ÉTUDES, DÉVELOPPEMENT & INTÉGRATION

LE LANGAGE Go

Les fondamentaux du langage

Frédéric G. Marand

LES + EN LIGNE

DUNOD

LE LANGAGE Go

Les fondamentaux du langage

Frédéric G. Marand
Consultant en architecture et qualité logicielle
Professeur de développement Go à l'EEMI

DUNOD

Illustration de couverture : https://thegrumpyunicorn.co
Inspirés de la mascotte Go gopher créée par Renée French, les pins illustrant la couverture font partie des créations de la société The Grumpy Unicorn, que vous pouvez vous procurer en ligne.

© Dunod, 2020
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
EAN 978-2-10-081442-8

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » [art. L. 122-4].
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

TABLE DES MATIÈRES

Table des listings XI

Avant-propos XV

1 Introduction : pourquoi Go ? 1

1.1 La programmation « in the large » 1

1.1.1 L'équation des coûts logiciels 2

1.1.2 Impact sur la conception du language 5

1.2 Nouvelles architectures 12

1.3 En pratique 14

1.3.1 Simplicité 14

1.3.2 Rapidité 15

2 Syntaxe 19

2.1 Alphabet 19

2.1.1 Jeu de caractères 19

2.1.2 Encodage 20

2.1.3 Classes de caractères 21

2.2 L'héritage syntaxique de C 21

2.3 Identificateurs 23

2.3.1 Forme et identificateurs réservés 23

2.3.2 Visibilité 25

2.3.3 Comparaison 26

2.4 Opérateurs 26

2.4.1 Opérateurs et types de données 26

2.4.2 Règles de priorité 28

2.4.3 Opérateurs multiplicatifs 30

2.4.4 Opérateurs additifs 31

2.4.5 Dépassement de capacité 31

2.4.6 Opérateurs de comparaison 32

2.4.7 Opérateurs logiques 34

2.4.8 Opérateurs d'adresse 35

2.4.9 Opérateur réception 36

2.5 Ponctuation 37

2.5.1 Opérateurs, instructions et ponctuation 37

2.5.2 Ponctuation générale 38

2.5.3 Espaces et autre formatages 38

2.6 Mots-clefs 40
2.6.1 Mots-clefs des définitions de données 40
2.6.2 Mots-clefs des structures de contrôle 41
2.6.3 Mots-clefs des paquets 42
2.6.4 Mots-clefs secrets 42

3 Types élémentaires 43
3.1 Types simples 43
3.1.1 Liste des types simples 43
3.1.2 Entiers et caractères 44
3.1.3 Nombres flottants réels et complexes 47
3.1.4 Chaînes de caractères 51
3.2 Types pointeurs 56
3.2.1 Définition 56
3.2.2 Déclaration de types pointeurs 57
3.2.3 Littéraux pointeurs 57
3.2.4 Manipulation des pointeurs 59
3.3 Types fonctions 63
3.3.1 Définition 63
3.3.2 Déclaration de types fonctions 63
3.3.3 Déclaration de fonctions 65
3.3.4 Littéraux fonction 65
3.3.5 Code des fonctions 66
3.3.6 Manipulation des fonctions 70
3.3.7 Chaînage de retours multiples 71
3.4 Méthodes d'un type 73
3.4.1 Définition 73
3.4.2 Déclaration de méthodes 74
3.4.3 Jeu de méthodes d'un type 74
3.4.4 Méthodes et champs fonctions 76
3.4.5 Utilité des méthodes sur les types non-structure 77
3.4.6 Récepteur pointeur ou récepteur valeur 80
3.4.7 Invocation dynamique 82
3.5 Types interfaces 85
3.5.1 Définition 85
3.5.2 Déclaration de types `interface` 86
3.5.3 Valeurs de types interfaces 89
3.5.4 Le type `interface{}` 91
3.5.5 Manipulation des interfaces 92
3.5.6 Organisation des interfaces 98

3.6 Compatibilité de types . . . 100
3.6.1 Identité de types . . . 100
3.6.2 Affectabilité . . . 102
3.6.3 Représentabilité des constantes . . . 105

4 Types composites . . . 109
4.1 Tableaux (`array`) . . . 109
4.1.1 Définition . . . 109
4.1.2 Déclaration de types tableaux . . . 110
4.1.3 Littéraux tableaux . . . 111
4.1.4 Manipulation des tableaux . . . 112
4.2 Tranches (`slice`) . . . 114
4.2.1 Définition . . . 114
4.2.2 Déclaration de types tranche . . . 115
4.2.3 Littéraux tranches . . . 116
4.2.4 Manipulation des tranches . . . 116
4.2.5 Les tranches comme paramètres . . . 120
4.3 Cartes (`map`) . . . 122
4.3.1 Définition . . . 122
4.3.2 Déclaration de types cartes . . . 122
4.3.3 Littéraux cartes . . . 122
4.3.4 Manipulation des cartes . . . 123
4.3.5 Les cartes comme paramètres . . . 125
4.4 Structures (`struct`) . . . 126
4.4.1 Définition . . . 126
4.4.2 Déclaration de types structure . . . 127
4.4.3 Champs intégrés et promotion . . . 129
4.4.4 Étiquettes de champs . . . 131
4.4.5 Littéraux structures . . . 132
4.4.6 Manipulation des structures . . . 134
4.5 Types canaux (`chan`) . . . 137
4.5.1 Définition . . . 137
4.5.2 Déclaration de types canaux . . . 138
4.5.3 Valeurs de types canaux . . . 139
4.5.4 Manipulation des canaux . . . 139
4.5.5 Extensions . . . 145

5 Données . . . 147
5.1 Variables . . . 147
5.1.1 Variables et types . . . 147
5.1.2 Déclaration des variables . . . 148
5.1.3 Utilisation des variables . . . 150

5.1.4 Quasi-variables : paramètres et retours . . . 150
5.1.5 Variables anonymes . . . 151
5.2 Constantes . . . 152
5.2.1 Constantes Go et autres langages . . . 152
5.2.2 Valeurs constantes . . . 152
5.2.3 Constantes et types . . . 154
5.2.4 Précision des valeurs constantes . . . 155
5.2.5 Déclaration des constantes . . . 156
5.3 Valeurs et types . . . 159
5.4 Valeurs vierges . . . 162
5.4.1 Définition . . . 162
5.4.2 Valeurs vierges des types de données . . . 163
5.4.3 Rendre la valeur vierge utile . . . 164
5.5 « `nil` » . . . 165
5.5.1 Définition . . . 165
5.5.2 `nil` pour les types interfaces . . . 166
5.5.3 `nil` comme récepteur . . . 166
5.5.4 Une erreur à un milliard de dollars ? . . . 170
5.6 Fin de vie . . . 170
5.6.1 Destructeurs . . . 170
5.6.2 Finaliseurs . . . 171
5.7 Réflexion et méta-programmation . . . 174
5.7.1 Définition . . . 174
5.7.2 Lecture d'étiquettes de structure . . . 174
5.7.3 Fonctions génériques . . . 177
5.7.4 API de réflexion . . . 178
5.7.5 Conclusion . . . 181

6 **Expressions** . . . 183
6.1 Opérateurs et opérandes . . . 183
6.2 Littéraux . . . 184
6.2.1 Règles générales . . . 184
6.2.2 Littéraux tableaux, tranches, et cartes . . . 184
6.2.3 Littéraux structures . . . 186
6.3 Qualification des identificateurs . . . 187
6.4 Expressions primaires . . . 188
6.5 Sélection . . . 189
6.6 Expressions de méthodes . . . 193
6.7 Valeurs de méthodes . . . 194
6.8 Indexation . . . 196
6.9 Tranchage (découpage) . . . 198

6.9.1 Définition . . . 198
6.9.2 Notation . . . 199
6.9.3 Contraintes et résultats . . . 200
6.9.4 Implantation mémoire . . . 201
6.10 Assertions de types . . . 204
6.10.1 Définition . . . 204
6.10.2 Notation . . . 205
6.10.3 Résultats . . . 205
6.11 Appels de fonctions/méthodes . . . 206
6.11.1 Définition et notation . . . 206
6.11.2 Arguments et résultats . . . 208
6.11.3 Chaînage d'appels . . . 209
6.12 Conversions . . . 211
6.12.1 Définition . . . 211
6.12.2 Notation . . . 212
6.12.3 Possibilités et limites . . . 214
6.12.4 Conversions numériques . . . 216
6.12.5 Conversions de chaînes . . . 217
6.12.6 Conversions d'interfaces . . . 218
6.13 Règles communes . . . 221
6.13.1 Expressions constantes . . . 221
6.13.2 Ordre d'évaluation . . . 222

7 Instructions . . . 225
7.1 Organisation de l'exécution . . . 225
7.1.1 Instructions terminales . . . 225
7.1.2 Instructions simples . . . 227
7.2 Instructions simples . . . 228
7.2.1 L'instruction vide . . . 228
7.2.2 Expressions . . . 228
7.2.3 Émission sur un canal . . . 230
7.2.4 Incrémentation / décrémentation . . . 230
7.2.5 Affectation . . . 231
7.2.6 Déclaration courte de variable . . . 234
7.3 Instructions générales . . . 237
7.3.1 Déclaration . . . 237
7.3.2 Instruction étiquetée . . . 240
7.3.3 Bloc . . . 241
7.3.4 Si - alors - sinon : `if - else` . . . 241
7.3.5 Commutateur d'expression `switch v` . . . 244
7.3.6 Commutateur de type `switch v.(type)` . . . 248
7.3.7 Sélecteur de communication `select` . . . 251

7.3.8 **for**, la boucle à tout faire . . . 257
7.3.9 **break**, l'interrupteur . . . 263
7.3.10 **continue**, le raccourci . . . 263
7.3.11 Déroutement **goto** . . . 265
7.4 Instructions liées aux fonctions . . . 268
7.4.1 Retour de fonction avec **return** . . . 268
7.4.2 Exécution concurrente avec **go** . . . 270
7.4.3 Code terminal avec **defer** . . . 272

8 **Builtins : fonctions intégrées et erreurs** . . . 275
8.1 Affichage : `print` et `println` . . . 276
8.2 Convertisseurs complexes . . . 277
8.3 Allocation : `make` et `new` . . . 278
8.4 Taille et capacité : `len`, `cap` . . . 280
8.5 Manipulation des tranches : `append`, `copy` . . . 282
8.6 Destruction d'éléments de carte : `delete` . . . 282
8.7 Fermeture de canaux par `close` . . . 282
8.8 Traitement des erreurs avec l'interface `error` . . . 283
8.8.1 Définition . . . 283
8.8.2 Le modèle Go de gestion des erreurs . . . 283
8.8.3 Création d'erreurs spécifiques . . . 285
8.8.4 Emballage et déballage d'erreurs . . . 286
8.8.5 L'expérimentation `check` / `handle` / `try` . . . 288
8.9 Situations de panique : `panic` et `recover` . . . 289
8.9.1 Définition . . . 289
8.9.2 Paniques d'exécution . . . 290
8.9.3 Utilisation appropriée . . . 292

9 **Organisation du code en paquets** . . . 295
9.1 Programmes et paquets . . . 295
9.2 La clause `package` . . . 297
9.2.1 Définition . . . 297
9.2.2 Clause paquets et compilation par fichier . . . 297
9.3 Répertoire d'un paquet . . . 298
9.4 Fichiers d'un paquet . . . 298
9.4.1 Répartition du code dans les fichiers . . . 298
9.4.2 Conventions de nommage . . . 299
9.5 Programmes exécutables et greffons : le paquet `main` . . . 300
9.6 Organisation des paquets . . . 303
9.6.1 Hiérarchisation des paquets . . . 303
9.6.2 Organisation des paquets d'un projet . . . 304
9.6.3 Réduction de la surface d'API avec les paquets `internal` . . . 304

9.7 Déclaration d'importation `import` . . . 305
9.8 Initialisation . . . 308
9.9 Chemins d'importation . . . 310
9.9.1 Notion de *workspace* et `GOPATH` . . . 310
9.9.2 Bibliothèque standard et `GOROOT` . . . 312
9.9.3 Paquets distants avec `go get` . . . 312
9.9.4 Organisation des paquets en mode `GOPATH` . . . 317
9.10 Modules VGO . . . 319
9.10.1 Inconvénients du mécanisme des espaces de travail . . . 320
9.10.2 Solutions antérieures aux modules . . . 320
9.10.3 Des objectifs du projet Go aux modules VGO . . . 322
9.10.4 Fonctionnement . . . 324
9.10.5 Utilisation au quotidien . . . 329

10 Programmation concurrente . . . 337
10.1 Concurrence et parallélisme . . . 337
10.2 Goroutines . . . 338
10.2.1 Définition . . . 338
10.2.2 Utilisation . . . 339
10.2.3 API . . . 340
10.2.4 Bonnes pratiques . . . 344
10.3 Contextes . . . 345
10.3.1 Définition . . . 345
10.3.2 Utilisation . . . 346
10.3.3 Exemple : annulation de handler HTTP . . . 348
10.4 Primitives de synchronisation : le paquet `sync` . . . 351
10.4.1 Groupe des tâches : `sync.WaitGroup` . . . 352
10.4.2 Exécution unique : `Once` . . . 354
10.4.3 Exclusion mutuelle : `Mutex / RWMutex` . . . 355
10.4.4 Allocations réutilisées : `Pool` . . . 357
10.4.5 Tableaux associatifs : `Map` . . . 358
10.4.6 Moniteurs : `Cond` . . . 359
10.5 Bas niveau : le paquet `sync/atomic` . . . 361
10.6 Mécanismes spécialisés : le paquet `golang.org/x/sync` . . . 362
10.6.1 Sémaphores : le paquet `semaphore` . . . 362
10.6.2 Regroupement d'erreurs et annulation : le paquet `errgroup` . . . 363
10.6.3 Dédoublonnage d'exécution : le paquet `singleflight` . . . 364
10.6.4 Cartes concurrentes : le paquet `syncmap` . . . 365
10.7 Situations de compétition : le *race detector* . . . 366

Index . . . 367

TABLE DES LISTINGS

2.1 Nécessité des accolades autour des blocs C 23
2.2 Masquage d'identificateur dans un bloc enfant 25
2.3 L'affectation comme opérateur en JavaScript 27
2.4 L'affectation comme non-opérateur en Go 27
2.5 Nécessité de types identiques pour les opérateurs binaires 28
2.6 Association gauche-droite des opérateurs unaires 29
2.7 Interférence de priorité entre opérateurs sur les nombres complexes 30
2.8 L'opérateur `AND NOT` 31
2.9 $a < a + 1$ n'est pas toujours vrai en Go 32
2.10 Division flottante par 0 33
2.11 Comparaison de structures 34
2.12 Adresse de littéraux composites 36
2.13 Le placement du **`else`** 39
2.14 C vs Go : `type`, `typedef` et `struct` 41

3.1 Dépassement de précision sur entiers 46
3.2 Types des builtins complexes 50
3.3 Imprécision des opérations en virgule flottante 51
3.4 Itération sur les chaînes 52
3.5 Littéraux pour pointeurs 58
3.6 Comparaison de pointeurs 59
3.7 Pointeurs sur données de longueur nulle 60
3.8 Pointeurs multi-niveaux 62
3.9 Paramètres et résultats de fonction nommés et anonymes 64
3.10 Littéraux fonction et récursion en ECMAScript 66
3.11 Littéraux fonctions et récursion en Go 67
3.12 Les quatre formes de retour de fonction 68
3.13 Comparaison de fonctions et pointeurs 70
3.14 Chaînage de fonction à retours multiples 72
3.15 Déclaration de méthodes 75
3.16 Méthodes de types intégrés 77
3.17 Sources de `Handler` et `HandlerFunc` 78
3.18 Utilisation de `Handler` et `HandlerFunc` 79
3.19 Récepteurs valeur et pointeur 81
3.20 Invocation dynamique par interface intégrée 83
3.21 Invocation dynamique par champ fonction 84
3.22 Récepteurs valeurs et adresse pour une même interface 87
3.23 Interfaces intégrées 88

3.24 Types statique et dynamique d'une interface . . . 89
3.25 Pointeurs et interfaces . . . 91
3.26 Conversion de composites d'interface vide . . . 93
3.27 `nil` est différent de `nil`, ou pas . . . 94
3.28 Conversion vers interface . . . 95
3.29 Commutateur de type interface . . . 97
3.30 Vérification d'existence d'une méthode . . . 99
3.31 Types définis et alias de types . . . 101
3.32 Identité de types structure entre paquets . . . 103
3.33 Affectabilité de `nil` . . . 104
3.34 Représentabilité numérique des constantes caractères . . . 106

4.1 Littéraux tableaux . . . 111
4.2 Conversion et comparaison de tableaux . . . 113
4.3 Adresse de tableau et d'éléments . . . 114
4.4 Littéraux tranches . . . 116
4.5 Adresse de tranche et d'élément . . . 117
4.6 Agrandissement de tranche avec `append` . . . 118
4.7 Fonction `copy` de tranches . . . 119
4.8 Tranches et tableaux comme paramètres de fonction . . . 121
4.9 Littéraux cartes . . . 123
4.10 Déclaration d'un type structure . . . 128
4.11 Champ intégré dans une structure . . . 129
4.12 Les deux formes des littéraux structure . . . 133
4.13 Comparaison de structures . . . 134
4.14 Conversion de structures . . . 135
4.15 Adresses de structures et champs . . . 136
4.16 Longueur d'un canal en situation de compétition . . . 143
4.17 La fermeture d'un canal comme mode de diffusion . . . 144

5.1 Type statique et dynamique . . . 148
5.2 Déclaration `var` multilignes . . . 149
5.3 Utilisation obligatoire des variables . . . 150
5.4 Valeur par défaut des retours nommés . . . 151
5.5 Expressions constantes avec `len` et `cap` . . . 154
5.6 Constantes entières de plus de 64 bits . . . 155
5.7 Constantes à initialisation parallèle . . . 157
5.8 Constantes à valeur répétée . . . 157
5.9 L'identificateur `iota` . . . 158
5.10 Expressions avec `iota` . . . 160
5.11 Comparaisons entre alias et types définis . . . 161
5.12 Initialisation récursive des valeurs vierges . . . 162
5.13 Redéfinition de `nil` . . . 166

5.14 `nil` n'est pas toujours égal à lui-même . . . 167
5.15 Appel de méthodes sur `nil` concret . . . 168
5.16 Appel de méthodes sur `nil` interface . . . 169
5.17 Finaliseur et `runtime.KeepAlive` . . . 172
5.18 Étiquettes structurées dans l'ORM GORM . . . 175
5.19 Lecture d'étiquettes de structure . . . 176
5.20 Comparaison par `reflect.DeepEqual` . . . 178
5.21 Permutation dans une tranche avec `reflect.Swapper` . . . 178
5.22 Introspection et intercession . . . 180

6.1 Littéraux tableaux et tranches . . . 185
6.2 Omission des types d'éléments dans les littéraux composites . . . 186
6.3 Littéraux structures et champs non exportés . . . 187
6.4 Sélection et profondeur de champs . . . 190
6.5 Sélecteurs homonymes et ambigus . . . 191
6.6 Sélecteurs sur expressions pointeurs . . . 191
6.7 Sélection sur `nil` . . . 192
6.8 Expressions de méthodes . . . 193
6.9 Valeurs de méthodes . . . 195
6.10 Exemples d'indexation . . . 197
6.11 Syntaxe du tranchage . . . 199
6.12 Tranchage de tableau et adressabilité . . . 201
6.13 Réutilisation mémoire lors du tranchage . . . 202
6.14 Implantation mémoire du tranchage . . . 203
6.15 Tranchage de `nil` . . . 204
6.16 Assertions de type . . . 206
6.17 Panique sur assertion de type . . . 207
6.18 Quatre formes d'expressions d'appel . . . 208
6.19 Appel de fonction variadique . . . 210
6.20 Chaînage de fonctions à retours multiples . . . 211
6.21 Notation des conversions explicites . . . 213
6.22 Possibilités et limites de conversion . . . 214
6.23 Conversion chaîne vers entier . . . 216
6.24 Conversion d'entiers . . . 216
6.25 Conversion de chaînes . . . 218
6.26 Expression de conversion entre interfaces . . . 219
6.27 Types des expressions à opérandes hétérogènes . . . 222

7.1 Instruction terminale d'une liste . . . 227
7.2 Instructions vides . . . 228
7.3 Instructions expression . . . 229
7.4 Instructions d'affectation . . . 232
7.5 Échange de valeurs . . . 234
7.6 Redéclarations courtes parallèles . . . 235

7.7 Variables temporaires . . . 236
7.8 Redéclaration dans une boucle de goroutines . . . 237
7.9 Les identificateurs « _ »et « init » . . . 239
7.10 Instructions **if**, **else** et ternaires . . . 242
7.11 L'instruction **switch** d'expression . . . 244
7.12 Instruction **fallthrough** . . . 248
7.13 Utilisation pratique de **fallthrough** . . . 249
7.14 L'instruction **switch** de type . . . 250
7.15 L'instruction **select** . . . 253
7.16 Temporisation avec `time.After` . . . 256
7.17 Quatre **for** de 0 à 9 . . . 258
7.18 Instructions simples de clause **for** . . . 259
7.19 Modification d'une expression de boucle **for** . . . 262
7.20 Interruption avec **break** . . . 264
7.21 Itération anticipée avec **continue** . . . 265
7.22 Émulation d'une boucle **for** avec **goto** . . . 266
7.23 Les quatre formes de **return** . . . 269
7.24 Exécution concurrente avec **go** . . . 271
7.25 Code terminal avec **defer** . . . 273
7.26 Accès aux résultats de fonction dans une fonction différée . . . 274

8.1 Différence entre une fonction normale et une fonction intégrée . . . 276
8.2 Conversions entre valeurs complexes et réelles . . . 278
8.3 Allocation et initialisation par `new` et `make` . . . 279
8.4 Longueur et capacité des tranches . . . 281
8.5 Équivalent de `delete` pour tranches et tableaux . . . 282
8.6 Erreurs constantes . . . 286
8.7 Situation de panique . . . 290
8.8 Utilisation de `panic` et `recover` . . . 291
8.9 Paniques d'exécution réelles et simulées . . . 292

9.1 Documentation de paquet : extrait de `fmt/doc.go` . . . 300
9.2 Fin d'exécution ordonnée avec et `os.Exit` . . . 302
9.3 Importation de paquets multiples sur une seule ligne . . . 307
9.4 Importation de code en langage C avec CGO Non utilisable sur . . . 309
9.5 Fichier `go.mod` minimal . . . 329

10.1 Informations de l'ordonnanceur avec `schedtrace` . . . 341
10.2 Informations sur les goroutines avec `schedtrace+scheddetail` . . . 341
10.3 Affichage de la pile d'une goroutine . . . 343
10.4 Trois approches pour un handler HTTP à durée limitée . . . 349
10.5 Utilisation de `sync.WaitGroup` . . . 353
10.6 Utilisation de `sync.Once` . . . 354

Avant-propos

« *Go is a programming language designed by Google to help solve Google's problems, and Google has big problems.* »

« *The language was designed by and for people who write—and read and debug and maintain—large software systems.* » [1]

Rob Pike
https://osinet.fr/go/intro/go-splash2012

Lorsque ce livre n'était qu'un projet, il était intitulé *Go deuxième langue*, et ces trois mots résument sa raison d'être.

Alors que Go présente une syntaxe extraordinairement simple, les logiciels développés avec sont en revanche souvent complexes et critiques, que ce soit par leur fonctionnalité propre ou par les niveaux de charge auxquels ils sont : il est couramment utilisé pour écrire des outils système (Docker), des bases de données spécialisées et des outils associés (Prometheus, InfluxDB, les outils de MongoDB), ou des services web à très forte charge comme https://dl.google.com

Ce sont là des tâches très différentes de celles du développement de sites web ou d'applications de gestion courantes.

De tels projets, par nature, ne sont pas confiés à des développeurs débutants mais à des programmeurs expérimentés, dont Go n'est jamais le premier langage mais au moins le deuxième, et souvent bien plus. Cette orientation se retrouve à tous les niveaux en Go, que ce soit dans les choix de conception du langage, dans sa documentation en ligne, ou dans les sujets présentés dans les nombreuses conférences organisées par la communauté Go.

Dès lors, au moment de concevoir le livre, une approche m'a paru s'imposer : plutôt que de simplement décrire le langage en ignorant le reste du monde comme il est de mise, le livre ne demanderait pas à ses lecteurs de faire abstraction de toutes leurs connaissances existantes mais s'appuierait au contraire sur elles, en présentant chaque nouvelle notation et chaque nouveau concept avec l'équivalent le plus proche dans d'autres langages.

1. « Go est un langage conçu par Google pour les problèmes rencontrés par Google, et les problèmes rencontrés par Google sont vastes. »
« Le langage a été conçu par et pour des gens qui écrivent – et lisent et déboguent et maintiennent – des systèmes logiciels à grande échelle. »

Cette approche combine deux mécanismes :

- ✓ la reconnaissance de motifs (« ah, oui, c'est comme XXX en YYY ! »), facilitant la mémorisation en tirant parti des capacités de l'esprit humain à classer l'information par similitude ;
- ✓ le pointage des différences, pour éviter que ces similitudes ne soient prises à tort pour une exacte identité, source d'erreurs difficiles à déboguer.

Voilà pourquoi, tout au long du livre, vous trouverez des exemples dans de multiples autres langages, rapprochant et opposant la version Go à celle de votre précédent langage.

Les langages les plus cités sont Python, en raison de sa popularité, et ceux de la même lignée syntaxique que Go : C, Java, JavaScript et PHP, qui représentent le principal vécu des nouveaux développeurs Go et donc – tel est notre espoir – le vôtre si vous lisez cet avant-propos.

Ce livre est-il pour vous ?

Ce livre a été conçu pour celles et ceux qui veulent apprendre Go, en vue de le maîtriser dans un contexte professionnel, et tout particulièrement si ce contexte est techniquement exigeant.

Si vous êtes déjà familier avec la programmation dans un autre langage, et en particulier si ce langage est JavaScript, PHP, Python, C ou Java, ce livre est tout particulièrement optimisé pour vous : vous allez retrouver dans Go des constructions déjà familières, mais dont les détails sémantiques changent d'un langage à l'autre ; et vous verrez, chaque fois qu'une telle différence existe, en quoi la version Go d'une fonctionnalité diffère de ses homologues dans ces autres langages, pour vous éviter les erreurs souvent subtiles résultant de ces différences en profondeur.

Si vous êtes déjà développeur Go mais que vous voulez approfondir votre connaissance, il est aussi pour vous : vous y découvrirez des détails d'implémentation pour mieux comprendre le comment et le pourquoi des mécanismes que vous utilisez au quotidien dans le langage et sa bibliothèque standard.

Enfin, si vous n'avez utilisé la programmation concurrente qu'au travers des canaux et goroutines, voire pas du tout, vous y découvrirez tout l'éventail des mécanismes fournis par le langage, sa bibliothèque standard (`sync` et `sync/atomic`) ou sa bibliothèque étendue (`golang.rg/x/sync`).

En revanche, il se peut que ce livre ne soit pas optimisé pour vous, dans deux principaux cas :

- ✓ Vous cherchez à apprendre la programmation en général, sans avoir une expérience dans d'autres langages : dans ce cas, Go lui-même n'est probablement pas le meilleur choix pour commencer ; Python pourrait bien être un meilleur choix, et le *Python précis et concis* de Mark Lutz une bonne référence.
- ✓ Vous cherchez simplement à découvrir le langage, pour une première impression : dans ce cas, le site https://go.dev et son didacticiel en ligne sur https://tour.golang.org seront plus appropriés pour ce premier contact.

Organisation

Le livre a été conçu selon une démarche globalement ascendante de fonctionnalité du langage, partant de la raison d'être pour finir dans les outils de programmation concurrente. Il décrit Go 1.14, les versions antérieures et indique des évolutions annoncées pour les version futures.

Le chapitre 1 décrit les problématiques qui ont conduit Google à valider et soutenir le projet de ce nouveau langage, comme solution possible aux problèmes rencontrés dans la programmation à grande échelle sur les architectures modernes, et en quoi Go répond à ces besoins.

À partir du chapitre 2 commence l'exploration du langage proprement dit, commençant par les éléments lexicaux, de l'encodage aux mots-clefs.

Les chapitres 3 et 4 décrivent les types de données disponibles dans le langage et la création des types métier, tandis que le chapitre 5 décrit les données pouvant être représentées par ces types.

Les chapitres 6, 7 et 8 montent d'un niveau dans la fonctionnalité, en décrivant les expressions, les instructions et les fonctions intégrées qui manipulent ces données, jusqu'aux situations d'erreur, dont le traitement en Go est inhabituel par rapport aux autres langages et a évolué récemment en Go 1.13.

Le chapitre 9 quitte l'organisation des blocs de code pour décrire les mécanismes d'organisation entre fichiers et répertoires, avec les notions de paquets et de modules.

Enfin, construit sur tout ce qui précède, le chapitre 10 décrit les mécanismes pour lesquels Go est le plus souvent mentionné : ceux liés à la programmation concurrente. Au-delà des outils de base que sont les canaux, goroutines et contextes, vous y découvrirez aussi bien des briques plus élémentaires comme les opérations atomiques ou les mutexes, que des types composés simplifiant la programmation de situations couramment rencontrées, comme les groupes d'attente, l'exécution unique, les lots d'allocation, les cartes à accès concurrent, et bien d'autres.

© Dunod – Toute reproduction non autorisée est un délit.

Les *Plus en ligne*

Le code source et les exemples de ce livre sont disponibles sur le Go Playground : chaque listing comporte un URL permettant d'y accéder et, pour la plupart d'entre eux, de les exécuter sans nécessiter d'installation locale.

Le site des *Plus en ligne* https://osinet.fr/go comporte divers compléments à ce livre, pour prolonger sa durée de vie au-delà du contenu initial :

- ✓ **Articles** : Actualités importantes du projet Go, PopQuiz avec solutions, articles originaux sur le quotidien du développement avec Go.
- ✓ **Dossiers** : Les dossiers examinent des sujets avancés qui n'avaient pas leur place dans le livre lui-même, comme l'utilisation du paquet `unsafe`, la création de greffons (*plugins*) ou le fonctionnement de l'ordonnanceur (*scheduler*) et du ramasse-miettes (*garbage collector*).
- ✓ **Errata** : Une liste d'erreurs ou imprécisions qui auront pu être identifiées dans le livre, avec leur discussion et leur correction.

Le site contient également la table de correspondance qui permet aux URL « humanisés » du livre de toujours pointer vers la bonne page de destination même si son URL réel évolue.

Conventions

Ce livre utilise quelques conventions pour faciliter sa lecture.

- ✓ **Typographie** :
  - L'utilisation du souligné dénote des URL, comme https://golang.org/ Dans les éditions numériques, ceux-ci sont en outre de couleur bleue et sont cliquables pour faciliter leur utilisation.
  - L'utilisation d'une police à chasse fixe dénote des éléments de code, quel qu'en soit le langage : commandes à saisir (`composer install`), exemples d'extensions de fichier (`*.jar`), noms de structures de données (`struct`), et plus généralement toute information manipulée sur la ligne de commande ou dans un éditeur de texte.
  - Lorsque ce texte à chasse fixe présente en outre une colorisation (éditions numériques) ou des niveaux de gris et de graisse variables, il représente des fragments écrits en Go, comme dans ce fragment : `var n int = 42`.
- ✓ **Encadrés** : certains paragraphes sont isolés dans un encadré avec une loupe ou un triangle danger dans le coin supérieur gauche. La loupe est utilisée pour indiquer un approfondissement lié à la section en cours mais n'en faisant pas directement partie, tandis que le triangle d'alerte signale un danger.

- ✓ **Exercices et solutions** : Les exercices suggérés proposent au lecteur de concevoir un fragment de code sur la base de ce qui vient d'être décrit. Ils sont mis en évidence par un liséré en marge commençant par un crayon.

Le paragraphe **solution** associé à un **exercice** contient une réalisation possible, présentée sur le Go Playground pour du code original, ou choisie dans le code source existant de la bibliothèque Go elle-même.

✓ **Listings** : Les listings hors texte sont de deux formes :

- ceux qui comportent une barre latérale entre les numéros de ligne et le code lui-même sont rédigés en Go ;
- ceux qui ne la comportent pas sont rédigés dans d'autres langages.

Tous ces listings sont numérotés, les numéros de ligne étant fréquemment utilisés dans le texte avoisinant pour rattacher précisément un point de syntaxe avec son utilisation pratique.

Certains de ces listings ne commencent pas à la ligne 1 : lorsque c'est le cas, les lignes omises le sont pour alléger l'exemple et se limitent à la déclaration de paquet et aux déclarations d'importation. Dans tous les cas, le code complet est présent sur le Go Playground, à l'URL indiqué dans la description du listing.

Prérequis

Aucune connaissance préalable en Go ou en aucun autre langage n'est nécessaire. En revanche, pour tirer le meilleur parti du livre, il est préférable :

✓ d'avoir déjà une première expérience de la programmation dans un autre langage, quel qu'il soit : Go n'est pas conçu pour l'apprentissage initial de la programmation. En particulier, la connaissance de JavaScript, PHP, Python, Java ou C facilitera la compréhension des concepts exposés ;

✓ d'avoir des notions de base sur les composants d'un ordinateur, et en particulier les processeurs, cœurs, registres, bus, caches, mémoire et disque.

La quasi-totalité des exemples, exercices et solutions ne nécessitent – grâce au Go Playground – qu'un navigateur et une connexion internet, même sur une simple tablette.

Pour travailler au quotidien, que ce soit pour créer des programmes multi-fichiers, lire les sources de la bibliothèque Go, ou utiliser un débogueur ou les outils en ligne de commande qui accompagnent le langage, il est toutefois nécessaire de disposer d'un poste de travail, avec un éditeur de texte ou un IDE comme Goland ou Visual Studio Code.

C'est sur Linux et FreeBSD que la simplicité de mise en œuvre du langage est la plus grande. L'expérience sur macOS est comparable, à condition de compléter l'installation standard par les outils ligne de commande de XCode. Le travail sous Windows est plus contraignant, mais possible également : la difficulté n'est pas au niveau du langage lui-même, mais plus dans tous les outils annexes.

© Dunod – Toute reproduction non autorisée est un délit.

Remerciements

Un livre n'existerait pas sans de multiples participants à sa création, autour de son auteur.

Je remercie donc tout particulièrement ma famille et amis :

✓ Aux origines, feu ma mère, à qui je dois en particulier à la fois toute mon éducation et l'envie d'écrire. Et tout le reste.

✓ Au quotidien, Dominique, qui m'a encouragé à entreprendre ce projet et soutenu pendant toute la durée de sa réalisation.

✓ Au cours du projet, Chantal, qui a relu les débuts du manuscrit et fourni des recommandations pour le rendre plus accessible et plus lisible.

Et, dans un second cercle, tous ceux qui sans être aussi proches ont impacté de façon parfois décisive l'existence et le contenu de ce livre :

✓ L'équipe des éditions Dunod, et en particulier mon éditeur, Jean-Luc Blanc, pour m'avoir encouragé à l'écriture de ce livre et aidé à cadrer son contenu, et Brice Martin pour ses relectures et son souci de précision dans l'organisation du texte.

✓ Les auteurs de Go, sans lesquels cet ouvrage n'aurait pas lieu d'être, et les contributeurs au projet Go et leurs blogues respectifs qui m'ont incité à lire le code source du langage et de sa bibliothèque standard, et en particulier Russ Cox, Dave Cheney, Jaana B. Dogan et Vincent Blanchon.

✓ Les étudiants de mes classes de Go à l'EEMI (https://eemi.com) : ils ont été les vaillants cobayes du cours qui a fourni la matière de ce livre et ont permis d'en améliorer le contenu par leurs critiques souvent pertinentes.

Introduction : pourquoi Go ?

1.1 LA PROGRAMMATION « IN THE LARGE »

Alors que les langages les plus utilisés dans les projets web comme PHP ou Java Script ont été à l'origine conçus pour des projets de petite ou moyenne taille, l'explosion des besoins des grands opérateurs comme Google – le créateur du langage Go – a mis en évidence que leurs besoins étaient différents, et relevaient d'une branche spécifique de l'informatique, la programmation *in the large* (à grande échelle).

Celle-ci se caractérise par l'un au moins de trois critères :

- ✓ code volumineux et/ou complexe ;
- ✓ grand nombre de développeurs et/ou parties prenantes ;
- ✓ longue durée de vie.

Elle a été caractérisée dès 1975 par deux articles :

- ✓ Fred Brooks : *The mythical man-month* [1], article séminal par excellence, au point d'avoir fait l'objet d'un livre complet portant le même titre, et d'avoir été réédité, toujours aussi pertinent, en 1995.
- ✓ Frank DeRemer / Hans Kron : *Programming in the large versus programming in the small* [2].

Les projets de ce type sont confrontés à des difficultés particulières, parmi lesquelles les plus notoires sont :

- ✓ La croissance quadratique du coût de la communication.

1. https://osinet.fr/go/intro/wiki-m3
2. https://osinet.fr/go/intro/deremer-inthelarge.pdf

En effet, dans une communauté de n développeurs, la communication requiert $n(n-1)$ échanges, induisant un temps de communication en $\mathcal{O}(n^2)$, alors que la production progresse au mieux en $\mathcal{O}(n)$, de sorte que le rapport du temps passé à communiquer sur le temps passé à produire tend vers $\mathcal{O}(n)$.
La production tend alors vers 0 en l'absence de mesures correctives adaptées, telles que la communication diffusée et la hiérarchisation et subdivision en projets concurrents à plus petite échelle.

- ✓ Le renouvellement des intervenants.
  En effet, pour une durée d'emploi donnée, plus le nombre d'intervenants devient long, plus la probabilité qu'un intervenant quitte le projet et doive être remplacé augmente. De même, plus la durée du projet s'allonge, plus cette probabilité augmente.
  Le projet doit alors supporter la charge de la perte de connaissances internes et l'acquisition des connaissances du projet – d'un volume sans cesse croissant – par les nouveaux entrants, réduisant là aussi la proportion de temps productif en l'absence de mesures correctives adaptées comme des actions fortes de documentation vivante, et l'industrialisation des déploiements d'environnements de développement, comme le permet la généralisation des systèmes de conteneurs, Docker[3] en tête.

1.1.1 L'équation des coûts logiciels

Au-delà de ces notions familières aux développeurs travaillant en entreprise, l'analyse de la programmation à grande échelle a mis en évidence une structure de coûts particulière dès lors que la dimension *temps* est intégrée.

La formule suivante, décrite par Mat Kanat-Alexander dans son ouvrage *Code Simplicity*[4], permet d'analyser la désirabilité d'un projet du point de vue de l'organisation qui le finance :

$$D(t) = \frac{Pv(t) * V_0}{C_0 + Cm(t)}$$

Avec :

- ✓ $D(t)$ désirabilité du projet au temps t ;
- ✓ $Pv(t)$ probabilité de valeur au temps t ;
- ✓ V_0 valeur d'implémentation initiale ;
- ✓ C_0 coût initial de développement ;
- ✓ $Cm(t)$ coût cumulé de la maintenance et de la possession au temps t (TCO : *Total Cost of Ownership*).

3. https://osinet.fr/go/intro/dunod-docker
4. https://osinet.fr/go/intro/kanat-code-simplicity

Au fil du temps :

$lim_{t\to\infty} Pv(t) = 1$	reflétant l'adéquation du produit aux besoins de sa cible ;
$lim_{t\to\infty} Cm(t) = \infty$	en raison du coût périodique de la maintenance évolutive, et des frais d'hébergement périodiques ;
V_0 et C_0	demeurent constants par nature.

Par conséquent, au fil du temps [5] :

$$lim_{t\to\infty} D(t) = lim_{t\to\infty} \frac{Pv(t) * V_0}{C_0 + Cm(t)}$$

Donc, puisque C_0 et V_0 sont constants :

$$lim_{t\to\infty} D(t) = \frac{lim_{t\to\infty}(Pv(t)) * V_0}{C_0 + lim_{t\to\infty} Cm(t)}$$

Soit, en un temps fini mais grand, s'agissant d'un projet à grande échelle, et compte tenu des tendances mentionnées pour $Pv(t)$ et $Cm(t)$:

$$D(t) \equiv \frac{V_0}{Cm(t)}$$

Il apparaît donc que, dans le cadre de tels projets à grande échelle, la maximisation de la désirabilité passe essentiellement par :

- ✓ la réduction de $Cm(t)$. Puisque ce coût est monotone croissant par nature, ceci se traduit par la minimisation de son rythme périodique de progression, généralement évalué comme coût annuel de maintenance et d'exploitation ;
- ✓ la maximisation de V_0, la valeur de l'implémentation initiale.

1.1.1.1 Réduire le coût total de possession Cm(t)

Les moyens par lesquels la conception d'un nouveau langage peut contribuer à réduire le coût de maintenance en contenant les facteurs de coûts ont été identifiés comme suit :

- ✓ **Rotation des équipes au fil de la vie du produit** : réduire le coût de la rotation de personnel :
  - réduire le niveau de formation nécessaire par un langage rapide à apprendre :
    - définir un langage avec une grammaire minimale et orthogonale,
    - utiliser la forme syntaxique la plus familière : celle du langage C ;
  - réduire le niveau de compétence nécessaire avec un langage souple mais aussi simplifié que possible :

5. Les notations symboliques utilisées ne le sont pas dans leur sens mathématique strict, mais dans leur approximation usuelle en matière d'évaluations informatiques.

© Dunod – Toute reproduction non autorisée est un délit.

  - absence d'arithmétique de pointeurs,
  - gestion mémoire par ramasse-miettes,
  - soutien natif de la programmation concurrente,
  - généralisation de la composition au lieu de l'héritage comme mécanisme de base de la programmation par objets ;
- réduire la variabilité des styles et outils :
  - définir avec le langage les règles de formatage pour éviter les divergences entre project,
  - fournir avec le langage les outils de mise en forme et de génération de documentation,
  - encourager des méthodes de codage idiomatiques en les définissant en complément de la référence du langage [6],
  - fournir une bibliothèque d'exécution suffisante pour une majorité de projets sans nécessiter de bibliothèques tierces ;
- réduire la variabilité des pratiques de qualité par un outillage standard :
  - inclure les outils de test et de benchmarking,
  - inclure les outils de profilage,
  - inclure les outils de déboguage du code concurrent.

✓ **Évolutions du langage et des bibliothèques** : réduire la variabilité du code source au fil du temps :
- garantir la compatibilité du langage au fil des versions. La *promesse de compatibilité Go 1* [7] garantit que tout code écrit avec le langage seul ou ses API publiques sera compatible avec toute version de Go 1.x. Cette promesse est tenue depuis 2009.

1.1.1.2 Augmenter la valeur d'implémentation V_0

Go étant un projet créé par Google plutôt qu'une production académique, les applications cibles justifiant sa création et son entretien ont été, au moins pendant la première décennie, les projets de Google, qui ont en commun d'être à grande échelle, à fort trafic, et déployés sous forme de services multi-répliqués.

Ceci a défini les grandes orientations sur la conception du langage [8] :

✓ **Maximisation de l'utilisation du matériel** : permettre au code de tirer parti au maximum des processeurs multi-cœurs, au moyen d'une programmation concurrente apte à être exécutée en mode parallélisé sur des cœurs multiples en fonction du matériel disponible.

6. https://osinet.fr/go/intro/golang-effective
7. https://osinet.fr/go/intro/golang-go1compat
8. https://osinet.fr/go/intro/golang-talks-20091030.pdf

- ✓ **Langage pour programmes système** : optimiser le langage et sa bibliothèque pour la réalisation de services plutôt qu'autour de l'interaction utilisateur.
- ✓ **Robustesse** : éviter les erreurs d'accès mémoire en gérant la mémoire avec un ramasse-miettes, et pour y parvenir supprimer leur principale cause, l'arithmétique de pointeurs.
- ✓ **Interface avec du code existant** : permettre au code Go de s'exécuter avec les conventions d'appel et d'édition de liens du langage C, dominant dans les couches basses d'infrastructure logicielle.
- ✓ **Simplification du déploiement** : Google déployant son code sur des grappes de machines massives dans ses centres de données, le choix d'un langage compilé avec édition de liens statiques permet de déployer la plupart des programmes par simple copie du binaire compilé, sans nécessiter de déploiement d'interpréteur, de machine virtuelle, de bibliothèques partagées, voire de programmes et bibliothèques de dépendances à la mode de PHP (`composer install`) et JavaScript (`npm install`).
- ✓ **Performance d'exécution** : langage compilé natif, fortement typé.
- ✓ **Maîtrise des dépendances et rapidité de compilation** : pour éviter les problèmes d'incompatibilité de versions et de lenteur de compilation de C/C++, liés entre autres à la séparation entre fichiers d'en-tête (`*.h`/`*.hpp`) et fichiers de code (`*.c`/`*.cpp`), construction d'un mécanisme de liaison entre modules de code ne nécessitant pas d'en-têtes séparés.

1.1.2 Impact sur la conception du language

1.1.2.1 Compilation anticipée

La mise en œuvre des langages de programmation passe par la traduction du code source en un format exécutable, pour lequelle il existe trois grandes stratégies :

- ✓ **Interprétation** : l'interpréteur analyse le source à la volée, empilant les jetons analysés au fil de l'analyse syntaxique, et les dépilant dès qu'il identifie un fragment correspondant à une instruction exécutable complète. C'est le modèle de la plupart des langages de scripts jusqu'à l'apparition de PERL, et aussi le modèle originel de PHP.
- ✓ **Compilation à la volée** : le compilateur analyse le source à la volée et produit un code intermédiaire, dit *bytecode*, qui est utilisé comme format pivot pour être ensuite soit interprété – en faisant cette fois l'économie de la passe d'analyse de source – soit compilé en format exécutable pour la machine sur laquelle l'exécution vient d'être demandée. Cette stratégie a permis à PERL, Python, ou Ruby d'atteindre des niveaux de performance suffisants pour nombre de projets web courants. Elle est en particulier directement observable en Python, qui produit le

© Dunod – Toute reproduction non autorisée est un délit.

code intermédiaire sous forme de fichiers (`*.pyc`), ou en Java avec sa machine virtuelle JVM, dont les programmes sont généralement distribués sous forme de code intermédiaire (`*.class`) ou d'archives de celui-ci (`*.jar`, `*.war`).
L'avantage de cette stratégie est de permettre d'optimiser la génération de code pour la machine physique sur laquelle le code est exécuté. Son inconvénient principal est la nécessité de procéder à cette compilation secondaire à chaque lancement du programme : peu impactant pour les programmes à longue durée de vie, ce temps de compilation est problématique pour les programmes à exécution courte tels que les utilitaires sytème, qui font partie de la cible Go (cf. § 1.1.1.2). Le terme usuel pour cette stratégie est *Just in Time compiling (JIT)*.

✓ **Compilation anticipée** : le compilateur est sollicité après l'écriture du code plutôt qu'au moment de l'exécution, et produit un code exécutable natif. L'urgence de compilation étant moindre à ce stade que lors de l'exécution, le compilateur peut passer plus de temps à produire un code optimisé que dans la stratégie JIT, permettant ainsi d'obtenir de meilleures performances à l'exécution, au prix du temps passé durant la compilation.

Figure 1.1 – Le problème numéro 1 de la compilation anticipée
©2010 Randall Munroe - Sous licence Creative Commons BY-NC 2.5

Le terme usuel pour cette stratégie est *Ahead of Time compiling (AOT)*. C'est celle choisie par les langages « classiques » comme C ou C++.

Par défaut, le code fabriqué cible l'architecture matérielle et le système d'exploitation de la machine sur laquelle la compilation a lieu. Il est aussi possible de compiler pour une autre cible, technique dite de *compilation croisée*. Ceci permet de faciliter le déploiement de code optimisé en préparant les binaires dès la machine de développement au lieu de requérir une étape de compilation finale sur la machine cible, susceptible d'être problématique pour les cibles de l'Internet des objets (*IoT : Internet of Things)*.

Afin d'obtenir la meilleure performance d'exécution tout en la combinant avec des déploiements simples, c'est la stratégie de compilation anticipée qui a été retenue pour la chaîne de développement Go standard, en incluant par défaut les mécanismes de compilation croisée.

1.1.2.2 Éditions de liens statique

Le choix d'une compilation anticipée et potentiellement croisée étant arrêté, une difficulté peut se présenter lors du déploiement sur un système cible, celui du format d'édition de liens.

En effet, pour des raisons tenant surtout au coût historique de la mémoire centrale, la plupart des systèmes d'exploitation encouragent l'utilisation de versions partagées de toutes les bibliothèques de code. Ce mécanisme est dénommé édition de liens dynamique, et il permet de minimiser la taille des exécutables et de mutualiser l'occupation mémoire des bibliothèques utilisées par plusieurs programmes présents simultanément en mémoire. En contrepartie, il conduit à un double problème :

- ✓ la compatibilité des versions de bibliothèques cohabitant sur un système, tout particulièrement sur les système de la famille Microsoft® Windows®[9], problème connu sous le nom de *DLL Hell*[10] ;
- ✓ la nécessité pour déployer un exécutable de s'assurer de la présence de l'ensemble des bibliothèques nécessaires, dans les versions nécessaires.

La solution radicale à ce problème est connue sous le nom d'édition de liens statique, et consiste à inclure dans le programme compilé l'ensemble des bibliothèques dont il dépend, à l'exception de quelques bibliothèques système. De la sorte, au prix d'une taille de fichier exécutable plus importante et d'une occupation mémoire non partagée pour les bibliothèques, le programme compilé peut être déployé sur la machine cible par une simple copie.

C'est cette stratégie qui a été retenue pour Go, en ligne avec l'objectif de minimiser le coût total de possession (cf. § 1.1.1.1).

9. Microsoft et Windows sont des marques déposées de Microsoft Corporation.
10. https://osinet.fr/go/intro/wiki-dll-hell

© Dunod – Toute reproduction non autorisée est un délit.

1.1.2.3 Système de types

Afin de définir la mise en œuvre des opérations, les données manipulées dans un programme sont toujours identifiées par le langage comme appartenant à un type particulier, et les langages diffèrent quant à leur exposition de ces types au programmeur. En particulier, deux approches s'opposent :

- ✓ **Faciliter le développement rapide en réduisant les déclarations** : dans cette approche, prédominante dans les langages de scripts et ceux conçus au départ pour les petits développements comme PHP ou JavaScript, le type des variables ne fait pas l'objet d'une déclaration explicite, et le langage stocke l'information de type avec la donnée elle-même, assurant la conversion automatique de la donnée d'un type à l'autre selon les attentes des opérations exécutées.
  Dans la plupart des cas, les *données* ont un type, mais les *variables* qui les contiennent n'ont pas de type fixe ; on parle alors de type dynamique.
  Ceci réduit le volume de code à écrire et facilite l'écriture des premiers programmes par les développeurs débutants, au prix d'une charge d'exécution plus lourde par le langage, conséquence de ce stockage de type additionnel et des conversions implicites.
- ✓ **Réduire le coût de développement et de maintenance des grands projets** : dans cette approche, le type n'est plus porté par les *données*, mais par les *variables*, ce qui permet de le résoudre à la compilation, évitant ainsi d'avoir à le faire persister en mémoire et à résoudre à l'exécution la stratégie à utiliser pour mettre en œuvre chaque opération.
  C'est l'approche traditionnelle des langages de programmation généralistes depuis Cobol ou Fortran, utilisée dans la famille des langages dérivés de C. On parle alors de type statique.

Certains langages, par exemple la famille BASIC, combinent les deux approches en définissant des types statiques – très limités dans le cas de BASIC – mais introduisent explicitement un type *variant* permettant de combiner l'efficacité des types statiques avec la souplesse des types dynamiques.

C'est une telle approche combinée qui a été retenue pour Go, dans une version modernisée. Elle consiste à utiliser autant que possible le typage statique pour sa performance et son aide à la réduction de maintenance, mais en le tempérant par l'usage d'un typage dynamique dans le cas des types *interface*, permettant de limiter le couplage entre composants par l'utilisation de paramètres interface - donc de type dynamique – plutôt que de types statiques, maximisant le potentiel de substitutions de Liskov [11] dans le cadre d'une programmation SOLID [12].

11. https://osinet.fr/go/intro/wiki-liskov
12. https://osinet.fr/go/intro/wiki-solid

Un type en particulier diffère beaucoup entre langages : celui des chaînes de caractères. Historiquement, les premières approches ont reposé sur une représentation à un octet par caractère (SBCS : *Single Byte Character Set*), avec sélection de jeux de caractères incompatibles comme le code EBCDIC [13] d'IBM™. [14] ou la norme états-unienne ASCII [15] et les déclinaisons nationales d'ISO 646, puis le *Latin 1* de la norme ISO 8859-1 et son évolution *Latin9* de la norme ISO 8859-15, solution de référence jusqu'aux années 90 en Europe de l'ouest.

Avec l'internationalisation croissante des développements informatiques, la limitation au jeu de caractères ASCII est devenue de moins en moins acceptable, et a conduit à la fin des années 80 à des approches divergentes : d'une part des structures de chaînes complexes comme les *CompoundString (XmString)* [16] d'OSF/Motif [17], et d'autre part la généralisation de jeux de caractère multi-octets (MBCS : *Multi-Byte Character Set*) et de leurs représentations, aboutissant à la norme ISO-10646 « Universal Character Set » (UCS) [18] pour ce qui est de la simple liste de caractères et à la norme évolutive Unicode [19] pour l'ensemble des considérations d'internationalisation.

Depuis le milieu des années 2010, une standardisation de fait – au moins dans l'univers du web – s'est opérée sur Unicode, dans sa représentation UTF-8 , un format astucieux combinant la compatibilité ascendante avec ASCII et la capacité d'encoder l'ensemble de l'UCS, inventé en 1992 par Ken Thompson et Rob Pike [20], deux des futurs concepteurs de Go. A mi-2019, d'après W3Techs, UTF-8 est utilisé par 93% des sites web, y compris dans le monde asiatique avec ses jeux de caractères multi-octets.

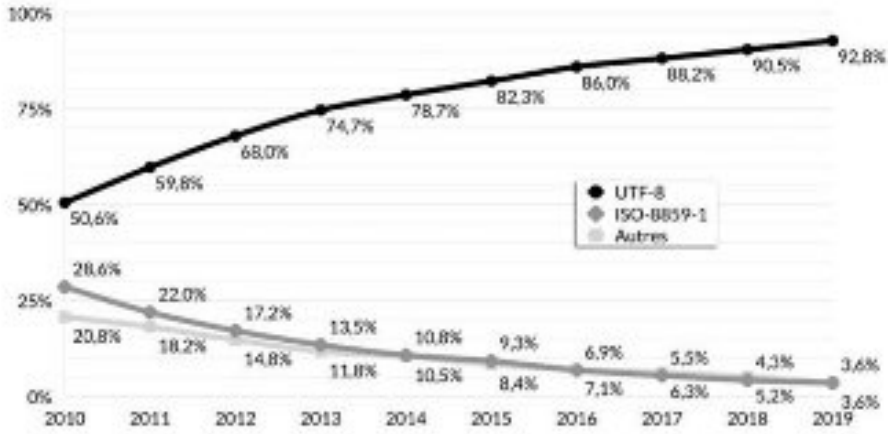
Cette technologie en constante évolution durant les années 90-2000 n'a pas facilité l'adoption par les langages de programmation.

Ainsi, le passage à un traitement complet d'Unicode dans Python 3 a été l'une des grandes difficultés de migration de Python 2 à Python 3, et la tentative pour faire de PHP un langage totalement compatible Unicode pour les versions 6.x a été un tel échec que les versions 6.x n'ont jamais été finalisées [21], le langage n'ayant retrouvé son élan qu'avec les versions 7.x, qui renonçaient à ces changements ambitieux pour préserver une compatibilité avec la base de code préexistante.

13. EBCDIC est un encodage sur un seul octet utilisé principalement sur les systèmes propriétaires d'IBM. L'une de ses particularités réside dans le fait que les caractères consécutifs dans l'alphabet ne sont pas toujours encodés par des valeurs numériques consécutives.
14. IBM est une marque déposée d'International Business Machines.
15. La norme ASCII : American Standard Code for Information Interchange définit un jeu de 96 caractères et 32 contrôles non affichables, de valeur décimale 0 à 127. Elle a fait l'objet de la norme ANSI X3.4-1963, régulièrement révisée jusqu'en 2012, renommée ANSI INCITS 4-2012.
16. https://osinet.fr/go/intro/motif-xmstring
17. https://osinet.fr/go/intro/wiki-motif
18. https://osinet.fr/go/intro/iec-iso-10646
19. https://osinet.fr/go/intro/unicode-12.1
20. https://osinet.fr/go/intro/wiki-utf8
21. https://osinet.fr/go/intro/lwn-php6

© Dunod – Toute reproduction non autorisée est un délit.

Figure 1.2 – Encodages observés sur le web, 2010-2019
©2019 Données W3Techs - https://osinet.fr/go/intro/w3techs-terms



Conçu à partir du 21/09/2007 par les deux inventeurs d'UTF-8 et Robert Griesemer [22] avec une première version publique en 2090, Go a naturellement incorporé le choix de l'encodage natif en UTF-8 pour son code source, et une compatibilité complète avec Unicode dès ses premières versions.

En pratique, cela a conduit à introduire dans le langage :

- ✓ le type natif *rune*, une donnée de 32 bits signée pouvant, du fait de sa largeur, représenter n'importe quel « point de code » [23] Unicode ;
- ✓ la notion de chaîne comme une séquence de runes plutôt qu'une simple séquence d'octets ;
- ✓ la capacité d'itérer nativement sur les runes d'une chaîne au lieu de devoir la parcourir octet par octet, ou devoir recourir à une bibliothèque additionnelle comme *mbstring* [24] en PHP.

1.1.2.4 Programmation concurrente native

La plupart des langages historiques permettent une programmation concurrente basée sur les processus, déléguant les tâches de synchronisation et communication à des bibliothèques externes s'appuyant sur les mécanismes fournis par le système d'exploitation, comme les *pipes* ou *IPC System V / POSIX.1* [25], ou les sockets TCP/IP. C'est un procédé d'une grande portabilité, mais très contraignant et relativement lent.

22. Source : https://osinet.fr/go/intro/yt-griesemer2015-evolution
23. Dans Unicode, le point de code est la combinaison d'un numéro et d'une description abstraite de caractère. Voir https://osinet.fr/go/intro/wiki-point-de-code
24. https://osinet.fr/go/intro/php-mbstring
25. UNIX System V a introduit trois types de communications inter-processus : de la mémoire partagée, des sémaphores, et des files de message. Ces mécanismes ont été unifiés et légèrement modifiés dans la norme POSIX.1 et la *Single Unix Specification*, adoptées pour la plupart des systèmes UNIX et dérivés, dont Linux.

Depuis, le support de fils d'exécution (*threads*) au-dessus de la bilbiothèque *pthread* standardisée par POSIX.3, ou des fibres, une variante destinée au multitâches coopératif plutôt que préemptif, se généralise progressivement, de même que le fonctionnement concurrent basé sur une boucle d'observation au-dessus d'une bibliothèque événementielle popularisé par Node.JS, et repris par exemple par ReactPHP [26], PHP AMP (ex-Icicle) [27], ou, côté Python, Twisted [28] et Tornado [29].

Le point commun de ces approches est le recours à une communication par partage de mémoire, introduisant un état mutable partagé entre les fils, les observateurs d'une boucle événementielle, voire les processus communiquant par une zone mémoire partagée sur IPC.

Au fil des ans, il est apparu que cet état mutable partagé était aussi la principale source de problème de ces approches, la synchronisation et les situations de compétition associées étant légion, et introduisant un non-déterminisme difficile à déboguer dans le code concurrent.

Parallèlement, une autre approche avait été imaginée et publiée par dès 1978 par C.A.R. Hoare, dans un article intitulé *Communicating Sequential Processes* [30] (CSP), décrivant une approche de la programmation concurrente basée sur la communication, qui allait inspirer divers langages parmi lesquels Occam (1983), Erlang (1987), Limbo (1995, Pike *et al.*), Clojure (2007), et finalement Go en 2009, qui est devenu le premier langage massivement populaire à adopter cette approche, en cessant de reposer sur l'état mutable partagé, comme décrit dans un article fondateur d'Andrew Gerrand : *Share memory by communicating* [31].

Dans cette approche, la communication entre fils par le biais d'un état mutable partagé est remplacée par des tâches concurrentes plus légères, dites *goroutines*, communiquant entre elles par des files de messages intégrées au langage, dites *canaux (channel)*. L'utilisation de ces mécanismes est détaillée au chapitre 10.

1.1.2.5 Simplifier la programmation orientée objets

Apparue dans Smalltalk (1971) après quelques rudiments dans Simula 67, la programmation orientée objets (POO) a progressivement émergé comme le paradigme dominant dans le développement logiciel à grande échelle à partir des années 1980 avec son apparition dans des langages dérivés de C : C++ (1983) et Objective-C (1984), avant de connaître son apogée avec Java (1995) et C# (2002), sur un modèle combinant les principes de base de la POO – typage, attributs, méthodes - avec le choix d'une implémentation reposant sur les notions de classe et d'héritage.

26. https://osinet.fr/go/intro/react-php
27. https://osinet.fr/go/intro/amphp
28. https://osinet.fr/go/intro/twisted
29. https://osinet.fr/go/intro/tornado
30. https://osinet.fr/go/intro/hoare1978-csp.pdf
31. https://osinet.fr/go/intro/golang-share

© Dunod – Toute reproduction non autorisée est un délit.

Cette situation dominante a conduit les éditeurs et les entreprises à construire de très grandes quantités de logiciels complexes sur ce modèle, et à populariser un modèle de hiérarchies de classes profondes, comme celles de J2SE [32].

Les difficulté de maintenance liés au couplage intrinsèque que représente par l'héritage ont été rapidement identifiés, comme l'impossibilité de l'héritage losange [33] (*diamond problem*), et la nécessité de réimplementation de méthodes inutiles dans des classes dérivées.

Parallèlement, l'explosion de la popularité de JavaScript (1995) avec son modèle d'héritage par prototype [34] a encouragé la recherche de solution alternatives à la POO basée sur les classes.

La combinaison des problèmes apparus dans le modèle de POO basé sur l'héritage de classes et le visible succès du modèle basé sur les prototypes a encouragé les praticiens à remettre en question le modèle classes/héritage sans renoncer pour autant aux acquis de la POO. L'un des moyens identifiés par les théoriciens des langages s'est révélé être l'utilisation du mécanisme de composition – relation *HAS_A* – de préférence à celui d'héritage [35] – relation *IS_A* – approche particulièrement appropriée car possible à mettre en œuvre même dans les langages disposant des systèmes de classes, au seul prix d'une verbosité syntaxique à chaque référence composée.

Les concepteurs de Go ont tiré parti de ces enseignements, en choisissant pour Go un modèle de POO éliminant les notions de classe et d'héritage, et en les remplaçant par un soutien au niveau langage pour l'approche de composition, afin d'éliminer la complexité syntaxique associée, avec deux simplifications de syntaxe rendant la composition aussi simple que l'héritage :

- ✓ l'implémentation implicite des types interfaces, y compris composées dans d'autres types interfaces ;
- ✓ la délégation automatique (*autoforward*) des accès aux champs et méthodes appartenant à des types composés.

Ce modèle est détaillé au chapitre 5.

1.2 NOUVELLES ARCHITECTURES

Pour la programmation à grande échelle, pour laquelle Go est conçu (cf. § 1.1), les tailles de programme progressent sans cesse, et avec elles le niveau d'abstraction des langages et les besoins de puissance pour préserver une fluidité d'interface utilisateur acceptable.

32. Java 2 Standard Edition, bibliothèque standard du langage Java, est composée de 5488 types, hiérarchisés sur neuf niveaux d'héritage, visible sur https://osinet.fr/go/intro/java8se-tree pour les utilisateurs connectés sur Oracle.com.
33. https://osinet.fr/go/intro/wiki-heritage-losange
34. L'héritage par prototype est apparu initialement dans Self (1987).
35. https://osinet.fr/go/intro/wiki-composition

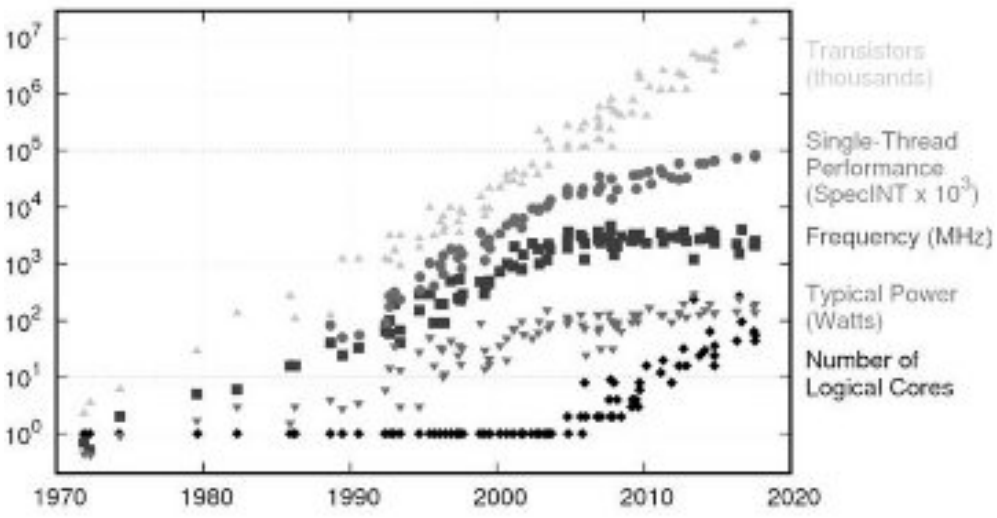
De 1973 à 2004 environ, l'accroissement exponentiel des fréquences d'horloge des processeurs courants du marché a couvert cette augmentation des besoins, comme l'illustrent les séries *Single-Thread Performance* et *Frequency* du graphique 1.3, représentant respectivement les fréquences d'horloge des processeurs commerciaux et leur puissance de calcul en nombres entiers.

Toutefois, dès 2002, cette tendance s'est infléchie, confrontée à une double contrainte :

- ✓ les limitations d'enveloppe thermique (*Thermal Design Power*, ou TDP) [36] des processeurs à refroidissement par air ;
- ✓ la limitation de longueur des circuits avec l'accroissement des fréquences, et la réduction inverse des longueurs d'onde des signaux correspondants.

C'est ce qu'illustrent les séries *Frequency* et *Typical power* du graphique 1.3 à partir de 2004, époque depuis laquelle les fréquences d'horloge commerciales ne progressent pratiquement plus, tandis que les enveloppes thermiques ne progressent plus que lentement, au prix de systèmes de refroidissement toujours plus imposants.

Figure 1.3 – 46 ans d'évolution des processeurs
©2017 Horowitz, Labonte, Shacham, Olukotun, Hammond, Batten, Rupp
Sous licence Creative Commons BY 4.0
https://osinet.fr/go/intro/processor-trends



Parallèlement, néanmoins, le nombre de transistors des microprocesseurs sur les puces de silicium continue sa progression exponentielle conformément à la deuxième loi de Moore [37] contribuant à la croissance des puissances thermiques desdits pro-

36. L'enveloppe thermique représente le flux thermique sortant maximum nécessaire pour le maintien en condition opérationnelle d'un processeur. Elle est mesurée en Watt, à l'image de l'énergie consommée.
37. https://osinet.fr/go/intro/moore1975-law.pdf

© Dunod – Toute reproduction non autorisée est un délit.

cesseurs alors même que leur rapport puissance de calcul sur puissance thermique s'améliore. L'utilisation de ces transistors est progressivement consacrée à deux axes d'amélioration :

✓ la présence de caches L1 et L2 toujours plus vastes et plus rapides ;

✓ l'augmentation du nombre de cœurs par processeur.

Les gains provenant de l'accroissement de la taille des caches sont pour l'essentiel automatiques et accessibles à tous les programmes, mais avec des retours rapidement décroissants, en particulier une fois que la taille de nombre de problèmes a pu tenir entièrement dans un cache de niveau 2, les accroissements additionnels de taille n'apportant plus aucun gain. C'est l'une des causes de la loi de Wirth [38].

L'utilisation de ces cœurs additionnels se retrouve donc être la seule piste pour tirer parti efficacement de ces nouvelles formes de gains de puissance, mais les langages traditionnels de grande utilisation n'intégraient pas nativement de soutien de la programmation concurrente et de son exécution parallèle, et reposent plutôt sur les mécanismes externes décrits au § 1.1.2.4.

Un article de Herb Sutter dans le DrDobb's Journal en 2005 détaille cette situation et a permis une prise de conscience générale du problème par les développeurs : *The free lunch is over* [39].

La conception de Go a intégré ces contraintes matérielles pour répondre à son objectif de soutien de la programmation à grande échelle, en créant le premier langage généraliste à être pensé nativement pour la programmation concurrente et l'exécution parallèle. Ces notions sont détaillées au chapitre 10.

1.3 EN PRATIQUE

1.3.1 Simplicité

En pratique, à quoi ressemble le fruit de cette démarche de création ? L'exemple de la figure 1.4 présente l'exemple canonique « *hello world* » à la mode Go.

Il s'agit de l'exemple de syntaxe de référence fourni par défaut sur le *playground* Go, situé sur https://play.golang.org/, un environnement d'exécution Go en ligne qui permet d'exécuter la plupart des programmes n'utilisant que des éléments de la bibliothèque d'exécution standard dans un navigateur sans avoir besoin d'installer d'outils de développement sur son poste de travail.

Cet environnement est suffisamment puissant pour que tous les exemples de cet ouvrage jusqu'au chapitre 9 y soient utilisables, et il permet d'échanger rapidement des fragments entre développeurs car les URL des sources qui y sont placés sont permanents.

38. https://osinet.fr/go/intro/wiki-loi-wirth
39. https://osinet.fr/go/intro/sutter2005-concurrency

Figure 1.4 – Le *playground* Go sur https://play.golang.org/

La persistance de ces contenus implique qu'il est important lors d'un développement professionnel de prendre soin de ne pas y exécuter de code pouvant être confidentiel.

1.3.2 Rapidité

Les besoins définis pour Go incluaient à la fois une recherche de rapidité de compilation et une recherche de performance d'exécution.

Côté rapidité de compilation, le démarrage du compilateur et la compilation du programme « *hello world* » ci-dessus prennent environ :

- ✓ 400 ms sur le playground ;
- ✓ 290 ms à la première compilation sur un Apple® MacBook Pro® 2016 ;
- ✓ 110 ms à partir de la deuxième compilation sur le même portable.

À titre de comparaison, sur le même portable, on relève :

- ✓ 320 ms de temps moyen pour la compilation du programme C++ équivalent avec le compilateur C++ inclus dans macOS® [40] Mojave ;
- ✓ 430 ms de temps moyen pour la compilation du programme Java équivalent avec le JDK 8, alors même que le résultat est un bytecode JVM (cf. § 1.1.2.1) et non du code exécutable, à la différence de Go et C++.

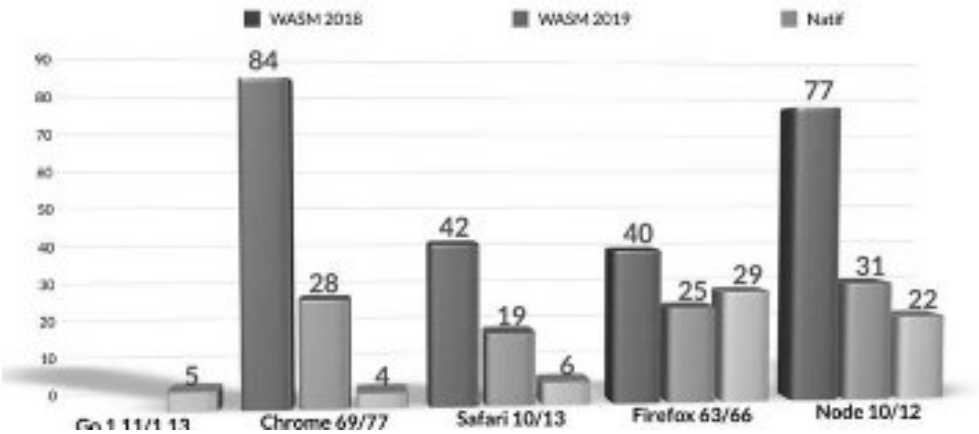
Pour ce qui est de la rapidité d'exécution, la figure 1.5 représente le temps d'exécution du calcul de la valeur de la suite de Fibonacci par le même algorithme naïf dans diverses technologies, sur le même portable, en n'utilisant qu'un seul cœur de processeur pour mettre les langages sur un pied d'égalité :

40. Apple, macOS, et MacBook Pro sont des marques déposées de Apple Inc.

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ code natif Go,
- ✓ WebAssembly compilé à partir d'un source Go dans les principaux navigateurs en 2018, puis en 2019,
- ✓ JavaScript dans Node.JS 10 et 12,
- ✓ JavaScript natif dans les principaux navigateurs.

Figure 1.5 – Temps de calcul naïf de Fibonacci(80), par plateforme



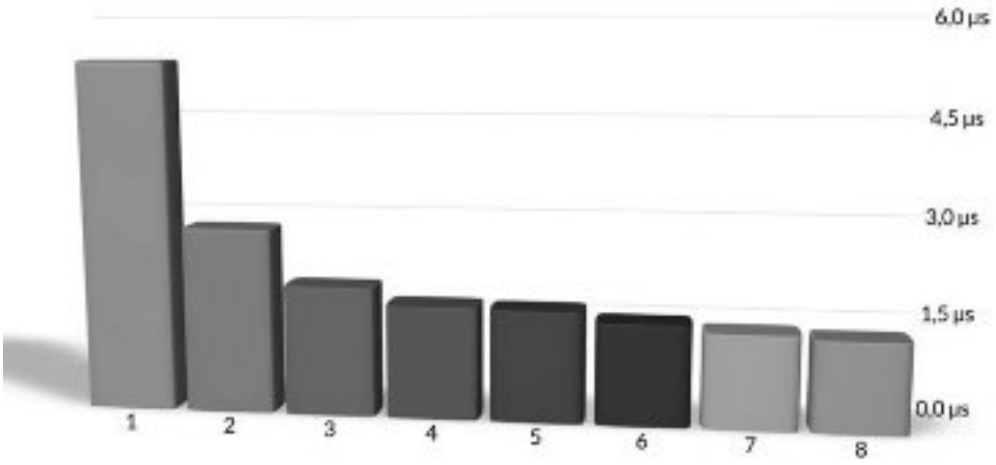
Il en ressort que cette performance sur un seul cœur est comparable à celle des compilateurs JIT modernes, mis en œuvre par Node.JS et les navigateurs courants, autour de 5 ms pour ce calcul purement séquentiel, réalisable dans tous les langages de programmation depuis les années 1950.

Mais Go a été conçu pour tirer parti des architectures actuelles, comme détaillé au § 1.2. La figure 1.6 représente le temps d'exécution du même calcul naïf de Fibonacci sur la même machine, mais en autorisant Go à utiliser un nombre progressivement plus élevé de cœurs sans optimiser l'algorithme lui-même.

Cet exemple, qui montre un temps d'exécution quatre fois moindre avec quatre cœeurs, et peu de gains au-delà, illustre deux points essentiels de l'optimisation sur les machines actuelles multicœurs :

- ✓ Go permet simplement de tirer parti des machines d'aujourd'hui pour atteindre des niveaux de performance inaccessibles aux langages non conçus pour elles.
- ✓ Les gains rendus possibles par l'exécution parallèle ont une limite intrinsèque pour un problème donné : multiplier les cœurs n'accélère plus l'exécution une fois un degré optimum atteint, tout programme concurrent étant limité en accélération à la fois par le coût de la synchronisation des unités d'exécution parallèle et par le temps passé dans la partie séquentielle.

Figure 1.6 – Temps de calcul naïf de Fibonacci(80) en fonction du nombre de cœurs autorisés



La limitation due à la partie séquentielle a été identifiée et popularisée dès 1969 par la loi d'Amdahl [41], qui donne la limite d'accélération d'un programme en fonction de sa proportion exécutée en parallèle et du nombre de processeurs utilisés pour l'exécution parallèle, généralement formulée comme suit :

$$S(n) = \frac{1}{r_s + \frac{r_p}{n}}$$

Dans laquelle :

$S(n)$	accélération obtenue avec n processeurs ;
r_s	proportion du temps passé dans la partie séquentielle ;
r_p	proportion du temps passé dans la partie parallélisée ;
n	nombre de processeurs sur lequel l'exécution est parallélisée.

41. Cette loi a été formulée lors de la conférence de l'AFIPS, et est expliquée dans le quatrième paragraphe de l'article de Gene Amdahl, disponible sur le site de l'université de Berkeley : https://osinet.fr/go/intro/amdahl1967-law.pdf
Une explication plus détaillée en est présentée sur https://osinet.fr/go/intro/wiki-loi-amdahl

2 Syntaxe

2.1 ALPHABET

2.1.1 Jeu de caractères

Au plus bas niveau de la spécification d'un langage se trouve l'encodage dans lequel les sources sont rédigés. Souvent implicitement ou explicitement ASCII (cf. § 1.1.2.3) [1], le jeu de caractères peut ne pas être spécifié comme en PHP. Toutefois, avec la prise en compte croissante des besoins d'internationalisation, certains langages requièrent l'utilisation de sources Unicode (cf. § 1.1.2.3), comme ECMAScript [2] 2018 qui spécifie explicitement que le code source est [3] :

> « *SourceCharacter* :: any Unicode code point »

ce qui est une forme très large. La version ECMAScript 5.1, longtemps utilisée comme référence, précisait de façon plus restrictive que le code source devait être exprimé dans Unicode version 3 ou plus, et avec la forme de normalisation canonique C définie dans Unicode TR15 [4], alors que la version ECMAScript 2018 ne précise plus de normalisation ni de version d'Unicode.

Le choix fait pour la syntaxe de Go est du même ordre : la spécification du langage précise que le code source Go est du texte Unicode, sans plus de précision, avec deux points de code traités spécifiquement, le NUL et la BOM [5] UTF-8 :

1. Fortran 77 spécifie un jeu de caractères composé d'un sous-ensemble de 83 caractères extraits d'ASCII-96.
2. ECMAScript est le nom de la version normalisée de JavaScript.
3. ECMA-262 9.0 (§ 10.1) : https://osinet.fr/go/syntaxe/ecma-262-9
4. https://osinet.fr/go/syntaxe/ecma-262-5
5. La *Byte Order Mark* ou BOM, correspond au point de code U+FEFF, et est parfois utilisée pour identifier l'ordre des octets dans les documents Unicode. Elle est principalement employée par des outils d'édition de texte utilisés sur Microsoft Windows, et peut être source de divers problèmes, cf. https://osinet.fr/go/syntaxe/wiki-bom

- ✓ tout compilateur *peut* interdire le point de code `U+0000` (ASCII NUL) dans les sources ;
- ✓ tout compilateur *peut* ignorer la BOM lorsqu'elle est présente comme premiers octets d'un source, et *peut* l'interdire à tout autre emplacement.

Le compilateur Go standard applique ces deux comportements.

Le langage spécifie également la distinction entre tous les points de code, en particulier entre une lettre majuscule et la minuscule associée (cf. § 2.3.2), ce qui aura un impact aussi important qu'inhabituel sur l'organisation du code.

Enfin, bien que la spécification textuelle du langage ne mentionne qu'Unicode en général pour le code source, et non une version particulière, les classes de caractères définies par les grammaires du langage (cf. § 2.1.3) spécifient explicitement la référence à Unicode 8.0.

2.1.2 Encodage

Un point plus rarement envisagé dans les spécifications de langages de programmation est l'encodage du texte, c'est-à-dire la représentation spécifique comme séquence d'octets de la suite des caractères (points de code Unicode) représentant le code source.

Ainsi, la même norme ECMAScript 2018 évoquée au § 2.1.1 comme autorisant tout le répertoire Unicode précise-t-elle que :

> « Les encodages effectivement utilisés pour stocker et échanger le code source ECMAScript ne sont pas du ressort de la spécification du langage. Quel que soit l'encodage, une implémentation conforme d'ECMAScript traite le texte source comme une séquence équivalente de valeurs *SourceCharacter*, chacun de ceux-ci étant un point de code Unicode. [6] »

Cette attitude permissive peut aboutir à des situations, liées notamment aux multiples formes de canonicalisation disponibles avec Unicode, dans lesquelles des fichiers différents devront être traités à l'identique, avec le risque d'introduire des difficultés de suivi de versions dans les environnements de projets à grande échelle où chaque développeur peut utiliser des outils différents sur un même projet.

Pour éviter ce problème, la spécification Go précise, outre le répertoire Unicode pour le jeu de caractères, le traitement explicite de l'encodage :

- ✓ utilisation exclusive du format UTF-8 (cf. § 1.1.2.3) ;
- ✓ absence de canonicalisation, et traitement explicite de tout caractère accentué résultant d'une canonicalisation comme différent du même caractère exprimé comme le point de code accentué équivalent.

6. Traduction de l'auteur.

2.1.3 Classes de caractères

À l'intérieur du jeu de caractères Unicode ainsi spécifié par Go, le langage distingue plusieurs classes de caractères, qui auront des usages différenciés dans la programmation.

- ✓ **Fin de ligne** : le point de code `U+000A`. À la différence de nombreux autres langage, Go ne reconnaît comme séparateur de ligne que ce point de code (ASCII LF) et ne traite pas le retour chariot (`U+000D`, ASCII CR) comme un séparateur de lignes.
- ✓ **Caractère Unicode** : tout point de code Unicode sauf la fin de ligne.
- ✓ **Lettre** : Go reconnaît comme « lettre » :
  - le caractère « _ » (blanc souligné / underscore) ;
  - tout point de code Unicode de l'une des classes `Lu` (uppercase), `Ll` (lowercase), `Lt` (titlecase), `Lm` (modifier) ou `Lo` (other), telles que définies dans la table 4-9 de la norme Unicode 8.0 [7].
- ✓ **Chiffre** : tout point de code de la class `Nd` (chiffre décimal) dans la même table 4-9 d'Unicode 8.0, soit en pratique :
  - les chiffres 0 à 9 du répertoire ASCII ;
  - mais aussi tous les chiffres spécifiques à une écriture particulière, comme défini dans Unicode 8.0 au § 4.6 du chapitre 4 et dans la table 22-3 du chapitre 22 [8].
- ✓ **Chiffre décimal** : les points de code des chiffres « `0` » à « `9` » du répertoire ASCII.
- ✓ **Chiffre octal** : les points de code des chiffres « `0` » à « `7` » du répertoire ASCII.
- ✓ **Chiffre hexadécimal** : les points de code des chiffres de 0 à 9, et des lettres de « `a` » à « `f` » et de « `A` » à « `F` » du répertoire ASCII.
- ✓ **Chiffre binaire** : les points de code des chiffres « `0` » et « `1` » du répertoire ASCII. Cette classe n'est définie que depuis Go 1.13.

Ces classes de caractères et leur utilisation dans la grammaire du langage ne concernent pas les limitations des données manipulées, ni les capacités de la bibliothèque d'internationalisation, qui a atteint Unicode 11 depuis Go 1.12, et Unicode 12 durant l'élaboration de Go 1.14.

2.2 L'HÉRITAGE SYNTAXIQUE DE C

Pour faciliter l'adoption du langage par la réduction du coût d'adoption (cf. § 1.1.1.1), les concepteurs de Go ont fait le choix d'une syntaxe aussi peu exotique que possible, fermement héritée du langage C :

- ✓ langage à base de blocs délimités par des accolades
- ✓ affectation par =, comparaison par `==` et `!=` [9]

7. https://osinet.fr/go/syntaxe/unicode8-ch4.pdf
8. https://osinet.fr/go/syntaxe/unicode8-ch22.pdf
9. À la différence des PHP ou JavaScript, également inspirés par C, Go est fortement typé, de sorte que les opérateurs « `===` » et « `!==` » n'auraient pas de sens.

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ commentaires de forme bloc `/* ... */` ou de forme ligne `// ...`
- ✓ espaces acceptés : `U+0009` (ASCII HT = tabulation), `U+000A` (ASCII LF), `U+000D` (ASCII CR), `U+0020` (espace)
- ✓ séparateurs de ligne `U+000A` (ASCII LF)
- ✓ tableaux et structures similaires définis par des crochets et initialisés par des accolades
- ✓ point d'entrée des programmes `main` et sortie des fonctions par **`return`**
- ✓ instructions complexes similaires : **`if`**, **`for`**, **`switch`** avec **`break`**, **`continue`**, **`goto`**
- ✓ définitions des types de données composites comme **`struct`**
- ✓ mêmes opérateurs arithmétiques et logiques : `+`, `-`, `*`, `/`, `++`, `--`, `&`

L'introduction d'un modèle d'organisation du code plus structuré amène le langage à définir les blocs un peu différemment de C :

- ✓ Du fait de l'organisation du code en paquets (cf. chapitre 9), existence d'un *bloc d'univers* et de *blocs de paquets* individuels, au lieu d'un unique bloc global en C.
- ✓ *Bloc de fichier*, contenant tout le code source Go présent dans un fichier.
- ✓ *Blocs d'instructions complexes* : instructions de type bloc, **`if`**, **`for`**, **`switch`**, et **`select`** (cf. § 7.3.3 et suivants).
- ✓ *Blocs de clauses* dans les instructions **`switch`** ou **`select`** (cf. § 7.3.5 et suivants).
- ✓ Suppression de la possibilité d'avoir des blocs d'une seule ligne ne nécessitant pas d'accolades : l'historique des bogues en C, JavaScript, et PHP a montré que le gain d'espace représenté n'était pas justifié par l'augmentation du risque de bogues résultant de cette fonctionnalité, à l'image de la faille de sécurité SSL/TLS corrigée en 2014 dans iOS™ [10], introduite par l'utilisation de blocs de ce type, comme dans l'extrait au listing 2.1.

Isolé dans le fragment présenté, le bogue est facile à repérer, mais situé au-delà de la ligne 500 d'un fichier de sources, il est demeuré présent pendant plusieurs versions de ce système d'exploitation avant d'être corrigé en 2014. Pour éviter par la syntaxe même du langage toute erreur de cette nature, de tels blocs sans accolades sont explicitement interdits en Go.

10. iOS est une marque déposée d'Apple.

Listing 2.1 – Nécessité des accolades autour des blocs C
https://osinet.fr/go/syntaxe/apple-55471

```
// Les /* ... */ représentent du code omis.

/* ... */
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
    SSLBuffer signedParams,
    uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;

    /* ... */
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    /* ... */

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}

/* ... */
```

2.3 IDENTIFICATEURS

2.3.1 Forme et identificateurs réservés

Les identificateurs Go, qu'ils soient prédéclarés ou créés par l'utilisateur, sont toujours composés d'une lettre suivie d'un ou plusieurs caractères Lettre ou Chiffre, au sens du § 2.1.3.

Afin de minimiser le temps d'apprentissage, Go ne définit qu'un très petit nombre d'identificateurs, 39 en tout [11], ce qui est l'une des listes d'identificateurs prédéclarés les plus limitées parmi les langages de programmation généralistes.

Cette liste est inchangée depuis la publication de Go 1.0 en 2009. À titre de comparaison, la figure 2.1 trace l'évolution du nombre d'identificateurs réservés au fil des versions d'ECMAScript, la version standardisée de JavaScript, avec un nombre d'identificateurs plus élevé que Go, et fluctuant au fil des versions, causant des incompatibilités sans équivalent pour les développeurs travaillant avec Go.

11. Certains textes indiquent des nombres d'identificateurs prédéclarés différents, car `print` et `println` ne font pas strictement partie du langage et leur usage est déprécié, tandis que l'identificateur vierge `_` n'est pas un simple identificateur prédéclaré mais implique une sémantique particulière.

© Dunod – Toute reproduction non autorisée est un délit.

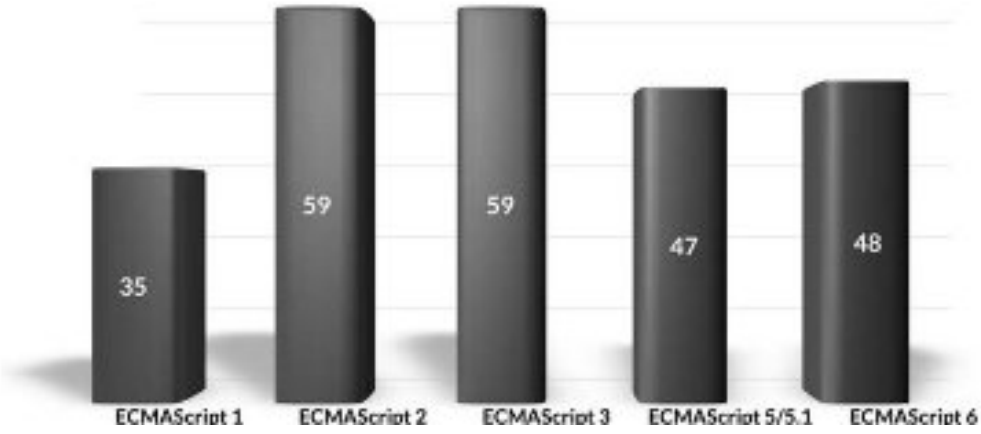
Tableau 2.1 – Identificateurs Go

Types	Booléens	`bool`

	Entiers	`uint int`
		`uint8 int8`
		`uint16 int16`
		`uint32 int32`
		`uint64 int64`

	Entiers spéciaux	`byte rune uintptr`
	Flottants	`float32 float64` `complex64 complex128`
	Chaînes de caractères	`string`
	Erreurs	`error`
Constantes	Vraies	`true false iota`
	Valeur vierge (0V)	`nil`
Fonctions génériques	Données composites	`append cap close` `copy delete len`
	Anomalies	`panic recover`
	Nombres complexes	`complex real imag`
	Allocation mémoire	`make new`
	Affichage	`print println`
Identificateur vierge		`_`

Figure 2.1 – Nombre d'identificateurs réservés ECMAScript



2.3.2 Visibilité

Les identificateurs sont définis dans la portée d'un bloc (cf. § 2.2), et sont visibles dans tous les blocs enfants du bloc où ils sont définis, à moins d'être masqués par une redéfinition.

En effet, un bloc de niveau inférieur (enfant) peut librement redéfinir tout identificateur défini dans un bloc de niveau supérieur (parent). On parle alors de *masquage* (*shadowing*), comme dans le listing 2.2.

Comme l'illustre ce listing, le masquage n'est pas limité à la valeur de la variable désignée par l'identificateur, mais peut aussi concerner son type.

Les identificateurs définis dans la portée d'un paquet, ou qui représentent des champs ou méthodes, peuvent être exportés pour être accessibles depuis d'autres paquets, sous la forme d'identificateurs qualifiés (cf. § 6.3).

Sur ce point, Go diffère de la majorité des langages en n'utilisant pas de clause explicite de visibilité comme `extern` en C, ou en partageant tous les symboles globaux comme JavaScript, mais en faisant reposer la visibilité des identificateurs hors de leur paquet sur le nom de l'identificateur :

- ✓ si le premier caractère d'un identificateur est une Lettre majuscule, définie par la classe *Lu* d'Unicode 12.1 [12], alors l'identificateur peut être référencé depuis l'extérieur du paquet ;
- ✓ sinon, il ne le peut.

Listing 2.2 – Masquage d'identificateur dans un bloc enfant
https://osinet.fr/go/syntaxe/play-id-shadowing

```
1  package main
2
3  func main() {
4          var x int = 5
5
6          println(x)
7          {
8                  var x string = "cinq"
9                  println(x)
10         }
11         println(x)
12 }
13 /* Résultat:
14 5
15 cinq
16 5
17  */
```

12. Table 4.9, chapitre 4 sur https://osinet.fr/go/syntaxe/unicode8-ch4.pdf

© Dunod – Toute reproduction non autorisée est un délit.

Ce comportement peu commun est compliqué par le fait que la classe *Lu* ne se limite pas aux 26 majuscules ASCII, mais comporte en fait 1788 caractères différents [13].

2.3.3 Comparaison

Go considère un identificateur comme *unique* s'il diffère de tous les autres.

Pour comparer les identificateurs, le langage applique deux critères dont l'un au moins doit être respecté :

✓ soit les identificateurs n'ont pas la même orthographe ;

✓ soit ils apparaissent dans deux paquets différents, et ne sont pas exportés.

Le premier critère n'est pas aussi évident qu'il pourrait le sembler si l'on envisage l'existence des problèmes de canonicalisation évoqués au fil des versions d'ECMAScript (cf. § 2.1.1).

Mais puisque la spécification Go exige que les identificateurs soient exclusivement composés de caractères des classes Lettre et Chiffre ou du souligné, comme défini au début de cette section, aucun caractère combinant ne peut appartenir à un identificateur ; de sorte que les identificateurs sont en pratique résistants à la canonicalisation, et qu'aucune ambiguïté ne peut en résulter.

Cette situation pourrait être amenée à changer dans le cadre de Go 2, pour lequel une évolution est envisagée [14] pour adopter la définition Unicode TR-31 des identificateurs [15].

Dans le cas d'identificateurs exportés du même nom dans deux paquets distincts, l'import par défaut (cf. chapitre 9) va entraîner une collision de nom empêchant la compilation, d'où l'importance de cette règle de différenciation des identificateurs.

Le second cas est la conséquence de la réécriture des identificateurs non exportés, qui sont en interne préfixés par le nom du paquet où ils sont définis.

2.4 OPÉRATEURS

2.4.1 Opérateurs et types de données

Du point de vue de la grammaire Go, un opérateur combine des opérandes dans une expression. Ainsi, certains jetons du langage parfois désignés comme opérateurs par simplification n'en sont-ils pas réellement.

13. Statistiques : https://osinet.fr/go/syntaxe/wiki-uppercase, liste : https://osinet.fr/go/syntaxe/uppercase
14. https://osinet.fr/go/syntaxe/gh-go-20706
15. https://osinet.fr/go/syntaxe/unicode-tr31

Le cas le plus notable est celui du symbole = utilisé dans les instructions d'affectation (*assignment*) : il combine bien deux opérandes – la destination et la source – mais ne produit pas de résultat qui serait utilisable dans une expression, à la différence d'autres langages, comme le listing 2.3 le montre pour JavaScript. La ponctuation autre que les opérateurs fait l'objet du § 2.5.

Listing 2.3 – L'affectation comme opérateur en JavaScript

```
const a = "une valeur";
let c = b = a;
console.log(a, b, c);
/* Résultat: une valeur une valeur une valeur */
```

A contrario, le code équivalent en Go, dans le listing 2.4, est rejeté par le compilateur, pour lequel l'affectation n'a pas de valeur, et ne constitue donc pas une expression, confirmant que le symbole = n'est pas un opérateur.

Listing 2.4 – L'affectation comme non-opérateur en Go
https://osinet.fr/go/syntaxe/play-assignment

```
package main

func main() {
        const a = "une valeur"
        var b, c string
        c = b = a
        println(a, b, c)
}
// Résultat: erreur de compilation:
//  syntax error: unexpected = at end of statement
```

Les opérateurs en Go ont toujours une arité 1 ou 2 : ils sont donc unaires ou binaires. À la différence de la plupart des autres langages dérivés de C, Go ne dispose en effet pas de l'opérateur ternaire `?:`, ni d'aucun autre opérateur ternaire comme le `BETWEEN` de SQL. Il s'agit d'un choix assumé des concepteurs du langage, qui indiquent dans la Foire Aux Questions Go [16] :

> « La raison pour laquelle `?:` est absent de Go est que les concepteurs du langage l'ont trop souvent vu être utilisé pour créer des expressions d'une complexité impénétrable. »

Comme pour d'autre aspects du langage, il s'agit de réduire le coût de possession du code source, comme détaillé au § 1.1.1.1.

16. https://osinet.fr/go/syntaxe/golang-faq. Traduction de l'auteur.

© Dunod – Toute reproduction non autorisée est un délit.

En raison du caractère fortement typé de Go, les opérateurs binaires s'appliquent à deux valeurs du même type, et non seulement de types convertibles, comme le montre le listing 2.5.

Listing 2.5 – Nécessité de types identiques pour les opérateurs binaires
https://osinet.fr/go/syntaxe/play-op-types

```
package main

func main() {
        var b int8 = 1
        var s int16 = 2
        println(b + s)
}
// Erreur de compilation:
//   "invalid operation: b + s (mismatched types int and int16)"
```

L'appréciation des types rencontre donc trois situations :

- ✓ Deux variables : elles doivent être du même type dynamique, sans nécessiter de conversion, comme dans le listing 2.5.
- ✓ Une constante non typée et autre chose (variable, expression, etc.) : en ce cas, Go convertit la constante au type de l'autre opérande. Pour les opérateurs de décalage de bits, l'opérande de droite doit pouvoir être représenté par un `uint`.
- ✓ Deux constantes non typées : le résultat est une constante non typée de même nature (booléen, numérique, chaîne). Mêmes restrictions que précédemment concernant les opérateurs de décalage de bits. Si les deux constantes ne sont pas de même nature, la conversion échoue.

2.4.2 Règles de priorité

Go priorise les opérations de façon régulière : les opérateurs unaires en premier, puis les binaires dans cinq groupes de priorité, comme listés dans le tableau 2.2. Les opérateurs binaires associent de gauche à droite comme dans l'écriture naturelle : `1 - 2 + 3` est évalué comme `(1 - 2) + 3`, et vaut donc `2`, et non `-4` comme ce serait le cas s'il associait de droite à gauche. Cette organisation est un choix simplificateur des concepteurs du langage en comparaison avec les 14 niveaux de priorité de C++.

Les opérateurs unaires n'ont pas besoin de groupes de priorité différents entre eux ; le principal cas de choix de priorité présent dans les langages dérivés de C est la résolution de la combinaison de l'opérateur de déréférencement avec un opérateur d'incrémentation ou décrémentation.

Tableau 2.2 – Priorité des opérateurs

Arité	Priorité	Groupe	Opérateurs						
1	sans objet		+	-	!	^	*	&	<-
2	5	multiplicatifs	*	/	%	<<	>>	&	&^
	4	additifs	+	-	|	^			
	3	comparaison	==	!=	<	<=	>	>=	
	2	ET logique	&&						
	1	OU logique	||						

Or, tout comme l'affectation évoquée précédemment, incrémentation et décrémentation ne sont pas des expressions en Go, et ne sont donc pas associées à des opérateurs, de sorte qu'une expression comme `*p++` n'a pas besoin d'une règle de priorité pour être résolue : une association `*(p++)` reviendrait à tenter de déréférencer le résultat de l'incrémentation, qui n'existe pas. La seule interprétation possible est donc `(*p)++`, comme le montre le listing 2.6.

Listing 2.6 – Association gauche-droite des opérateurs unaires
https://osinet.fr/go/syntaxe/play-op-unary

```
1  package main
2
3  func main() {
4          var a float64 = 2.4
5          var pa = &a
6
7          *pa++
8          println(a)
9  }
10 // Résultat: +3.400000e+000
```

Cet ensemble de règles de priorité est assez coutumier, mais il interagit de façon parfois surprenante avec la notation des constantes complexes, comme le montre le listing 2.7.

En effet, à première lecture, le code de la ligne 4 peut sembler correspondre au produit des deux nombres complexes $(1 + 2i) * (3 + 5i) = -7 + 11i$; mais du fait de la priorité de la multiplication sur l'addition, ce code est en fait interprété comme $1 + (2i * 3) + 5i = 1 + 11i$. Deux façons d'obtenir le résultat souhaité sont indiquées aux lignes 5 et 6.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 2.7 – Interférence de priorité entre opérateurs sur les nombres complexes
https://osinet.fr/go/syntaxe/play-op-priority

```
package main

func main() {
        println(1+2i * 3+5i)
        println((1 + 2i) * (3 + 5i))
        println(complex(1, 2) * complex(3, 5))
}

// Résultat:
// (+1.000000e+000+1.100000e+001i)
// (-7.000000e+000+1.100000e+001i)
// (-7.000000e+000+1.100000e+001i)
```

2.4.3 Opérateurs multiplicatifs

Go définit sept opérateurs comme multiplicatifs.

Tableau 2.3 – Opérateurs multiplicatifs

Opérateur	Description	Types des opérandes
*	La multiplication classique	Tout type numérique
/	La division classique	Tout type numérique
%	L'opérateur modulo	Types entiers
<< et >>	Le décalage de bits, vers la gauche ou la droite	Limité aux entiers
&	Le ET bit à bit	Types entiers
&^	Le ET NON bit à bit	Types entiers

Cette liste peut sembler étonnamment étendue. Il s'agit en fait d'une acception large du terme, basée sur la combinaison des produits numériques et de la multiplication booléenne appliquée aux bits.

L'opérateur `&^` est rarement fourni par les langages de la famille C ; il réalise le `ET` bit à bit de l'opérande gauche avec le `NON` bit à bit de l'opérande droit, comme illustré par le listing 2.8.

À la différence de certains langages comme Python, l'opérateur de multiplication n'est pas applicable en Go entre une chaîne et un nombre pour produire une chaîne résultant de la répétition de l'opérande chaîne un nombre de fois égal à l'opérande numérique.

Listing 2.8 – L'opérateur AND NOT
https://osinet.fr/go/syntaxe/play-op-andnot

```
package main

func main() {
        var l byte = 0x3
        var r byte = 0x5
        println(l &^ r)
}

// Résultat: 2. Explication:
// 0x3    = 0b00000011
// 0x5    = 0b00000101
// 3 &^ 5 = 0b00000011 & NOT 0b00000101
// 3 &^ 5 = 0b00000011 &     0b11111010
// 3 &^ 5 = 0b00000010 = 2
```

Limitation : avant Go 1.13, le second opérande des opérateurs de décalage << et >> devait être un entier non signé ; cette restriction a été levée à partir depuis. En revanche, la valeur de ce second opérande doit toujours être positive : une valeur de décalage négative provoquera une erreur de compilation si la valeur est une constante, ou une panique d'exécution si c'est une valeur calculée à l'exécution.

2.4.4 Opérateurs additifs

Go définit cinq opérateurs comme additifs, à l'instar de la plupart des langages usuels.

Tableau 2.4 – Opérateurs additifs

Opérateur	Description	Types des opérandes
+	La concaténation	Types chaînes
+	L'addition classique	Tout type numérique
–	La soustraction classique	Tout type numérique
|	Le OU bit à bit	Types entiers
^	Le XOR bit à bit	Types entiers

2.4.5 Dépassement de capacité

Les opérateurs numériques additifs et multiplicatifs, entiers comme flottants [17], peuvent entraîner des dépassements de capacité de plusieurs façons :

✓ la soustraction de deux nombres non signés peut donner un résultat négatif ;

17. Sauf indication particulière, les nombres complexes – composés de deux flottants – suivent les règles des flottants.

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ toutes les opérations sur les entiers peuvent donner un résultat impossible à représenter dans le type des opérandes ;
- ✓ la division peut produire des résultats infinis ou indéterminés.

Pour tous ces cas l'approche choisie par Go est inhabituelle : le dépassement de capacité (*overflow*) ainsi causé ne provoque **pas** d'erreur d'exécution ; au contraire, le résultat est silencieusement tronqué à la taille des opérandes, donnant un résultat reproductible basé sur le format binaire des nombres.

Ce choix très particulier déplace la responsabilité du contrôle d'étendue des opérandes numériques du langage vers le développeur, en requérant de sa part la prise en compte des grandeurs manipulées et de leurs limitations numériques. En contrepartie de cet accroissement de charge de travail lors du développement initial, le gain obtenu est la suppression des corrections « magiques » pratiquées par d'autres langages avec des résultats imprévisibles.

L'une des conséquences les plus inattendues, signalée dans la spécification du langage, est le fait que la relation $x < x + 1$ n'est pas nécessairement vraie dans un programme Go, comme le montre le listing 2.9.

Listing 2.9 – $a < a + 1$ n'est pas toujours vrai en Go
https://osinet.fr/go/syntaxe/play-plus-less

```
1 package main
2
3 func main() {
4         var a uint32 = 1<<32 - 1
5
6         println(a, a+1, a < a+1)
7 }
```

Le cas de la division par zéro est encore plus particulier :

- ✓ Si le zéro est une constante, il est intercepté à la compilation, sinon…
- ✓ Dans le cas d'opérandes entiers, le programme va déclencher une erreur d'exécution (*panic*).
- ✓ Dans le cas d'opérandes flottants, le langage spécifie l'application de la norme IEEE-754 qui définit des retours `+Inf`, `-Inf`, et `NaN`, mais laisse toute latitude aux implémentations du langage pour déclencher ou non une erreur d'exécution en ce cas. Comme le montre le listing 2.10, le compilateur standard employé sur le Playground Go ne déclenche pas cette erreur.

2.4.6 Opérateurs de comparaison

Go définit seulement les six opérateurs de comparaison traditionnels. Ils sont tous dyadiques et renvoient une valeur booléenne.

Listing 2.10 – Division flottante par 0
https://osinet.fr/go/syntaxe/play-zero-divide

```
1 package main
2
3 func main() {
4         var a, b = 1., -1.
5         var d = 0.
6
7         println(a / d, b / d, d / d)
8 }
```

Tableau 2.5 – Opérateurs de comparaison

Opérateurs	Description	Détail
== et !=	L'égalité	Tous types comparables
<= et >=	L'ordre	Tous types ordonnés
< et >	L'ordre strict	Tous types ordonnés

Les opérateurs sont génériques : leur disponibilité entre deux opérandes ne dépend pas directement du type des opérandes mais de ses propriétés :

✓ l'un des deux opérateurs doit être affectable au type de l'autre ;

✓ tout type ordonné est comparable ;

✓ pour certains types, seule la comparaison à `nil` est autorisée.

Tableau 2.6 – Comparaisons possibles par type

Types	Égalité à `nil`	Égalité générique	Ordre
Booléens		Oui	
Entiers		Oui	Oui
Flottants		Oui	Oui
Complexes		Oui	
Chaînes		Limitations	Limitations
Pointeurs	Oui	Limitations	
Tranches	Oui		
Cartes	Oui		
Tableaux		Oui	
Structures		Limitations	
Interfaces	Oui	Oui	
Interface et type concret		Limitations	

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ **Chaînes** : la comparaison d'égalité ne prend pas en compte les équivalences de canonicalisation Unicode, mais est une simple comparaison octet par octet.
- ✓ **Chaînes** : la comparaison d'ordre ne prend pas en compte les règles associées à une langue ou une localisation, mais est une simple comparaison octet par octet, dans laquelle une chaîne est supérieure à toutes ses sous-chaînes commençant à la première position, par exemple `"alpha" < "alphabet"`.
- ✓ **Pointeurs** : la comparaison d'égalité entre deux pointeurs vers des valeurs de taille nulle [18] est syntaxiquement correcte, mais le résultat peut indifféremment être vrai ou faux ; la comparaison ne doit donc pas être utilisée dans ces situations.
- ✓ **Structures** : la comparaison entre des valeurs de deux types structures n'est possible que si les deux types contiennent les mêmes champs. Si les deux types ne sont pas les deux mêmes, ce n'est possible que si au moins l'un des deux est un type anonyme, car tous les types définis sont différents, comme dans le listing 2.11.
- ✓ **Interface et type concret** : la comparaison n'est possible que si le type concret implémente l'interface.

Listing 2.11 – Comparaison de structures
https://osinet.fr/go/syntaxe/play-compare-struct

```
1  package main
2
3  import "fmt"
4
5  type T1 struct{}
6  type T2 T1
7
8  func main() {
9          var t1 T1
10         var t2 T2
11         var t3 struct{}
12
13         // Comparaison invalide: T1 et T2 sont deux types définis.
14         // fmt.Println(t1, t2, t3, t1 == t2)
15
16         // Comparaison valide: le type de t3 n'est pas défini.
17         fmt.Println(t1, t2, t3, t1 == t3)
18 }
```

2.4.7 Opérateurs logiques

Les opérateurs logiques prennent un ou deux arguments booléens, et renvoient un résultat booléen. Comme la plupart des langages, Go en définit trois (cf. tableau 2.7).

18. Par exemple : `struct{}`

Tableau 2.7 – Opérateurs logiques

Arité	Opérateur	Signification
1	!	Négation
2	&&	ET logique
	||	OU logique

La spécification de Go garantit le court-circuit d'évaluation sans employer le terme, en précisant que pour les deux opérateurs dyadiques, l'opérande droit est évalué conditionnellement.

2.4.8 Opérateurs d'adresse

Comme Go inclut un mécanisme de pointeurs, il inclut les deux opérateurs monadiques associés à ce mécanisme :

- ✓ **Adresse** : l'opérateur d'adresse utilise le même symbole `&` que l'opérateur du ET bit à bit (cf. § 2.4.3), sans ambiguïté puisque ce dernier est dyadique.
- ✓ **Indirection / Déréférencement** : l'opérateur d'indirection utilise le même symbole `*` que l'opérateur de multiplication (cf. § 2.4.3), sans ambiguïté puisque ce dernier est dyadique.

L'opérateur d'adresse ne s'applique qu'à certaines valeurs :

- ✓ **Valeur adressable** : plusieurs sortes de données sont adressables :
  - variables ;
  - résultats d'une indirection (déréférencement) de pointeur, par exemple `&(*foo)` ;
  - résultat d'un tranchage, par exemple `&someSlice[1:]`.
- ✓ **Décalage dans une valeur adressable** : à l'intérieur d'un objet adressable en propre :
  - sélecteur de champ dans une structure adressable, par exemple `&objet.Champ` ;
  - résultat de l'indexation dans un tableau adressable, par exemple `&table[3]`.
- ✓ **Littéral composite** : comme dans l'exemple du listing 2.12. Comme ces littéraux ne sont pas des variables, le langage renvoie l'adresse d'une variable à usage unique, initialisée à la valeur du littéral.

En particulier, le fait qu'une valeur ait une forme adressable n'est pas suffisante pour qu'elle soit réellement adressable : ainsi, l'indexation dans une variable de type tableau est adressable, mais la même indexation dans une valeur de type tableau n'est pas adressable si le tableau est un littéral tableau, ou le résultat d'un appel de fonction.

Puisque l'opérande doit exister en mémoire pour avoir une adresse que l'opérateur pourra renvoyer, elle doit pouvoir être évaluée. À défaut, l'exécution déclenchera une `panic`.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 2.12 – Adresse de littéraux composites
https://osinet.fr/go/syntaxe/play-literal-address

```
package main

import "fmt"

func main() {
        // Correct: adresse d'un littéral composite.
        px := &[...]int{1, 1, 2, 3, 5}

        // Incorrect: adresse de l'indexation d'un littéral.
        // py := &[...]int{1, 1, 2, 3, 5}[2]

        // Correct: adresse d'une variable construite par
        // l'indexation d'un littéral.
        z := [...]int{1, 1, 2, 3, 5}[2]
        pz := &z

        fmt.Printf("%p %p\n%#v %#v\n", px, pz, px, pz)
}
```

L'opérateur d'indirection / déréférencement est le complémentaire de l'opérateur d'adresse : à partir d'un opérande pointeur, il renvoie la donnée présente à l'adresse représentée par le pointeur. Si le pointeur est `nil`, l'exécution déclenchera une `panic`.

Adresses et constantes

Au contraire des variables, les constantes ne sont pas adressables, donc il n'est pas possible de leur appliquer l'opérateur d'adresse.

2.4.9 Opérateur réception

L'opérateur de réception `<-` est l'une des spécificités de Go. C'est un opérateur monadique dont l'opérande est un canal de type `T` permettant les réceptions, de la forme `chan T` ou `<-chan T` :

- ✓ Si une donnée est disponible dans le canal, l'opérateur renvoie cette donnée – du type `T` – et la retire du canal.
- ✓ Si aucune donnée n'est disponible sur un canal ouvert, le flux d'exécution (*goroutine*) dans lequel se trouve l'opérateur est suspendu. En particulier, un canal `nil` ne pouvant recevoir de données depuis un émetteur, il est toujours vide et, ne pouvant être fermé, une réception sera toujours bloquante.

✓ Sur un canal fermé, en revanche, la réception des données présentes a lieu comme sur un canal ouvert puis, une fois toutes les données drainées, la réception renvoie perpétuellement la valeur vierge du type `T` des éléments du canal, sans bloquer.

Pour différencier les valeurs vierges valides reçues d'un canal de celles reçues depuis un canal vide et fermé, Go dispose d'une particularité syntaxique permettant à l'opérateur de renvoyer un *ou deux* résultats.

La syntaxe basique utilise un seul résultat, sous la forme `element := <-canal`, mais une syntaxe alternative est disponible et recommandée dans la plupart des cas : `element, ok := <-canal`. Dans cette version, `ok` est vrai pour les valeurs reçues correspondant à des valeurs effectivement présentes dans le canal, mais il est faux pour les valeurs vierges émises par un canal fermé et vide.

Une question peut venir à l'esprit : « S'il y a un opérateur réception, où est l'opérateur d'émission ? » En fait, l'émission vers un canal ne renvoie aucune valeur, de sorte que du point de vue de la grammaire de Go, l'instruction d'émission n'est pas un opérateur, tous les opérateurs renvoyant une ou plusieurs valeurs. L'instruction d'émission est détaillée au chapitre 7.2.3.

2.5 PONCTUATION

2.5.1 Opérateurs, instructions et ponctuation

La majorité des symboles de ponctuation de Go sont les opérateurs, déjà décrits au § 2.4.

Tableau 2.8 – Ponctuation : opérateurs

+	-	*	/	%	&	|
^	<<	>>	&^	&&	||	<-
==	<	>	!	<=	>=	!=

D'autres sont utilisés pour exprimer des instructions mais ne sont pas des opérateurs car ils ne renvoient pas de résultat ; ils sont décrits au chapitre 7.

Tableau 2.9 – Ponctuation : instructions

+=	-=	*=	/=	%=	&=
|=	^=	<<=	>>=	&^=	++
--	<-	=	:=	[	]

Le symbole <- présent dans ces deux listes est dans le premier cas celui de l'opérateur de réception et dans la second celui de l'instruction d'émission.

© Dunod – Toute reproduction non autorisée est un délit.

2.5.2 Ponctuation générale

Les symboles de ponctuation hors de ces deux catégories sont les suivants :

✓ **« . . . »** : ce symbole est utilisé de trois façons différentes en Go :
- la définition automatique de la taille d'un tableau initialisé par un littéral composite ;
- la définition du dernier argument d'une fonction variadique ;
- la transformation d'une tranche en liste d'arguments passés à une fonction variadique.

✓ **« { »** et **« } »** : cette paire de symboles est utilisée, toujours en association l'un avec l'autre, de deux façons différentes :
- définition d'un littéral composite ;
- définition d'un bloc de code.

✓ **« (»** et **«) »** : cette paire de symboles est utilisée, toujours en association l'un avec l'autre, de trois façons différentes :
- invocation d'une fonction ou méthode ;
- regroupement de valeurs dans une expression ;
- regroupement de déclarations d'importation en une seule instruction.

✓ **« , »** : la virgule n'est pas un opérateur en Go, à la différence de C/C++, mais est utilisée de multiples façons pour constituer des listes :
- affectation parallèle (cf. § 7.2.5) ;
- liste de paramètres d'une fonction/méthode, ou passés à l'appel d'une fonction/méthode ;
- liste de paquets dans une instruction d'importation ;
- liste de valeurs dans un littéral composite ;
- liste de valeurs la définition d'un retour multiple de fonction/méthode, ou dans une instruction de retour multiple (cf. § 7.4.1) ;
- liste de valeurs dans la clause `case` d'une instruction `switch` ou `select`.

✓ **« ; »** : le point-virgule est utilisé de deux façons :
- séparation des éléments de l'instruction `for` (cf. § 7.3.8) ;
- terminaison des instructions. Dans ce rôle, il est implicite et généralement inséré automatiquement [19], sauf dans le cas des « instructions courtes » présentes dans une instruction de contrôle. Cette technique de l'insertion automatique est un héritage de BCPL, dont Ken Thompson était aussi l'un des créateurs.

2.5.3 Espaces et autre formatages

Au-delà de ces symboles de ponctuation visibles, demeurent les invisibles : espaces, tabulations, et autres caractères de contrôle, qui définissent le format du code.

19. Règles d'insertion automatique : https://osinet.fr/go/syntaxe/golang-semicolons

Dans l'ensemble, Go est – dans la tradition de C et au contraire de Python – un langage à forme essentiellement libre, dans lequel les choix d'espacement, indentation, et retours à la ligne n'ont généralement pas d'impact.

La principale restriction sur ce point concerne le cas des retours à la ligne, en raison de l'insertion automatique des points-virgules, qui a pour effet d'imposer quelques choix de mise en forme, dont le principal est la nécessité de placer les **`else`** sur la même ligne que l'accolade fermante du premier bloc d'une instruction **`if`** : en le mettant sur la ligne suivante, le langage insèrerait un point-virgule après l'accolade, provoquant une erreur de compilation, le **`else`** se retrouvant interprété comme une instruction, alors qu'il n'en est pas une, comme l'illustre le listing 2.13.

Listing 2.13 – Le placement du **`else`**
https://osinet.fr/go/syntaxe/play-else

```
1  package main
2
3  func main() {
4          if 1 == 2 {
5                  println("Erreur de syntaxe")
6          } // Un ; est inséré à la fin du bloc.
7          else {
8                  println("Else n'est pas une instruction")
9          }
10 }
```

Pour éviter la divergence des pratiques, Go a défini et outillé des règles de formatage, fournissant avec `gofmt` un outil permettant d'appliquer à tout code les règles standard, déchargeant les développeurs du souci de formater leur code selon celles-ci puisqu'un outil standard pourra le faire pour eux avant d'enregistrer le code dans un gestionnaire de versions.

Les principales règles liées à ces espaces et formats sont les suivantes :

- ✓ Les indentations utilisent des tabulations, pas des espaces [20].
- ✓ Dans les listes verticales – par exemple les champs d'une structure – les symboles sont alignés au lieu d'avoir un espacement constant d'un seul espace comme dans nombre d'autres normes de codage.
- ✓ La longueur des lignes est laissée à l'appréciation des développeurs, au lieu d'imposer des longueurs historiques comme 80 ou 120 colonnes à la façon des terminaux et télétypes des années 1970.

20. Au-delà des interminables débats en faveur d'une convention ou l'autre a émergé courant 2019 la perception d'un avantage décisif en matière d'accessibilité pour l'usage des tabulations : https://osinet.fr/go/syntaxe/reddit-tabs-a11y

© Dunod – Toute reproduction non autorisée est un délit.

✓ Les parenthèses sont déconseillées lorsqu'elles ne sont pas indispensables, par exemple autour des conditions de `if`.

✓ Pas d'ajout systématique d'espace entre les opérateurs dyadiques et leurs opérandes.

2.6 MOTS-CLEFS

Pour faciliter l'adoption du langage par la réduction du coût d'adoption (cf. § 1.1.1.1), les concepteurs de Go se sont attachés à minimiser le nombre de mots-clefs du langage, reprenant pour l'essentiel ceux du langage C et n'ajoutant que les quelques mots nécessaires aux fonctionnalités originales du langage, avec pour résultat la présence de seulement 25 mots-clefs.

Ils se répartissent en trois groupes principaux :

✓ la définition des données et de leurs types ;

✓ les structures de contrôle ;

✓ l'organisation du code en paquets.

… plus quelques *secrets*, sans utilisation réelle, dans les versions anciennes du langage.

2.6.1 Mots-clefs des définitions de données

Tableau 2.10 – Mots-clefs des définitions de données

Mot-clef	Exemple	Explication
`chan`	`type c chan int`	Définition d'un type « canal d'entiers »
`const`	`const n = 62`	Définition d'une constante
`func`	`var a = func () {}`	Définition d'une fonction
`interface`	`type i interface{}`	Définition d'un type « interface »
`map`	`type m map[int]int`	Définition d'un type « carte »
`struct`	`type s struct {}`	Définition d'un type « structure »
`type`	`type foo bar`	Définition d'un type
`var`	`var m int`	Définition d'une variable

Bien que ces mots-clefs proviennent largement de C, ils ne sont pas strictement équivalents à leurs versions dans d'autres langages :

✓ **const** : n'est utilisé que pour définir des constantes, et ne peut pas être utilisé sur des paramètres de fonction. De plus, à la différence de JavaScript, il n'est pas applicable à toute variable initialisée, mais uniquement à des données dont la valeur peut être calculée à la compilation.

- ✓ **struct** : est utilisé dans une définition de type ou de littéral, mais n'est pas une déclaration autonome de type comme en C (cf. listing 2.14).
- ✓ **type** : est comparable au `typedef` de C (cf. listing 2.14).
- ✓ **var** : ne présente pas les étrangetés (*hissage* ou *hoisting*) de son homologue en JavaScript.

Listing 2.14 – C vs Go : `type`, `typedef` et `struct`
https://osinet.fr/go/syntaxe/play-type

```c
#include <stdio.h>

typedef struct foo {
    int bar;
} baz;

int main() {
        baz f;
        printf("%p\n", &f);
}
```

```go
package main

import "fmt"

type baz struct {
        bar int
}

func main() {
        var f baz
        fmt.Printf("%#v\n", &f)
}
```

2.6.2 Mots-clefs des structures de contrôle

L'usage de **break** peut être une source d'erreurs pour les codeurs familiers avec d'autres langages : il sert à sortir d'une structure de contrôle, et non à terminer une clause **case**, du fait que les clauses d'un **switch** ont par défaut le comportement d'une clause terminée par un **break** dans la plupart des autres langages.

Ceci est détaillé aux § 7.3.5 et 7.3.6 pour les sélecteurs de cas, 7.3.7 pour le sélecteur de communication, et 7.3.9 pour l'instruction **break** elle-même.

© Dunod – Toute reproduction non autorisée est un délit.

Tableau 2.11 – Mots-clefs des définitions de données

Structure	Mots-clefs
Si / sinon	`if else`
Sélecteur de cas	`switch case default break fallthrough`
Sélecteur de communication	`select case default break`
Interruption de séquence	`break`
Boucle	`for range continue break`
Exécution en arrière-plan	`go`
Exécution en fin de fonction	`defer`
Retour de fonction	`return`
Saut arbitraire	`goto`

2.6.3 Mots-clefs des paquets

L'organisation du code Go en paquets, similaire aux espaces de noms (*namespaces*) de PHP ou aux modules JavaScript, est la raison d'être des deux derniers mots-clefs officiels du langage :

✓ `package` : définition d'un paquet ;

✓ `import` : importation des symboles d'un paquet dans le paquet courant.

L'ensemble des notions autour des paquets est décrit au chapitre 9.

2.6.4 Mots-clefs secrets

Jusqu'à la version 1.9, l'analyseur lexical de Go reconnaissait cinq autres mots-clefs, non documentés et utilisés – semble-t-il – uniquement pour valider le fonctionnement du jeton `LIGNORE` [21] de cet analyseur.

Ces mots-clefs étaient :

✓ `notwithstanding`

✓ `thetruthofthematter`

✓ `despiteallobjections`

✓ `whereas`

✓ `insofaras`

Ils ont été retirés de l'analyseur un peu avant la sortie de Go 1.8-beta1, lors du retrait, au commit `7df2f` [22] du couple lexer/parser réalisé lors du replacement des analyseurs d'origine – écrits en C – par des équivalents écrits en Go dans Go 1.5 [23].

21. https://osinet.fr/go/syntaxe/gh-go-lignore
22. https://osinet.fr/go/syntaxe/gh-go-nosecret
23. https://osinet.fr/go/syntaxe/golang-1.5

3 Types élémentaires

3.1 TYPES SIMPLES

3.1.1 Liste des types simples

Go fournit 18 types simples, pour chacune des natures de valeurs ; ce sont ces types qui sont utilisés pour fabriquer les types composites. À l'exception des types complexes – qui n'existent pas dans certains de ces langages – ces types sont les mêmes que les types *scalaires* de PHP, Python [1], ou JavaScript.

Le tableau 3.1 représente une classification logique de ces types.

Tableau 3.1 – Types simples

Nature	Variante	Tailles définies				Implicite
Booléens						`bool`
Entiers	Signés	`int8`	`int16`	`int32`	`int64`	`int`
	Non signés	`uint8`	`uint16`	`uint32`	`uint64`	`uint`
	Spéciaux	`byte`		`rune`		`uintptr`
Flottants	Réels			`float32`	`float64`	
	Complexes	`complex64`		`complex128`		
Chaînes						`string`

1. Python définit deux autres types scalaires : `bytes`, une simple séquence d'octets, qui est un type composite en Go, et `NoneType` qui évoque de loin le `nil` de Go.

3.1.2 Entiers et caractères

3.1.2.1 Liste des types entiers et caractère

Les entiers sont représentés en Go depuis sa création par des littéraux similaires à ceux des langages dérivés de C, pour les littéraux décimaux, octaux, et hexadécimaux.

Par extension au format traditionnel comme `1234567`, Go 1.13 introduit la possibilité de séparer les groupes de chiffres par le caractère `_`, afin d'améliorer la lisibilité, suivant ici l'exemple d'Ada (1983), repris depuis par de nombreux autres langages, autorisant des notations comme `1_234_567`.

Go 1.13 a également introduit de nouvelles notations pour les littéraux binaires et octaux. Une extension de notation avait été envisagée pour des bases arbitraires de 2 à 36[2], mais a été finalement écartée car peu utile une fois les nouveaux formats adoptés. Le tableau 3.2 présente les notations disponibles pour les entiers, dont les caractères (**`rune`**).

Outre les classiques types **`int`**`[8|16|32|64]` et leurs équivalents non signés, Go inclut trois types entiers spéciaux[3] :

- ✓ **byte** : est un alias (cf. page 160) du type entier non signé **`uint8`**, reprenant une notation extrêmement courante des programmeurs C.
- ✓ **rune** : est un alias du type entier signé **`int32`** utilisé pour la représentation des caractères Unicode, à la différence d'un type comme le `char` de C/C++, limité à un octet.
- ✓ **uintptr** : est un type entier spécifié comme étant d'une taille suffisante pour contenir la représentation binaire de tout pointeur. Il est principalement utilisé dans les opérations de bas niveau devant effectuer des manipulations numériques sur les pointeurs.

Tableau 3.2 – Syntaxe des entiers

Base	Exemple	Valeur décimale	Remarque
2	`0b1111011`	64 + 32 + 16 + 8 + 2 + 1 = 123	Go ⩾ 1.13
8	`0173` `0o173`	1 * 64 + 7 * 8 + 3 * 1 = 123	Traditionnel Go ⩾ 1.13
10	`123`	1 * 100 + 2 * 10 + 3 * 1 = 123	Traditionnel
16	`0x7B`	7 * 16 + 11 * 1 = 123	Traditionnel
rune	`'{'`	Unicode `U+007B` : 123	Traditionnel

2. https://osinet.fr/go/types/gh-go-28256
3. https://osinet.fr/go/types/gh-go-entiers

3.1.2.2 Précision des opérations sur les entiers

Les entiers ayant une représentation machine d'une taille déterminée selon leur type, les opérations arithmétiques peuvent occasionner un dépassement de capacité (« *overflow* »). Go précise que celui-ci est toujours ignoré et ne provoque aucune erreur, en appliquant des règles distinctes pour les types entiers signés et non signés :

- ✓ **Entiers non signés** : les opérations d'addition, soustraction, multiplication et décalage vers la gauche sont effectuées modulo $2^{\textit{bits du type}}$, empêchant de fait tout dépassement de capacité.
- ✓ **Entiers signés** : les mêmes opérations, ainsi que la division, peuvent dans ce cas causer un dépassement de capacité, mais le résultat est défini et déterminé par la représentation des entiers signés sur l'architecture concernée, sans causer d'erreur là non plus.

Ces règles interdisent certaines optimisations arithmétiques aux compilateurs, qui doivent prendre en compte l'existence du dépassement de capacité et ses conséquences, dont trois cas sont détaillés dans le listing 3.1 :

- ✓ Le premier cas est probablement le plus familier : une variable non signée 32 bits est initialisée avec $2^{32} - 1$.
  L'ajout de 1 donne $2^{32} \bmod 2^{32} = 0$, violant l'inégalité arithmétique $\forall x \in \mathbb{Z}, x < x+1$ que certains compilateurs pourraient utiliser en optimisation.
- ✓ Le deuxième cas est similaire avec un entier 32 bits signé : la variable est initialisée avec $2^{31} - 1$, qui est la plus haute valeur possible pour un entier signé 32 bits, représentée par `0x7fffffff`.
  L'ajout de 1 donne `0x80000000`, qui est $-(2^{32})$, la plus basse valeur possible pour ce type, violant de nouveau la même inégalité.
- ✓ Le troisième cas est moins commun : la variable `b` est initialisée avec -128, qui est la plus basse valeur possible pour un entier signé 8 bits, représentée par `0x80`.
  La division par −1 donne $-128/-1 = 128$, qui est représenté par `0x80`, soit de nouveau −128, violant la relation $\forall x \in \mathbb{Z}, -x = x/-1$.

Ces règles peuvent sembler être des subtilités destinées uniquement aux auteurs de compilateurs, mais dans la pratique elles ont pour conséquence que toute opération effectuée sur des variables entières en Go doit prendre en compte au niveau applicatif le risque de perte de précision :

- ✓ soit en vérifiant les valeurs en amont ;
- ✓ soit en étendant les valeurs au type de taille supérieure *avant* l'utilisation dans une opération si le type de taille supérieure existe, ce qui n'est pas le cas pour les entiers 64 bits.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 3.1 – Dépassement de précision sur entiers
https://osinet.fr/go/types/play-precision

```
package main

import "fmt"

func main() {
        var u uint32 = 4294967295 // 1<<32-1
        fmt.Println(u, u+1, u < u+1)

        var i int32 = 2147483647 // 1<<31-1
        fmt.Println(i, i+1, i < i+1)

        var b int8 = -128 // -1<<7
        fmt.Println(b, b/-1, b != b/-1)
}

// Résultat
// 4294967295 0 false
// 2147483647 -2147483648 false
// -128 -128 false
```

3.1.2.3 Littéraux entiers `rune` : caractères

La dernière ligne du tableau 3.2 concernant la représentation **`rune`** des entiers pouvait surprendre, n'étant pas une base numérique, mais il s'agit de la même mécanique qu'en C et ses dérivés, associant un point de code Unicode à l'entier correspondant au même motif binaire 32 bits.

Outre les caractères ordinaires, la syntaxe Go reconnaît également pour la définition des entiers **`rune`** plusieurs formes de mécanismes d'échappement, permettant de saisir des points de code non affichables, selon les règles du tableau 3.3.

Ces échappements, à deux exceptions près pour l'apostrophe et le guillemet, sont applicables aussi bien pour les littéraux de type **`rune`** que pour la définition de runes à l'intérieur de chaînes de caractères.

Les deux mécanismes de définition de caractères Unicode par le numéro de leur point de code aux deux dernières lignes du tableau 3.3 permet de définir des points de code arbitraires, et n'effectue aucune validation par rapport aux tables Unicode.

Les deux types de caractères invalides les plus courants sont :

✓ Les points de code supérieurs à 0x10FFFF, limite supérieure d'Unicode depuis la version 2.0 et toujours en vigueur [4] dans la version 12 de mars 2019.

4. https://osinet.fr/go/types/unicode-bits

Tableau 3.3 – Échappements dans les runes

Notation	Unicode	Description
`\a`	`U+0007`	`^G` : ASCII `BEL` : bip console
`\b`	`U+0008`	`^H` : ASCII `BS` : retour d'un caractère
`\f`	`U+000C`	`^L` : ASCII `FF` : saut de page
`\n`	`U+000A`	`^J` : ASCII `LF` : descente d'une ligne
`\r`	`U+000D`	`^M` : ASCII `CR` : retour ligne
`\t`	`U+0009`	`^I` : ASCII `HT` : tabulation horizontale
`\v`	`U+000B`	`^K` : ASCII `VT` : tabulation verticale (effacement d'écran)
`\\`	`U+005C`	barre oblique inverse (\)
`\'`	`U+0027`	apostrophe, uniquement dans un littéral rune
`\"`	`U+0022`	guillemet, uniquement pour une rune dans un littéral chaîne
Notation	**Point de code**	**Exemple d'évaluation**
`\`*nnn*	Octal	`'\101' → 1*64+1*1 → 65 → 'A'`
`\x`*nn*	Hexa 8 bits	`'\x41' → 4*16+1*1 → 65 → 'A'`
`\u`*nnnn*	Hexa 16 bits	`'\x0041' → 4*16+1*1 → 65 → 'A'`
`\U`*nnnnnnnn*	Hexa 32 bits	`'\x00000041' → 4*16+1*1 → 65 → 'A'`

✓ Les points de code correspondant aux demi-paires subrogées, dans les plages 0xD800-0xDBFF (1024 points de codes subrogés hauts) et 0xDC00-0xDFFF (1024 points de code subrogés bas) : leur apparition résulte en général d'une erreur de décodage UTF-16.

À titre d'exemple, la clef de sol, point de code U+1D11E, est représentée en UTF-16 par deux demi-paires subrogées U+D834 U+DD1E qui, stockées dans une valeur **`rune`**, sont invalides séparément. Les données encodées en UTF-16 doivent être traitées comme des flux d'octets bruts et non comme des flux de **`rune`** ou des chaînes. Leur représentation interne en Go est analysée au § 3.1.4 à partir du listing 3.4.

3.1.3 Nombres flottants réels et complexes

3.1.3.1 Liste des types flottants

Comme la plupart des langages de programmation, Go propose des nombres à virgule flottante, suivant la norme IEEE-754.

De façon moins courante, il propose non seulement des types destinés aux nombres réels, mais aussi aux nombres complexes.

© Dunod – Toute reproduction non autorisée est un délit.

Pour les parties de la norme qu'il implémente, il s'aligne sur la version 2008 du standard ; mais il n'inclut pas certaines nouveautés de cette version comme les nouveaux types binaires 16 et 128 bits ni les types décimaux : ces derniers font l'objet d'une proposition pour Go 2 [5].

En l'absence des formats 16 et 128 bits de cette révision, les flottants Go peuvent être simple précision (32 bits) ou double précision (64 bits), correspondant respectivement aux types **`float32`** et **`float64`**.

Les complexes étant réalisés par la combinaison d'une partie réelle et d'une partie imaginaire toutes deux flottantes, les types associés sont **`complex64`** et **`complex128`**.

Et les quaternions ?

Il est possible de trouver par endroits mention du soutien natif de Go pour les quaternions (cf. https://osinet.fr/go/types/wiki-quaternion), une généralisation des nombres réels et complexes.

En réalité, ce soutien a bien fait l'objet d'une proposition formelle : https://osinet.fr/go/types/gh-go-19813, mais il s'agissait d'un poisson d'avril ; la gestion des quaternions n'est disponible que sous forme de paquets contribués, parmi lesquels :

- ✓ https://osinet.fr/go/types/gh-quaternion-westphae
- ✓ https://osinet.fr/go/types/gh-quaternion-thisisneal

Le paquet d'interfaçage Protocol Buffers dans les API Google inclut une représentation des quaternions, utilisable comme référence pour toute API devant manipuler ces données : https://osinet.fr/go/types/gh-quaternion-google

3.1.3.2 Littéraux flottants

Les littéraux flottants sont de deux natures : réels ou imaginaires purs, combinant un littéral numérique et le caractère `i` ; en effet la représentation d'une valeur combinant une partie réelle et une partie imaginaire, comme $2 + 3i$ n'est pas un littéral, mais une expression constante combinant deux littéraux flottants, respectivement 2 et $3i$.

Syntaxiquement, ils sont similaires à ceux des autres langages dérivés de C, composés de :

- ✓ un signe + ou –, le + étant implicite ;
- ✓ une partie entière ;
- ✓ un délimiteur `.` ;
- ✓ une partie décimale (fractionnelle) ;
- ✓ un délimiteur d'exposant décimal `e` ou `E`, ou binaire `p` ou `P` ;
- ✓ un signe d'exposant décimal ou binaire + ou –, le + étant implicite ;
- ✓ un exposant décimal ou binaire ;
- ✓ dans le cas des flottants imaginaires purs, le symbole `i`.

5. https://osinet.fr/go/types/gh-go-19787

Dans le cas des exposants binaires `p` et `P`, la mantisse doit être sous forme hexadécimale, comme `-0x10P-4`, qui s'évalue à $-16 * 2^{-4} = -16/16 = -1$. Ce format est disponible depuis Go 1.13.

Il est possible d'omettre la partie entière *ou* la partie fractionnelle, et le délimiteur `.` ou le délimiteur d'exposant décimal. À l'image des littéraux entiers, il est possible à partir de Go 1.13 de séparer les chiffres de la partie entière, de la partie fractionnelle, ou de l'exposant décimal par des caractères _ pour faciliter la lecture en regroupant les chiffres au choix de l'auteur :

```
1_23.45_67E-2_3 == +1.234567e-21.
```

Les seules différences entre les deux natures sont, pour les valeurs imaginaires, la présence du caractère `i` en fin de littéral, et le fait qu'un délimiteur de partie fractionnelle n'est pas nécessaire : dès lors que `i` est présent, le littéral numérique qui le précède est interprété comme une valeur flottante, avec entre autres conséquences l'absence d'interprétation du zéro initial comme indicateur de base octale :

- ✓ `i` seul n'est pas la représentation du nombre imaginaire *i*, mais un identificateur. Un littéral dénotant le nombre *i* est `1i`.
- ✓ `2i == 2*1i` et `2i == 2.0i`.
- ✓ `010i == 10.0i`, mais `010+0i != 10.0+0i` car cette représentation n'est pas un littéral complexe, mais une expression combinant une valeur non typée `010`, qui vaut donc 8, avec la valeur imaginaire `0i`, donc `010+0i == 8.0+0i`.
- ✓ Au contraire `010.0 == 10.0` car la présence du délimiteur décimal indique une valeur flottante, qui désactive le 0 comme préfixe de représentation octale.

3.1.3.3 Opérations sur les flottants

Les quatre opérateurs arithmétiques de base [6] s'appliquent aux types flottants – y compris les types complexes – comme aux autres types numériques. L'opérateur de modulo, en revanche, ne leur est pas applicable.

Le résultat de la division par 0, sur les réels comme sur les complexes, est spécifié par la norme IEEE 754 : `+Inf` / `-Inf`, ou `NaN` selon le dividende. Toutefois un compilateur conforme à la spécification peut provoquer une erreur d'exécution au lieu de renvoyer cette valeur : la norme IEEE 754 précise en effet à la fois la valeur si elle est renvoyée *et* le signalement d'une erreur.

De même, les opérateurs de comparaison sur les valeurs complexes sont limités à l'égalité et l'inégalité, aucune relation d'ordre total ne pouvant être définie sur le corps des nombres complexes. L'existence des opérateurs d'égalité/différence ne signifie pas qu'il soit opportun de les utiliser, en raison des restrictions liées à la précision sur les nombres en virgule flottante, détaillée au paragraphe suivant.

6. Voir § 2.4, pages 30-33.

© Dunod – Toute reproduction non autorisée est un délit.

En complément des opérateurs, le langage inclut trois fonctions intégrées (*builtins*) pour la manipulation des valeurs complexes :

- ✓ **complex** : la fonction `complex()` prend deux valeurs flottantes réelles, et les combine comme partie réelle et partie imaginaire d'une valeur complexe qu'elle renvoie comme résultat.
- ✓ **real** : la fonction `real()` prend une valeur complexe et renvoie sa partie réelle sous forme d'une valeur flottante réelle.
- ✓ **imag** : la fonction `imag()` prend une valeur complexe et renvoie sa partie imaginaire sous forme d'une valeur flottante réelle.

Ces trois fonctions s'appliquent aux deux tailles de valeurs flottantes, et renvoient les types associés, comme illustré par le listing 3.2. Si tous leurs arguments sont constants, leur résultat est une valeur constante également.

Listing 3.2 – Types des builtins complexes
https://osinet.fr/go/types/play-builtins-complexes

```
package main

import "fmt"

func main() {
        var s, t float32 = 1, 2
        var d, e float64 = 3, 4

        u := complex(s, t)
        f := complex(d, e)
        fmt.Printf("2*float32: %T\n2*float64: %T\n", u, f)

        fmt.Printf("complex64: %T/%T\ncomplex128 %T/%T\n",
                real(u), imag(u),
                real(f), imag(f),
        )
}

// Résultat:
// 2*float32: complex64
// 2*float64: complex128
// complex64: float32/float32
// complex128 float64/float64
```

3.1.3.4 Précision des opérations sur les flottants

La précision des opérations sur les flottants est définie par la norme IEEE 754, et ne peut – par la nature même de la représentation en virgule flottante – être mathématiquement exacte : seules peuvent être exactes les opérations effectuées sur des nombres flottants correspondant à des valeurs entières représentables sans perte de précision.

Ceci peut survenir même sur des valeurs très courantes, comme dans le listing 3.3 : le résultat devrait être $0 + 1i$, mais le résultat est correct pour la partie imaginaire et non pour la partie réelle.

Listing 3.3 – Imprécision des opérations en virgule flottante
https://osinet.fr/go/types/play-precision-flottants

```
1  package main
2
3  import "math/cmplx"
4
5  func main() {
6          println(cmplx.Pow(-1, 0.5))
7  }
8
9  // Résultat: (+6.123234e-017+1.000000e+000i)
```

Depuis Go 1.12, le langage autorise explicitement l'utilisation des opérateurs de produit/somme fusionnés (*FMA : fused multiply-add*) [7], comme ceux inclus dans les processeurs x86-64 récents. L'avantage de cette opération est l'application d'une seule étape d'arrondi sur la combinaison des deux opérations au lieu du traitement de chaque étape d'une expression comme une valeur sujette à arrondi. Cette utilisation est possible dès lors que l'expression n'inclut pas une étape intermédiaire explicite de conversion dans un type spécifique ; en effet dans ce cas, la valeur est arrondie lors de la conversion avant de poursuivre l'évaluation.

✓ FMA possible : `a = b*c + d;`

✓ FMA impossible : `a = float64(b*c) + d`.

3.1.4 Chaînes de caractères

3.1.4.1 Représentation

Les chaînes de caractères en Go sont traitées comme des suites de caractères Unicode, et non comme de simples suites d'octets à l'instar de nombreux autres langages.

7. https://osinet.fr/go/types/wiki-fma

© Dunod – Toute reproduction non autorisée est un délit.

Les valeurs de type chaîne sont *immuables*, ce qui autorise l'optimisation de leur utilisation en mémoire, leur caractère immuable permettant de réutiliser une représentation partout où elle apparaît sans craindre d'écrasement par une quelconque modification.

Ces chaînes sont représentées en UTF-8 afin de pouvoir encoder tout le répertoire Unicode, suivant la démarche décrite au § 1.1.2.3. Le langage permet nativement d'itérer sur les caractères plutôt que sur les octets.

Elles sont convertibles sans perte depuis et vers le type « tranche d'octets » [8], et le listing 3.4 illustre la différence de traitement entre la manipulation de chaînes et la manipulation de niveau octet comme en PHP, en l'appliquant au caractère multi-octets « clef de sol » dont la représentation UTF-16 a été évoquée à la fin du § 3.1.2.

Listing 3.4 – Itération sur les chaînes
https://osinet.fr/go/types/play-iteration-chaines

```
package main

import "fmt"

func main() {
        const a = "\U0001D11E" // Symbole clef de sol.
        var s = []byte(a)      // Tranche d'octets.

        fmt.Printf("Longueurs: a = %d, s = %d\n", len(a), len(s))
        fmt.Println("\nBoucle sur la chaîne")
        for ak, av := range a {
                fmt.Printf("%d 0x%08x\n", ak, av)
        }
        fmt.Println("\nBoucle sur la tranche")
        for sk, sv := range s {
                fmt.Printf("%d 0x%02x = 0b%08b\n", sk, sv, sv)
        }
}

/* Résultat:
Longueurs: a = 4, s = 4

Boucle sur la chaîne
0 0x0001d11e

Boucle sur la tranche
0 0xf0 = 0b11110000
1 0x9d = 0b10011101
2 0x84 = 0b10000100
3 0x9e = 0b10011110
*/
```

8. Les types tranche sont décrits au § 4.2.

Dans ce programme, la boucle sur la chaîne ne réalise qu'une seule itération, absorbant le caractère clef de sol en une seule fois, avec une valeur hexadécimale `0x0001d11e` correspondant exactement à la séquence d'échappement utilisée pour définir la rune dans la chaîne.

Au contraire, la boucle sur la version convertie en tranche d'octets réalise quatre itérations, pour chacun des quatre octets composant la rune clef de sol.

Particularité de ces boucles : alors que le point de code de cette rune est `0x0001d11e`, les octets renvoyés par l'itération ne sont pas comme on pourrait s'y attendre `0x00 0x01 0xd1 0x1e`, mais `0xf0 0x9d 0x84 0x9e`.

Les valeurs de ces octets démontrent en fait l'encodage UTF-8 interne des chaînes [9]. Examinons comment cet encodage permet de reconstituer la chaîne d'origine :

- ✓ **Longueurs** : que ce soit pour la version chaîne ou la version tranche d'octets, le nombre d'octets utilisés est toujours 4.
- ✓ **`0xf0 = 1111 0000`** : la présence des trois bits de poids fort à 111 dénote la représentation sur quatre octets, nécessaire pour les points de code de U+10000 à U+1FFFF. La donnée utile sur ce premier octet se compose des trois bits de poids faible : `000`, précisant un caractère compris entre U+10000 et U+3FFFF.
- ✓ **`0x9d = 1001 1101`** : les quatre bits de poids fort étant `1001`, la partie utile se compose des six bits de poids faible : 011101, précisant que le caractère est compris entre U+10000 et U+1FFFF.
- ✓ **`0x84 = 1000 0100`** : seuls les six bits de poids faible sont utiles : `000100`.
- ✓ **`0x9e = 1001 1110`** : seuls les six bits de poids faible sont utiles : `011110`.
- ✓ **Réassemblage des octets** :
  - résultat en bits : `000 011101 000100 011110` ;
  - groupé en octets : `(000)0 0001 1101 0001 0001 1110` ;
  - soit en hexadécimal `0x01d11e`.

On retrouve bien le point de code `U+0001D11E` de la clef de sol, comme attendu.

UTF-8 ou pas ?

Comme indiqué plus haut, les chaînes sont traitées comme des suites de caractères Unicode encodés en UTF-8 mais, à l'exception des littéraux chaîne, le langage ne fournit pas de garantie qu'une variable de type chaîne donnée est effectivement une séquence de caractères valides.
À titre d'exemple, une chaîne initialisée par la conversion d'une tranche d'octets quelconques peut parfaitement contenir des séquences UTF-8 invalides, qui peuvent représenter un risque de sécurité à l'affichage notamment dans un navigateur, comme dans le modèle d'attaque « CAPEC-80 », décrit sur https://osinet.fr/go/types/capec-80
Toute valeur chaîne obtenue depuis une autre source que le code du programme doit donc être validée au moyen de la fonction `utf8.ValidString(s string) bool` avant d'être utilisée comme une chaîne.

9. Le tableau de décodage de Wikipedia est l'une des ressources les plus claires pour suivre ce décodage. Voir https://osinet.fr/go/types/wiki-utf8

© Dunod – Toute reproduction non autorisée est un délit.

3.1.4.2 Littéraux chaînes

Les littéraux chaînes peuvent être déclarés avec deux syntaxes :

- ✓ **Chaîne interprétée** : délimités par les guillemets classiques hérités de C, les littéraux chaîne interprétée sont ceux démontrés depuis le début de cet ouvrage, par exemple dans le listing 3.4. Ils permettent la saisie de caractères non-ASCII au moyen des séquences d'échappement décrites à propos du type **`rune`** dans le tableau 3.3 page 47. Ces littéraux ne peuvent contenir de délimiteur de fin de ligne, ce qui les limite à représenter des chaînes d'une seule ligne. À la différence des littéraux **`rune`**, les guillemets, utilisés comme délimiteurs du littéral, doivent être échappés.
- ✓ **Chaîne brute** : délimités par des accents graves (*backtick, backquote*), les littéraux chaîne brute permettent au contraire de saisir des chaînes multi-lignes, acceptant tous les caractères sauf l'accent grave, pour lequel aucune séquence d'échappement n'est disponible.

Bien que les littéraux de chaîne brute acceptent tous les caractères, d'après la spécification des chaînes Go, tous ne sont pas entièrement utilisables :

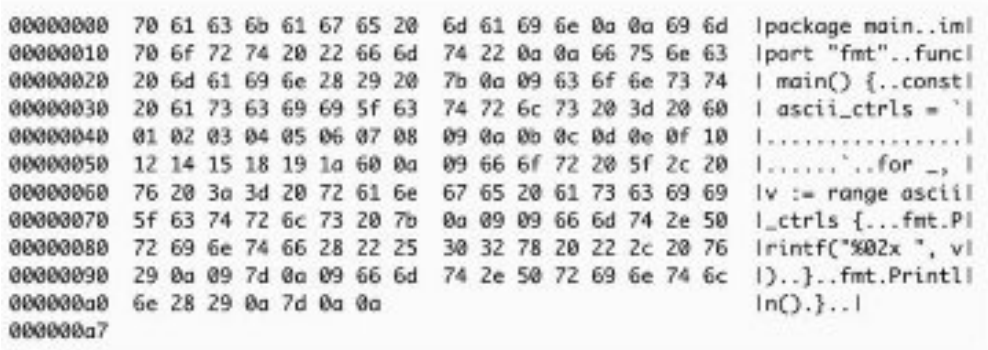
- ✓ Comme indiqué au § 2.1.1, tout compilateur Go peut indifféremment rejeter ou accepter le caractère ASCII `NUL`, de valeur entière 0, dans les codes source. En particulier, le compilateur standard a fait le choix de le rejeter. Son utilisation est donc vivement déconseillée.
- ✓ Le caractère `CR`, de valeur entière 13, est au contraire purement et simplement supprimé lors de la transcription du littéral en valeur chaîne.

Les figures 3.1 et 3.2 représentent un programme Go incluant les caractères de contrôle ASCII correspondant aux valeurs entières 01 à 26, respectivement dans `vim` et en dump hexadécimal canonique, à la ligne 00000040.

Figure 3.1 – Caractères de contrôle dans `vim`

```
1 package main
2
3 import "fmt"
4
5 func main() {
6         const ascii_ctrls = `^A^B^C^D^E^F^G^H
7 ^K^L^M^N^O^P^R^T^U^X^Y^Z`
8         for _, v := range ascii_ctrls {
9                 fmt.Printf("%02x ", v)
10        }
11        fmt.Println()
12 }
```

Figure 3.2 – Caractères de contrôle en hexadécimal



À l'exécution, ce programme renvoie :

```
01 02 03 04 05 06 07 08 09 0a 0b 0c 0a 0e 0f 10 12 14 15 18 19 1a
```

Bien que les deux vues du source montrent bien la présence du caractère `CR`, sous la forme d'un `^M` dans `vim` et de la valeur hexadécimale `0d` dans le dump, le résultat liste bien tous les caractères, sauf celui-ci, confirmant son élimination dans les littéraux bruts.

3.1.4.3 Opérations sur les chaînes

Le caractère immuable des chaînes limite les opérations qui leur sont applicables :

- ✓ **Indexation** : il est possible d'accéder aux octets individuels – et non aux runes – en utilisant la syntaxe des tableaux, par exemple `"golang"[3] == 'a'`. Cette opération d'indexation est limitée à la lecture et ne peut être utilisée pour modifier les octets comme c'est habituellement possible dans les langages de la famille de C.
- ✓ **Adresse** : il est possible de prendre l'adresse d'une variable chaîne avec l'opérateur `&`, mais – à la différence d'un tableau d'octets – il n'est pas possible de prendre l'adresse du résultat d'une indexation de chaîne : cela fournirait en effet un moyen indirect de violer la règle d'immuabilité de ce type.
- ✓ **Concaténation** : la seule opération véritablement utile sur les chaînes est la concaténation, dénotée par l'opérateur `+`[10]. Le résultat de la concaténation de deux chaînes de caractères est une troisième chaîne, sans modification des chaînes opérandes.
- ✓ **Longueur** : la fonction intégrée `len` renvoie le nombre d'octets d'une chaîne, tandis que `utf8.RuneCountInString` renvoie le nombre de caractères.

10. Voir § 2.4.4 page 31.

© Dunod – Toute reproduction non autorisée est un délit.

✓ **Bibliothèque** : la bibliothèque standard du langage contient deux principaux paquets destinés à la manipulation des chaînes de caractère :

- **`strings`** : destiné spécifiquement aux opérations usuelles sur les variables chaînes ;
- **`unicode`** : et ses sous paquets `utf8` et `utf16` destinés à la manipulation des données Unicode en prenant en compte l'encodage, dont la capacité de convertir des paires de valeurs en runes.

3.2 TYPES POINTEURS

3.2.1 Définition

Les pointeurs sont un type de données courant dans les langage de programmation généraux comme C/C++/C#, Java ou Pascal. En revanche, ils ne sont pas accessibles aux développeurs d'applications dans les langages de scripts usuels comme PHP, JavaScript, Python ou Ruby, sauf parfois dans les bibliothèques d'interfaçage avec les librairies C, comme le paquet Python `ctypes` [11], ou l'extension FFI en PHP 7.4.

Dans une mémoire traitée comme un espace linéaire, les variables sont stockées sur des positions consécutives – à l'alignement près – à partir d'un point de départ dans l'un des espaces mémoire alloués au programme. Pour chaque variable, l'emplacement du premier octet qu'elle occupe dans cet espace est dénommé l'*adresse* de la variable, et un *pointeur* sur cette variable est une valeur représentant cette adresse [12].

Cette valeur est en pratique un nombre entier positif, et Go définit un type spécifique, **`uintptr`**, comme étant un entier d'une taille appropriée pour stocker la représentation d'un pointeur sur l'architecture cible pour laquelle le programme est compilé.

Taille des pointeurs

La taille des pointeurs n'est pas liée à la taille marketing des processeurs, qui se réfère à la largeur des mots utilisés pour les calculs et non à la largeur du bus d'adresse ou des adresses virtuelles possibles. Ainsi, sur x86-64, les types pointeurs sont d'une largeur de 64 bits, mais en pratique seul un nombre limité de bits de poids faible est utilisé : 48 bits depuis les premiers processeurs Opteron™ (2007), autorisant d'adresser jusqu'à 2^{48} = 256 To de mémoire.

Dans un type composite, l'adresse d'un élément est calculée par le langage en ajoutant un décalage correspondant à l'élément (*offset*) au pointeur sur la variable.

11. Types `c_char_p`, `c_wchar_p`, `c_void_p` : https://osinet.fr/go/types/python-ctypes
12. Plus de détails sur https://osinet.fr/go/types/cheney-pointeurs

Ce mécanisme d'arithmétique sur les pointeurs, très largement utilisée dans la programmation C traditionnelle, a été reconnu comme une source de bogues, que les langages ultérieurs ont tenté d'éliminer, parfois en supprimant l'accès aux pointeurs, comme dans les langages de script web usuels, parfois en leur proposant des alternatives.

Dans le cas de Go, le choix a été de maintenir la notion de pointeur pour les programmeurs d'application, pour conserver un modèle mental de programmation proche du système, mais d'interdire l'usage direct de l'arithmétique de pointeurs à moins de passer par des mécanismes expressément désignés comme dangereux, regroupé dans le paquet `unsafe`, décrit dans les *Plus en ligne*.

3.2.2 Déclaration de types pointeurs

Les types pointeurs représentent l'ensemble des pointeurs vers des variables d'un type donné, qui est dénommé *type de base* (*base type*) du pointeur.

Ils sont définis par un astérisque suivi du type de base vers lequel ils pointent :

```
type PT *T
```

Dans cet exemple, le type `PT` est un type pointeur sur le type de base `T`.

Pour tous les types pointeurs, la valeur vierge est **`nil`**. Bien que ce point soit absent de la spécification du langage, **`nil`** correspond à la valeur numérique zéro dans le type **`uintptr`** pour tous les compilateurs usuels.

Les types pointeurs sont l'une des deux seules catégories de types définis à ne pas supporter la définition de méthodes – l'autre étant les types interfaces : en effet, lorsqu'une méthode est définie avec un receveur pointeur, elle est considérée comme définie sur le type sous-jacent et non sur le type pointeur lui-même. La définition des méthodes est décrite au § 3.4.

La représentation des valeurs de certains types qui ne sont pas spécifiés comme des types pointeurs est également au format pointeur. C'est par exemple le cas des valeurs de type fonction ou interface.

3.2.3 Littéraux pointeurs

Il n'existe pas strictement de littéraux pointeurs : initialiser une variable de type pointeur sur un type de base `Base` passe par l'utilisation d'une expression de ce type. Les principaux types d'expressions pouvant être utilisées sont illustrés par le listing 3.5 :

- ✓ **Fonction `new`** : la fonction intégrée `new(Base)` permet de renvoyer un pointeur sur une valeur vierge de type `Base`.
- ✓ **Adresse d'un littéral** : si les littéraux du type `Base` sont adressables, comme c'est le cas pour les types structure, l'opérateur d'adresse `&` (cf. § 2.4.8) peut être appliqué à un littéral de type `Base` pour produire un pointeur sur `Base`.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 3.5 – Littéraux pour pointeurs
https://osinet.fr/go/types/play-litteraux-pointeurs

```
package main

import "fmt"

type Base struct{ x int }
type Pointeur *Base

func main() {
        // Fonction intégrée new.
        var p0 Pointeur = new(Base)

        // Adresse d'un littéral adressable.
        var p1 Pointeur = &Base{1}

        // Utilisation d'une variable.
        var b Base = Base{2}
        var p2 Pointeur = &b

        // Expression renvoyant un Pointeur.
        var p3 Pointeur = func(x int) Pointeur {
                return &Base{x}
        }(3)

        fmt.Printf("%T %v\n%T %v\n%T %v\n%T %v\n",
                p0, p0, p1, p1, p2, p2, p3, p3)
}

// Résultat
// main.Pointeur &{0}
// main.Pointeur &{1}
// main.Pointeur &{2}
// main.Pointeur &{3}
// invalid operation: p1a == p2 (mismatched types P1 and P2)
```

- ✓ **Adresse d'une variable** : le passage par une variable de type `Base` permet d'utiliser l'opérateur d'adresse `&` (cf. § 2.4.8) à la variable pour obtenir un pointeur du type souhaité, avec une valeur qui peut être non vierge.
- ✓ **Expressions générales** : il est possible d'utiliser d'autres mécanismes renvoyant un résultat du type souhaité, comme des fonctions ou l'utilisation de l'opérateur adresse sur d'autres expressions adressables, mais les deux premières formes sont plus naturelles.

3.2.4 Manipulation des pointeurs

Les expressions utilisant des opérandes pointeurs sont les suivantes :

✓ **Comparaison** : les pointeurs du même type peuvent être comparés, y compris à `nil`, mais uniquement par les opérateurs `==` et `!=` : aucune relation d'ordre n'est définie entre eux, comme dans le listing 3.6.

Listing 3.6 – Comparaison de pointeurs
https://osinet.fr/go/types/play-comparaison-pointeurs

```
1  package main
2
3  import (
4          "fmt"
5          "unsafe"
6  )
7
8  type P1 *int
9  type P2 P1
10
11 func main() {
12         var i int = 36
13         var p1a, p1b P1 = &i, &i
14         var p2 P2 = &i
15
16         fmt.Println(p1a, p1b, p2, p1a == p1b,
17                 unsafe.Pointer(p1a) == unsafe.Pointer(p2))
18
19         // Erreur de compilation: types différents.
20         // fmt.Println(p1a == p2)
21 }
22
23 // Résultat: 0x414020 0x414020 0x414020 true true
```

Comme l'illustre ce listing, il ne suffit pas que les pointeurs soient des pointeurs vers le même type de base, ni même que l'un soit un type défini vers l'autre : la ligne 20, décommentée, provoque une erreur de compilation.
La seule possibilité pour une comparaison entre deux types différents est que l'un des types soit un alias de l'autre, par exemple en modifiant le listing pour déclarer `type P2 = P1` au lieu de `type P2 P1` à la ligne 9.
Deux pointeurs sont égaux s'ils pointent la même variable, ou sont tous les deux égaux à `nil`. Ils peuvent également – de façon imprévisible car non spécifiée – être égaux s'ils pointent deux variables différentes de longueur nulle.
En effet, quel que soit leur type, les valeurs de longueur nulle n'ont aucune occupation mémoire réelle, et peuvent avoir la même adresse ou non comme le montre

© Dunod – Toute reproduction non autorisée est un délit.

le listing 3.7 : ce point n'est pas spécifié par le langage ; de ce fait, leurs pointeurs peuvent être égaux ou non à l'exécution. Lorsqu'ils sont alloués explicitement, leur valeur est celle de la constante globale `runtime.zerobase`, définie dans `malloc.go`.

Listing 3.7 – Pointeurs sur données de longueur nulle
https://osinet.fr/go/types/play-pointeurs-sur-vide

```
1  package main
2
3  import "fmt"
4
5  type Empty struct{}
6
7  func main() {
8          t1, t2, t3 := &Empty{}, &Empty{}, &Empty{}
9          fmt.Printf("%p %p %t %t", t1, t2, t1 == t2, t1 == t3)
10 }
11
12 // Résultat: 0x1c4c84 0x1c4c84 true false
```

Dans cet exemple, l'exécution sur le playground Go montre que les deux premiers pointeurs vers des valeurs distinctes sont égaux, mais pas le troisième [13] : c'est donc une comparaison qu'il faut veiller à éviter. L'une des conséquences est qu'il n'est pas souhaitable d'utiliser des types pointeur comme clefs de types carte.

- ✓ **Conversion** : une valeur pointeur peut être « convertie » en une valeur d'un type alias de son type courant, cette opération n'accomplissant rien.
  Elle peut également être convertie en une valeur d'un type basé sur le même type sous-jacent, ce qui correspond aux types définis basés l'un sur l'autre.
  Pour des opérations non protégées comme l'arithmétique de pointeurs, tout pointeur peut également être converti dans le type `unsafe.Pointer`, le plus souvent pour le convertir ensuite en **`uintptr`** pour affichage.
- ✓ **Déréférencement** : la principale utilisation d'un pointeur consiste à accéder à la valeur située à l'adresse mémoire désignée par la valeur numérique du pointeur : c'est l'opération de *déréférencement*. Fonctionnellement son rôle est comparable à celui de l'indexation pour un tableau ou tranche, ou de la sélection pour une structure : partir d'une donnée pour accéder à une autre.
  Déréférencer **`nil`**, la valeur vierge des pointeurs, provoque une panique d'exécution : il est donc nécessaire lors de tout déréférencement de valeur pointeur de

13. À titre d'exemple, ajouter l'affichage de `t3` au `fmt.Printf` : le troisième pointeur devient égal aux deux premiers. Inversement, supprimer l'affichage des deux premiers pointeurs : ils deviennent différents.

s'assurer que le pointeur n'est pas `nil`, soit par construction, soit en vérifiant la valeur par une comparaison du pointeur avec `nil`.
Bien que `unsafe.Pointer` soit en apparence déclaré dans le paquet `unsafe` comme un pointeur `type Pointer *ArbitraryType` (cf. *Plus en ligne*), il présente la particularité de ne pas pouvoir être déréférencé, comme le montre le listing 3.8 : décommenter la ligne 20 provoquerait une erreur de compilation.

✓ **Adresse** : l'opérateur d'adresse `&` (cf. § 2.4.8) est applicable aux pointeurs, et il est possible de déréférencer des pointeurs à plusieurs niveaux comme dans le listing 3.8, où la valeur 5 est obtenue en déréférençant en cascade le pointeur sur pointeur `pp`.

Enfin, pour initialiser un pointeur trois possibilités existent :

✓ **Adresse** : l'opérateur d'adresse appliqué à une valeur adresssable – donc en particulier pas une constante – renvoie un pointeur sur la valeur. C'est le procédé le plus courant.

✓ **Fonction `new()`** : `p := new(B)`
Dans cette initialisation rarement utilisée, la fonction intégrée `new()` renvoie un pointeur sur une valeur vierge du type `B`. Il est plus idiomatique d'écrire `p := &B{}`.

✓ **Déclaration seule** : comme pour toute variable Go, la simple déclaration d'une variable de type pointeur l'initialise à la valeur vierge `nil` du type pointeur choisi.

Exercice Concevoir un programme qui utilise l'arithmétique de pointeurs avec les types et fonctions du paquet `unsafe` pour parcourir l'implantation mémoire réelle des chaînes de caractère.

Le programme commence par allouer une chaîne `"unsafe"`, puis prend son pointeur, parcourt les données pointées, et reconstitue les éléments de la chaîne à partir de là : son pointeur interne, et sa longueur, et enfin utilise ces éléments pour parcourir les octets de la chaîne d'origine sur la bonne longueur.

Solution https://osinet.fr/go/types/play-pointeurs-arithmetique

Exercice Après avoir réalisé l'exercice précédent sur le playground ou sur un poste de développement, l'exécuter sur l'autre environnement, et comparer les résultats.

Identifier la différence (les valeurs numériques des pointeurs n'ont pas d'importance) et la raison de cette différence.

Solution https://osinet.fr/go/types/play-pointeurs-architecture

© Dunod – Toute reproduction non autorisée est un délit.

Listing 3.8 – Pointeurs multi-niveaux
https://osinet.fr/go/types/play-pointeurs-multi-niveaux

```
package main

import (
        "fmt"
        "unsafe"
)

type P *int
type PP *P

func main() {
        var i int = 5
        var p P = &i
        var pp PP = &p
        var up unsafe.Pointer = unsafe.Pointer(p)

        fmt.Printf("%#v\n%#v\n%#v\n%#v\n%#v\n",
                up, p, pp, *pp, **pp)

        // invalid indirect of up (type unsafe.Pointer)
        // fmt.Printf("%#v\n", *up)
}

// Résultat:
// (unsafe.Pointer)(0x414020)
// (main.P)(0x414020)
// (main.PP)(0x40c128)
// (main.P)(0x414020)
// 5
```

Et les références C++ ?

Dans la famille des langages inspirés par C, le C++ définit la notion de *référence*, connexe à celle de pointeur : une référence de variable est une autre variable occupant la même adresse, de sorte que les pointeurs sur une variable et sur une référence à cette variable sont égaux.

Ce concept n'existe pas en Go, et ne figure pas dans les évolutions Go 2.

Hormis pour les valeurs de longueur nulle cf. *supra*, toute variable a une adresse différente de toute autre variable.

3.3 TYPES FONCTIONS

3.3.1 Définition

Les fonctions Go sont similaires aux fonctions des autres langages usuels, avec quelques particularités :

- ✓ Elles peuvent recevoir un nombre variable de paramètres.
- ✓ Elles peuvent renvoyer plusieurs valeurs au lieu d'une seule comme dans la majorité des langages.
- ✓ Les valeurs de retour peuvent être nommées, et leurs identificateurs utilisés comme ceux des paramètres et des variables et constantes locales.
- ✓ Elles peuvent être nommées ou anonymes, et les fonctions anonymes sont des fermetures (*closure*) [14] sur leur environnement.
- ✓ Elle peuvent être appelées récursivement.
- ✓ Elles sont des objets de première classe [15], ce qui permet de définir des fonctions d'ordre supérieur (*higher order function*) sans particularités de notation. Cette capacité est en particulier utilisée dans le développement web pour la création de médiateurs (*middleware*).
- ✓ Elles ne peuvent pas avoir de paramètres nommés, ni de paramètres optionnels avec des valeurs par défaut.

Pour ce qui concerne la récursion, certains compilateurs peuvent inclure l'optimisation de la récursion terminale dans certains cas [16], mais ce n'est pas une garantie du langage.

3.3.2 Déclaration de types fonctions

Les types de fonctions sont une représentation de la signature des fonctions :

- ✓ **Paramètres** : le nombre et les types des paramètres font partie du type ; en revanche les noms des paramètres n'en font pas partie.
- ✓ **Retours** : le nombre et les types des valeurs renvoyées font partie du type ; en revanche leurs noms – ou leur caractère anonyme – n'en font pas partie.

La déclaration d'un type fonction se compose :

- ✓ du mot-clef `type` ;
- ✓ d'un identificateur qui sera le nom du type ;
- ✓ du mot-clef `func` ;

14. https://osinet.fr/go/types/wiki-fermeture
15. https://osinet.fr/go/types/wiki-premiere-classe
16. https://osinet.fr/go/types/nuts-recursion-terminale

© Dunod – Toute reproduction non autorisée est un délit.

✓ des paramètres des fonctions du type, sous la forme :
- d'une parenthèse ouvrante,
- optionnel : d'une liste de paramètres séparés par des virgules, se composant :
  - de zéro, un, ou plusieurs identificateurs de paramètres, y compris l'identificateur vierge, séparés par des virgules,
  - optionnel, pour le dernier paramètre seulement : du symbole trois-points,
  - d'un type, identifié ou non,
- d'une parenthèse fermante ;

✓ optionnel : du résultat des fonctions du type :
- soit sous la même forme qu'une liste de paramètres, hormis le cas du symbole trois-points,
- soit sous la forme de la déclaration d'un type, défini ou non.

Si cet élément est absent, la fonction ne renvoie aucun résultat.

Dans les paramètres comme dans les résultats, si un au moins des paramètres est nommé par un identificateur, y compris par l'identificateur vierge, alors tous les paramètres doivent l'être, bien que le type ne tienne pas compte de ces identificateurs : il n'est pas possible de mélanger paramètres nommés et non nommés. Ceci peut donner lieu à des situations troublantes comme celle du listing 3.9 :

Listing 3.9 – Paramètres et résultats de fonction nommés et anonymes
https://osinet.fr/go/types/play-parametres-anonymes

```
1  package main
2
3  type melange func(string, float64, bool, x int)
4
5  func fonction(string, float64, bool, x int) {
6          println(string, float64, bool, x)
7  }
8
9  func main() {
10         var f melange = fonction
11         f(1, 2, 3, 4)
12 }
13
14 // Résultat: 1 2 3 4
```

Dans cet exemple, le type `melange` semble définir des fonctions prenant trois paramètres anonymes de types **`string`**, **`float64`** et **`bool`**, et un paramètre nommé `x` de type **`int`**, en violation de la règle voulant que les identificateurs soient tous nommés ou qu'aucun ne le soit.

L'exécution montre que la situation est différente : les identificateurs des types n'étant que des identificateurs prédéfinis par le langage et non des mots-clefs, ils sont

ici surchargés par une utilisation comme noms de paramètres, et le type `melange` définit en fait des fonctions à quatre paramètres nommés de type `int`.

3.3.3 Déclaration de fonctions

Les fonctions peuvent être :

- ✓ nommées, lorsqu'elles sont définies par une déclaration ;
- ✓ anonymes, lorsqu'elles sont définies par un littéral fonction.

Les déclarations de fonction sont obligatoirement réalisées dans la portée d'un paquet : il n'est pas possible de déclarer une fonction à l'intérieur d'une autre fonction, ou de tout autre bloc.

Une déclaration de fonction se compose des éléments suivants, comme aux lignes 5-7 du listing 3.9 :

- ✓ le mot-clef `func` ;
- ✓ un identificateur qui servira de nom à la fonction ;
- ✓ une définition de signature, similaire à celles décrites à propos des types fonctions : paramètres, et résultats facultatifs ;
- ✓ optionnel : un corps de fonction, sous la forme d'un bloc de code délimité par accolades.

Dans cet exemple, la fonction comporte le corps de fonction, ce qui est représentatif de l'immense majorité des fonctions déclarées. Les déclarations qui n'en comportent pas servent à déclarer les fonctions externes, par exemple celles écrites en assembleur et rattachées au code au moment de l'édition de liens. Dans la bibliothèque standard Go, de telles fonctions externes sont essentiellement utilisées pour l'interfaçage avec les fonctions système et la définition des fonctions intégrées de l'environnement d'exécution.

3.3.4 Littéraux fonction

L'autre façon d'introduire une fonction est la définition d'un littéral fonction, qui permet de créer une fonction anonyme (*lambda*). Cette fonction anonyme peut alors être affectée à une variable de n'importe quel type fonction compatible, ou être invoquée directement. La documentation Go n'utilise pas le terme *lambda* pour les fonctions anonymes.

Un littéral fonction se compose des éléments suivants :

- ✓ le mot-clef `func` ;
- ✓ une définition de signature, similaire à celles décrites à propos des types fonctions : paramètres, et résultats facultatifs ;
- ✓ un corps de fonction, sous la forme d'un bloc de code délimité par accolades.

© Dunod – Toute reproduction non autorisée est un délit.

À la différence des fonctions déclarées, l'identificateur ne peut pas être présent, et le corps de fonction n'est pas optionnel.

Les littéraux fonctions ne sont pas limités à la portée paquet comme les fonctions déclarées.

Les fonctions anonymes ainsi définies sont des fermetures (*closure*) sur leur environnement lexical : elles peuvent faire référence à tous les symboles visibles dans la portée où elles sont définies. Lorsqu'une telle référence existe dans une fonction, la variable concernée est partagée entre le contexte externe et le littéral fonction, de sorte que la suppression des références à la donnée par le contexte externe ne suffit pas à la rendre libérable pour le ramasse-miettes tant que la fonction anonyme existe aussi.

Le fait que les fonctions anonymes ne puissent pas porter un identificateur optionnel comme en ECMAScript introduit une contrainte sur les fonctions récursives. En effet, il est possible en ECMAScript d'écrire une fonction anonyme auto-référentielle comme dans le listing 3.10.

Listing 3.10 – Littéraux fonction et récursion en ECMAScript

```
const f = function fact(n) {
        if (n > 1) {
                return n * fact(n - 1);
        }
        return 1;
};
console.log(f(5));
```

Dans cet exemple, la fonction anonyme ECMAScript s'invoque récursivement, alors que la fonction anonyme Go équivalente ne dispose pas d'un identificateur par lequel s'appeler ; il est donc nécessaire de passer par deux instructions séparées comme dans le listing 3.11 : la première pour déclarer une variable fonction anonyme, la seconde pour définir par un littéral la fonction anonyme récursive utilisant cette variable et l'assigner à la variable.

3.3.5 Code des fonctions

Le code d'une fonction présente trois principales particularités par rapport à celui d'un bloc de code quelconque : le traitement des paramètres, celui des résultats, et le code terminal.

Listing 3.11 – Littéraux fonctions et récursion en Go
https://osinet.fr/go/types/play-litteraux-fonctions

```
1  package main
2
3  func main() {
4          var fact func(int) int
5          fact = func(n int) int {
6                  if n > 1 {
7                          return n * fact(n-1)
8                  }
9                  return 1
10         }
11         println(fact(5))
12 }
```

Pour ce qui est des paramètres, peu de particularités :

- ✓ Tous les arguments d'une fonction lui sont passés par valeur. Du fait du passage par valeur, à la sortie de la fonction, la valeur de tous les arguments est inchangée, même si elle a été modifiée dans le code de la fonction ; si cette valeur est un pointeur, ou une structure contenant des pointeurs, comme dans l'exemple des tranches au listing 4.8 page 121, ceci n'empêche pas la fonction de modifier la donnée pointée : le passage par valeur n'est pas une garantie d'immuabilité, le passage par valeur ne réalisant qu'une copie creuse (*shallow copy*).
- ✓ Une fonction est *variadique* si son dernier paramètre est de la forme `...T`.
  En pareil cas elle peut recevoir, outre les paramètres fixes définis avant ce dernier paramètre, autant de paramètre que l'appelant le souhaite, dès lors que leurs valeurs sont affectables (cf. § 3.6.2) au type `T`. En particulier, si son dernier paramètre est de type `...interface{}`, elle peut recevoir un nombre quelconque de paramètres de types arbitraires.
  Si ce paramètre est nommé – ce qui est le seul moyen d'y accéder – l'identificateur associé est celui d'un paramètre du type `[]T`, que la fonction peut alors exploiter comme n'importe quelle autre variable tranche.
  L'exemple canonique de ce type de fonctions est `fmt.Printf` [17], qui accepte un premier paramètre format contenant assez d'information pour définir les types de tous les arguments attendus, et un second paramètre variadique recevant les arguments arbitraires à afficher.

La définition des résultats, renvoyée par **`return`** (cf. § 7.4.1), est plus particulière, comme démontré dans le listing 3.12.

17. https://osinet.fr/go/types/go-printf

© Dunod – Toute reproduction non autorisée est un délit.

Listing 3.12 – Les quatre formes de retour de fonction
https://osinet.fr/go/types/play-retours

```
package main

import "fmt"

// Retour simple défini par un type.
func simple() string {
        return "11 Rue Paul Bert"
}

// Retours multiples définis par des types.
func multiples() (int, string) {
        return 92240, "Malakoff"
}

// Retours multiples nommés avec retour implicite.
func nommés() (rue string, cp int, ville string) {
        rue = "11 Paul Bert"
        cp = 92240
        ville = "Malakoff"
        return
}

// Retours multiples nommés avec retour explicite.
func renommés() (rue string, cp int, ville string) {
        rue = "11 Paul Bert"
        cp = 92240
        ville = "Malakoff"
        return "21 Montparnasse", 75006, "Paris"
}

func main() {
        fmt.Println(simple())     // 11 Paul Bert
        fmt.Println(multiples()) // 92240 Malakoff
        fmt.Println(nommés())     // 11 Paul Bert 92240 Malakoff
        fmt.Println(renommés())   // 21 Montparnasse 75006 Paris
}
```

- ✓ **Retour simple** : la première forme de retour correspond aux langages usuels dérivés de C : pour une fonction définie comme `func (/* ... */) T`, le retour est déclenché par une instruction `return t`, dans laquelle `t` est de type `T`.
  C'est le cas de la fonction `simple()` de l'exemple.
- ✓ **Retours multiples anonymes** : une fonction peut renvoyer plusieurs résultats, et déclare dans ce cas une liste parenthésée de types de ses résultats ; l'instruction `return` est suivie des valeurs à renvoyer.
  C'est le cas de la fonction `multiple()` de l'exemple.

- ✓ **Retours nommés implicites** : outre les définitions de résultats par le simple type, il est également possible de fournir des identificateurs pour la liste des valeurs renvoyées, de manière similaire à la liste des paramètres à l'exception du mécanisme variadique.
  Dans cette forme de définition, les résultats sont alloués et initialisés comme des variables locales avant la première ligne de code de la fonction. Celle-ci peut ensuite manipuler leurs valeurs librement, et renvoyer ses résultats par l'instruction **`return`** sans liste de résultats : la fonction renvoie alors l'état courant des variables de retour ainsi déclarées au moment de l'exécution de **`return`**.
  L'exécution de la fonction doit se terminer par un **`return`** mais, en particulier par le jeu des boucles et **`goto`**, celui-ci n'est pas nécessairement la dernière instruction du code source.
  C'est le cas de la fonction `nommés()` de l'exemple.
- ✓ **Retours nommés explicites** : même lorsque le format des résultats nommés est utilisé, ceux-ci peuvent être surchargés par un **`return`** comportant une liste de résultats.
  Cela n'affecte en rien la manipulation des résultats dans la fonction, mais ils sont remplacés lors du retour par ceux passés explicitement dans l'instruction **`return`**.
  C'est le cas de la fonction `renommés()` de l'exemple.

Le mécanisme des retours nommés explicites est une bonne pratique : il permet de limiter la documentation en documentant directement la signification des valeurs renvoyées par une fonction, pour peu que les identificateurs soient bien choisis. Les retours nommés implicites, bien qu'ils présentent le même avantage, obligent le lecteur du code à reconstituer mentalement l'état des variables au moment du retour : ils sont donc moins lisibles.

Exercice L'identificateur vierge est-il utilisable dans les listes des résultats nommés ? Rédiger un exemple de fonction avec retours nommés implicites dans laquelle l'un des résultats est nommé par l'identificateur vierge. D'où provient la valeur renvoyée par le résultat associé ?

Solution https://osinet.fr/go/types/play-retour-vierge

La particularité la plus originale du code des fonctions est le mécanisme de code terminal, introduit dans Go par le mot clef **`defer`**.

Par ce mécanisme, un bloc de code peut être exécuté lors du retour d'une fonction, sans avoir besoin d'être invoqué explicitement, ce qui est particulièrement commode pour déclencher automatiquement les libérations de ressources allouées dans la fonction, par exemple les connexions de base de données ou les descripteurs de fichiers.

Du fait de ses implications complexes, il fait l'objet de sa propre documentation au § 7.4.3.

© Dunod – Toute reproduction non autorisée est un délit.

3.3.6 Manipulation des fonctions

Les expressions utilisant des opérandes fonctions sont les suivantes :

✓ **Comparaison** : les fonctions ne peuvent être comparées que pour l'égalité ou inégalité à `nil`. Cette comparaison n'est utile que pour les fonctions anonymes, toutes les fonctions nommées étant non-`nil` par le mécanisme même de leur déclaration littérale : bien que la comparaison soit possible pour les fonctions nommées, elle est relevée comme une anomalie par l'outil d'analyse de code `go vet` [18]. C'est ce que montre le listing 3.13.

Listing 3.13 – Comparaison de fonctions et pointeurs
https://osinet.fr/go/types/play-comparaison-fonctions

```
package main

import "fmt"

type TF func() string

func nommée() string {
        return "Bonjour"
}

func main() {
        var anonyme = nommée
        var pointeur = &anonyme

        fmt.Printf(`
Nommée:       %T
Anonyme:      %T
*Pointeur:    %T
Non-nil:      %t %t
Pointeurs:    %t
Appel:        %s
`,
                nommée, anonyme, *pointeur,
                nommée != nil, anonyme != nil,
                pointeur == &anonyme,
                (*pointeur)(),
        )
}
```

✓ **Conversion** : une valeur fonction peut être « convertie » en une valeur d'un type alias de son type courant, cette opération n'accomplissant rien.

18. https://osinet.fr/go/types/go-vet

Elle peut également être convertie en une valeur d'un type basé sur le même type sous-jacent, ce qui correspond aux types définis basés l'un sur l'autre ou aux types non liés mais portant la même signature, ignorant les noms des paramètres et résultats.
La conversion est examinée dans le cas général au § 6.12.

✓ **Adresse** : l'opérateur d'adresse `&` (cf. § 2.4.8) est applicable aux fonctions, comme illustré dans le listing 3.13.
Intérieurement, une valeur fonction est représentée comme un pointeur, mais cette information demeure opaque et ne fait pas partie de la spécification du langage. L'opérateur adresse appliqué à une valeur fonction renvoie un pointeur sur ce pointeur, et l'opacité va jusqu'à empêcher la conversion directe d'une valeur fonction vers un pointeur générique `unsafe.Pointer` (cf. *Plus en ligne*).

✓ **Appel** : l'appel (invocation) d'une fonction demeure son utilisation la plus fréquente. Sa principale particularité en Go est la capacité de renvoyer des résultats multiples, qui est généralement combinée avec la capacité d'effectuer des affectations parallèles pour définir les valeurs de plusieurs variables, sans nécessiter le passage par des structures intermédiaires comme dans le plupart des autres langages comme dans les exemples du tableau 3.4.

Tableau 3.4 – Affectation parallèle de retours multiples

Langage	Définition d'une lambda	Résultats multiples
Go	`f := func(c int) (int, int) { return c, c*c }`	`a, b := f(2)`
PHP	`$f = function($c) { return [$c, $c*$c]; };`	`[$a, $b] = $f(2);`
ES6	`f = c => [c, c*c];`	`[a, b] = f(2);`
Python	`f = lambda c: [c, c*c]`	`(a, b) = f(2)`

3.3.7 Chaînage de retours multiples

L'affectation parallèle de retours multiples comme dans l'exemple précédent permet de traiter en une seule instruction l'affectation de plusieurs résultats de fonction à des variables indépendantes.

Cette capacité d'utilisation parallèle est plus limitée lorsqu'il s'agit de passer directement des retours multiples à une autre fonction acceptant des paramètres multiples, au lieu de devoir passer par des variables intermédiaires.

Le listing 3.14 présente les trois principales façons de réaliser ce chaînage, à partir des deux résultats renvoyés par une fonction `ops`.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 3.14 – Chaînage de fonction à retours multiples
https://osinet.fr/go/types/play-chainage-fonctions

```
1  package main
2
3  import "fmt"
4
5  // Exemple de fonction renvoyant 2 valeurs.
6  func ops(a, b int) (int, int) {
7          return a + b, a - b
8  }
9
10 // Conversion de valeurs multiples en une seule tranche.
11 func tranche(args ...int) []int {
12         return args
13 }
14
15 // Fonction non variadique avec le bon nombre de paramètres.
16 func accepte1(v1, v2 int) {
17         fmt.Printf("%#v\n", []int{v1, v2})
18 }
19
20 // Fonction variadique à un seul paramètre.
21 func accepte2(valeurs ...int) {
22         fmt.Printf("%#v\n", valeurs)
23 }
24
25 // Fonction non variadique acceptant une tranche.
26 func accepte3(format string, values []int) {
27         fmt.Printf(format, values)
28 }
29
30 func main() {
31         accepte1(ops(7, 2))
32         accepte2(ops(7, 2))
33         accepte3("%#v\n", tranche(ops(7, 2)))
34 }
35
36 // Résultat: []int{9, 5} dans les 3 cas
```

Le cas le plus naturel est celui qui consiste à passer les retours multiples à une fonction acceptant le nombre exact de paramètres de la première fonction. C'est le cas de la la fonction `accepte1`, lignes 15-18 et 31.

Un cas presque similaire quoique un peu plus limité est celui d'une fonction variadique à un seul paramètre variadique, comme la fonction `accepte2`, lignes 20-23 et 32. L'invocation est tout aussi naturelle que dans le premier cas, mais ce modèle

est limité par le fait que les valeurs multiples doivent toutes être du type du paramètre variadique ; si cette limitation est contournée en déclarant un paramètre variadique de la forme `...interface{}`, la fonction devient limitée par le fait qu'elle ne dispose plus d'aucune information de type sur les paramètres.

En pareil cas, la solution commune, inspirée de `fmt.Printf`, consiste à passer un premier paramètre décrivant les divers types dynamiques des paramètres suivants. Mais en pareil cas, la syntaxe Go bloque, ne réalisant pas automatiquement l'ajout de la liste de résultats multiples – une valeur de type `Tuple` dans le runtime Go [19] (*n-uplet* en français) – comme des valeurs additionnelles à la liste de paramètres, provoquant une erreur `multiple-value ops() in single-value context`.

La solution consiste, comme dans la fonction `accepte3` lignes 25-28 et 33, à passer par une fonction intermédiaire de la deuxième forme ci-dessus, dont le seul rôle est de convertir les paramètres multiples en une tranche du type attendu par la fonction chaînée devant recevoir les dites valeurs multiples. C'est ce que réalise la fonction `tranche` lignes 10-13 et 33.

3.4 MÉTHODES D'UN TYPE

3.4.1 Définition

Les méthodes Go sont des fonctions, avec un argument supplémentaire noté différemment des autres. Il n'existe donc pas de type méthode spécifique ni de littéraux méthodes. Elles présentent quatre principales différences fonctionnelles par rapport aux langages ECMAScript, Java, PHP ou Python :

- ✓ Elles peuvent être créées sur tout type défini (cf. § 5.3) autre que les types pointeurs et interfaces, et non seulement sur les types `struct`, bien que ce soit l'usage le plus traditionnel.
  Il n'est pas possible de *définir* des méthodes sur des types non définis, que ce soient des types intégrés ou des littéraux types, y compris des identificateurs d'alias de type. Il est toutefois possible de *déclarer* des méthodes sur des types interface non définis, ceux-ci n'en portant pas l'implémentation.
- ✓ L'instance d'objet sur lequel la méthode est définie, dite le *récepteur* [20] ne porte pas de nom réservé comme `self`, `this`, ou `$this`, mais est défini explicitement avec un nom librement choisi par le développeur.
- ✓ Cette instance n'est pas passée obligatoirement par pointeur comme le sont habituellement les récepteurs `self` ou `($)this`, mais peut également être passée par valeur. Si la méthode modifie la valeur du récepteur, les modifications sont

19. Voir `$GOROOT/src/go/types/type.go`.
20. Les articles français sur Go utilisent également le mot *receveur*, plus proche du terme utilisé en anglais, le *receiver*.

© Dunod – Toute reproduction non autorisée est un délit.

ignorées au retour de la méthode, dans les limites liées au passage d'arguments par valeur comme détaillé au § 3.3.5.
C'est ce que montre l'exemple de la méthode `Valeur` lignes 15-18, 32-33, et 37 du listing 3.15 : dans la méthode, le récepteur `o` vaut 2, mais au retour de la méthode, il retrouve bien sa valeur 1.

✓ Les méthodes portent toujours sur une instance du type, et non sur le type lui-même : il n'existe pas de méthodes statiques, l'équivalent en Go étant de déclarer des fonctions nommées dans le même paquet que le type.

3.4.2 Déclaration de méthodes

Les méthodes étant pour l'essentiel des fonctions nommées, elles sont définies dans la portée paquet, avec une syntaxe similaire aux fonctions, plutôt que d'être visuellement incluses dans la définition des structures, ce qui ne serait pas toujours possible puisque les méthodes peuvent être définies sur des types non-structure.

Une déclaration de méthode sur un type `T` se compose des éléments suivants, comme dans l'exemple du listing 3.15 :

✓ le mot-clef **`func`** ;

✓ une liste de paramètres à un seul élément, le récepteur, qui peut être nommé ou non, y compris par l'identificateur vierge ;

✓ un identificateur qui servira de nom à la méthode, qui doit être non vierge, différent pour chaque méthode, et ne peut être l'identificateur d'un champ pour les types structure ;

✓ une définition de signature, similaire à celles décrites à propos des types fonctions : paramètres, et résultats facultatifs ;

✓ optionnel : un corps de méthode, sous la forme d'un bloc de code délimité par accolades. S'il est absent, il sera rattaché au moment de l'édition de liens.

Cette notation à récepteur explicite, très différente des autres langages à objets usuels, provient d'Oberon-2.

3.4.3 Jeu de méthodes d'un type

Comme décrit à propos des types, un type définit entre autres le jeu de méthodes disponibles sur ses variables ; et comme mentionné au § 3.4.1, un jeu de méthodes peut être défini et implémenté sur tout type autre que pointeur et interface.

✓ Dans le cas des interfaces, décrites au § 3.5, celles-ci définissent seulement un « contrat » correspondant à un jeu de méthodes à implémenter, et non une implémentation. Le jeu de méthodes d'un type interface est donc l'interface elle-même.

✓ Pour les autres types, le jeu de méthodes du type `T` est composé de toutes les méthodes définies avec un récepteur de type `T`.

Listing 3.15 – Déclaration de méthodes
https://osinet.fr/go/types/play-methodes

```
package main

import "fmt"

type octet byte

func (octet) Anonyme() string {
        return "0"
}

func (_ octet) Vierge() string {
        return "0"
}

func (o octet) Valeur() string {
        o = 2
        return fmt.Sprint(o)
}

func (o *octet) Pointeur() string {
        *o = 3
        return fmt.Sprint(*o)
}

// Externe pourrait être réalisée en assembleur.
// func (o octet) Externe();

func main() {
        var un octet = 1
        initial := un
        an, vi := un.Anonyme(), un.Vierge()
        va := un.Valeur()
        aprèsValeur := un
        po := un.Pointeur()
        aprèsPointeur := un
        fmt.Println(initial, an, vi,
                va, aprèsValeur,
                po, aprèsPointeur)
}

// Résultat: 1 0 0 2 1 3 3
```

© Dunod – Toute reproduction non autorisée est un délit.

✓ Cas particulier pour les types pointeurs, le jeu de méthodes d'un type pointeur `*T` est composé de deux parties :
  - les méthodes définies avec un récepteur de type `*T`,
  - mais aussi les méthodes définies avec un récepteur de type `T`.

✓ Dans le cas des types structure comportant des champs intégrés comme décrits au § 4.4.3, leurs méthodes sont promues sur le type structure qui les porte, de façon transparente.
  La dualité entre les méthodes à receveur `T`/`*T` se retrouve dans ce cas. Le tableau 3.5 détaille, selon que le champ intégré est de type `T` ou de son pointeur `*T`, les méthodes du jeu de méthodes du type intégré qui sont ajoutées au jeu de méthodes du receveur du type `Porteur`.
  Comme le montre ce tableau, ce n'est que lors de l'utilisation d'une variable du type `Porteur` avec un champ intégré du type `T` que les méthodes avec un receveur de type `*T` ne sont pas disponibles : dans tous les autres cas, l'utilisation de la méthode est directement possible par la promotion du champ intégré.

Tableau 3.5 – Types de récepteurs en présence de types intégrés

Champ intégré	Méthodes de `Porteur`		Méthodes de `*Porteur`	
`T`	`T`		`T`	`*T`
`*T`	`T`	`*T`	`T`	`*T`

Les méthodes des champs intégrés à un type structure font également partie du jeu de méthodes de celui-ci, mais leur récepteur est bien du type défini pour la méthode, et non du type de la variable du type extérieur, comme dans le listing 3.16.

Ceci est une conséquence du fait que la notation des types intégrés n'est qu'un *sucre syntaxique* (*syntaxic sugar*) pour la notation plus complète dans laquelle le champ intégré est inséré dans l'appel de méthode : dans cet exemple, `e.Type()` est compilé et exécuté comme `e.Message.Type()`, ce qui explique que le type imprimé par la méthode `Type` du type `Enveloppe` soit bien `Message`.

3.4.4 Méthodes et champs fonctions

Pour une instance `v` d'un type structure `T`, la notation `v.foo(a, b)` peut donner lieu à deux interprétations différentes par le compilateur :

✓ **Méthode** : `foo` peut être une méthode avec récepteur de type `T` ou `*T` et deux arguments visibles, qui est compilée comme une fonction recevant à l'exécution trois arguments : le récepteur `v` et les deux arguments visibles.

✓ **Champ fonction** : `foo` peut être un champ d'un type fonction acceptant deux arguments seulement : les deux arguments explicites, mais pas le récepteur `v`.

Listing 3.16 – Méthodes de types intégrés
https://osinet.fr/go/types/play-methodes-integres

```
package main

import "fmt"

type Message string

type Enveloppe struct{ Message }

func (m Message) Type() { fmt.Printf("%T\t", m) }

func main() {
        e := Enveloppe{Message: "Bonjour"}
        e.Type()
        e.Message.Type()
}

// Résultat: main.Message main.Message
```

L'ambiguïté n'existe pas pour les types autres que les structures, ces types étant les seuls à pouvoir porter des champs ; pour tout autre type, la première interprétation est donc la seule possible. Pour les structure, c'est la définition du type `T` qui résout l'ambiguïté : un même type ne peut pas avoir de champ homonyme d'une méthode sans provoquer une erreur de compilation.

La technique des champs fonction est principalement utilisée pour l'invocation dynamique, examinée au § 3.4.7. Le paquet `sync` des primitives de concurrence l'utilise pour le type `sync.Pool`.

3.4.5 Utilité des méthodes sur les types non-structure

Pour les développeurs venant de langages à objets et classes traditionnels, l'intérêt de pouvoir définir des méthodes sur des types autres que les **`struct`** n'est pas évident : à quels messages un objet sans état interne structuré pourrait-il bien répondre ?

La bibliothèque standard fournit un exemple canonique en la matière : le type fonction `http.HandlerFunc`, en relation avec le type interface `http.Handler`, dont l'extrait pertinent est donné dans le listing 3.17.

Sans entrer dans les détails des types interface, qui font l'objet du § 3.5, le type `http.Handler` définit une interface composée d'une seule méthode, `ServeHTTP`. Sa vocation est d'être le type d'interface commun utilisé par les contrôleurs web.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 3.17 – Sources de `Handler` et `HandlerFunc`
https://osinet.fr/go/types/go-handler-source

```
3  // Extraits de la bibliothèque standard Go:
4  // $GOROOT/src/net/http/server.go
5
6  // [...]
7  type Handler interface {
8          ServeHTTP(ResponseWriter, *Request)
9  }
10 // [...]
11
12 type HandlerFunc func(ResponseWriter, *Request)
13
14 // ServeHTTP calls f(w, r).
15 func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
16         f(w, r)
17 }
18
19 // [...]
```

Le type `http.HandlerFunc`, lui, est un type fonction. Sa vocation est d'être le type commun utilisé par les contrôleurs web simples, qui n'ont pas besoin d'embarquer des données additionnelles en complément de la requête et de la destination de réponse.

Sa signature est visiblement la même que celle de l'interface `http.Handler`. Imaginons une fonction nommée affectable à ce type, comme la fonction `Contrôleur` dans le listing 3.18.

Cette fonction présente bien la signature attendue mais, implémentant directement la signature souhaitée au lieu de porter une méthode implémentant cette signature, elle n'implémente pas l'interface `http.Handler`.

C'est ici qu'intervient la définition de méthodes sur un type quelconque : comme le montre le listing 3.17, le type `http.HandlerFunc` n'est pas défini seulement par la signature en question, mais aussi par un jeu de méthodes composé en tout et pour tout de la méthode `ServeHTTP` avec ladite signature ; il implémente donc l'interface `http.Handler`.

Dès lors, il est possible de convertir `Contrôleur` au type `http.HandlerFunc` comme la fonction `ctrl` à la ligne 15 du listing 3.18 : la signature est la même. Et à l'issue de cette conversion, `ctrl` est bien une fonction portant la méthode `ServeHTTP` souhaitée.

Listing 3.18 – Utilisation de `Handler` et `HandlerFunc`
https://osinet.fr/go/types/play-handler-usage

```
1  package main
2
3  import (
4          "fmt"
5          "net/http"
6  )
7
8  // Simple fonction, pas de méthode, pas un HandlerFunc.
9  func Contrôleur(w http.ResponseWriter, r *http.Request) {
10         /* ...logique du contrôleur...*/
11 }
12
13 func main() {
14         // ctrl est de type HandlerFunc
15         var ctrl http.Handler = http.HandlerFunc(Contrôleur)
16
17         // ctrl implémente-t-il l'interface Handler ?
18         _, estUnHandler := ctrl.(http.Handler)
19
20         // Type statique: HandlerFunc, implémente Handler.
21         fmt.Printf("%T %t\n", ctrl, estUnHandler)
22 }
23
24 // Résultat: http.HandlerFunc true
```

Il ne reste plus à l'implémentation de cette méthode dans `http.HandlerFunc` qu'à invoquer la fonction récepteur elle-même, avec les mêmes arguments, pour être une implémentation parfaite de l'interface `http.Handler`. C'est ce que réalise la ligne 16 dans le listing 3.17.

Ce mécanisme permet lors de l'écriture de services web d'utiliser de simples fonctions comme contrôleurs au lieu d'avoir à créer des types structure uniquement pour respecter l'interface `http.Handler`.

Il est adaptable à tout autre type et toute autre signature, la version la plus répandue consistant à ajouter sur des types définis la méthode :

```
func (t UnTypeDéfini) String() string
```

Cet ajout leur fait implémenter l'interface `fmt.Stringer`, équivalent plus explicite d'un mécanisme comme la méthode magique `__toString()` en PHP, ou la méthode intégrée `Object.prototype.toString` en ECMAScript [21].

21. Pour un exemple d'interfaces multiples, voir le blogue de Matt Butcher : https://osinet.fr/go/types/butcher2014-methodes-de-fonctions

© Dunod – Toute reproduction non autorisée est un délit.

Exercice Retrouver ce mécanisme à l'œuvre dans le type `syscall.Errno`, et imaginer son utilisation pour un traitement fin des erreurs.

Solution https://osinet.fr/go/types/cheney-traitement-erreurs

3.4.6 Récepteur pointeur ou récepteur valeur

De la même façon que tout paramètre, le récepteur d'une méthode d'un type `T` peut être passé par valeur ou par adresse, associant respectivement la méthode au jeu de méthodes de `T` ou de `*T` :

- ✓ **Par valeur** : `func (récepteur T) Méthode()`
- ✓ **Par adresse** : `func (récepteur *T) Méthode()`

En revanche, alors que la distinction est clairement visible lors de la déclaration de la méthode, elle n'est pas nécessairement apparente à l'appel de la méthode : Go convertit en effet automatiquement la variable utilisée pour l'appel entre valeur et pointeur dans la mesure du possible, comme le montre le listing 3.19, selon les règles suivantes :

- ✓ **Récepteur valeur, instance valeur** : dans le premier cas, la méthode attend et reçoit une instance valeur : aucune réécriture n'est nécessaire.
- ✓ **Récepteur pointeur, instance pointeur** : dans le deuxième cas, la méthode attend et reçoit une instance pointeur : aucune réécriture n'est nécessaire.
- ✓ **Récepteur valeur, instance pointeur** : dans le troisième cas, la méthode attend une instance valeur mais est invoquée avec un argument pointeur : Go déréférence automatiquement le pointeur pour obtenir la valeur pointée et la passe comme valeur à la méthode.
- ✓ **Récepteur pointeur, instance valeur adressable** : dans le quatrième cas, la méthode attend une instance pointeur mais est invoquée avec un argument valeur : la valeur étant adressable, Go prend automatiquement son adresse et la passe comme pointeur à la méthode.
- ✓ **Récepteur pointeur, instance valeur non adressable** : dans le cinquième cas, la méthode attend une instance pointeur mais est invoquée avec un argument valeur constante : les constantes n'étant pas adressables (cf. § 5.2.1), Go émet une erreur à la compilation.

Lors de la définition de types portant des méthodes se pose donc la question du choix du type de récepteur : valeur ou adresse ?

Le passage par valeur est tentant de prime abord, par la garantie que les éventuelles modifications apportées par la méthode sur la donnée passée seront dépilées au retour de la fonction, mais puisque la valeur copiée n'est qu'une copie creuse, la portée de cette « sécurité » est des plus limitées, concernant essentiellement les types numériques et chaînes.

Listing 3.19 – Récepteurs valeur et pointeur
https://osinet.fr/go/types/play-recepteur

```
1  package main
2
3  import "fmt"
4
5  type Octet byte
6
7  func (t Octet) parValeur() Octet { return t }
8
9  func (t *Octet) parPointeur() Octet { return *t }
10
11 func main() {
12         const c Octet = 5
13         var t Octet = c
14
15         fmt.Println(
16                 // Cas normaux.
17                 t.parValeur(), (&t).parPointeur(),
18                 // Réécritures automatiques.
19                 (&t).parValeur(), t.parPointeur(),
20                 // Réécriture impossible.
21                 // c.parPointeur()
22         )
23 }
24
25 // Résultat: 5 5 5 5
```

Par ailleurs, le passage par valeur nécessite une copie creuse de l'argument sur la pile lors de l'appel, et dès lors que la taille de l'instance devient significative, le coût de cette copie devient plus important que celui du passage du pointeur, Go ne pouvant faire appel à une optimisation *copy on write* du fait de son fonctionnement intrinsèquement concurrent [22].

Au contraire, le passage par adresse est plus généralement recommandé :

- ✓ Il est utilisable aussi bien pour les fonctions modifiant le récepteur que pour celles qui ne le modifient pas.
- ✓ Du fait des réécritures automatiques décrites ci-dessus, l'utilisation d'un récepteur pointeur n'aura souvent aucun impact sur l'écriture du code utilisant la méthode.
- ✓ Il reproduit le schéma du récepteur prédéfini (`this`, etc) avec lequel la plupart des développeurs venus d'autres langages sont familiers.

22. https://osinet.fr/go/types/nuts-passage-arguments

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ Certains types très utilisés comme `sync.Mutex` [23] ne doivent pas être copiés sous peine de perdre leurs garanties liées à la programmation concurrente, et il impliquent de ce fait l'utilisation du récepteur pointeur, puisqu'ils seraient copiés sur la pile dans le cas de l'appel par valeur. Cet effet est démultiplié par l'utilisation de champs intégrés qui peuvent eux-mêmes comporter de tels types non copiables.
  De ce fait, nombre de types auront au moins une méthode portant un récepteur pointeur.
- ✓ Les bonnes pratiques Go recommandent que toutes les méthodes d'un type portent un récepteur du même type : soit valeur, soit adresse ; et accessoirement que le récepteur porte toujours le même identificateur, pour simplifier la lecture.

3.4.7 Invocation dynamique

Comme les méthodes Go sont de simples fonctions, dont l'invocation ne passe pas par une table de résolution de méthodes virtuelles comme dans les langages à objets les plus courants [24], aucune surcharge ou invocation dynamique (*dynamic dispatch*) dans laquelle la méthode concrète invoquée serait choisie à l'exécution n'est disponible sur les types concrets.

Les deux paragraphes suivants vont illustrer les principales approches pour un type `Dynamique` ayant besoin d'une méthode dynamique pour effectuer un traitement variable choisi à l'exécution. Dans cet exemple le traitement consiste à imprimer une valeur en chaîne de caractères par deux méthodes différentes selon que la valeur est entière ou flottante.

Ces deux approches ne s'appliquent qu'à des types structure.

3.4.7.1 Utilisation de champs intégrés interface

La première approche consiste à intégrer au type `Dynamique` un champ de type interface dont l'interface est précisément la méthode à invoquer dynamiquement, comme dans le listing 3.20.

Pour cela, le type intègre un champ interface `Affichable`, et les valeurs du type `Dynamique` initialisent ce champ avec des types concrets différents, dont chacun comporte donc une implémentation concrète différente.

À l'exécution, la méthode exécutée est résolue comme la méthode concrète du type dynamique du champ, `Entier` ou `Flottant` selon la valeur.

23. https://osinet.fr/go/types/go-mutex
24. https://osinet.fr/go/types/wiki-vmt

Listing 3.20 – Invocation dynamique par interface intégrée
https://osinet.fr/go/types/play-appel-dynamique-interface

```
package main

import (
        "fmt"
        "strconv"
)

type Affichable interface {
        Affiche() string
}

// Dynamique n'implémente pas Affichable
// directement, mais par un champ intégré.
type Dynamique struct{ Affichable }

type Entier int // Implémente Affichable.

func (e Entier) Affiche() string {
        return strconv.Itoa(int(e))
}

type Flottant float32 // Implémente Affichable.

func (f Flottant) Affiche() string {
        return fmt.Sprint(f)
}

func main() {
        var e Entier = 2
        var f Flottant = 3.14

        d := Dynamique{Affichable: e}
        fmt.Println(d.Affiche())

        d.Affichable = f
        fmt.Println(d.Affiche())
}

// Résultat: 2 3.14.
```

© Dunod – Toute reproduction non autorisée est un délit.

3.4.7.2 Utilisation de champs fonction

Une autre approche consiste à ne pas utiliser des méthodes au sens strict, mais des champs fonction, en suivant l'inspiration du patron de conception *Stratégie* [25], comme dans le listing 3.21.

Listing 3.21 – Invocation dynamique par champ fonction
https://osinet.fr/go/types/play-appel-dynamique-champ

```
package main

import (
        "fmt"
        "strconv"
)

type Stratégie func(interface{}) string

// Dynamique n'implémente pas Affiche
// directement, mais par un champ fonction.
type Dynamique struct {
        Affiche Stratégie
}

func AfficheEntier(e interface{}) string {
        return strconv.Itoa(int(e.(int)))
}

func AfficheFlottant(f interface{}) string {
        return fmt.Sprint(f.(float32))
}

func main() {
        d := Dynamique{}
        var e int = 2
        var f float32 = 3.14

        d.Affiche = AfficheEntier
        fmt.Println(d.Affiche(e))

        d.Affiche = AfficheFlottant
        fmt.Println(d.Affiche(f))
}

// Résultat: 2 3.14.
```

25. https://osinet.fr/go/types/wiki-strategie

Pour cela le type `Dynamique` est doté d'un champ de type fonction implémentant la stratégie choisie à l'exécution, comme dans le listing 3.20.

À l'exécution, la « méthode » exécutée est résolue comme la valeur courante du champ de stratégie. Au contraire d'une véritable méthode, la fonction stratégie n'a pas de récepteur, et doit donc recevoir la valeur de l'instance sur laquelle elle est invoquée comme un paramètre explicite traditionnel.

L'exemple du listing 3.21 est évidemment très largement simplifié par rapport à une utilisation réelle, dans laquelle les stratégies seraient probablement regroupées dans une carte interne à l'instance du type `Dynamique` permettant de calculer la stratégie à appliquer en fonction des données manipulées : en l'état de l'exemple, il serait plus simple d'invoquer directement les fonctions `Affiche*` sans les associer au type `Dynamique`, dont aucune donnée n'est en fait utilisée.

Exercice Vérifier que les méthodes sont bien de simples fonctions avec un premier argument égal au récepteur valeur ou adresse :

- ✓ créer un type avec une méthode à receveur valeur et une à receveur adresse ;
- ✓ créer deux fonctions anonymes initialisées respectivement par l'une et l'autre méthodes ;
- ✓ vérifier le bon fonctionnement en invoquant ces deux fonctions, l'une sur une instance du type créé et l'autre sur le pointeur vers cette instance.

Solution

- ✓ Exemple simple : https://osinet.fr/go/types/play-methodes-fonctions
- ✓ Exemple dans la bibliothèque standard : `setupFd` dans la méthode `exec.Cmd.start()`, dans le fichier `src/os/exec/exec.go` (indice : que sont les valeurs `stdin`, `stdout` et `stderr` ?).

3.5 TYPES INTERFACES

3.5.1 Définition

Le terme *interface* en Go est utilisé de plusieurs façons toutes liées à la notion d'un contrat de fonctionnalité. L'usage le plus basique est comme synonyme de « jeu de méthodes » (cf. § 3.4.3).

© Dunod – Toute reproduction non autorisée est un délit.

En effet, à la différence des interfaces dans d'autres langages, les interfaces sont de pures listes de méthodes [26] :

✓ pas de constantes comme dans les interfaces PHP [27] ;

✓ pas de champs comme dans les interfaces TypeScript [28] ou C# [29] ;

✓ pas d'implémentations partielles.

Les types interfaces sont des types définis, caractérisés par leur jeu de méthodes, également dénommé *interface*. Ce sont, avec les pointeurs, les seuls types définis à ne pas pouvoir porter de méthodes, puisque leur rôle est précisément de spécifier un contrat sous la forme d'une liste de méthodes, et non d'en fournir une implémentation à la façon d'une classe de base abstraite, ce qui serait inapplicable à un langage excluant le mécanisme d'héritage.

Particularité de Go : la satisfaction des interfaces est implicite : cela signifie que tout type comportant l'ensemble des méthodes d'une interface `I` est reconnu comme implémentant cette interface, sans qu'il soit nécessaire de faire une déclaration explicite d'implémentation. Il n'existe donc aucun mot-clef similaire au mot-clef `implements` de PHP, TypeScript, et autres.

La satisfaction implicite des interfaces entraîne des pratiques déclaratives différentes de ces langages à satisfaction explicite, comme expliqué au § 3.5.6.

3.5.2 Déclaration de types `interface`

Les types interfaces sont déclarés comme dans le listing 3.23, par :

✓ Le mot-clef **`type`**.

✓ L'identificateur du type, qui ne peut pas être vierge.

✓ Le mot-clef **`interface`**.

✓ Un bloc entre accolades, contenant une liste de déclarations de méthodes, ou d'identificateurs d'autres types d'interface, intégrés de façon similaire aux champs intégrés dans les structures (cf. § 4.4.3). Dans ce bloc :

- Chaque méthode doit être nommée.
- Aucun identificateur ne peut être l'identificateur vierge.
- Chaque identificateur de méthode doit être unique dans un type donné. Jusqu'en Go 1.13, cette restriction s'applique aussi entre les méthodes définies localement et celles définies par les interfaces intégrées, ce qui est différent du cas des champs intégrés.

---

26. Leur réalisation dans Go 1.0 est décrite par Russ Cox dans son article consacré à leur implémentation : https://osinet.fr/go/types/cox-interfaces
Hormis la conversion du code de Go lui-même de C vers Go lors de la version 1.5, le principe de fonctionnement est demeuré essentiellement inchangé depuis.
27. https://osinet.fr/go/types/php-interfaces
28. https://osinet.fr/go/types/typescript-interfaces
29. https://osinet.fr/go/types/csharp-interfaces

Cette restriction est levée en Go 1.14, qui autorise des interfaces intégrées comportant des homonymes, à condition qu'elles aient la même signature, autorisant des situations comme l'intégration simultanée d'interfaces comme `io.ReadCloser` et `io.WriteCloser` qui comportent toutes deux une identique méthode `Close`. L'interface résultante est composée de l'union des méthodes définies localement et de celles définies par les interfaces intégrées.

- Chaque méthode est spécifiée comme une fonction, sans mention du récepteur : pour le même type interface, certains types peuvent comporter une implémentation avec un récepteur valeur et d'autres avec une valeur pointeur, comme dans le listing 3.22, où les deux types `C1` et `C2` implémentent l'interface `fmt.Stringer`, l'un avec un récepteur valeur, l'autre avec un récepteur pointeur.

Listing 3.22 – Récepteurs valeurs et adresse pour une même interface
https://osinet.fr/go/types/play-recepteurs-interface

```
8  type C1 int
9
10 func (c C1) String() string { return strconv.Itoa(int(c)) }
11
12 type C2 string
13
14 func (c *C2) String() string { return string(*c) }
15
16 func exemple(x fmt.Stringer) {
17         fmt.Printf("%#v\n", x.String())
18 }
19
20 func main() {
21         var c1 C1 = 1
22         var c2 C2 = "deux"
23         exemple(c1)
24         exemple(&c2)
25 }
26
27 // Résultat: "1" "deux"
```

Si des méthodes sont déclarées sur la même ligne, elles doivent être séparées par des points-virgules.

Les noms des paramètres de chaque méthode peuvent être soit tous présents, soit tous absents. Il en va de même pour les résultats nommés mais, en tout état de cause, ces noms sont ignorés pour la mise en correspondance des méthodes avec les implémentations : leur seul intérêt est documentaire.

Une interface ne peut s'intégrer elle-même, ni directement ni récursivement.

© Dunod – Toute reproduction non autorisée est un délit.

L'exemple du listing 3.23 démontre ces aspects :

- ✓ Une interface peut composer d'autres interfaces : `I` intègre `A` et `C`.
- ✓ Chaque méthode ne doit être définie qu'une fois, que ce soit par un identificateur de méthode local ou par une interface intégrée : ici `A`, définissant `X()`, empêche à la fois d'intégrer `B` et de déclarer une méthode `X()` dans `I`.
- ✓ Une méthode et une interface intégrée peuvent porter le même identificateur, comme ici la méthode `A()` et l'interface `A`.
- ✓ Le jeu de méthodes du type se compose de l'union des méthodes définies localement sur le type en cours de déclaration et de celles déclarées sur toutes les interfaces intégrées. Dans cet exemple, l'interface `I` définit donc :
  - `X()`, apportée par l'interface intégrée `A` ;
  - `Y()`, apportée par l'interface intégrée `B` ;
  - `A()`, définie localement sur `I`.

Listing 3.23 – Interfaces intégrées
https://osinet.fr/go/types/play-interfaces-integrees

```
3  type A interface{ X() }
4  type B interface{ X() }
5  type C interface{ Y() }
6
7  type I interface {
8          A
9          C
10         A()
11
12         // Les doublons suivants sont autorisés
13         // à partir de Go 1.14 seulement:
14         // - par une interface intégrée.
15         B
16         // - par le type principal.
17         X()
18 }
```

Les bonnes pratiques Go définissent également des conventions de nommage pour les types interfaces :

- ✓ Pour les interfaces composées d'une seule méthode, construire un nom d'interface à partir du nom de la méthode et d'un suffixe terminé par `er`. Par exemple, les interfaces `Reader`, nombreuses dans la bibliothèque standard, se composent de la seule méthode `Read`. Adapter la règle si le résultat est inintelligible.
- ✓ Pour les interfaces composées d'un petit nombre de méthodes, notamment celles composant d'autres interfaces unitaires, appliquer la même règle à l'ensemble des noms. Par exemple `io.ReadWriteCloser` est une interface intégrant les interfaces unitaires `io.Reader` (méthode : `Read`), `io.Writer` (méthode : `Write`) et `io.Closer` (méthode : `Close`).

✓ Lorsqu'une interface spécifie des méthodes avec la même signature que des méthodes de la bibliothèque standard, utiliser le même nom que pour les méthodes de la bibliothèque standard : `Reader` en est un parfait exemple, présent en quatorze implémentations dans la seule bibliothèque standard Go 1.14. De même, une méthode de conversion en chaîne reprendra le nom de la méthode bien connue `String` de l'interface `fmt.Stringer` plutôt que de créer un nouveau nom comme `__toString`, même s'il est familier dans un autre langage.

✓ Inversement, ne pas utiliser le nom d'une interface ou des méthodes de la bibliothèque standard lorsque la signature en diffère.

3.5.3 Valeurs de types interfaces

En mémoire, les valeurs de types interfaces se composent de deux parties : la définition du type concret implémentant l'interface, et la donnée proprement dite, directement ou sous la forme d'un pointeur opaque au développeur. Elles portent donc du point de vue de l'utilisateur deux types à la fois :

✓ **statique** : le type *statique* est le type interface sous lequel elles sont déclarées ;

✓ **dynamique** : le type *dynamique* est le type concret de la valeur au moment de l'exécution ; il implémente le type interface.

C'est ce que démontre le listing 3.24 : la variable `x` est déclarée avec le type interface `fmt.Stringer`, qui est donc son type statique, et initialisée successivement avec des valeurs de deux types dynamiques différents implémentant tous deux ce type statique interface.

Listing 3.24 – Types statique et dynamique d'une interface
https://osinet.fr/go/types/play-type-dynamique

```
9  func main() {
10         var x fmt.Stringer
11
12         x = net.HardwareAddr{0xa0, 0xce, 0xc8, 0x05, 0x24, 0xd8}
13         fmt.Printf("%-16T %v\n", x, x)
14
15         x = big.Accuracy(0)
16         fmt.Printf("%-16T %v\n", x, x)
17 }
18
19 // Résultat:
20 // net.HardwareAddr a0:ce:c8:05:24:d8
21 // big.Accuracy     Exact
```

L'existence de ces données dynamiques interdit la définition de constantes typées avec des types interfaces.

© Dunod – Toute reproduction non autorisée est un délit.

Les valeurs de types interfaces peuvent exister dans toutes les portées, aussi bien pour des variables que des paramètres ou des résultats de fonction ou méthode.

Bonnes pratiques

En la matière, la bonne pratique consiste à ce que les fonctions et méthodes :

- ✓ définissent le type des paramètres par des types interfaces comportant le strict jeu de méthodes nécessaires durant l'exécution de la fonction : cela permet de simplifier au plus haut degré la réalisation des tests sur la fonction, et sa maintenance en général, en limitant l'étendue des possibles pour le domaine des paramètres ;
- ✓ définissent le type des résultats par des types concrets : cela évite la multiplication des interfaces et n'impacte pas leur réutilisation puisque ces types concrets, du fait de la satisfaction implicite des interfaces, peuvent être utilisés en aval en tant que d'autres types interface qu'ils implémentent sans même en avoir connaissance.

Les valeurs de types interfaces sont presque toujours utilisées telles quelles et non au travers d'un pointeur : utiliser un pointeur sur une valeur interface est en effet presque toujours une erreur, signalée à la compilation : de tels pointeurs ne peuvent en effet être passés comme arguments qu'à des fonctions ou méthodes spécifiant pour le paramètre correspondant le type `interface{}`, mais même cela correspond généralement à une erreur de logique.

La raison courante pour vouloir passer un tel pointeur sur une valeur interface est de vouloir utiliser une méthode définie avec un récepteur pointeur, mais dans ce cas, il faut initialiser la valeur interface avec le pointeur sur le type concret et utiliser cette valeur interface, au lieu de l'initialiser avec la valeur du type concret et tenter d'utiliser le pointeur sur l'interface.

C'est ce que démontre le listing 3.25, où une valeur `i` du type interface `I` est utilisée successivement avec des méthodes à récepteur pointeur et valeur :

- ✓ Bien que la méthode `pointeur()` soit définie avec un récepteur pointeur, c'est bien avec une variable de type interface qu'elle est invoquée, et non avec un pointeur sur cette variable.
- ✓ En revanche cette variable interface est initialisée avec un pointeur sur le type concret.
- ✓ Cela ne l'empêche pas d'être également utilisée pour invoquer la méthode `valeur()` qui attend un récepteur valeur, du fait de la réécriture automatique des récepteurs, comme décrit au § 3.4.6.

Le langage prédéfinit un seul type interface, le type `error`, détaillé au § 8.8, page 283.

Listing 3.25 – Pointeurs et interfaces
https://osinet.fr/go/types/play-pointeurs-interfaces

```
1  package main
2
3  type I interface {
4          pointeur()
5          valeur()
6  }
7
8  type T []bool
9  func (t *T) pointeur() { println("pointeur") }
10 func (t T) valeur() { println("valeur") }
11
12 func main() {
13         var i I = &T{}
14
15         i.pointeur()
16         i.valeur()
17 }
```

3.5.4 Le type `interface{}`

« *`interface{}` says nothing.* »

Rob Pike - *Go Proverbs*
https://osinet.fr/go/types/yt-pike2015-nothing

Puisque les types interface sont définis par une liste d'identificateurs de méthodes placée dans un bloc, le type **`interface`**`{}` désigne une interface ne requérant aucune méthode pour être satisfaite.

La satisfaction implicite des interfaces, conjuguée au fait que la seule caractéristique d'un type interface soit son jeu de méthodes, implique que cette interface est implémentée par tous les autres types, puisque ceux-ci n'ont besoin de comporter aucune méthode pour satisfaire l'interface [30].

De ce fait, un paramètre de fonction de type **`interface`**`{}` peut recevoir des arguments de type quelconque, à l'image des fonctions et méthodes des langages dynamiques habituels comme ECMAScript, Python, ou PHP sans le *type hinting*. Cette flexibilité lui permet d'être le point d'entrée de l'API de réflexion, décrite au § 5.7.

Utilisé comme élément d'un type carte comme **`map`**`[`**`string`**`]`**`interface`**`{}`, c'est même souvent l'un des premiers types mis en œuvre par les développeurs venant de ces langages.

30. Description par Vincent Blanchon : https://osinet.fr/go/types/blanchon-empty

© Dunod – Toute reproduction non autorisée est un délit.

Le problème est que, puisqu'il ne spécifie aucune méthode du fait de son jeu de méthodes vides, et qu'il ne comporte aucun champ ou valeur propre puisqu'il n'est qu'un type interface, un tel argument nécessite une conversion vers un type concret pour être utilisé ; mais cela pose la question de la connaissance du type vers lequel cette interface doit être convertie : d'où peut provenir l'information ?

- ✓ Elle peut provenir d'un autre argument, à la façon des chaînes de format de `fmt.Printf`. En pareil cas, il est toujours utile de se demander si l'appelant, connaissant le type de la donnée – puisqu'il passe ce type dans le premier argument – ne pourrait pas utiliser une fonction plus spécialisée spécifiant un type plus utile. Ce n'est pas toujours le cas, et les fonctions du paquet `fmt` en sont un parfait exemple, mais lorsque ce changement est possible, il réduit la complexité du code.
- ✓ Elle peut provenir d'une connaissance extérieure au programme, appartenant au développeur. En pareil cas, modifier la signature de la fonction pour spécifier cette connaissance en requérant une interface plus fonctionnelle, permet de codifier cette connaissance, réduisant le besoin de documentation et simplifiant la maintenance future.

Une contrainte non triviale du type `interface{}` est le fait qu'il ne soit pas directement utilisable comme remplacement de tout autre type dans les types composites, à la différence de sa flexibilité dans les appels de fonction.

Ainsi, il n'est pas possible de convertir un tableau ou une tranche d'un type donné – même un autre type interface – en tranche d'`interface{}` : les deux n'ont pas la même représentation en mémoire. Dans ce genre de situation, il est nécessaire d'opérer une copie élément par élément, dans une boucle, comme dans le listing 3.26.

3.5.5 Manipulation des interfaces

Les expressions et instructions utilisant des données de types interfaces sont les suivantes :

- ✓ **Comparaison** : les valeurs interfaces du même type statique peuvent être comparées, uniquement pour ce qui est de l'égalité : aucune relation d'ordre n'est en effet définie entre interfaces, même si leurs types dynamiques supportent une relation d'ordre. Elles sont égales si :
  - soit elles sont du même type dynamique, si les valeurs de ce type dynamique sont comparables, et si ces valeurs comparées sont égales ;
  - soit elles sont toutes les deux `nil`.

  Cette double comparaison entraîne une particularité autour des interfaces `nil` : puisque la comparaison ne se limite pas à comparer les pointeurs ou valeurs mais aussi leur type dynamique, deux valeurs `nil` peuvent être différentes si leurs types dynamiques sont différents.

Listing 3.26 – Conversion de composites d'interface vide
https://osinet.fr/go/types/play-conversion-composites

```
package main

import "fmt"

type TR [3]rune
type TI [3]interface{}

func main() {
        var tr TR = TR{'a', 'b', 'c'}
        var ti TI

        // Conversion directe impossible.
        // ti = TI(tr)

        for i, r := range tr {
                ti[i] = r
        }
        fmt.Printf("%#v\n", ti)
}

// Résultat: main.TI{97, 98, 99}
```

De même la valeur vierge d'un type interface est un **nil** de type dynamique **nil**, qui ne peut être converti en aucun autre type, à la différence d'un **nil** portant un type dynamique non **nil**.

Ce sont ces deux cas que démontre le listing 3.27.

Depuis Go 1.13, des fonctions spécifiques destinées à la comparaison des valeurs du type interface prédéfini **error** sont disponibles dans le paquet `errors`, et leur utilisation est recommandée pour la comparaison de ces valeurs. Ces aspects sont détaillés dans la description du type **error**, au § 8.8.

✓ **Conversion** : une interface structure peut être « convertie » en une valeur d'un type alias de son type courant ou d'un type défini identique à son type courant, cette opération n'accomplissant rien (cf. § 6.12.6).

Plus utilement, elle peut être convertie en tout autre valeur d'un type interface spécifiant le même jeu de méthodes, comme `fmt.Stringer` et `expvar.Var`, qui définissent tous deux la seule méthode `String() string`.

La conversion depuis un type concret ou interface vers un type interface est implicite comme le montre le listing 3.28, même si une conversion explicite est autorisée.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 3.27 – `nil` est différent de `nil`, ou pas
https://osinet.fr/go/types/play-nil-different-nil

```go
package main

import "fmt"

type Vide interface{}

func main() {
        var unOctet *uint8
        var unCourt *uint16

        var o Vide = unOctet
        var c Vide = unCourt
        var n Vide

        // Un nil typé peut être converti
        c2 := c.(Vide)

        // Mais un nil de type nil ne peut l'être
        // n2 = n.(Vide)

        // Les 4 valeurs sont nil.
        fmt.Printf("%v %v %v %v\n", o, c, n, c2)

        // nil != nil, ou pas.
        fmt.Printf("%t %t %t %t\n",
                o != c, o != n, c != n, c != c2)

        // C'est une question de type dynamique
        fmt.Printf("%T %T %T %T\n", o, c, n, c2)

}

// Résultat:
// <nil> <nil> <nil> <nil>
// true true true false
// *uint8 *uint16 <nil> *uint16
```

✓ **Adresse** : l'opérateur d'adresse est applicable aux valeurs interface. Toutefois, comme détaillé au § 3.5.3, le résultat est essentiellement inutilisable : dans la vaste majorité des cas, l'utilisation d'un pointeur sur interface dénote une erreur de logique.

Listing 3.28 – Conversion vers interface
https://osinet.fr/go/types/play-conversion-interface

```
package main

import "fmt"

type Chaîneur interface{ String() string }

type Concret int

func (c Concret) String() string {
        return fmt.Sprint(string(c))
}

func main() {
        const concret Concret = 3

        // Conversion implicite dans les trois cas.
        var stringer fmt.Stringer = concret
        var importé Chaîneur = stringer
        var local Chaîneur = concret

        fmt.Printf("%T %#v %T %#v %t\n",
                importé, importé,
                local, local,
                importé == local,
        )
}

// Résultat: main.Concret 3 main.Concret 3 true
```

✓ **Assertion de type** : l'assertion de type (*type assertion*) permet de réaliser une conversion contrôlée d'une interface vers une valeur de son type dynamique au point de conversion (cf. § 6.10). Elle est déclarée en appliquant à la variable interface le nom du type destination entre parenthèses, évoquant un accès champ sur une structure ; par exemple pour une variable `i` vers le type `int` : `i.(int)`.

Le résultat renvoie une liste de deux valeurs :

- une valeur du type destination. Si l'assertion a réussi parce que le type dynamique était bien le type souhaité, la valeur est bien la valeur d'origine, convertie. Sinon, la valeur est la valeur vierge du type destination.
  Comme pour les conversions explicites, une assertion de type sur une valeur d'interface `nil` de type `nil` ne peut qu'échouer ;
- une valeur booléenne indiquant le succès ou l'échec de l'assertion.

© Dunod – Toute reproduction non autorisée est un délit.

Si seule la première des deux valeurs est affectée et que l'assertion échoue, le programme rencontre une erreur d'exécution. Pour cette raison, il est préférable de toujours utiliser les deux valeurs.
Si, pour des raisons de logique interne, l'assertion ne peut échouer, il est possible d'affecter la seconde valeur à l'identificateur vierge.
Le type cible de l'assertion est habituellement un type concret, qui doit implémenter l'interface sans quoi l'assertion ne peut réussir. Il peut aussi être une interface, et en ce cas l'assertion vérifie que le type dynamique de la valeur implémente l'interface cible.

✓ **Commutateur de type** : une troisième forme de conversion des valeurs interfaces est le commutateur de type (*type switch*) (cf. § 7.3.6).
Dans la conversion simple vue en premier lieu, l'auteur du code sait vers quel type une valeur interface doit être convertie, et utilise une conversion explicite.
Dans l'assertion de type, l'auteur s'attend à ce que l'interface soit implémentée par un type dynamique spécifique ou implémentant une autre interface également connue.
Reste le cas où le type dynamique n'est pas connu à l'avance par l'auteur du code : plutôt que de devoir construire une cascade d'assertions de type successives, Go inclut une syntaxe de choix multiple avec l'instruction `switch`, similaire à la façon dont le `switch` habituel dans les langages dérivés de C permet de remplacer des cascades de `if`/`else`.
Dans cette syntaxe, l'expression évaluée pour une variable interface `i` s'écrit, comme dans le listing 3.29 :

- **mot-clef** : `switch` ;
- **instruction simple** : optionnelle : une « instruction simple », telle que définie au § 7.1.2, terminée par un point-virgule ;
- **clause de garde** : une expression de la forme `i.(type)`, similaire à une assertion de type dans laquelle le type cible aurait été remplacé par le mot-clef `type`.
  Cette expression peut être assignée à une variable par une déclaration courte, auquel cas la clause est de la forme `v := i.(type)` et déclare une variable `v` dans le bloc de chacune des clauses de cas du commutateur ;
- **accolade ouvrante** : début d'une liste de cas ;
- **liste de clauses de cas** : dans laquelle chaque clause de cas est composée :
  - soit du mot-clef `case` suivi d'une liste de types séparés par des virgules et terminée par un caractère deux-points ;
  - soit du mot-clef `case` suivi de l'identificateur `nil`, puis d'un séparateur deux-points ;
  - soit du mot-clef `default` suivi d'un séparateur deux-points.

Aucune clause ne peut en répéter une autre, même partiellement.
Chaque clause de cas définit implicitement un bloc, dans la portée duquel est définie l'éventuelle variable initialisée dans la clause de garde, si elle est présente.
Dans ce bloc, ladite variable est du type spécifié par la clause de cas si un seul type est spécifié, ou du type dynamique de la variable à laquelle s'applique le commutateur, type qui est l'un de ceux listés pour les clauses autres que `case nil:` et `default:`.
Dans une clause `case nil:`, la valeur est **`nil`**.
Dans une clause `default:`, la variable est du type dynamique de la variable d'origine. Ce type n'étant plus analysable autrement que par l'API d'introspection du paquet `reflect` (cf. § 5.7), ces clauses sont principalement utilisées pour l'affichage ou la journalisation des valeurs inattendues ;

- **accolade fermante** : fin de la liste de cas.

Listing 3.29 – Commutateur de type interface
https://osinet.fr/go/types/play-commutateur-type

```
1  package main
2
3  import (
4          "fmt"
5          "math/big"
6          "net"
7          "net/mail"
8  )
9
10 func main() {
11         var i fmt.Stringer = &mail.Address{}
12
13         switch v := i.(type) {
14         case big.Accuracy, net.HardwareAddr:
15                 fmt.Printf("Acc|Har %T\n", v)
16         case *mail.Address:
17                 fmt.Printf("Adresse %T\n", v)
18         case nil:
19                 fmt.Println("Nil")
20         default:
21                 fmt.Printf("Autre   %T\n", v)
22         }
23 }
24
25 // Résultat: Address *mail.Address
```

© Dunod – Toute reproduction non autorisée est un délit.

3.5.6 Organisation des interfaces

« *The bigger the interface, the weaker the abstraction.* »

Rob Pike - *Go Proverbs*
https://osinet.fr/go/types/yt-pike2015-bigger

La stratégie recommandée pour la création de méthodes en Go est très différente de celle d'autres langages comme Java, dans lequel les interfaces ont tendance à comporter un grand nombre de méthodes : le développement en Go a cristallisé l'idée que les interfaces les plus utiles sont celles qui définissent le moins de méthodes, conduisant aux interfaces unitaires comme `io.Reader` ou `io.Writer`, voire à l'interface vide `interface{}`.

Le fait que les interfaces soient satisfaites implicitement est central dans cette inversion d'approche : en raison de cette satisfaction implicite, il est très facile pour un type d'un paquet d'implémenter une interface définie dans un autre paquet, même sans en avoir connaissance.

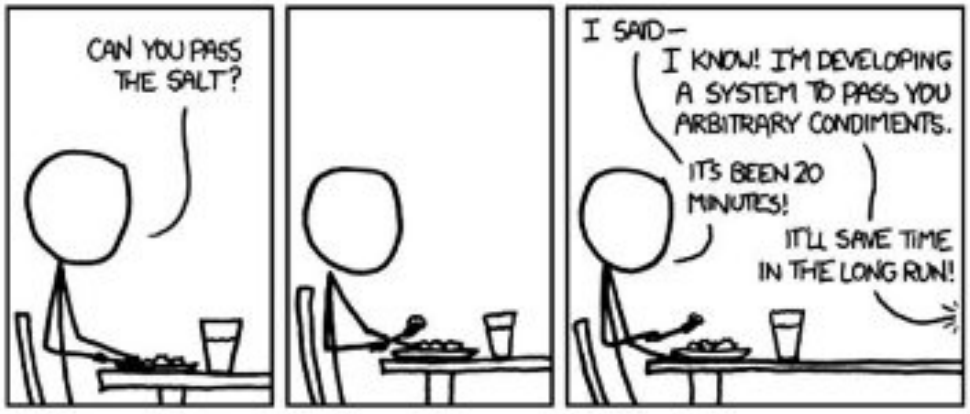
Ainsi, un paquet spécifique à une application peut parfaitement définir une interface qui lui est propre, mais qui se trouve être composée d'un ensemble de méthodes déjà présentes dans un type défini par un paquet tiers, par exemple dans la bibliothèque standard. Dès lors, sans que les paquets tiers n'aient connaissance de cette interface, les valeurs dont les types se trouvent implémenter ce nouveau type interface peuvent être passées comme paramètres à des fonctions attendant des paramètres dudit interface.

Mener cette approche à sa conclusion logique conduit à deux particularités dans l'organisation du code :

- ✓ **Résultats concrets plutôt qu'interfaces** : puisqu'un type peut implémenter une interface sans en avoir connaissance, écrire du code utilisable largement ne nécessite pas de créer l'abstraction que représentent les interfaces au lieu de production des données, en tentant d'imaginer les usages futurs, avec les risques bien connus de cette approche, comme dans la figure 3.3 : délais et complexité excessifs, coût de maintenance, et inadaptation aux besoins futurs réels.
  Une conséquence en est le soin permanent de ne pas définir des interfaces – et en particulier des interfaces publiques – dans les paquets, mais de ne définir que les interfaces présentant une utilité d'abstraction immédiate pour le paquet lui-même ou les premiers consommateurs identifiés [31].
- ✓ **Définition des interfaces au point de consommation** : puisqu'une interface peut être définie à tout moment, y compris dans une portée interne au sein d'un bloc, il est possible de les définir au plus près de l'utilisation, par exemple pour

31. Cette approche hygiéniste des interfaces évite la pollution de l'espace de noms par des interfaces inutiles, et est détaillée par Jaana B. Dogan sur https://osinet.fr/go/types/dogan-interface-pollution

Figure 3.3 – Choisir quand créer une interface
©2011 Randall Munroe - Sous licence Creative Commons BY-NC 2.5



vérifier qu'une valeur implémente une méthode plutôt que de vérifier qu'elle est d'un type particulier, suivant le modèle "*HAS_A* plutôt que *IS_A*" de Go (cf. § 1.1.2.5) : vérifier la présence d'une méthode est réalisable par une assertion de type sur un type interface créé à la volée pour la vérification et composé de cette seule méthode.
Cette pratique est détaillée dans l'article de Dave Cheney sur les interfaces, appliqué à la reconnaissance des erreurs de type `Timeout`, résumé par le listing 3.30.

Listing 3.30 – Vérification d'existence d'une méthode
https://osinet.fr/go/types/cheney-method-exists

```
3  func isTimeout(err error) bool {
4          type timeout interface {
5                  Timeout() bool
6          }
7          te, ok := err.(timeout)
8          return ok && te.Timeout()
9  }
```

Exercice Définir des classes de conteneurs d'utilisation générale utilisables pour regrouper des données de type arbitraire (indice : `interface{}`), et en particulier des structures de tas, de tampon cyclique, et de liste doublement chaînée.

Solution Ces structures sont fournies dans la bibliothèque standard du langage, dans le paquet `container` : https://osinet.fr/go/types/go-container. Leur implémentation fournit un exemple des bonnes pratiques en la matière.

© Dunod – Toute reproduction non autorisée est un délit.

3.6 COMPATIBILITÉ DE TYPES

Les critères de compatibilité entre valeurs et types définissent les règles liées aux types selon lesquelles des valeurs peuvent être comparées, assignées à des variables ou, dans le cas de constantes non typées, représentées par des valeurs d'un type donné.

3.6.1 Identité de types

L'identité de types est le premier critère appliqué à la comparaison de valeurs, qui n'est toujours possible en Go qu'entre valeurs de même types, à la différence de certains environnements proposant un ordre total entre valeurs de types différents comme les comparaisons BSON dans MongoDB [32].

Comme mentionné au § 5.3, les types de données Go peuvent être *définis* ou non.

Les types définis sont tous différents et sont tous associés à un identificateur.

Cela ne signifie pas pour autant que tous les types portant un identificateur soient des types définis, et donc tous différents. En effet, la définition d'un type par une instruction de la forme :

```
type <id> <expression de type>
```

crée un type défini, et lui associe l'identificateur `<id>` spécifié. En revanche, la notation visuellement proche :

```
type <id> = <expression de type>
```

associe bien l'identificateur `<id>` à un type, mais ne crée pas pour autant un nouveau type défini : elle associe seulement un identificateur au résultat de l'expression de type, que ce résultat soit lui-même un type défini ou non. Ce mécanisme, dit *alias de type* (*type aliasing*) et introduit en Go 1.9, permet à des variables portant des types décrits par des identificateurs différents d'être associées au même type, comme dans le listing 3.31.

Comme le montre cet exemple, une définition de type comme celles de `T1` et de `T2` crée de nouveaux types définis, différents les uns des autres, alors qu'une définition d'alias de type, comme celle de `T3` et de `T4` ne crée que de nouveaux identificateurs sans créer de nouveaux types, et ce même si le type faisant l'objet de la création d'un alias n'est pas un type défini.

Ceci illustre le principal intérêt des alias de types : puisqu'il est possible de les associer à des littéraux de type et non seulement à des types définis, ils fournissent un moyen de bénéficier de la possibilité propre aux types anonymes d'être identiques, tout en conservant la souplesse de disposer d'un identifiant plutôt que de devoir mentionner le littéral de type correspondant à chaque utilisation.

32. https://osinet.fr/go/types/mongodb-comparaisons

Listing 3.31 – Types définis et alias de types
https://osinet.fr/go/types/play-alias-types

```
package main

import "fmt"

type T1 int   // Un type défini.
type T2 int   // Un autre type défini.
type T3 = int // Un autre identificateur pour "int".
type T4 = T1  // Un autre identificateur du premier.

func main() {
        const cinq = 5
        const t1 T1 = cinq
        const t2 T2 = cinq
        const t3 T3 = cinq
        const t4 T4 = cinq

        fmt.Printf("T1 défini       %T\n", t1)
        fmt.Printf("T2 défini       %T\n", t2)
        fmt.Printf("T3 alias de int %T\n", t3)
        fmt.Printf("T4 alias de T1  %T\n", t4)
}

// Résultat
// T1 défini       main.T1
// T2 défini       main.T2
// T3 alias de int int
// T4 alias de T1  main.T1
```

Les types non définis, et leurs alias ainsi décrits, peuvent être identiques selon des critères propres à chaque nature de type, listés dans le tableau 3.6.

Pour apprécier l'éventuelle identité entre deux types non définis portant des champs (types structures) ou des méthodes (types interfaces), il est nécessaire de prendre en compte le nom réel des champs ou méthodes : là où les identificateurs des champs ou méthode exportés (cf. § 2.3.2) sont identiques à leur apparence dans le source, les noms de leurs pendants non exportés sont préfixés par le nom du paquet d'origine, de sorte qu'ils ne sont pas identiques entre paquet importé et paquet importateur.

C'est ce qu'explicite le listing 3.32 :

✓ Les types `X` du paquet principal et du paquet importé définissent le même champ exporté `A` :
`x == ix` est vrai car les variables sont de types identiques.

© Dunod – Toute reproduction non autorisée est un délit.

Tableau 3.6 – Conditions d'identité entre types non définis

Nature de type	Conditions d'identité	Non-conditions
types simples	type lui-même	
pointeurs	type de base	
tableaux	type d'éléments et longueur	
tranches	type d'éléments	
structures	champs : nom, ordre, types, étiquettes	
cartes	types de clef et d'éléments	capacité
canaux	type d'éléments, directionnalité	capacité
fonctions	signature : type et ordre des arguments et retours, variadicité	
interfaces	jeu de méthodes : noms et signatures	ordre

- ✓ Les types `Y` du paquet principal et du paquet importé définissent le même champ non exporté `a` :
  `y == iy` échoue à la compilation, car le type de `y` est `struct { a int }` alors que le type de `iy` est affecté par l'import et est `struct { importé.a int}` : le nom du champ n'est donc pas le même et les deux types ne sont pas identiques.
- ✓ Le jeu de méthodes du type `Exemple` du paquet principal et du type interface `I` du paquet importé est apparemment le même, se composant d'une unique méthode non exportée `i`.
  Lorsque la fonction `fonc(importé.I) {}` est invoquée avec pour argument `e` qui est une instance de `Exemple`, la compilation échoue, le nom de la méthode déclarée par `importé.I` est affecté par l'import et est `importé.i` au lieu du simple `i` porté par le type `Exemple`.

3.6.2 Affectabilité

L'affectabilité est la capacité d'affecter une valeur `v` de type `V` à une variable d'un type `T`, défini ou non.

Cette possibilité existe dans six cas :

- ✓ Le plus évident est lorsque `V` est identique à `T`, au sens vu précédemment.
- ✓ Un cas moins limitatif est lorsque les types sous-jacents de `V` et `T` sont identiques, et que l'un au moins des deux n'est pas un type défini.
- ✓ L'affectabilité existe aussi si `T` est un type interface et si `V` implémente cette interface.
- ✓ Dans le cas des canaux, elle existe si `V` bidirectionnel, si les types d'éléments de `V` et `T` sont identiques, et si au moins l'un des deux types `T` et `V` n'est pas un type défini.

Listing 3.32 – Identité de types structure entre paquets

```
package importé

type X = struct{ A int }
type Y = struct{ a int }
type I interface { i() }
```

```
package main

import (
        "fmt"

        "./importé"
)

type X = struct{ A int }
type Y = struct{ a int }

type Exemple struct{}

func (Exemple) i() {}

func fonc(importé.I) {}

func main() {
        var x X
        var ix importé.X
        var y Y
        var iy importé.Y
        var e Exemple

        // Succès
        fmt.Println(x == ix)

        // Erreurs de compilation.
        fmt.Println(y == iy)
        fonc(e)
}
```

Le tableau 4.1, page 140, décrit la matrice d'affectabilité dans ce cas, groupée avec les éventuelles conversions nécessaires lorsque l'affectabilité directe n'est pas présente, mais qu'une conversion est possible, correspondant aux conversions notées comme implicites dans le tableau.

© Dunod – Toute reproduction non autorisée est un délit.

✓ La valeur **nil** est affectable à tout type pointeur, fonction, tranche, carte, canal, ou interface, c'est-à-dire tous les types dont il est la valeur vierge.
C'est ce que représente le listing 3.33, dans lequel les variables sont explicitement initialisées à **nil** pour la lisibilité, bien qu'il s'agisse de la valeur par défaut de chacun de ces types.

Listing 3.33 – Affectabilité de **nil**
https://osinet.fr/go/types/play-affectabilite-nil

```
1  package main
2
3  import "fmt"
4
5  type P *int
6  type F func()
7  type T []int
8  type M map[string]int
9  type C chan int
10 type I interface{ F() }
11
12 func (C) F() {}
13
14 func main() {
15         var p P = nil
16         var f F = nil
17         var t T = nil
18         var m M = nil
19         var c C = nil
20         var i I = c
21         fmt.Printf("%T %T %T %T %T %T\n", p, f, t, m, c, i)
22 }
23
24 // Résultat main.P main.F main.T main.M main.C <nil>
```

Le point particulier de cet exemple est le fait que le type rapporté pour la variable `i` soit `<nil>`, bien qu'il ait reçu un type dynamique concret `C` : cela tient au caractère particulier de la valeur **nil** affectée à un type interface, qui positionne à la valeur vierge à la fois le pointeur vers la donnée dynamique et vers la description de son type, comme décrit au § 3.5.5.

✓ Le cas où `v` est une constante non typée *représentable* par une valeur de type `T` : en ce cas, c'est cette dernière valeur qui est *in fine* assignée à la variable cible.
La représentabilité des constantes est définie au paragraphe suivant.

3.6.3 Représentabilité des constantes

Les constantes et expressions constantes ont pour particularité de n'avoir pas nécessairement de type associé, mais seulement une nature de données, comme détaillé au § 5.2.3.

Les opérations ne mettant en œuvre que des valeurs constantes sont évaluées à la compilation et ne sont pas concernées par les considérations de types, tant qu'elles ne sont pas associées à une déclaration de constante typée.

En revanche, pour pouvoir les utiliser dans une opération impliquant des valeurs non constantes, comme pour pouvoir les utiliser dans l'initialisation d'une constante typée, leur valeur doit soit faire directement partie de l'ensemble des valeurs représentables par le type envisagé, soit être convertible automatiquement à la compilation pour rentrer dans cet ensemble de valeurs.

Pour les types `bool`, `rune` et tous les types entiers, signés ou non, le domaine des valeurs possibles est directement défini par la taille du type, et limite directement les valeurs représentables :

- ✓ **Règles communes** : pour tous les types, une constante est représentable par une valeur d'un type donné si sa valeur fait partie de celles définies par le type. Les paragraphes suivants précisent ou limitent cette règle.
- ✓ **Type `bool`** : les deux valeurs booléennes `true` et `false` uniquement : en pratique, aucune constante booléenne ne peut de toute façon porter une autre valeur, de sorte que toute expression booléenne est représentable par le type **`bool`**.
- ✓ **Types entiers signés et non signés** : les valeurs entières ou flottantes, réelles ou complexes, dont la valeur correspond exactement à une valeur entière du type cible, sont représentables dans ce type cible, comme dans l'initialisation

```
var i int8 = 1.0 + 0i
```

- ✓ **Types entiers signés, dont `rune`** : $-2^n \ldots 2^{n-1} - 1$, où n est le nombre de bits du type ; et tout littéral caractère représentant un point de code dont le numéro est dans le même intervalle.

  Pour les types 8 bits, ceci inclut en particulier l'ensemble du jeu de caractères ASCII, et rien de plus, comme le montre le listing 3.34, où le glyphe `tilde` est représentable par une constante de type `int8`, avec la valeur 126, mais où le `y tréma` ne l'est pas et provoque un dépassement de capacité, alors qu'il est représentable dans le type `int16`.
- ✓ **Types entiers non signés, dont `byte` et `uintptr`** : $0 \ldots 2^n - 1$, où n est le nombre de bits du type ; et tout littéral caractère représentant un point de code dont le numéro est dans le même intervalle.

© Dunod – Toute reproduction non autorisée est un délit.

Pour les types 8 bits, ceci inclut toujours le jeu de caractères ASCII, mais aussi le sous-ensemble du jeu de caractères occidental WGL-4 [33] correspondant à la norme ISO 8859-1, comme le montre le listing 3.34, où le glyphe `y tréma` est représentable dans le type `uint8`, avec la valeur 255, mais où le `A tilde` ne l'est pas et provoque un dépassement de capacité, alors qu'il est représentable dans le type `uint16`.

Listing 3.34 – Représentabilité numérique des constantes caractères
https://osinet.fr/go/types/play-representabilite-caracteres

```
package main

func main() {
        const tilde = '~'  // U+007E: int8 et plus
        const ytréma = 'ÿ' // U+00FF: uint8 et plus
        const Atilde = 'Ã' // U+0100: int16 et plus

        const o int8 = tilde

        // Dépassement de capacité signée
        // const p int8 = ytréma

        const s int16 = ytréma

        println(o, s) // Résultat: 126 162

        const b byte = ytréma

        // Dépassement de capacité non signée.
        // const c uint8 = Atilde

        const us uint16 = Atilde

        println(b, us) // Résultat: 255 256
}
```

✓ **Type `string`** : bien que cela ne soit pas expressément spécifié par le langage, la longueur des chaînes représentables jusqu'à Go 1.13 est limitée :

- à $2 * 10^9$ octets sur toutes les architectures en raison d'un contrôle de taille limite dans le compilateur (fonction `gc.dsname()`) ;
- à 2^{n-1} octets, où n est le nombre de bits du type `int` sur l'architecture cible, en raison du codage de la longueur sur un type `int`, même si la limite précédente était levée.

33. https://osinet.fr/go/types/wiki-wgl-4

- ✓ **Types flottants réels** : outre les valeurs correspondant directement à une représentation IEEE-754, toute valeur numérique pouvant être arrondie à la précision du type **`float32`** ou **`float64`** choisi sans dépassement ou soupassement [34] de capacité est représentable dans ces types.
  En particulier, les expressions complexes dont la partie imaginaire est évaluée à 0, comme `var f float32 = 1 - 0i` sont représentables dans ces types, mais celles dont cette partie est dénormalisée, dénotant un soupassement comme `var f float32 = 1 - 1E-200i` ne le sont pas, provoquant une erreur de compilation.
  L'arrondi utilisé est un arrondi vers la valeur la plus proche ayant son chiffre de poids faible pair. En complément, la valeur IEEE-754 « zéro négatif » qui ne peut être la valeur d'une constante (cf. § 5.2.2) mais peut résulter d'un tel arrondi, est représentée par un zéro positif.
  Outre le zéro négatif, `NaN` et les deux infinis sont des valeurs des types flottants qui ne correspondent à la représentation d'aucune constante, la spécification du langage précisant qu'aucune constante ne peut porter l'une de ces valeurs.
- ✓ **Types flottants complexes** : les constantes représentables dans les deux types **`complex64`** et **`complex128`** sont celles dont chacune des deux parties – réelle et imaginaire – est représentable dans le type flottant réel correspondant, soit respectivement **`float32`** et **`float64`**.

34. https://osinet.fr/go/types/wiki-soupassement

4

Types composites

Go désigne comme types composites tous les types autres que les types simples décrits au § 3.1. Quatre d'entre eux sont associés aux structures de données correspondant à des conteneurs ou agrégats de données simples [1] :

- ✓ **tableaux** : les tableaux sont comparables aux tableaux C, avec une taille fixe ;
- ✓ **tranches** : les tranches sont un type original de Go, qui se comporte comme un tableau dynamique ;
- ✓ **cartes** : les cartes sont des tableaux associatifs ;
- ✓ **structures** : les structures sont le type Go le plus proche des `struct` de C/C++/C#, ou des classes de la plupart des langages à objets, mais avec des différences importantes.

4.1 TABLEAUX (`array`)

4.1.1 Définition

Les tableaux Go sont des structures légères, équivalentes aux tableaux C : ce sont des collections ordonnées à une dimension d'éléments d'un même type, adjacents en mémoire, sans structures annexes, mais avec éventuellement un alignement approprié à l'architecture matérielle, qui peut provoquer l'ajout d'octets de remplissage entre les éléments.

1. Les représentations internes associées à ces structures sont détaillées sur https://osinet.fr/go/types/cox-slices
Attention, comme tout article externe au site officiel du langage, et en particulier pour ce qui concerne le fonctionnement interne du langage – qui est exclu de la promesse de compatibilité Go 1 (cf. §1.1.1.1) – les informations représentent l'état du langage à la date de l'article, mais peuvent être obsolètes dans les versions plus récentes.

La taille des tableaux est fixe et les éléments, indexés à partir de 0, sont initialisés à la valeur vierge de leur type (cf. § 5.4) lors de l'initialisation du tableau.

Il n'existe pas de tableaux constants en Go (cf. § 5.2.2).

Du fait de cette implantation linéaire d'éléments de taille fixe en mémoire, les éléments sont indexés par une simple arithmétique de pointeurs dans le code compilé, ajoutant *n* fois la taille du type des éléments à l'adresse de base du tableau pour localiser l'élément d'index *n*.

Le type des éléments peut être n'importe quel type Go, y compris un tableau : ceci permet notamment de constituer des tableaux multidimensionnels, chaque dimension étant un tableau dont les éléments sont un tableau avec une dimension de moins.

Du fait de cette organisation dans laquelle les tableaux multi-dimensionnels sont des tableaux de tableaux, l'accès aux éléments individuels passe par des indexations successives, par exemple :

```
var z = untableau[x][y]
```

est équivalent à

```
var z = (untableau[x])[y]
```

… comme en C, JavaScript, PHP ou Python, et non par un jeu de crochets uniques avec plusieurs dimensions d'indexation séparées par des virgules comme en Ada ou en Pascal : `untableau[x, y]`.

4.1.2 Déclaration de types tableaux

Les types tableaux sont définis par une taille entre crochets, suivie du type des éléments. Cette taille peut être toute expression constante, comme dans cette déclaration d'un tableau de deux entiers :

```
type t1 [2]int
```

Lors de la déclaration de tableaux multidimensionnels, les parenthèses implicites vont de droite à gauche, à l'inverse des parenthèses implicites lors de l'accès aux éléments par indexation.

Ainsi, cette déclaration d'un tableau de deux tableaux de cinq chaînes de caractères :

```
type t1 [2][5]string
```

est équivalent à

```
type t1 [2]([5]string)
```

4.1.3 Littéraux tableaux

Comme pour les autres types composites, les littéraux de types tableaux sont représentés par le type, suivi d'un jeu d'expressions du type des éléments.

Chacun des types impliqués peut être anonyme, ou être le nom d'un type existant.

Comme il n'existe pas de tableaux constants (cf. § 5.2.2), les littéraux tableaux ne peuvent pas être utilisés pour initialiser des constantes.

Pour faciliter la rédaction de littéraux tableaux :

- ✓ Il est possible de n'inclure dans le littéral que les premières valeurs du tableau, commençant à l'indice 0. Les éventuelles valeurs manquantes à la fin du littéral sont initialisées à la valeur vierge du type des éléments du tableau, comme pour la variable `a` dans le listing 4.1.
- ✓ Lors de la création de tableaux de types anonymes, afin d'éviter un décompte manuel du nombre d'éléments pour définir la taille constante nécessaire à la déclaration d'un littéral de type tableau, le symbole « trois-points » `...` peut être utilisé au lieu d'une constante numérique explicite, comme pour la variable `b` du listing 4.1. La taille est alors évaluée lors de la compilation au nombre effectif d'éléments présents dans le littéral.
  Ce symbole est composé de trois points ASCII, et ne peut pas être remplacé par le caractère Unicode « Horizontal Ellipsis » (*U* + 2026) « … » qui peut être visuellement similaire mais est inconnu du langage.

Listing 4.1 – Littéraux tableaux
https://osinet.fr/go/types/play-litteraux-tableaux

```go
package main

import "fmt"

func main() {
        a := [3]int{3, 7}
        b := [...]int{3, 7, 0}
        fmt.Printf("a: %#v\nb: %#v\na == b ? %t", a, b, a == b)
}

// Résultat
// a: [3]int{3, 7, 0}
// b: [3]int{3, 7, 0}
// a == b ? true
```

© Dunod – Toute reproduction non autorisée est un délit.

Comme le montre le résultat dans ce listing, les deux tableaux ont le même type `[3]int` : un tableau de trois entiers, et ils sont même égaux, bien que :

- ✓ le premier n'ait été initialisé qu'avec deux valeurs pour un tableau de longueur 3 : les éléments non déclarés dans un littéral tableau sont initialisés avec la valeur vierge du type de l'élément, qui est 0 pour les types numériques ;
- ✓ la taille du second ait été calculée automatiquement à la compilation alors que la taille du premier utilisait une valeur explicite : cette taille automatique ne change pas le type et, les types étant anonymes, il est possible d'utiliser l'opérateur `==` pour comparer les deux valeurs [2], qui se retrouvent être du même type (cf. § 3.6.1).

Exercice Définir une variable `a` de type « cube de chaînes à 3 dimensions de taille 2 », initialisée par un littéral de ce type, dont chaque élément est l'une des $2^3 = 8$ chaînes de `"aaa"` à `"bbb"`.

Solution https://osinet.fr/go/types/play-cubes

4.1.4 Manipulation des tableaux

Les tableaux ne sont pas destinés à un usage direct par la plupart des programmes, qui manipuleront plutôt les types tranche, plus pratiques. Ils ne supportent qu'un nombre limité d'opérations :

- ✓ **Comparaison** : les tableaux du même type peuvent être comparés uniquement par les opérateurs `==` et `!=` : aucune relation d'ordre n'est définie entre eux.
- ✓ **Conversion** : une valeur tableau peut être « convertie » en une valeur d'un type alias de son type courant, cette opération n'accomplissant rien. Elle peut également être convertie en une valeur d'un type basé sur son type courant, comme dans le listing 4.2. Le cas général des conversions est détaillé au § 6.12.
- ✓ **Indexation** : il est possible de référencer un élément d'un tableau à partir de son index, et c'est la principale opération effectuée sur les tableaux, en lecture ou en écriture. Si l'index est négatif ou supérieur à la limite du tableau, une erreur d'exécution *runtime panic* est déclenchée par Go. L'indexation, sur tous les types applicables, est détaillée au § 6.8.
- ✓ **Tranchage** : le tranchage (*slicing*) permet d'extraire une séquence d'éléments d'un tableau sous la forme d'une tranche du même type d'éléments que le tableau d'origine, qui utilisera le tableau d'origine comme support. Si l'un des index de l'expression de tranchage est négatif ou supérieur à la limite du tableau, une erreur d'exécution *runtime panic* est déclenchée par Go. Le tranchage, sur tous les types applicables est détaillé au § 6.9.

2. Voir § 5.3 page 159 pour la différence entre types définis et alias de types.

✓ **Adresse** : l'opérateur d'adresse `&` (cf. § 2.4.8) est applicable aux tableaux et aux indexations de tableau. Comme le montre le listing 4.3, l'opérateur d'adresse appliqué au tableau renvoie un pointeur du type du tableau, tandis que, appliqué à une indexation, il renvoie un pointeur du type des éléments du tableau.

✓ **Fonctions `len` et `cap`** : les fonctions intégrées (`builtins`) de taille et capacité s'appliquent aux tableaux. Dans le cas des tableaux, leur résultat est constant et correspond à la taille constante du tableau. Ces fonctions sont détaillées, pour tous les types applicables, au § 8.4.

Le listing 4.3 montre également le caractère compact de la représentation des tableaux. En effet :

✓ L'adresse du pointeur sur l'élément d'index 1 est supérieure de 4 à celle du pointeur vers l'élément d'index 0. Cette taille 4 correspond exactement à la taille du type **`rune`** : 32 bits, soit quatre octets.

✓ L'adresse du pointeur vers l'élément d'index 0 est la même que celle du tableau lui-même, n'en différant que par le type.

Listing 4.2 – Conversion et comparaison de tableaux
https://osinet.fr/go/types/play-conversion-tableaux

```go
package main

import "fmt"

type BytesPair [2]byte
type PairAlias = [2]byte

func (bp BytesPair) sum() byte {
        return bp[0] + bp[1]
}

func main() {
        a := [...]byte{1, 2}
        b := BytesPair(a)
        c := PairAlias(a)
        const format = "%-14T %v\tÉgal ? %t\n"
        fmt.Printf(format, b, b, b == a)
        fmt.Printf(format, c, c, c == a)
}

// Résultat
// main.BytesPair [1 2]         Égal ? true
// [2]uint8       [1 2]         Égal ? true
```

© Dunod – Toute reproduction non autorisée est un délit.

Listing 4.3 – Adresse de tableau et d'éléments
https://osinet.fr/go/types/play-adresse-tableau

```
package main

import (
        "fmt"
)

func main() {
        a := [...]rune{'a', 'b'}
        fmt.Printf("%9T %08p\n%9T %08p\n%9T %08p\n",
                &a, &a,
                &a[0], &a[0],
                &a[1], &a[1])
}

// Résultats
// *[2]int32 0x00414020
//     *int32 0x00414020
//     *int32 0x00414024
```

De façon générale, à moins d'avoir à réaliser une fonction de bas niveau à utilisation très fréquente, l'avantage en performance et occupation mémoire des tableaux ne compense pas la lourdeur de leur manipulation, et l'utilisation des tranches comme une alternative bien plus fonctionnelle est préférable.

4.2 TRANCHES (`slice`)

Le choix d'avoir des tableaux de taille fixe nécessite une structure de données complémentaire de taille variable pour la plupart des usages, et en Go cette structure est incluse dans le langage et porte le nom de *slice*, ou « tranche » en français.

4.2.1 Définition

Comme un tableau, une tranche permet de manipuler un lot d'éléments du même type avec des notations d'indexation et tranchage par crochets `[]` mais, à la différence du tableau, son type ne comporte pas la définition de la taille.

Intérieurement, une tranche est une « fenêtre » mobile appliquée devant un tableau [3]. C'est une structure légère composée :

✓ d'un pointeur vers un tableau ;

✓ d'une longueur ;

✓ d'une capacité.

Le pointeur pointe vers un élément dans un tableau : ce n'est pas forcément le premier élément, plusieurs tranches pouvant faire référence au même tableau sous-jacent.

La longueur indique le nombre d'éléments du tableau sur lesquels la tranche donne une visibilité : elle permet à la tranche de ne représenter qu'un sous-ensemble des données du tableau sous-jacent.

La capacité indique la taille du tableau sous-jacent : elle indique la limite du nombre de valeurs que la tranche pourra enregistrer avant de devoir changer de tableau.

Cas particulier : le valeur vierge du type, une tranche `nil`, ne pointe sur aucun tableau, sa longueur et sa capacité sont égales à 0, et elle ne contient donc aucun élément.

4.2.2 Déclaration de types tranche

Les types tranches sont définis par une paire de crochets similaire à celle d'un type tableau, mais sans taille entre les crochets, suivie du type des éléments.

Comme les tableaux, les types tranches peuvent être multidimensionnels, avec une notation par crochets successifs, par exemple :

```
type bidimensionnel [][]byte
```

Comme pour les tableaux multidimensionnels, les parenthèses implicites vont de droite à gauche, à l'inverse des parenthèses implicites lors de l'accès aux éléments par indexation.

Puisque les types tranche n'ont pas de taille fixe, une tranche multidimensionnelle peut contenir des tranches de dimensions différentes les unes des autres, comme dans le listing 4.4, où chaque ligne de la tranche bi-dimensionnelle contient une tranche de longueur différente des autres.

3. Pour aller plus loin :
✓ Leur structure est celle des types `slice` et `notInHeapSlice` dans le fichier `runtime/slice.go`.
✓ Leur fonctionnement fait l'objet d'un article sur le blogue officiel Go : https://osinet.fr/go/types/go-slices
✓ L'exemple fourni sur https://osinet.fr/go/types/play-slices montre comment examiner la structure d'une tranche depuis un programme en Go.
✓ Le blogue de Russ Cox, l'un des contributeurs du langage, examine ces structures plus en détail : https://osinet.fr/go/types/cox-slices

© Dunod – Toute reproduction non autorisée est un délit.

Listing 4.4 – Littéraux tranches
https://osinet.fr/go/types/play-litteraux-tranches

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          type bytes [][]byte
7
8          var a = bytes{
9                  {1, 2, 3, 4},
10                 {5, 6},
11                 {},
12         }
13         fmt.Println(a)
14 }
15
16 // Résultat: [[1 2 3 4] [5 6] []]
```

4.2.3 Littéraux tranches

Comme pour les autres types composites, les littéraux de types tranches sont représentés par le type, suivi d'un jeu d'expressions du type des éléments, à l'image du listing 4.4.

Chacun des types impliqués peut être anonyme, ou être le nom d'un type existant.

Comme il n'existe pas de tableaux constants (cf. § 4.1), les littéraux tranches qui les référencent ne peuvent pas être utilisés pour initialiser des constantes tranches, qui ne peuvent donc exister.

À la différence des littéraux tableaux, comme la taille d'une tranche n'est pas définie par le type tranche, il n'est pas possible de n'inclure dans le littéral que les premières valeurs de la tranche : chaque tranche est initialisée avec un tableau de la taille exacte correspondant au nombre d'éléments du littéral, comme lors de l'utilisation de la notation `...` pour un littéral tableau. Le listing 4.4 représente précisément ce cas.

4.2.4 Manipulation des tranches

Les tranches supportent presque les mêmes opérations que les tableaux, décrites au § 4.1.4) :

✓ **Comparaison** : les tranches du même type ne peuvent être comparées qu'à la valeur vierge `nil` : aucune relation d'égalité ou ordre n'est définie entre tranches.

✓ **Conversion** : similaire aux tableaux, plus un cas particulier pour la conversion `[]byte` ↔ `string`.

Cette conversion répond à un besoin très courant : nombre de logiques fonctionnelles sont conçues pour des chaînes de caractère, en tant que suite de runes, alors que les opérations d'entrées/sorties sont dans l'ensemble conçues pour des suites d'octets ; le traitement de flux de données texte inclut donc fréquemment cette conversion dans les deux directions.

- ✓ **Indexation** : similaire aux tableaux, la limite supérieure n'étant pas la capacité de la tranche mais sa longueur.
  L'indexation, sur tous les types applicables est détaillée au § 6.8.
- ✓ **Tranchage** : similaire aux tableaux, mais à la différence de l'indexation, la limite supérieure n'est pas la longueur de la tranche, mais sa capacité.
  La tranche résultat est basée sur le même tableau sous-jacent que la tranche d'origine. Ceci peut entraîner des écrasements lors d'écritures sur des tranches d'un même tableau, que ce soit le résultat d'un tranchage direct du tableau ou celui du tranchage d'une tranche déjà tranchée depuis le tableau.
  Le tranchage, sur tous les types applicables, est détaillé au § 6.9.
- ✓ **Adresse** : similaire aux tableaux, mais comme le montre le listing 4.5, l'opérateur d'adresse appliqué à la tranche ne renvoie pas la même que lorsqu'il est appliqué à son premier élément, à la différence des tableaux.
- ✓ **Fonctions `len` et `cap`** : les fonctions intégrées (*builtins*) de taille et capacité s'appliquent aux tranches. Dans le cas des tranches, la capacité est celle de l'espace disponible dans le tableau sous-jacent, tandis que la longueur est celle de la tranche proprement dite, comprise entre 0 et la capacité.
  Ces fonctions sont détaillées, pour tous les types applicables, au § 8.4.

Listing 4.5 – Adresse de tranche et d'élément
https://osinet.fr/go/types/play-adresse-tranche

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var a = [...]byte{1, 3}
7          var s = []byte{1, 3}
8
9          fmt.Printf("Tableau: %p == %p\n", &a, &a[0])
10         fmt.Printf("Tranche: %p != %p\n", &s, &s[0])
11 }
12
13 // Résultat
14 // Tableau: 0x414020 == 0x414020
15 // Tranche: 0x40a0e0 != 0x414022
```

© Dunod – Toute reproduction non autorisée est un délit.

Le langage inclut d'autres mécanismes intégrés pour les tranches :

✓ **append** : fonction intégrée, qui permet d'ajouter à une tranche existante d'autres éléments du même type, en renvoyant une nouvelle tranche contenant les valeurs précédentes, plus les valeurs ajoutées.
Tant que la capacité du tableau sous-jacent est suffisante pour contenir les éléments ajoutés par `append()`, les valeurs ajoutées y sont stockées et la tranche résultante est créée sur le tableau d'origine. Mais lorsque cette capacité est atteinte, `append()` alloue un nouveau tableau plus grand [4], et y recopie les données de l'ancien tableau, puis ajoute ses arguments dans ce nouveau tableau, et la tranche résultante est basée sur ce dernier, comme le montre le listing 4.6.

Listing 4.6 – Agrandissement de tranche avec `append`
https://osinet.fr/go/types/play-append

```
package main

import "fmt"

func p(a []string) {
        fmt.Printf("%d %d %v\n", len(a), cap(a), a)
}

func main() {
        a := make([]string, 2, 4)
        p(a)                            // 2 4 []
        a[0] = "a"
        a[1] = "b"
        p(a)                            // 2 4 [a b]
        a = append(a, "c", "d")
        p(a)                            // 4 4 [a b c d]
        a = append(a, "e", "f")
        p(a)                            // 6 8 [a b c d e f]
}
```

4. La stratégie de choix de la taille est optimisée pour les situations courantes d'évolution de taille, et se trouve dans la fonction `runtime.growslice`.
Sommairement, la taille choisie double à chaque agrandissement automatique jusqu'à 1 kilo-octet, choisit exactement la taille demandée si elle est supérieure au double de la précédente, ou choisit la précédente taille multipliée par la première puissance de 1,25 supérieure à la taille demandée ; puis arrondit le résultat vers le haut en fonction des besoins de l'allocateur mémoire et de la taille du type d'élément de la tranche.

Comme pour les conversions `[]byte` ↔ `string`, `append()` dispose d'une syntaxe spécifique permettant d'ajouter à une tranche `[]byte` le contenu d'une chaîne sans passer par une conversion explicite de celle-ci, comme dans cet exemple :

```
            append([]byte("Bonjour "), "vous"...)
plutôt que  append([]byte("Bonjour "), []byte("vous")...)
ou          append([]byte("Bonjour "), 'v', 'o', 'u', 's')
```

✓ **copy** : fonction intégrée, qui permet de copier les éléments d'une tranche dans une autre tranche préexistante du même type, en gérant correctement les éventuels recouvrements susceptibles de survenir si les deux tranches sont basées sur le même tableau sous-jacent. Si la destination est plus longue que la source, les éléments au-delà de la longueur copiée sont inchangés, comme dans le listing 4.7. Comme `append()`, elle accepte pour la source un argument chaîne sans nécessiter de conversion explicite en `[]byte`.
À la différence d'`append()` :

- elle n'effectue pas de réallocation des tableaux sous-jacents aux deux tranches, de sorte que le nombre d'éléments effectivement copiés est la plus petite des longueurs des deux tranches ;
- elle ne renvoie pas une tranche, mais le nombre d'éléments [5] effectivement copiés : `nombre := copy(destination, source)`.

Listing 4.7 – Fonction `copy` de tranches
https://osinet.fr/go/types/play-copy

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          destination := []byte{1, 2, 3}
7          source := []byte{5, 4}
8          copiés := copy(destination, source)
9
10         fmt.Printf("Copiés: %d, résultat: %v\n",
11                 copiés, destination)
12 }
13
14 // Résultat: Copiés: 2, résultat: [5 4 3]
```

5. Attention aux habitudes héritées du C, dans lequel `memcpy` est très similaire, mais renvoie le nombre d'octets et non un nombre d'éléments.

© Dunod – Toute reproduction non autorisée est un délit.

Dans le listing 4.7, l'intégralité de la tranche source de deux octets a été copiée, mais comme la destination était plus longue, la fin de la tranche destination n'a pas été écrasée : le dernier élément est resté inchangé, et l'appel `copy(destination, source)` a renvoyé `2`, correspondant au nombre d'éléments copiés.

Enfin, pour initialiser les tranches, cinq possibilités existent. Pour un type d'élément `T`, l'initialisation d'une tranche `[]T` est possible par :

- ✓ **Expression** : toute expression renvoyant une tranche peut être utilisée pour initialiser une variable tranche du même type.
- ✓ **Littéral** : `slice := []T{t1, t2 /* ... */}` où chaque élément est de type `T`.
  Dans cette initialisation, le tableau sous-jacent est alloué au nombre exact d'éléments listés dans le littéral, de sorte que longueur et capacité de la tranche sont égales, et aucune sur-allocation n'a lieu. La tranche est initialisée avec les valeurs d'éléments listées dans le littéral.
- ✓ **Fonction `make` à deux arguments** : `slice := make([]T, n)` où `n` est un entier.
  Dans cette initialisation, le tableau est alloué à la longueur `n`, de sorte que là aussi longueur et capacité sont égales. La tranche est initialisée avec `n` éléments dont la valeur est la valeur vierge du type `T`.
- ✓ **Fonction `make` à trois arguments** : `slice := make([]T, n, c)` où `n` et `c` sont des entiers et $n \leqslant c$.
  Dans cette initialisation, le tableau est alloué à la capacité `c`, tandis que la longueur de fenêtre est définie à `n`. Comme dans la forme précédente, la tranche est initialisée avec `n` éléments dont la valeur est la valeur vierge du type `T`.
- ✓ **Fonction `new`** : `slice := new([]T)`
  Dans cette initialisation rarement utilisée, la fonction intégrée `new()` renvoie un pointeur sur une tranche **`nil`**. Son utilisation est équivalente à une simple déclaration de la variable sans initialisation explicite, qui produit aussi une tranche **`nil`**.

4.2.5 Les tranches comme paramètres

La structure interne des tranches donne lieu à des comportements qui peuvent surprendre, en particulier lors de leur utilisation comme paramètres de fonction, comme le montre le listing 4.8.

En effet, pour toute variable d'un type `T`, son utilisation comme paramètre de fonction donne lieu à un passage par valeur et non par référence, de sorte que les éventuelles modifications appliquées au paramètre à l'intérieur de la fonction sont appliquées à la valeur copiée sur la pile lors de l'appel et éliminées à la sortie.

Le listing 4.8 le montre aux lignes 14-16, dans lesquelles la permutation appliquée au tableau est ignorée au retour de la fonction.

Mais comme le montrent les lignes 18-20, ceci semble ne pas s'appliquer au paramètre tranche : bien qu'il soit passé par valeur à la fonction, la tranche est bien modifiée dans l'appelant au retour de la fonction :

- ✓ Lors de l'initialisation de la tranche ligne 18, un tableau de deux entiers est alloué et initialisé avec les valeurs 1 et 2, et défini comme tableau pour la tranche.
- ✓ Lors de l'appel, la tranche est copiée sur la pile d'appel à la fonction.
- ✓ Dans la fonction, lorsque l'échange des valeurs a lieu, les données ne sont pas modifiées dans la copie de la tranche passée sur la pile, mais dans le tableau sous-jacent de cette copie, qui est le même tableau sous-jacent que dans la variable tranche de l'appelant ;
- ✓ Au retour de l'appel, la copie de la tranche est dépilée, mais le tableau partagé entre la tranche `s` et sa copie dorénavant détruite est bien modifié, comme le montre la ligne 20.

Enfin, les lignes 22-24 démontrent le même phénomène, appliqué cette fois non seulement au tableau interne de la tranche `t` mais aussi au tableau par le tranchage duquel elle a été initialisée.

Listing 4.8 – Tranches et tableaux comme paramètres de fonction
https://osinet.fr/go/types/play-parametres-tranches

```
5  func fa(x [2]int) {
6          x[0], x[1] = x[1], x[0]
7  }
8
9  func fs(x []int) {
10         x[0], x[1] = x[1], x[0]
11 }
12
13 func main() {
14         a := [...]int{1, 2}
15         fa(a)
16         fmt.Println(a) // [1 2]
17
18         s := []int{1, 2}
19         fs(s)
20         fmt.Println(s) // [2 1]
21
22         t := a[:]
23         fs(t)
24         fmt.Println(t, a) // [2 1] [2 1]
25 }
```

© Dunod – Toute reproduction non autorisée est un délit.

Tout ceci est une illustration d'un principe plus général dans Go et nombre d'autres langages : passer un paramètre par valeur ne garantit pas l'immuabilité de son contenu, dès lors que ce paramètre contient lui-même des pointeurs.

C'est une propriété familière avec les types objets dans d'autres langages, et leur équivalent **`struct`** en Go : les tranches sont seulement le seul type natif à inclure un pointeur interne causant ce comportement différent des tableaux alors que leur forme et utilisation sont si proches.

4.3 CARTES (`map`)

4.3.1 Définition

Les cartes (*map*) sont la version Go des tableaux associatifs, connus dans d'autres langages sous divers noms, parmi lesquels : carte (*map* ECMAScript 6), dictionnaire (Python), hash (PERL, Ruby), voire simplement tableaux (PHP).

Comme en Python, PERL, ou Ruby avant 1.9, et à la différence d'ECMAScript 6, Ruby ⩾ 1.9 et PHP, les cartes ne sont pas traitées comme ordonnées.

4.3.2 Déclaration de types cartes

Les types cartes sont déclarés comme **`map`**`[type_clefs]type_éléments`.

Les éléments d'une carte sont tous du même type, qui peut être tout type valide.

À la différence des tableaux et tranches, la clef utilisée pour accéder aux éléments n'est pas nécessairement un entier positif, mais peut être de n'importe quel type dès lors que celui-ci supporte entièrement les opérateurs `==` et `!=`.

- ✓ Ceci exclut les types qui ne supportent que la comparaison à **`nil`**, comme les fonctions, tranches, cartes.
- ✓ Mais ceci autorise des types de clefs inhabituels comme les tableaux, ainsi que le montre le listing 4.9.
- ✓ Pour les types **`interface`**, ces opérateurs doivent être définis pour le type dynamique des clefs : les interfaces ne définissant que des méthodes, le compilateur ne peut pas déterminer à la compilation si le type est approprié, de sorte que les éventuelles erreurs ne seront détectées qu'à l'exécution.
  L'utilisation de types interface comme type clef de carte nécessite donc des précautions et n'est pas recommandable en général.

Leur structure interne est complexe, et définie par un jeu de types dont la partie commune est le type `hmap` dans le fichier `runtime/map.go`.

4.3.3 Littéraux cartes

Comme pour les autres types composites, les littéraux de types tranches sont représentés par le type, suivi d'un jeu de paires d'expressions séparées par un carac-

tère `:`, l'expression de gauche définissant la clef et celle de droite la valeur, à l'image du listing 4.9 qui démontre la définition d'une carte d'entiers indexés par des tableaux.

Listing 4.9 – Littéraux cartes
https://osinet.fr/go/types/play-litteraux-cartes

```
package main

import "fmt"

func main() {
        type stringPair = [2]string

        type stringMap map[stringPair]int

        var a = stringMap{
                {"un", "UN"}:     1,
                {"deux", "DEUX"}: 2,
        }
        fmt.Println(a)
}

// Résultat: map[[deux DEUX]:2 [un UN]:1]
```

Chacun des types impliqués peut être anonyme, ou être le nom d'un type existant.

À la différence des littéraux tableaux et tranche, la taille d'une carte est entièrement dynamique, et réallouée au fil des ajouts et suppressions de données dans la carte.

4.3.4 Manipulation des cartes

Les expressions et instructions manipulant des données cartes sont les suivantes :

- ✓ **Comparaison** : les cartes du même type ne peuvent être comparées qu'à la valeur vierge **`nil`** : aucune relation d'égalité ou ordre n'est définie entre cartes.
- ✓ **Conversion** : la conversion entre types cartes nécessite que les types des clefs et des éléments soient convertibles entre les deux types cartes, ce qui en pratique la rend rarement utile.
  La conversion est examinée dans le cas général au § 6.12.
- ✓ **Indexation** : elle est dénotée par l'utilisation de crochets `[]` comme pour les tableaux et tranches, elle en diffère en n'échouant jamais, même sur une carte **`nil`** : comme pour un objet JavaScript, si elle est utilisée avec une clef inexistante, aucune erreur ne survient, et une valeur par défaut est renvoyée qui est la valeur vierge du type des éléments de la carte.

© Dunod – Toute reproduction non autorisée est un délit.

Puisque cette valeur vierge peut parfaitement être une valeur valide, Go propose pour différencier une valeur vierge existante d'une valeur renvoyée en l'absence de la clef spécifiée un mécanisme reposant sur une affectation parallèle comme :

```
e, ok := carte[k].
```

Dans cette notation, si la clef existe, `ok` est vrai, sinon il est faux.
Ceci évite d'avoir à recourir à une fonction séparée d'existence des clefs comme `array_key_exists()` en complément de `isset()` en PHP.
L'indexation, sur tous les types applicables est détaillée au § 6.8.

✓ **Adresse** : à la différence des tableaux et tranches, il est bien possible d'utiliser l'opérateur `&` pour obtenir l'adresse d'une carte, mais pas pour obtenir l'adresse de ses éléments, qui est mobile au fil de l'évolution de la carte.

✓ **Fonction `len`** : la fonction intégrée (*builtin*) de taille s'applique aux cartes, et donne le nombre d'éléments présents dans la carte. La fonction `cap` n'est en revanche pas définie : il n'est pas possible en se limitant à la spécification du langage d'obtenir la taille du support de données interne des cartes.
Cette fonction est détaillée, pour tous les types applicables, au § 8.4.

✓ **Définition d'élément** : dans une carte non-`nil`, il est possible d'insérer un élément associé à une clef par une expression d'affectation comme :

```
carte[clef] = valeur
```

Le langage ne propose pas de moyen de savoir si une valeur existait antérieurement et a été redéfinie, ou s'il s'agit d'une insertion.
Toute tentative d'insérer un élément dans une carte `nil` provoque une erreur d'exécution.

✓ **Suppression d'élément** : la fonction intégrée `delete` permet de supprimer un élément par sa clef, comme dans `delete(carte, clef)` [6].
Elle ne provoque pas d'erreur si la carte ne comporte déjà aucun élément pour la clef spécifiée, y compris sur une carte `nil`, ce qui lui permet d'être idempotente.

Les cartes ne supposant pas de relation d'ordre sur les clefs, il serait impossible de donner un sens à une expression de tranchage, qui n'existe donc pas sur les cartes.

Enfin, pour initialiser les cartes, cinq possibilités existent. Pour un type de clef `K` et un type d'élément `V`, l'initialisation d'une tranche `map[K]V` est possible par :

✓ **Expression** : toute expression renvoyant une carte peut être utilisée pour initialiser une variable tranche du même type.

6. À la différence du `delete` JavaScript et du `unset` PHP, la carte et la clef sont deux arguments séparés.

- ✓ **Littéral** : `slice := map[K]V{k1: v1, k2: v2 /* ... */}` où chaque clef est de type `K` et chaque élément est de type `V`.
  La liste vide `{}` est valide et crée une carte vide non-`nil`.
- ✓ **Fonction `make` à un argument** : `carte := make(map[K]V)`
  Dans cette initialisation, la carte est initialisée vide, avec une capacité par défaut dépendant de l'implémentation, non spécifiée par le langage.
- ✓ **Fonction `make` à deux arguments** : `carte := make(map[K]V, c)` où `c` est un entier positif ou nul.
  Dans cette initialisation, la carte est initialisée vide, avec une capacité initiale égale au paramètre `c`.
- ✓ **Fonction `new`** : `carte := new(map[K]V)`
  Dans cette initialisation rarement utilisée, la fonction intégrée `new()` renvoie un pointeur sur une carte `nil`. Son utilisation est équivalente à une simple déclaration de la variable sans initialisation explicite, qui produit aussi une carte `nil`.

Au quotidien, les cartes Go ont moins le caractère d'objet « bon à tout, propre à rien » qui est fréquemment associé à leur homologues dans les langages dynamiques du web, du fait du typage strict des clefs et des éléments. Ceci rend leur utilisation comme cible de désérialisation des flux réseau JSON et autres moins immédiate que dans ces autres langages, forçant les développeurs à adopter en règle générale une approche plus structurée des flux.

Le type carte le plus adapté à cette utilisation est dans la plupart des cas une carte `map[string]interface{}` : la plupart des flux utilisent en effet des chaînes ou des nombres – représentables comme chaînes – comme clefs ; la principale difficulté réside alors dans l'interprétation des éléments, qui nécessite soit une connaissance externe par un schéma, soit l'utilisation d'un mécanisme de réflexion/introspection, décrit au § 5.7. La relative complexité du code basé sur la réflexion et l'existence usuelle de schéma ont tendance à décourager cette approche au profit de types plus structurés.

4.3.5 Les cartes comme paramètres

Comme pour les tranches, la structure interne des cartes donne lieu à des comportements qui peuvent surprendre, en particulier lors de leur utilisation comme paramètres de fonction : bien que passées par valeur, les cartes peuvent être modifiées par toute fonction qui les reçoit comme paramètres.

Cela provient du fait que la structure interne représentant les cartes est en fait un pointeur sur la structure `hmap`, et non une copie de cette valeur : de ce fait, tout son contenu peut être modifié dans toute fonction appelée avec un paramètre carte.

© Dunod – Toute reproduction non autorisée est un délit.

Évolution du type

Dans les premiers temps du langage, bien avant la version 1.0 et la promesse de compatibilité, le type aujourd'hui connu comme `map` était en fait déclaré comme un pointeur `*map`, pour rendre ce passage de pointeur explicite, mais il a été masqué par la suite pour adopter la forme actuelle[7], dans un souci de simplification syntaxique.

4.4 STRUCTURES (`struct`)

4.4.1 Définition

Les trois précédents types composites – tableau, tranche, et carte – avaient en commun de contenir un nombre variable d'éléments du même type. Les types structure, au contraire, contiennent un nombre fixe et ordonné d'éléments de types potentiellement différents, appelés *champs* (*fields*).

Les structures sont les types Go les plus proches des classes trouvées dans nombre d'autres langages comme C++, ECMAScript 6, Java, PHP, ou Python ; elles en diffèrent toutefois fondamentalement sur plusieurs aspects, pour les raisons détaillées au § 1.1.2.5 :

- ✓ absence d'héritage, remplacé par la composition ;
- ✓ absence des types partiels (traits, *mixins*), remplacé par la composition ;
- ✓ délégation automatique (*autoforward*) des accès aux champs et méthodes appartenant à des types composés ;
- ✓ implémentation automatique des interfaces, détaillée au § 3.5 consacré aux interfaces ;
- ✓ absence de propriétés et méthodes statiques ;
- ✓ absence de modificateurs de types et champs comme `abstract` ou `final` ; en l'absence de d'héritage :
  - toute structure est intrinsèquement `final` au sens Java/PHP,
  - une structure `abstract` n'aurait aucune utilité, ne pouvant être héritée, et n'ayant aucune implémentation à inclure dans une composition.

À la différence des classes Ruby, les types `struct` sont figés : il n'est pas possible de leur ajouter des propriétés dynamiquement (*monkey patching*), pour respecter l'organisation mémoire définie à la compilation.

Les structures diffèrent également des `struct` du langage C, dont elles sont pourtant inspirées, sur trois points :

- ✓ Les structures C sont limitées à pouvoir comporter des champs pointeurs de fonctions, alors que les structures Go possèdent de tels champs de types fonctions, mais possèdent aussi séparément des méthodes avec un receveur.

7. https://osinet.fr/go/types/nuts-map

La différence entre ces deux fonctionnalités est détaillée au § 3.4.

✓ Les structures Go n'ont pas d'équivalent aux champs bits de C/C++.

✓ À la différence de C, qui garantit une implantation mémoire compacte [8], ou de Pascal qui dispose de l'attribut `packed` pour forcer une organisation des champs d'un `record` (l'équivalent Pascal des **`struct`**) à l'octet près, Go ne spécifie pas l'alignement interne des champs dans un type structure, laissant ces détails à la combinaison du compilateur et de l'architecture cible.
Ainsi, la seule garantie fournie par la spécification du langage en la matière est que l'alignement total d'une structure est égal à l'alignement du plus large de ses champs [9], de sorte que le même champ octet peut être aligné sur 32 bits (donc quatre octets) dans un type structure mais sur 64 bits (donc huit octets) dans un autre type structure d'un même programme sur la même architecture cible.

Une autre différence avec d'autres langages orientés objets est le fait que les types classe sont souvent les seuls sur lesquels des méthodes peuvent être définies, alors qu'en Go la plupart des types définis peuvent supporter des méthodes, les seules exceptions étant les types pointeurs et interfaces.

4.4.2 Déclaration de types structure

Dans un type structure, les champs peuvent être explicites ou implicites, et portent un nom qui est un identificateur du langage, comme défini au § 2.3.1.

Les identificateurs autres que l'identificateur vierge (_) doivent être uniques dans un type structure donné. Les champs portant l'identificateur vierge peuvent être utiles par exemple pour attribuer – de manière non sûre mais immédiate, par le paquet `unsafe` – décrit dans les *Plus en ligne* – une structure typée sur le contenu d'un tampon mémoire, notamment lors d'opérations d'entrée/sortie (*E/S*, *I/O*) de bas niveau. Ils sont en ce sens équivalents aux `FILLER` des « enregistrements » Cobol.

La liste de champs d'un type peut être zéro, correspondant à une structure vide, de longueur zéro en mémoire, ce qui peut être utile comme élément d'autres types composites où seule l'existence d'un élément est utilisée alors que la valeur ne l'est pas : en pareil cas, l'occupation mémoire est réduite au minimum puisqu'aucune donnée n'est présente. Deux utilisations courantes [10] sont :

✓ **contenu de canaux** : ces données de longueur vides sont utiles pour des canaux (cf. § 4.5) de pure signalisation / contrôle de flux entre goroutines ;

8. Utilisation de cette propriété, en C, Go et Rust : https://osinet.fr/go/types/raymond2014-packing
9. https://osinet.fr/go/types/go-alignement
10. Dave Cheney sur l'utilisation de ces structures vides : https://osinet.fr/go/types/cheney-structures-vides et sur un cas particulier où la longueur résultante de cette structure vide n'était pas zéro dans les versions intermédiaires de Go 1.5 beta : https://osinet.fr/go/types/cheney-alignement

© Dunod – Toute reproduction non autorisée est un délit.

✓ **ensembles** : Go n'ayant pas de type « ensemble » (*set*) au niveau langage, un type carte `map[T]struct{}` fournit une implémentation native du type ensemble pour des éléments de type `T` sans gaspiller d'espace pour stocker des valeurs inutiles.

Les types structure sont déclarés par le mot clef **`struct`**, suivi d'une liste de champs entre accolades, comme dans le listing 4.10. Chaque ligne de champ est composée :

✓ d'un ou plusieurs identificateurs de champ, optionnels, séparés par des virgules, et qui peuvent être l'identificateur vierge _ ;

✓ du type du champ, obligatoire et potentiellement anonyme ;

✓ d'une chaîne de caractères, optionnelle, dite étiquette (*tag*) du champ.

Listing 4.10 – Déclaration d'un type structure
https://osinet.fr/go/types/play-structure

```
package main

import "fmt"

type Foo struct {
        Entier uint16 `json:"n" yaml:"num"`
        Rejet  int8   `json:"-" yaml:"-"`
        _      byte
        Chaîne string `json:"s,omitempty" yaml:"msg,omitempty"`
}

func main() {
        f1 := Foo{
                Entier: 1,
                Chaîne: "un",
        }
        f2 := Foo{Entier: 2, Chaîne: "deux"}
        f3 := Foo{
                3,
                30,
                0,
                "trois",
        }
        f4 := Foo{4, 40, 0, "quatre"}
        fmt.Println([...]Foo{f1, f2, f3, f4})
}

// Résultat:
// [{1 0 0 un} {2 0 0 deux} {3 30 0 trois} {4 40 0 quatre}]
```

Aux lignes 5-10, ce listing illustre un type **struct** basique, comportant quatre champs de types différents, avec des étiquettes (cf. § 4.4.4), et dont l'un est désigné par l'identificateur vierge. Les lignes suivantes démontrent les quatre formes de littéraux disponibles pour l'initialisation d'une valeur de type structure, détaillée un peu plus bas, au paragraphe 4.4.5.

4.4.3 Champs intégrés et promotion

Afin de faciliter l'utilisation de la composition, les types **struct** peuvent inclure des champs intégrés (*embedded fields*) : ce sont des champs qui sont déclarés uniquement par l'identificateur de leur type, sans identificateur de champ explicite, comme le type `Interne` à la ligne 11 du listing 4.11.

Listing 4.11 – Champ intégré dans une structure
https://osinet.fr/go/types/play-champ

```
1  package main
2
3  import "fmt"
4
5  type Interne struct {
6          x byte
7          s string
8  }
9
10 type Externe struct {
11         Interne
12         x string
13 }
14
15 func main() {
16         e := Externe{
17                 Interne{2, "interne"},
18                 "externe",
19         }
20         fmt.Printf(`Structure: %v
21 Explicite: x = %-8v s = %-8v
22 Implicite: x = %-8v s = %-8v`,
23                 e, e.Interne.x, e.Interne.s, e.x, e.s)
24 }
25
26 // Résultat:
27 // Structure: {{2 interne} externe}
28 // Explicite: x = 2        s = interne
29 // Implicite: x = externe  s = interne
```

© Dunod – Toute reproduction non autorisée est un délit.

Pour de tels champs :

- ✓ L'identificateur du type sert également d'identificateur du champ.
- ✓ Les méthodes et les éventuels champs de la valeur portée par le champ intégré sont accessibles comme s'ils étaient portés directement par la structure externe, sans nécessiter un passage par une référence au champ intégré.
- ✓ Cette intégration se limite aux champs et méthodes du champ, et ne permet par d'utiliser la `struct` porteuse comme une instance du type du champ, par exemple pour l'utiliser dans une expression.
  En particulier, lors de l'initialisation par un littéral structure, le champ intégré doit être utilisé pour l'initialisation, comme à la ligne 17 du listing 4.11 : il n'est pas possible d'initialiser la structure externe directement avec les champs de la structure interne du champ intégré.
- ✓ Du fait de cette utilisation automatique de l'identificateur, un seul champ par type peut être intégré dans une structure : si plusieurs champs sont du même type, un seul peut être intégré automatiquement ainsi ; les autres ont nécessairement des identificateurs explicites pour respecter la règle d'unicité des identificateurs de champs.
  En pareil cas, il est préférable que tous les champs du type concerné portent des identificateurs explicites pour éviter toute ambiguïté de lecture.
- ✓ Le type n'a pas obligation d'être un type défini : `type T struct { int }` est une déclaration valide, qui définit un type `T` comportant un champ intégré entier, mais `type T struct { []int }` n'est pas valide, `[]int` n'étant pas un identificateur.

L'accès aux champs et méthodes des types intégrés est appelé « promotion » du champ ou de la méthode.

Si une ambiguïté de nommage existe entre des champs intégrés à divers niveaux, comme dans le cas du champ `x` dans le listing 4.11, qui est défini à la fois sur les types `Interne` et `Externe`, la référence au champ ambigu est effectuée par le compilateur sur le champ le moins profondément imbriqué, comme dans le résultat :

- ✓ Le résultat de la sélection `e.s`, non ambigu, correspond bien au champ `s` du type `Interne`.
- ✓ Mais le résultat de la sélection `e.x`, ambigu, correspond au champ `x` du type `Externe`, qui est le moins imbriqué, comme le montre la ligne 29.
- ✓ Il est nécessaire pour accéder au champ imbriqué de passer par son chemin explicite, `e.Interne.x`, en ignorant l'intégration, comme à la ligne 28.

La même règle s'applique pour la résolution des ambiguïtés sur les méthodes.

4.4.4 Étiquettes de champs

Comme mentionné dans la définition des types structures au § 4.4.2, les déclarations de champs dans les types structure peuvent comporter – après le nom du type – une chaîne de caractères, le plus souvent au format brut (cf. § 3.1.4.2) dite « étiquette » (*tag*). Le listing 4.10 montre trois champs ainsi étiquetés : `Entier`, `Rejet`, et `Chaîne`.

Ces étiquettes sont lisibles au moyen de l'API de réflexion (introspection) du paquet `reflect` (cf. § 5.7.2) et sont utilisées pour les comparaisons d'identité (cf. § 3.6.1) de types **`struct`**, mais n'affectent pas le stockage des données ou les opérations disponibles.

Une étiquette composée d'une chaîne vide et une étiquette absente sont équivalentes.

Le contenu des étiquettes est libre mais, en pratique, la bibliothèque standard du langage et nombre d'outils tiers attendent qu'elles respectent un format spécifique pour permettre leur exploitation simplifiée par les outils de réflexion. Les cas d'utilisation les plus typiques sont :

✓ **Les sérialiseurs** : pour les bibliothèques de sérialisation – JSON, YAML, protocol buffers, XML, etc – les étiquettes permettent de définir les options de sérialisation directement au niveau des champs plutôt que de devoir les définir dans un objet d'options séparé.
Les propriétés dépendent des besoins du sérialiseur, mais incluent souvent au moins le nom du champ dans la version sérialisée, et une option `omitempty` pour indiquer que le champ ne doit pas être inclus – plutôt que d'être inclus avec une valeur vide – dans la version sérialisée s'il est vide, et une option « - » indiquant que le champ doit être ignoré.

✓ **Les mappeurs objet-relationnel (ORM)** : peuvent utiliser les étiquettes pour définir les propriétés des colonnes de tables associées aux champs : type de données, taille, unicité, incrémentation automatique, etc ; ainsi que les contraintes d'intégrité référentielle et les index.

Ce format se compose de paires clef-valeur séparées par un espace :

✓ La clef est une chaîne non vide composée de caractères autres que
- contrôles ASCII (valeur ⩽ 32 = 0x20) ;
- espace (valeur 32 = 0x20) ;
- deux-points (valeur 58 = 0x3A).

En pratique, il s'agit presque toujours d'un mot au format des identificateurs.

© Dunod – Toute reproduction non autorisée est un délit.

Chaque paquet utilisant les étiquettes définit une ou plusieurs clefs qu'il utilise, habituellement une seule clef égale au nom du paquet.

- ✓ Le séparateur clef-valeur est le symbole deux-points (valeur 58).
- ✓ La clef est une chaîne interprétée, donc entourée de guillemets (valeur 34). L'interprétation du contenu de la chaîne est libre, selon les besoins du paquet qui l'exploite ; en pratique, il s'agit souvent d'une liste de propriétés ou de paires clef-valeur, séparées par des virgules.

L'exemple du listing 4.10 page 128 illustre ces règles :

- ✓ Le champ `Entier` serait par défaut sérialisé sous le même nom, mais l'étiquette définit qu'en JSON il sera sérialisé sous le nom `n`, alors qu'en YAML il le sera sous le nom `num`.
- ✓ Le champ `Rejet` serait par défaut sérialisé sous le même nom, mais l'étiquette indique qu'il est ignoré par les sérialiseurs JSON et YAML, du fait de la présence de la valeur `-`.
- ✓ Le champ `Chaîne` serait par défaut sérialisé sous le même nom, mais l'étiquette définit, outre des noms différents, le fait que – en JSON comme en YAML – il ne sera pas inclus dans la version sérialisée s'il est vide, du fait de la valeur `omitempty`.
- ✓ Il n'est pas nécessaire que le code définissant des étiquettes pour un paquet importe le paquet correspondant : cet exemple définit des étiquettes pour JSON et YAML mais ne les importe pas.

L'API de réflexion est décrite au § 5.7 consacré au paquet `reflect`.

4.4.5 Littéraux structures

Comme pour les autres types composites, les littéraux de types structure sont représentés par le type, suivi d'un jeu d'expressions du type des champs de la structure, à l'image des quatre variables `f1` à `f4` du listing 4.10 et de la variable `e` du listing 4.11.

Compte tenu de la diversité des types de champs dans une structure, les règles sont un peu plus nombreuses pour les littéraux structure que pour ceux des tableaux, tranches et cartes, où tous les éléments sont du même type. Ce sont les cinq règles suivantes :

- ✓ **Liste vide** : une liste de champs vide est valide, et désigne la valeur vierge du type structure, composée des valeurs vierges de chacun de ses champs.
- ✓ **Liste de valeurs** : si le littéral ne contient aucune clef, mais seulement des valeurs, comme aux lignes 10-13 du listing 4.12, les valeurs sont implicitement affectées aux champs dans l'ordre des champs déclarés dans le type de la structure.

Dans ce format, des valeurs doivent être présentes pour l'intégralité des champs du type structure, y compris les champs intégrés et ceux dont la clef est l'identificateur vierge, comme à la ligne 11 où surviendrait une erreur de compilation, et ce bien que la valeur du troisième champ soit en pratique inutile du fait de l'identificateur vierge, comme le montre le résultat, où sa valeur est à 0 alors que le littéral comporte un 3.

Listing 4.12 – Les deux formes des littéraux structure
https://osinet.fr/go/types/play-litteraux-structures

```
1  package main
2
3  import "fmt"
4
5  type Structure struct {
6          a, b, _ int
7  }
8
9  func main() {
10         // Erreur: "too few values in Structure literal"
11         // x := Structure{1, 2}
12
13         x := Structure{1, 2, 3}
14         fmt.Println(x)
15
16         // Erreur: "invalid field name _ in struct initializer"
17         // y := Structure{a: 1, b: 2, _: 3}
18
19         y := Structure{a: 1, b: 2}
20         fmt.Println(y)
21 }
22
23 // Résultats: {1 2 0} et {1 2 0}.
```

✓ **Liste de clefs et valeurs** : dans ce format, comme aux lignes 16-19, seuls les champs dont la valeur n'est pas la valeur vierge de leur type ont besoin d'être déclarés : les champs dont la clef n'est pas listée dans le littéral sont initialisés avec la valeur vierge de leur type. Déclarer une valeur pour les champs portant l'identificateur vierge comme à la ligne 16 n'est pas autorisé.
L'ordre des clefs dans ce format n'affecte pas les résultats ; en particulier, si un champ dépend d'un autre, le compilateur ne fournit pas de garantie sur l'ordre dans lequel les champs sont initialisés, qui est indépendant de l'ordre des lignes du littéral.
Les seules valeurs autorisées pour les clefs sont les noms des champs du type structure initialisé, qui peuvent être les noms implicites des champs intégrés.

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ **Liste mixte** : la combinaison de champs au format clef+valeur et de champs au format valeur seule est incorrecte.
- ✓ **Types d'autres paquets** : si le type que le littéral initialise appartient à un autre paquet, ses champs non exportés ne peuvent pas être initialisés, leurs identificateurs n'étant pas visibles dans le paquet courant.
  Si le littéral est au format liste de valeurs, cette liste devra comporter l'initialisation des champs non exportés pour respecter l'obligation d'inclure tous les champs : bien que ce ne soit pas absolument incorrect, c'est une pratique déconseillée, signalée par les outils de contrôle de qualité de code : elle introduit en effet un couplage entre le fonctionnement interne du paquet importé et du paquet qui l'importe, sans aucune garantie de stabilité.

4.4.6 Manipulation des structures

Les expressions utilisant des opérandes structures sont les suivantes :

- ✓ **Comparaison** : les structures du même type – y compris par alias – peuvent être comparées uniquement par les opérateurs `==` et `!=` : aucune relation d'ordre n'est définie entre elles.
  Cette comparaison n'est possible que si tous les champs du type structure comparés sont eux-mêmes comparables : les deux structures sont égales si tous leurs champs non vierges sont égaux, comme dans le listing 4.13.
- ✓ **Conversion** : une valeur structure peut être « convertie » en une valeur d'un type alias de son type courant, cette opération n'accomplissant rien.

Listing 4.13 – Comparaison de structures
https://osinet.fr/go/types/play-comparaison-structures

```
1  package main
2
3  type Comparable struct{ x, _ int }
4
5  func main() {
6          t1 := Comparable{3, 5}
7          t2 := Comparable{3, 7}
8          println(t1 == t2)
9  }
10
11 // Résultat: true
```

Elle peut également être convertie en une valeur d'un type basé sur le même type sous-jacent, ce qui correspond aux types définis basés l'un sur l'autre ou aux types non liés mais composés exactement des mêmes champs, y compris les champs à identificateur vierge, mais en ne prenant pas en compte les étiquettes de structure, comme à la ligne 17 du listing 4.14 [11].
La conversion est examinée dans le cas général au § 6.12.

Listing 4.14 – Conversion de structures
https://osinet.fr/go/types/play-conversion-structures

```
1  package main
2
3  import "fmt"
4
5  type T1 struct {
6          x int `foo:"bar"`
7  }
8
9  type T2 T1 // Type basé sur T1
10
11 type T3 struct {
12         x int `foo:"baz"` // Etiquette différente
13 }
14
15 func main() {
16         t1, t2, t3 := T1{1}, T2{1}, T3{1}
17         fmt.Println(t1, T1(t2), T1(t3))
18 }
19
20 // Résultat: {1} {1} {1}
```

✓ **Sélection** : il est possible de référencer un élément d'une structure à partir de son champ, et c'est la principale opération effectuée sur les tableaux, en lecture ou en écriture : ce mécanisme est l'équivalent pour les structures de l'opération d'indexation.
À la différence de JavaScript ou PHP qui l'autorisent sans restriction et ajoutent dynamiquement les champs indéfinis, la sélection d'un champ non défini en Go provoque une erreur de compilation.

11. Les premières versions de Go requéraient que les étiquettes soient aussi les mêmes. Cette limitation a été levée en Go 1.8.

© Dunod – Toute reproduction non autorisée est un délit.

Go ne propose pas non plus de moyen intégré d'accéder à un champ par son nom de façon dynamique : cela est possible uniquement par l'API de réflexion (cf. § 5.7). Si une telle sélection dynamique est exercée au moyen de cette API sur un champ inexistant, le langage émet une panique d'exécution.

✓ **Adresse** : l'opérateur d'adresse `&` (cf. § 2.4.8) est applicable aux structures et sélections de champs dans les structures.
Comme le montre le listing 4.15 :

- l'opérateur d'adresse appliqué à la structure renvoie un pointeur du type de la structure, tandis qu'appliqué à une sélection il renvoie un pointeur du type du champ sélectionné ;
- l'adresse du premier champ de la structure est numériquement égale à celle de la structure entière.

Listing 4.15 – Adresses de structures et champs
https://osinet.fr/go/types/play-adresse-structure

```
package main

import "fmt"

func main() {
        a := struct{ r, s rune }{'r', 's'}

        fmt.Printf("%9T %08p\n%9T %08p\n%9T %08p\n",
                &a, &a,
                &a.r, &a.r,
                &a.s, &a.s)
}

// Résultats
// *struct { r int32; s int32 } 0x00414020
//     *int32 0x00414020
//     *int32 0x00414024
```

Enfin pour initialiser une structure, Go ne propose pas de mécanisme spécifique de constructeur, à la différence des langages à base de classes comme Java, PHP, ou ECMAscript : dans la plupart des cas, les types Go sont conçus pour être utilisables à partir d'une simple déclaration, leur valeur vierge étant prête à l'emploi comme décrit au § 5.4.

Lorsque ce n'est pas le cas, plutôt que de définir un mécanisme rigide de constructeur unique par classe comme en PHP, ou de devoir mettre en œuvre un mécanisme de surcharge autorisant des constructeurs multiples comme en C++ ou Java, les

constructeurs Go sont de simples fonctions – et non des méthodes – définies dans le même paquet que le type qu'elles construisent, dont elles renvoient une instance initialisée.

Pour les identifier plus facilement, plusieurs conventions existent :

✓ Pour le choix du nom, une fonction jouant le rôle de constructeur pour un type `T` est conventionnellement dénommée :
- `MakeT` si elle renvoie une instance de `T` ;
- `New` si elle renvoie un pointeur sur `T` et que ce type est le seul exporté par le paquet auquel il appartient, et que le paquet est dénommé `t` ;
- `NewT` si elle renvoie un pointeur sur `T` et que le paquet auquel il appartient définit plusieurs types exportés, auquel cas le paquet n'est normalement pas dénommé `t`.

✓ Pour le type de résultat, si un type n'existe que pour fournir une implémentation d'une interface et qu'il n'est pas exporté lui-même, le constructeur renvoie l'instance sous la forme du type interface en question plutôt que comme une instance du type concret, qui n'est pas exporté. C'est une exception à la pratique recommandée usuelle consistant à renvoyer des types concrets.
Des exemples de ce genre de situations se trouvent sur les algorithmes de hachage, où les constructeurs présents dans les paquets `hash.crc32` et `adler32` renvoient des instances de l'interface `hash.Hash32`, tout comme ceux présents dans les paquets `hash.crc64` et `hash.fnv` renvoient des instances de l'interface `hash.Hash64`.

Exercice Définir un type « ensemble de chaînes » implémentant les méthodes `Add()`, `Remove(s)`, `Contains()` et implémentant `fmt.Stringer`, en utilisant des concepts de ce chapitre : types définis, carte, structure, méthodes, interface. Cela nécessite moins de 50 lignes.

Solution https://osinet.fr/go/types/play-string-set

4.5 TYPES CANAUX (chan)

4.5.1 Définition

Les canaux (*channels*) sont des files d'attente premier entré-premier sorti (*FIFO : First In First Out*) à capacité fixe, inspirés de l'article CSP (*Communicating Sequential Processes* [12]) de C.A.R Hoare de 1978, et similaires aux canaux du langage Occam, augmentés d'un mécanisme de tamponnage et de la capacité de transmettre des canaux sur des canaux, rejoignant par là-même la version 2015 de CSP [13].

12. https://osinet.fr/go/types/hoare1978-csp.pdf
13. https://osinet.fr/go/types/hoare2015-csp

© Dunod – Toute reproduction non autorisée est un délit.

Cette capacité de transmettre des canaux sur d'autres canaux est décrite dans la documentation Go comme l'une des propriétés les plus importantes du langage [14].

Ces canaux diffèrent des canaux Erlang par le fait qu'ils sont cantonnés à l'espace mémoire d'un processus unique, au lieu d'être répartis sur tout un réseau.

Les canaux supportent deux opérations principales et synchrones : envoyer et recevoir un élément, et sont à la base du modèle de programmation concurrente (cf. chapitre 10) recommandé en Go.

Ils sont caractérisés par leur capacité, qui représente le nombre d'éléments pouvant être à la fois présents dans le canal, et par leur longueur, représentant le nombre de ces éléments effectivement présents à l'instant de la mesure.

Ils démarrent leur cycle de vie dans un état ouvert, autorisant les émissions et réceptions, et peuvent par la suite être fermés pour marquer la fin des émissions, après quoi il n'est plus possible de les réouvrir.

4.5.2 Déclaration de types canaux

Les types canaux définissent des files d'éléments d'un même type, qui est le type d'éléments du canal. L'autre élément qui les caractérise est la directionnalité : ils peuvent être limités à l'émission ou à la réception, ou autoriser les deux.

Pour apprécier si un canal est en émission ou en réception, le point de vue est toujours celui de l'expression ou instruction utilisant le canal, et pas du canal lui-même : la réception d'une donnée depuis le canal, ou l'émission d'une donnée vers le canal.

Pour un type d'éléments `T`, le type canal associé est déclaré comme suit :

- ✓ **canal bidirectionnel** : `chan T`
- ✓ **canal en émission** : `chan<- T`
- ✓ **canal en réception** : `<-chan T`

Au niveau syntaxique, pour lever une potentielle ambiguïté, la notation `<-` s'applique toujours au groupe de symboles le plus à gauche possible. Ainsi, pour un type d'éléments `T` :

- ✓ `chan<- <-chan T` est équivalent à `chan<- (<-chan T)` : un type « canal en émission d'éléments canaux en réception d'éléments de type T » ;
- ✓ `<-chan <-chan T` est équivalent à `<-chan (<-chan T)` : un type « canal en réception d'éléments canaux en réception d'éléments de type T ».

14. https://osinet.fr/go/types/go-canal-de-canaux

4.5.3 Valeurs de types canaux

De la même façon que les cartes et tranches, les valeurs de types canaux se comportent comme des pointeurs, dans la mesure où il n'est pas nécessaire de les passer par adresse pour pouvoir les modifier par les opérations d'émission et réception. La raison en est la même : la valeur canal est en réalité un pointeur opaque vers une structure, et c'est celle-ci qui est modifiée plutôt que la valeur de type canal elle-même.

C'est aussi pourquoi la valeur vierge des canaux est **`nil`**, et pourquoi il n'est pas possible de définir des constantes de type canal. C'est une valeur valide pour un canal, mais son utilité est rare, car toute émission ou réception sur un canal **`nil`** est bloquante.

À l'utilisation, un canal d'éléments de type `T` se comporte comme pourrait le faire une structure comportant un champ privé de type tranche de `T`, pourvue au niveau compilateur du sucre syntaxique renvoyant les opérations intégrées à ses méthodes de FIFO : le stockage tampon du canal expose à la fois une capacité, représentant la taille d'un tableau sous-jacent, et une longueur, représentant le nombre d'éléments émis vers le canal et pas encore reçus.

En pratique toutefois, la structure [15] est plus complexe, et présente des limitations numériques. Pour Go 1.13, ces limites sont les suivantes :

- ✓ **Taille du tampon** : limitée en nombre d'éléments : 2^{32} sur les machines 32 bits, 2^{64} sur les machines 64 bits, du fait de la définition du nombre comme une valeur de type `uint` ;
- ✓ **Taille des éléments** : limitée à 64ko, du fait de la définition de la taille d'élément comme une valeur de type `uint16`.

Ces limites ne font pas partie de la spécification du langage, et peuvent donc changer au fil des versions. En particulier, en Go 1.0, le nombre d'éléments était limité à 2^{32} indépendamment de l'architecture, et il est passé à cette taille variable dans 1.1rc2, comme conséquence implicite du changement de définition du type `uint` [16].

4.5.4 Manipulation des canaux

Les expressions et instructions manipulant des canaux sont les suivantes :

- ✓ **Comparaison** : les canaux du même type peuvent être comparés entre eux, et à la valeur vierge **`nil`**, mais uniquement par les opérateurs `==` et `!=` : aucune relation d'ordre n'est définie entre canaux.
  Pour être égaux, deux canaux, comme des pointeurs, doivent avoir été créés par la même initialisation, ou être tous les deux **`nil`**.

15. Les canaux sont définis par le type structure `hchan`, dans `src/runtime/chan.go`.
16. https://osinet.fr/go/types/go-taille-int

© Dunod – Toute reproduction non autorisée est un délit.

✓ **Conversion** : hormis le cas trivial de la conversion entre types alias, la conversion entre valeurs de types canaux nécessite à la fois que les éléments soient de types compatibles et que les canaux soient de même nature (bidirectionnel ou même direction) ou de bidirectionnel vers unidirectionnel.
Le tableau 4.1 résume les possibilités de conversion entre canaux, avec comme conventions :

- ✓ tous les types de canaux listés ont le même type d'éléments ;
- ✓ `DB1` et `DB2` sont des types définis de canaux bidirectionnels ;
- ✓ `DR1`, `DR2`, `DE1`, `DE2` sont des types définis de canaux unidirectionnels, respectivement en réception et en émission ;
- ✓ `AB`, `AR` et `AE` sont des types anonymes de canaux respectivement bidirectionnel et unidirectionnels en réception et émission ;
- ✓ « dir. » indique que les données sont directement utilisables sans conversion, étant du même type ;
- ✓ « impl. » indique qu'une conversion est opérée implicitement par le compilateur lors d'une affectation ou d'un passage d'argument à une fonction ;
- ✓ « expl. » indique qu'une conversion explicite dans le code source est nécessaire pour pouvoir opérer une affectation ou un passage d'argument à une fonction.

Tableau 4.1 – Conversions entre types canaux

De - Vers	DB1	DB2	DR1	DR2	DE1	DE2	AB	AR	AE
DB1	dir.	expl.					impl.	impl.	impl.
DR1			dir.	expl.				impl.	
DE1					dir.	expl.			impli.
AB	impl.	impl.	impl.	impl.	impl.	impl.	dir.	impl.	impl.
AR			impl.	impl.				dir.	
AE					impl.	impl.			dir.

La conversion est examinée dans le cas général au § 6.12.

✓ **Adresse** : l'opérateur adresse `&` est applicable aux canaux, mais en pratique il est de peu d'utilité : il n'est pas nécessaire de passer un pointeur sur canal à une fonction pour que celle-ci puisse émettre ou recevoir dessus, et il n'est pas nécessaire de comparer les pointeurs sur deux canaux pour vérifier qu'ils proviennent de la même initialisation.
À la différence du traitement des appels méthodes, il n'y a pas de conversion automatique entre pointeur et valeur pour les canaux dans les opérations d'émission et réception.

Il est donc généralement préférable d'utiliser directement les valeurs canal plutôt que leurs pointeurs, bien qu'il existe des situations en programmation concurrente où l'utilisation du pointeur sur canal puisse faire sens [17].

✓ **Émission** : l'émission d'une valeur `t` de type `T` sur un canal `c` de type `chan T` est notée par `c <- t`. Le formatage standard inclut les espaces de part et d'autre de l'opérateur `<-` pour cette opération.
L'émission n'est possible que sur un canal qui n'a pas encore été fermé : tenter d'émettre une valeur sur un canal fermé déclenche une panique.
L'émission est une opération synchrone, qui revient immédiatement si le canal recevant la valeur émise dispose de la capacité pour stocker l'élément dans son tampon, ou qui bloque l'exécution de la goroutine en cours jusqu'à ce qu'une autre goroutine (unité d'exécution, cf. § 10.2) exécute une réception sur le même canal, libérant du même coup un emplacement pour la valeur émise si le canal a une capacité non nulle, ou recevant directement la valeur émise si cette capacité est nulle. Par cohérence, l'émission sur un canal `nil` est également bloquante.
En cas de blocage, l'ordonnanceur interne de Go transfère le contrôle à une autre goroutine. Si aucune autre goroutine n'est prête à l'exécution, une panique d'exécution est émise.

✓ **Réception** : la réception est le pendant de l'émission : elle extrait une valeur du canal concerné. Pour affecter à une variable `v` de type `T` le résultat d'une réception depuis un canal `c` de type `chan T`, la notation est `v = <-c` ou, de préférence, une affectation parallèle comme `v, ok = <-c`. Le formatage standard n'inclut pas d'espace à droite de l'opérateur `<-` pour cette opération, à la différence de l'émission.
Au contraire de l'émission, elle fonctionne aussi sur un canal fermé.
Comme l'émission, la réception est une opération synchrone, qui revient immédiatement avec une paire de valeurs si :

- le canal utilisé est ouvert, de capacité non nulle, et une goroutine était bloquée sur une émission : en ce cas, la valeur reçue est valide et accompagnée d'un `true` ;
- le canal utilisé est ouvert ou fermé, de capacité non nulle, et contient au moins une valeur : en ce cas, la valeur reçue est valide et accompagnée d'un `true` ;
- le canal est fermé, et est soit de capacité nulle, soit de capacité non nulle mais ne contenant plus aucun élément : en ce cas, la valeur reçue est la valeur vierge du type d'éléments du canal et accompagnée d'un `false`.

  Elle est bloquante dans tous les autres cas. Si elle est appliquée à un canal `nil`, le cas du retour non bloquant des canaux fermés ne s'applique pas, un canal devant être initialisé pour pouvoir être fermé.

17. Exemple d'utilisation effective de canaux `nil` : https://osinet.fr/go/types/canaux-nil

© Dunod – Toute reproduction non autorisée est un délit.

✓ **Fonctions `len` et `cap`** : les fonctions intégrées (*builtins*) de longueur et capacité s'appliquent aux canaux.
La capacité est constante et correspond à la valeur utilisée à la définition du canal : elle définit la limite supérieure du nombre d'éléments qui pourront être émis vers le canal sans être reçus ni causer un blocage. Elle est nulle pour un canal **`nil`**.
Le longueur correspond au nombre d'éléments présents dans le canal à l'instant de la mesure. Elle est nulle pour un canal **`nil`**.

Danger : concurrence

Les canaux n'implémentant aucun mécanisme de type transactionnel, la mesure d'une longueur `n` n'implique pas qu'il soit possible de recevoir au moins `n` éléments : d'autres goroutines peuvent s'exécuter concurremment, et recevoir des éléments, comme dans le listing 4.16, où la goroutine `g1` ne reçoit pas autant d'éléments que le longueur du canal ne pourrait le laisser penser, du fait d'une situation de compétition (*race condition*) avec la goroutine `g2`.

Ces fonctions sont détaillées, pour tous les types applicables, au § 8.4.

✓ **Fonction `close`** : la fonction intégrée `close` bascule un canal de son état initial ouvert à l'état fermé.
Cette fonction n'est pas idempotente : l'invoquer sur un canal fermé provoque une panique. Un canal **`nil`** n'étant pas initialisé n'est pas ouvert, l'utilisation de `close` sur un tel canal provoque donc une panique.
Une fois fermé, un canal écoule les données qu'il peut encore contenir, et les réceptions ont lieu sans bloquer, avec indication d'échec, mais il n'accepte plus de nouvelles émissions, comme décrit ci-dessus à propos de ces deux opérations.

✓ **Mot-clef `range`** : l'itérateur générique de Go s'applique aux canaux sous la forme :

```
for éléments := range canal
```

Il exécute une réception, et renvoie le premier élément de la paire reçue, tout en consommant le second élément pour déterminer la fin de l'itération.

Listing 4.16 – Longueur d'un canal en situation de compétition
https://osinet.fr/go/types/play-longueur-canal

```
package main

import (
        "fmt"
        "time"
)

func g1(c chan int) {
        l := len(c)
        fmt.Printf("Longueur: %d, capacité: %d\n", l, cap(c))
        time.Sleep(50 * time.Millisecond)

        // Peut-on recevoir "l" éléments ?
        for i := 0; i < l; i++ {
                v, ok := <-c
                if !ok {
                        break
                }
                fmt.Println(v)
        }
}

func g2(c chan int, fin chan bool) {
        time.Sleep(10 * time.Millisecond)
        for i := 0; i < 5; i++ {
                <-c
        }
        close(c)
        time.Sleep(100 * time.Millisecond)
        fin <- true
}

func main() {
        c := make(chan int, 10)
        fing2 := make(chan bool)

        for i := 0; i < 10; i++ {
                c <- i
        }
        go g1(c)
        go g2(c, fing2)
        <-fing2 // Ne pas finir avant la fin de g2.
}

// Résultat: g1 reçoit moins de 10 éléments.
```

© Dunod – Toute reproduction non autorisée est un délit.

Listing 4.17 – La fermeture d'un canal comme mode de diffusion
https://osinet.fr/go/types/play-fermeture-canal

```
type vide struct{}

func client(rang int, barrière, fin chan vide) {
        att := time.Duration(rand.Intn(1000)) * time.Millisecond
        time.Sleep(att)
        fmt.Printf("Client %d à la barrière\n", rang)
        <-barrière
        fmt.Printf("Client %d passé\n", rang)
        fin <- vide{}
}

func main() {
        const limite = 3
        barrière, fin := make(chan vide), make(chan vide)
        for i := 0; i < limite; i++ {
                go client(i, barrière, fin)
        }
        time.Sleep(limite * time.Second)
        close(barrière)
        libérés := 0
        for _ = range fin {
                libérés++
                if libérés == limite {
                        close(fin)
                }
        }
}
```

Danger : réception bloquante

La réception étant bloquante sur les canaux ouverts, si une telle itération est utilisée sur un canal non fermé, elle bloquera une fois le dernier élément reçu : la méthode la plus simple consiste à n'utiliser cette itération que sur un canal fermé, mais il est aussi possible de réaliser la fermeture dans le bloc itéré, comme dans le listing 4.17.

Dans cet exemple, la réception dans les clients est bloquée sur le canal `barrière`, et les clients arrivent successivement à intervalles pseudo-aléatoires[18].

Lorsque la fonction `main` ferme le canal `barrière`, toutes ces réceptions sont débloquées : la fermeture a fourni un moyen de diffuser le signal à tous les clients en attente sans devoir itérer sur une liste maintenue dans `main`. Ce mécanisme de diffusion par fermeture est central dans le fonctionnement des « contextes » (cf. § 10.3) en programmation concurrente.

La fonction `main` reçoit leur signal de fin sur le canal éponyme, et ferme une fois le nombre total de clients passés obtenu, pour permettre à l'itération de sortir naturellement.

18. Exemple inspiré d'une présentation d'Andrew Gerrand : https://osinet.fr/go/types/go-fermeture-canal

Enfin, pour définir des variables de types canaux, il n'existe pas de littéraux ; pour un type d'élément `T`, l'initialisation d'un canal de `chan T` est possible par :

- ✓ **Expression** : toute expression renvoyant un canal peut être utilisée pour initialiser une variable canal du même type.
- ✓ **Fonction `make` à un argument** : `canal := make(chan T)`, ou l'équivalent pour les canaux unidirectionnels.
  Dans cette initialisation, le canal est créé avec une capacité 0 et initialisé à l'état ouvert.
- ✓ **Fonction `make` à deux arguments** : `canal := make(chan T, c)`, ou l'équivalent pour les canaux unidirectionnels.
  Dans cette initialisation, le canal est créé avec une capacité de `c` éléments, et initialisé à l'état ouvert.
- ✓ **Fonction `new`** : `canal := new(chan T)`, ou l'équivalent pour les canaux unidirectionnels.
  Dans cette initialisation rarement utilisée, la fonction intégrée `new()` renvoie un pointeur sur un canal `nil`, toujours bloquant et ne pouvant être fermé.
  Tous les canaux ainsi initialisés sont égaux entre eux.

Plus généralement, l'association canaux + goroutines est au cœur du modèle de programmation concurrente promu par Go, qui fait l'objet du chapitre 10.

4.5.5 Extensions

La limitation des canaux à l'espace d'un processus est problématique pour les applications souhaitant répartir une charge de travail au-delà d'une machine unique, à l'image des possibilités d'Erlang, tout en conservant une logique à base de canaux plutôt que de devoir restructurer son modèle sur une file d'attente classique.

Deux extensions au moins existent en Open Source pour répondre à ce besoin :

- ✓ Le paquet `libchan` de Docker, disponible depuis 2014 et utilisé en production par Docker, avec un soutien intégré pour plusieurs transports de bas niveau : canaux Go, socket UNIX, TCP brut, TLS, HTTP2/SPDY, et Websocket.
  Github : https://osinet.fr/go/types/gh-docker-libchan
- ✓ Le paquet `vice` de Mat Ryer, plus récent et plus idiomatique, avec un soutien intégré pour plusieurs transports de type file d'attente : NATS, NSQ, RabbitMQ, Redis, et Amazon SQS.
  Github : https://osinet.fr/go/types/gh-matryer-vice

5 Données

5.1 VARIABLES

5.1.1 Variables et types

Comme dans la plupart des langages, une variable Go est un espace de stockage contenant une valeur. Le domaine des valeurs que peut adopter une variable constitue le type [1] de celle-ci. Il est défini, en tant que *type statique* de la variable :

- ✓ lors de la déclaration d'une variable, d'un élément d'une variable structurée tel un champ de structure ou un élément de tableau ;
- ✓ lors d'une allocation dynamique par l'allocateur générique `new` ;
- ✓ lors de la définition d'un littéral composite.

Si c'est un type concret, c'est le seul que connaîtra la variable, et il définit directement son stockage et ses méthodes. Mais Go permet également d'instancier des variables d'un type **`interface`** [2], au lieu de se limiter à la déclaration de variables de types concrets implémentant des interfaces.

Comme une variable doit contenir des données et non simplement respecter un contrat d'interface, il est nécessaire de décrire le stockage effectif de la valeur désignée par la variable, et pour cela le type statique défini par la déclaration est complété à l'exécution par un *type dynamique*, qui est le type concret implémentant l'interface spécifiée par le type statique de la variable.

1. Le fonctionnement du système de types est décrit dans la suite de ce chapitre, à partir du § 5.3.
2. Les interfaces sont décrites au § 3.5.

C'est ce qu'illustre le listing 5.1 :

- ✓ À la ligne 10, la variable `s` est déclarée comme étant de type statique **`error`**, type **`interface`** défini nativement par le langage comme implémentant une méthode `Error()` renvoyant une chaîne : https://osinet.fr/go/data/error.
- ✓ À la ligne 12, `s` est initialisée par la fonction fabrique `errors.New`, et contient dès lors une donnée d'un type dynamique implémentant le contrat d'interface d'**`error`**, mais inconnu.
- ✓ L'impression à la ligne 13 révèle ce type dynamique : `*errors.errorString`.

Listing 5.1 – Type statique et type dynamique
https://osinet.fr/go/data/play-statique-dynamique

```
1  package main
2
3  import (
4          "errors"
5          "fmt"
6  )
7
8  func main() {
9          // Voir https://godoc.org/builtin#error
10         var s error
11
12         s = errors.New("Hello, playground")
13         fmt.Printf("Type dynamique: %T\nValeur: %s\n", s, s)
14 }
15
16 // Résultat:
17 // Type dynamique: *errors.errorString
18 // Valeur: Hello, playground
```

Cet exemple reflète le fait que ce mécanisme de types statique et dynamique permet de déclarer et manipuler des variables définies uniquement par un contrat d'interface, mécanisme essentiel pour supporter la substitution de Liskov de l'approche SOLID (cf. note 12 page 8).

5.1.2 Déclaration des variables

Le mot-clef utilisé pour définir une variable est **`var`**, suivi d'un ou plusieurs identificateurs de variables, puis du type commun à ces variables, sans séparateur autre que l'espace entre l'identificateur de la variable et la désignation du type :

```
var s, t string
```

Une variable ne peut être déclarée par **`var`** qu'une seule fois dans une portée.

Après une déclaration de cette forme, les variables sont déclarées, et définies avec la valeur vierge [3] du type statique de la variable. Dans le cas d'un type **`interface`**, en l'absence d'un type concret, la valeur est initialisée à la valeur spéciale **`nil`** [4].

Il est aussi possible de combiner déclaration et initialisation des variables en ajoutant une instruction d'affectation sur la même ligne. S'il y a plusieurs variables, il faut le même nombre de valeurs pour une affectation parallèle :

```
var s, t string = "Bonjour", "Bonsoir"
```

Dans cette situation d'initialisation, le type des variables peut souvent être déduit du type des valeurs utilisées pour initialiser les variables, qui est **`string`** dans l'exemple ci-dessus. De ce fait, Go permet de ne pas définir le type de la variable : il est automatiquement défini par l'initialiseur et peut différer pour chaque variable dans une même initialisation :

```
var s, n = "Bonjour, numéro", 6
```

Il est possible de grouper plusieurs déclarations de variables en les répartissant sur plusieurs lignes avec des parenthèses, comme dans le listing 5.2.

Listing 5.2 – Déclaration `var` multilignes
https://osinet.fr/go/data/play-var-multi

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var (
7                  s, t    = "Bonjour", "Bonsoir"
8                  i, j, k = 1, 3, 4
9          )
10         fmt.Println(s, t, i, j, k)
11 }
12
13 // Résultat: Bonjour Bonsoir 1 3 4
```

Enfin, pour simplifier encore la déclaration et la définition des variables, Go dispose d'une déclaration courte (*short variable declaration*), équivalente à l'initialisation par **`var`** avec initialisation :

```
s, t := "Bonjour", "Bonsoir"
```

3. La notion de valeur vierge est détaillée au § 5.4.
4. **`nil`** est défini et détaillé au § 5.5.

© Dunod – Toute reproduction non autorisée est un délit.

Cette déclaration courte est une instruction dite *simple*, dont les particularités sont décrites au § 7.2.6. Dans la majorité des cas, elle est préférée aux autres formes de déclaration de variables du fait de sa lisibilité et sa simplicité d'utilisation.

5.1.3 Utilisation des variables

Pour encourager l'hygiène du code, la syntaxe de Go interdit les variables non utilisées : une variable utilisée n'est pas simplement la cause d'un avertissement, mais déclenche une erreur de compilation.

C'est ce qu'illustre le listing 5.3 : l'utilisation en écriture à la ligne 9 de la variable `n` ne compte pas comme une utilisation : seules comptent les utilisations en lecture : utilisation dans une expression, modification par un opérateur d'incrémentation/décrémentation, passage à une fonction comme paramètre, etc.

Listing 5.3 – Utilisation obligatoire des variables
https://osinet.fr/go/data/play-usage-variables

```
1 package main
2
3 func main() {
4         var n int
5
6         n = 29
7 }
8
9 // Erreur de compilation: "n declared and not used"
```

5.1.4 Quasi-variables : paramètres et retours

Go permet de déclarer des fonctions avec des retours nommés et potentiellement multiples [5], avec une syntaxe de la forme :

```
func foo(p1 type1, p2 type2) (r1 type3, r2 type4) { /* ... */ }
```

Dans ces contextes de fonction, les paramètres nommés, tout comme les retours nommés, sont équivalents à des variables, bien qu'ils ne soient pas déclarés par une instruction **`var`** ou une déclaration courte.

Le cas des paramètres est classique, et ceux-ci ont toujours une valeur, définie par l'appelant de la fonction, mais le cas des retours nommés est plus inhabituel : du fait de ce statut de quasi-variable, les retours nommés sont initialisés comme des variables, donnant lieu à une allocation mémoire définie par le type, et à une initialisation à la valeur vierge du type statique.

5. Les retours de fonction sont détaillés au § 7.4.1.

Comme pour une vraie variable, si le type statique est une **interface**, l'initialisation a lieu à la valeur **nil**, jusqu'à ce que la logique de la fonction fournisse une valeur concrète permettant de définir le type dynamique.

C'est ce qu'illustre le listing 5.4, qui montre les valeurs par défaut pour un type interface (**error**) et un type concret (**string**) :

Listing 5.4 – Valeur par défaut des retours nommés
https://osinet.fr/go/data/play-retours-nommes

```
1  package main
2
3  import "fmt"
4
5  func foo() (err error, s string) {
6          fmt.Printf("Err: %#v, S: %#v\n", err, s)
7          return
8  }
9
10 func main() {
11         foo()
12 }
13 // Résultat: Err: <nil>, S: ""
```

5.1.5 Variables anonymes

Enfin, d'autres données sans identificateur propre se comportent comme des variables :

- ✓ Les éléments des tableaux et tranches, ainsi que les champs des types structure peuvent être pointés par l'opérateur d'adresse (cf. § 2.4.8) et sont manipulés comme des variables.
- ✓ Lorsque l'opérateur adresse est utilisé pour prendre l'adresse d'un littéral composite, une variable anonyme est créée pour que l'opérateur puisse obtenir son adresse.
- ✓ L'utilisation de l'allocateur générique `new` alloue et initialise une variable anonyme dont il renvoie l'adresse.

Dans tous ces cas, les variables ainsi obtenues sont manipulées par indirection du pointeur vers la variable anonyme.

© Dunod – Toute reproduction non autorisée est un délit.

5.2 CONSTANTES

5.2.1 Constantes Go et autres langages

La définition des constantes dans la spécification de Go est complexe, mais cette complexité découle pour l'essentiel d'une différence d'approche avec les langages de programmation courants, dont les constantes sont définies essentiellement comme des variables à ne pas réassigner :

- ✓ Python ne les définit pas, hormis les constantes natives du langage lui-même [6], et s'appuie uniquement pour les constantes utilisateur sur des conventions de notation, comme la casse.
- ✓ JavaScript, dans sa version ECMAScript 6, le précise explicitement en introduisant le mot-clef `const` : *... const declarations define variables...* [7].
- ✓ PHP définit les constantes globales par `define` [8] à partir d'expressions quelconques.

À l'inverse de cette approche, les constantes Go sont des données évaluées lors de la compilation, soit directement à partir du langage lui-même comme les constantes internes **`true`** ou **`false`**, soit par la valeur d'une expression constante, donc dans tous les cas sans requérir une évaluation à l'exécution. L'opérateur d'adresse ne leur est pas applicable. C'est une approche comparable aux constantes de classes PHP [9].

Conserver cette interprétation à l'esprit permet de mieux comprendre les règles qui en découlent, tant pour les constantes que pour les valeurs constantes, et sont détaillées dans les paragraphes suivants.

Outre la spécification du langage, sur laquelle est basée cette section, les auteurs de Go ont documenté les raisons de leurs choix sur le blogue Go, à la page :

https://osinet.fr/go/data/go-const

5.2.2 Valeurs constantes

Les valeurs constantes sont de plusieurs natures, et évaluées à partir de diverses représentations dans le code source :

- ✓ **Identificateur** : l'identificateur d'une constante déjà définie en amont, ou l'un des identificateurs définis par le langage.
- ✓ **Littéral** : la grammaire de Go permet de définir des littéraux pour chaque nature de donnée pouvant être constante, comme dans le tableau 5.1.

6. https://osinet.fr/go/data/constantes-python
7. Dans ECMA-262 9th Edition / June 2018, § 13.3.1 : https://osinet.fr/go/data/constantes-es6
8. https://osinet.fr/go/data/constantes-php-define
9. https://osinet.fr/go/data/constantes-php-class

✓ **Expression constante** : un sous-ensemble des expressions est applicable à des valeurs constantes et renvoie des valeurs constantes :

- les expressions composées d'opérandes constants et de tous les opérateurs sauf ceux d'adresse et de réception ;
- les conversions de types appliquées à des valeurs et expressions constantes, comme `const n = int(3.0)` ;
- l'appel de certaines fonctions intégrées au langage :
  - celles liées aux valeurs complexes `real`, `imag`, `complex`,
  - la fonction de longueur `len` pour les chaînes,
  - les fonctions de longueur et capacités `len` et `cap` appliquées à des tableaux et pointeurs sur tableaux [10], comme dans l'exemple 5.5,
  - la fonction `unsafe.Sizeof` de calcul d'occupation mémoire [11].

Tableau 5.1 – Natures de constantes et littéraux

Nature de valeur	Identificateurs et exemples de littéraux
booléens	identificateurs natifs : **`true`**, **`false`**
rune	`a`, `\U0001F600`
entiers	-3, 11, identificateur natif **`iota`**
flottants	`0.25`, `2.E196`, mais pas `-0`, `NaN`, `+Inf`, ni `-Inf`
complexes	imaginaires purs : `i`, `2.3e-5i`
chaîne	`"Dunod"`, `"\U0001F1EB\U0001F1F7"`

Trois points méritent d'être soulignés dans ce tableau :

✓ **Valeurs flottantes** : les valeurs constantes sont des valeurs exactes, de précision infinie ; pour cette raison, les valeurs de types flottantes IEEE-754 qui ne représentent pas de véritables nombres – `NaN`, les deux infinis et le zéro négatif – ne peuvent être utilisés pour initialiser une constante.

✓ **Zéro négatif** : bien que le zéro négatif [12] ne puisse être utilisé pour initialiser une constante flottante, une initialisation par `-0.0` demeure valide ; en effet, en ce cas, l'initialisation est compilée comme une expression constante de l'opérateur monadique – avec l'opérande flottant `0.0` et a pour valeur le zéro flottant usuel.

✓ **Valeurs complexes** : seul un nombre imaginaire pur peut être représenté par un littéral ; une valeur complexe avec une partie réelle est valide, mais traitée par le langage comme une expressions constante ajoutant un flottant et un imaginaire pur, et non comme un littéral complexe.

10. Comment Go calcule-t-il ces valeurs constantes, alors que les tableaux ne sont pas des valeurs constantes ? En fait, le compilateur n'a pas besoin du contenu du tableau concerné, mais uniquement de son type, qui est bel et bien constant, comme dans l'exemple 5.5.
11. https://osinet.fr/go/data/go-sizeof
12. https://osinet.fr/go/data/wiki-zero-negatif

© Dunod – Toute reproduction non autorisée est un délit.

Listing 5.5 – Expressions constantes avec `len` et `cap`
https://osinet.fr/go/data/play-constantes-len-cap

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          x := [5]int{2, 9}
7          const a, b, c, d = len(x), cap(x), len(&x), cap(&x)
8          fmt.Println(a, b, c, d, x)
9  }
10
11 // Résultat: 5 5 5 5 [2 9 0 0 0]
```

5.2.3 Constantes et types

Alors que les valeurs constantes sont d'une nature définie – par exemple « entier » – elles peuvent être typées – par exemple **`uint32`** – ou non typées, comme défini par le tableau 5.2 :

Tableau 5.2 – Constantes typées et non typées

Valeur	Origine
non typée	littéraux, **`true`**, **`false`**, **`iota`**, déclarations non typées, expressions constantes composées d'opérandes non typés
typée explicitement	déclaration typée, conversion
typée implicitement	déclaration initialisée ou affectation à une variable opérande d'une expression non constante

Les constantes décrites ici non typées ont un *type par défaut* dans lequel elles sont converties lorsqu'un type est nécessaire mais non disponible. C'est par exemple le cas pour les déclarations courtes de la forme `v := 31` : en pareil cas, `31` est un littéral, donc une valeur constante non typée, et comme il se retrouve affecté à l'initialisation d'une variable elle aussi non typée, le langage doit choisir un type, puisque les variables sont toujours typées.

Le type utilisé est alors le type par défaut appliqué à la valeur.

Le tableau 5.3 représente les diverses natures de valeurs constantes et leur type par défaut.

Tableau 5.3 – Types par défaut des constantes

Nature	Type par défaut
booléens	`bool`
rune	`rune`
entiers	`int`
flottants	`float64`
complexes	`complex128`
chaîne	`string`

5.2.4 Précision des valeurs constantes

Une particularité des constantes Go est la précision : les expressions constantes sont évaluées de manière à aboutir à des valeurs exactes de la nature de valeur, aussi bien pour les littéraux que pour les éventuelles valeurs intermédiaires.

Cela signifie en particulier que des constantes peuvent être déclarées avec des valeurs situées hors du domaine de définition de tous les types du langage, comme dans le listing 5.6, qui démontre l'utilisation d'une constante entière de 101 bits, très au-dessus de la plus grande valeur positive d'une variable entière `uint64`, qui est de $2^{64} - 1$, alors que ce listing manipule une valeur $2^{100} + 1$.

Listing 5.6 – Constantes entières de plus de 64 bits
https://osinet.fr/go/data/play-constantes-larges

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          const Huge = 1<<100 + 1
7          const hi uint64 = Huge >> 64
8          const lo uint64 = Huge & 0xffffffffffffffff
9
10         fmt.Printf("%016x %016x\n", hi, lo)
11 }
12
13 // Résultat: 0000001000000000 0000000000000001
```

Examinons les étapes de ce programme :

- ✓ Ligne 6 : la constante non typée `Huge` est initialisée avec la valeur de l'expression constante `1<<100 + 1`, soit $2^{100} + 1$. Cette valeur nécessiterait un type `int128` si elle devait être typée, mais en l'absence d'un type, le compilateur peut la manipuler normalement.
- ✓ Lignes 7 et 8, deux autres constantes typées `uint64` sont initialisées avec respectivement les 64 bits de poids fort et de poids faible de `Huge`. Puisque ce sont

© Dunod – Toute reproduction non autorisée est un délit.

des constantes typées, les expressions constantes qui les initialisent doivent respecter le type de destination, ce qui est bien le cas.

✓ Lignes 10 et 13, l'affichage démontre que le calcul a bien été effectué sans perte de précision.

Ce type de calcul sur les constantes avec une précision étendue par rapport aux calculs sur les valeurs de variables est possible car les expressions constantes sont évaluées durant la compilation, et peuvent faire appel à du calcul en précision étendue [13] en s'appuyant sur le paquet `math/big` également disponible pour les programmes utilisateurs.

Dans l'absolu, rien n'empêche ces formules de requérir une précision arbitraire, mais les compilateurs réels ont toujours des limites concrètes. Afin que celles-ci ne soient pas trop limitatives, la spécification du langage précise les exigences relatives aux valeurs constantes que les compilateurs conformes doivent pouvoir traiter, aussi bien pour les littéraux que lors de l'évaluation des expressions constantes :

✓ Traiter correctement au moins 256 bits pour les entiers.

✓ Traiter correctement au moins 256 bits de mantisse et 16 bits d'exposant signé pour les flottants, y compris les parties réelle et imaginaire des valeurs complexes.

✓ Détecter et signaler les erreurs au cas où une valeur entière ne peut être représentée par l'implémentation.

✓ Détecter et signaler les erreurs résultant de l'incapacité de l'implémentation à représenter une valeur flottante en raison d'un dépassement de capacité.

✓ Arrondir une valeur flottante ou complexe à la constante la plus proche que l'implémentation peut représenter compte tenu de ses limites de précision.

5.2.5 Déclaration des constantes

5.2.5.1 Le mot-clef `const`

Les valeurs constantes peuvent être affectées à des constantes déclarées avec le mot-clef **`const`**, comme déjà illustré dans les listings 5.5 et 5.6.

Une déclaration de constante se compose d'un ou plusieurs identificateurs, du signe = et de valeurs constantes comme définies dans les paragraphes précédents. En pareil cas, le type des constantes déclarées demeure indéfini, et l'identificateur de chaque constante propage la valeur constante en précision étendue utilisée pour son initialisation.

Il est aussi possible de définir explicitement le type de ces constantes. En ce cas, la valeur constante de précision étendue est convertie dans le type spécifié, et peut donner lieu à une erreur de compilation si la valeur dépasse l'intervalle défini par le type, par exemple `const x byte = 150`.

13. Le paquet source réalisant ces évaluations est `go/constants`.

Un autre format plus compact regroupe, comme pour les variables, des identificateurs sur une même instruction **const** sans parenthèses, comme dans le listing 5.7, pour une initialisation parallèle. Dans ce cas, les constantes sont définies comme s'il s'agissait d'instructions indépendantes : si un type est défini, il s'applique à toutes les constantes, comme à la ligne 4 ; sinon chacune est une constante indépendante non typée, potentiellement de nature différente des autres de la même ligne, comme à la ligne 5.

Listing 5.7 – Constantes à initialisation parallèle
https://osinet.fr/go/data/play-init-parallele

```
1  package main
2
3  func main() {
4          const a, b int = 5, 8.0
5          const c, d = 6, "go"
6          println(a, b, c, d)
7  }
8
9  // Résultat: 5 8 6 go
```

Enfin, que le type soit défini ou non, une liste d'identificateurs de constantes peut être déclarée avec une seule instruction **const**, et un bloc de lignes entre parenthèses comportant un ou plusieurs identificateurs par ligne et un nombre potentiellement plus réduit de valeurs constantes affectées, avec des types potentiellement différents, comme dans le listing 5.8, à l'image de la déclaration groupée décrite pour les variables avec l'exemple 5.2 page 149.

Listing 5.8 – Constantes à valeur répétée
https://osinet.fr/go/data/play-init-repetee

```
1  package main
2
3  func main() {
4          const (
5                  a, b = 5, 3
6                  c, d
7                  e, f
8                  g float64 = 8
9                  h
10         )
11
12         println(a, b, c, d, e, f, g, h)
13 }
14
15 // Résultat: 5 3 5 3 5 3 +8.000000e+000 +8.000000e+000
```

En pareil cas, sur les lignes ne comportant pas de valeur définie, l'identificateur est associé à la valeur et au type définis sur la ligne précédente, que ceux-ci soient

© Dunod – Toute reproduction non autorisée est un délit.

explicites comme aux lignes 8 et 9, ou résultant d'une précédente affectation par cette même règle, comme à la ligne 7.

Comme l'illustre le listing 5.8 :

✓ Le nombre de valeurs sur une ligne doit être soit zéro, soit égal au nombre d'identificateurs.

✓ Le nombre d'identificateurs sur une ligne sans valeurs explicites doit être le même que celui de la ligne précédente.

5.2.5.2 L'identificateur `iota`

Le format de déclarations groupées en liste décrit en dernier au paragraphe précédent peut être associé avec un identificateur interne de Go : **`iota`**.

Celui-ci est évalué comme une expression constante dont la valeur, comme le montre le listing 5.9 :

✓ est constante sur une ligne d'affectation donnée ;

✓ augmente de 1 à chaque ligne d'affectation, mais pas à chaque constante.

Listing 5.9 – L'identificateur **`iota`**
https://osinet.fr/go/data/play-iota

```
1  package main
2
3  func main() {
4          const (
5                  a, b = iota, iota
6                  c, d
7                  e, f, g = 5, 8, 13
8                  h       = iota
9                  i
10         )
11         const (
12                 j = iota
13         )
14         println(a, b, c, d, e, f, g, h, i, j)
15 }
16
17 // Résultat: 0 0 1 1 5 8 13 3 4 0
```

✓ **Ligne 5** : initialisation parallèle de deux constantes non typées. Comme les deux **`iota`** font partie de la même instruction d'affectation, ils ont la même valeur initiale : zéro.

✓ **Ligne 6** : initialisation parallèle réutilisant l'expression de la ligne précédente. Comme il s'agit d'une nouvelle instruction d'affectation, **`iota`** est incrémenté. Comme **`iota`** ne dépend que du nombre d'instructions d'affectation et non du

nombre de constantes initialisées dans une instruction, `iota` n'est incrémenté que de 1 bien que l'instruction précédente ait initialisé deux constantes.

- ✓ **Ligne 7** : initialisation d'un nombre de constantes différent du groupe précédent.
- ✓ **Ligne 8** : initialisation à partir d'un `iota` d'un nombre de constantes différent du groupe précédent. Comme il s'agit du même groupe parenthésé, `iota` reste dans la même séquence, mais est incrémenté de 2 et non de 1, pour prendre en compte la ligne 7.
- ✓ **Ligne 9** : initialisation réutilisant le nouveau iota.
- ✓ **Ligne 12** : initialisation d'une constante iota. Comme il s'agit d'un nouveau groupe parenthésé, `iota` recommence à zéro.

Si l'initialisation `iota` se limitait à des valeurs constantes, ce serait déjà un élément utile pour la génération de séquences de constantes, mais le mécanisme `const` + `iota` va plus loin, en permettant à la valeur constante utilisée sur les lignes sans valeur explicite d'être une expression, évaluée à chaque ligne [14].

Avec l'incrémentation automatique de `iota`, elle permet d'utiliser des expressions pour produire des séquences de constantes variant autrement que par un simple incrément de 1, comme l'illustre le listing 5.10

5.3 VALEURS ET TYPES

Les types en Go sont la combinaison de l'ensemble des valeurs et des actions disponibles pour les variables et constantes manipulées par les programmes, qu'il s'agisse d'opérateurs, de fonctions intégrées ou non pouvant les manipuler, ou de méthodes disponibles [15].

Le cas des constantes non typées et de leurs expressions se situe en-dehors de ce cadre, l'évaluation correspondante étant réalisée lors de la compilation et non à l'exécution, comme décrit au § 5.2.3.

De la même façon que des valeurs littérales ne sont pas forcément associées à une déclaration de variable ou constante, tous les types Go ne sont pas nécessairement associés à un identificateur de type, mais peuvent être définis par un littéral définissant un type pour l'utiliser immédiatement sans jamais le nommer.

Les types non anonymes sont déclarés par le mot-clef `type`, un identificateur, puis :

- ✓ Soit une définition de type, qui peut être élaborée ou se résumer à l'identificateur d'un type existant, créant un nouveau *type défini* dont le type existant est le type sous-jacent (*underlying type*).

14. Cette capacité existe aussi pour les affectations groupée sans `iota`, mais n'a pas d'intérêt spécifique, répétant la même valeur, qu'il est plus lisible d'exprimer en clair.
15. La définition de méthodes sur des types définis est décrite au § 3.4.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 5.10 – Expressions avec `iota`
https://osinet.fr/go/data/play-iota-expressions

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          const (
7                  a = 1 << 2
8                  b
9                  c = 1 + 1<<iota
10                 d
11                 e
12         )
13         fmt.Printf("%08b\n%08b\n%08b\n%08b\n%08b\n",
14                 a, b, c, d, e,
15         )
16 }
17
18 // Résultat:
19 // 00000100
20 // 00000100
21 // 00000101
22 // 00001001
23 // 00010001
```

Le nouveau type a la même structure de données que le type sous-jacent, mais n'hérite pas de ses méthodes.

✓ Soit le signe = et l'identificateur d'un type existant, créant un *alias* d'un type existant ; le nouveau type est identique au précédent hormis son nom.
Cette notion a été introduite en Go 1.9.

Comme pour les déclarations de variables et constantes, il est possible de regrouper les déclarations de type avec un seul mot-clef `type` et les diverses lignes de définition de types regroupées entre parenthèses.

Les types définis sont tous différents et les valeurs de deux types définis différents ne sont pas comparables ou affectables sans conversion explicite de types, détaillée au § 6.12.

Le listing 5.11 illustre les différences entre ces deux formes de déclarations :

✓ **Lignes 8-10** : Il est possible de définir des méthodes sur `T`, car c'est un type défini du programme. Inversement, tenter de définir une méthode sur `S` provoquerait une erreur, car `S` n'est qu'un alias du type natif `string`, qu'il n'est pas possible de modifier dans un programme.

✓ **Comparaisons, lignes 20 et 24** : bien que `s1` et `s2` soient nominalement de types différents, il est possible de les comparer et les trouver égales car `S` – le type de `s1` – n'est qu'un alias de **`string`**, le type de `s2`. Inversement, il est nécessaire de convertir `t` dans le type `S` pour la comparer à `s1`, car son type `T` est un type différent de **`string`**, qui est son type sous-jacent, ce qui permet à la conversion de fonctionner pour réaliser cette comparaison.

✓ **Types, lignes 21 et 24** : lorsque Go représente le type de `s2`, qui est une variable du type `S`, il renvoie **`string`** et non `S`, confirmant que `S` n'est qu'un alias de **`string`** et non un type en propre. Inversement, bien que `T` soit basé sur **`string`**, la représentation du type de `t` renvoie bien une référence à `T` et non **`string`**, confirmant que `T` est bien d'un véritable type différent de **`string`**.

Listing 5.11 – Comparaisons entre alias et types définis
https://osinet.fr/go/data/play-comparaisons-alias-types

```
1  package main
2
3  import "fmt"
4
5  type (
6          S = string
7          T string
8  )
9
10 func (t T) len() int {
11         return len(t)
12 }
13
14 func main() {
15         var s1 string = "A"
16         var s2 S = "A"
17         var t T = "A"
18
19         fmt.Printf("%v %v %T %T %T %d\n",
20                 s1 == s2, s1 == S(t),
21                 s1, s2, t, t.len())
22 }
23
24 // Résultat: true true string string main.T 1
```

Le mécanisme de définition des types ainsi mis en place permet de créer les types non natifs au langage, mais composés par le programmeur pour définir des données composites à partir des types simples, fournis nativement par le langage.

© Dunod – Toute reproduction non autorisée est un délit.

go/types

Pour mieux comprendre la façon dont Go voit les types dans un programme, et faciliter l'écriture de programme manipulant directement des codes sources Go, le programme `gotypes` est maintenu par Alan Donovan.

Il fait partie du dépôt d'exemples du projet Go :

https://osinet.fr/go/data/gh-go-gotypes

Reposant sur le paquet `go/types` introduit dans Go 1.5, il n'est utilisable tel quel qu'avec cette version de Go et celles plus récentes.

5.4 VALEURS VIERGES

5.4.1 Définition

La valeur vierge (*zero value*) – parfois abrégée en `0V` – est, pour chaque type de données, la valeur à laquelle une variable de ce type est initialisée lors de sa déclaration si elle ne fait pas l'objet d'une initialisation explicite.

Pour un type `T`, c'est la valeur obtenue à la fois lors d'une initialisation de la forme `var t T` ou par la fonction intégrée `new`, comme dans le listing 5.12, qui montre les

Listing 5.12 – Initialisation récursive des valeurs vierges
https://osinet.fr/go/data/play-init-0v

```
package main

import "fmt"

type S struct {
        s [2]string
}

type T struct {
        S
        c complex64
}

func main() {
        var t1 T
        var t2 = new(T)
        fmt.Printf("%#v\n%#v\n%t\n", t1, *t2, t1 == *t2)
}

// Résultat:
// main.T{S:main.S{s:[2]string{"", ""}}, c:(0+0i)}
// main.T{S:main.S{s:[2]string{"", ""}}, c:(0+0i)}
// true
```

deux formes d'initialisation, et l'affectation des valeurs vierges aux champs normaux (`T.c`), intégrés (`T.S`,) et aux types d'éléments des tableaux (`S.s`).

Le concept n'est pas original à Go, mais l'insistance sur la formalisation de ces valeurs peut être déroutante en venant d'autes langages, en particulier les langages utilisant abondamment classes et héritage, Java en tête.

Elle s'inscrit dans la même logique que le choix de la composition plutôt que l'héritage (cf. § 1.1.2.5) et la recherche de la simplification du code : là où une donnée complexe dans un tel langage va généralement être modélisée comme un type classe et systématiquement initialisée par un constructeur, qui va récursivement appeler les constructeurs parents, pour initialiser progressivement chacun des champs de la structure, utilisant des constructeurs pour chacun des champs, l'approche idiomatique en Go va être totalement différente : en l'absence d'héritage, la pratique recherchée pour un type structure va consister à définir les champs directs ou intégrés de sorte que la valeur fournie lors de l'initialisation soit directement utilisable, sans qu'il soit besoin d'aucun constructeur, ni pour la structure enveloppante, ni pour aucun de ses champs, et ainsi de suite récursivement pour la valeur vierge de chacun de ses sous-champs.

Cette approche matérialise un choix en matière d'organisation de code :

- ✓ En limitant le recours à des constructeurs, elle limite la taille de l'API de chaque paquet exportant des types et simplifie l'initialisation des données, en particulier lorsqu'elles sont intégrées dans des structures composées plus complexes.
- ✓ En contrepartie, cela peut impliquer un code un peu moins simple dans les types ainsi inclus, la valeur vierge (0V) de chaque champ utilisée n'étant pas nécessairement celle qui se prête à l'utilisation la plus intuitive.

Comme dans la plupart des cas en Go, ce choix priorise la simplicité de consommation et la minimalité des API sur celle d'écriture initiale, afin d'obtenir du code moins couplé, plus résistant aux évolutions indépendantes, caractéristiques de code à grande échelle.

Il n'interdit pas d'avoir resources à des constructeurs, et la bibliothèque standard n'en manque pas, comme les 8 `NewEncoder()` ou les 41 `New()` présents dans Go 1.13 ; mais recommande simplement de n'en créer que s'ils ont une vraie raison d'être, et non par simple habitude.

5.4.2 Valeurs vierges des types de données

Ces valeurs vierges sont les suivantes, en fonction des natures de types :

Le cas particulier de **`nil`** et de ses usages multiples est traité au § 5.5.

© Dunod – Toute reproduction non autorisée est un délit.

Tableau 5.4 – Valeurs vierges (0V) par nature de type

Nature de type	Valeur vierge
Booléens	`false`
Entiers signés ou non	`0`
Rune	`'\x00'`
Flottants réels et complexes	`0.0` (`== 0+0i`)
Chaînes	`""`
Pointeurs	**`nil`**
Cartes	**`nil`**
Tranches	**`nil`**
Canaux	**`nil`**
Fonctions	**`nil`**
Interfaces	**`nil`**
Tableaux	Tableau valide où chaque élément est initialisé à la valeur vierge du type d'élément
Structure	Structure valide où chaque champ est initialisé à la valeur vierge de son type

5.4.3 Rendre la valeur vierge utile

« *Make the zero value useful.* »

Rob Pike - *Go Proverbs*
https://osinet.fr/go/data/yt-pike2015-0v

Comment mettre en pratique ce proverbe souvent cité ?

La bibliothèque standard fourmille d'exemples : examinons `strings.Builder`, la façon recommandée de construire une chaîne par morceaux au lieu d'utiliser une simple concaténation, pour minimiser les allocations et copies mémoire. Le code interne de Go 1.13 pour ce type est disponible sur :

https://osinet.fr/go/data/gh-go-strings-builder

Le type `Builder` y est défini par les méthodes de quatre interfaces unitaires :

- ✓ **fmt.Stringer** : `String() string` renvoie le contenu sous forme d'une chaîne de caractères ;
- ✓ **io.ByteWriter** : `WriteByte(c byte) error` ajoute un octet à la chaîne en cours de construction ;

- ✓ **io.StringWriter** : `WriteString(s string) (n int, err error)` concatène une chaîne à celle en cours de construction ;
- ✓ **io.Writer** : `(p []byte) (n int, err error)` ajoute les octets d'une tranche d'octets à la chaîne en cours de construction.

Venant d'une approche traditionnelle à la mode C, il semblerait naturel que ce type soit initialisé par un constructeur, qui allouerait probablement un tampon d'écriture et des valeurs initiales pour la position d'écriture et la longueur allouée.

Rien de tout celà en Go : la valeur vierge du type est directement utilisable. Le type est un type structure à deux champs :

- ✓ une tranche d'octets ;
- ✓ un pointeur vers le récepteur pour la détection de recouvrement, simple mesure de sécurité pour le cas de recouvrements mémoire.

Puisqu'il s'agit d'un type structure, il est automatiquement initialisé avec la 0V de ses deux champs : **`nil`** pour le pointeur, et la 0V d'une tranche d'octets.

Suivant ce principe de valeurs vierges utiles, la 0V des types tranche (cf. § 4.2.1) est une tranche de longueur nulle. Les trois méthodes d'écriture peuvent donc allonger son contenu dynamiquement sans avoir besoin d'utiliser un constructeur – ne serait-ce qu'un `make []byte` – et la méthode de sortie `String()` peut directement convertir cette valeur vierge, si aucune écriture n'a été réalisée, sous la forme de la valeur vierge du type chaîne : une chaîne vide.

Cet exemple illustre comment concevoir un type de données complexe sans nécessiter de constructeurs, en s'appuyant et en respectant le principe des valeurs vierges utiles.

Une autre possibilité pour le cas où certaines méthodes du type doivent renvoyer des valeurs par défaut qui ne sont pas les valeurs vierges de leur type de retour, en évitant l'utilisation de constructeurs, consiste à tester dans les méthodes l'utilisation d'un récepteur **`nil`**, comme décrit au § 5.5.3.

Pour aller plus loin, d'autres exemples notables de types complexes sans constructeur dans la bibliothèque standard sont `sync.Mutex` et `bytes.Buffer`.

© Dunod – Toute reproduction non autorisée est un délit.

5.5 « nil »

5.5.1 Définition

Le mot **`nil`** est un identificateur prédéfini, et non un mot-clef du langage. En tant que tel, il peut être surchargé, avec des conséquences généralement non souhaitées, comme le montre le listing 5.13.

Listing 5.13 – Redéfinition de `nil`
https://osinet.fr/go/data/play-surcharge-nil

```
package main

func main() {
        { var nil = 42;       println(nil) } // 42
        { const nil = "nil"; println(nil) } // "nil"
        { nil := func() {};  println(nil) } // 0xa8c40
}
```

Lorsqu'il n'est pas redéfini, ce qui est la situation normale d'un programme Go, sa valeur est la valeur vierge (cf. § 5.4.1) des types pointeurs, cartes, tranches, fonctions, et interfaces.

5.5.2 `nil` pour les types interfaces

Comme détaillé au § 3.5.3, les valeurs interfaces sont représentées par une paire de pointeurs, respectivement vers le type dynamique et vers la valeur.

Une conséquence est que, à la différence de C et d'autres langages de la même lignée, deux valeurs `nil` du même type statique interface peuvent ne pas être égales entre elles, pour peu que leur type dynamique soit différent, comme dans le listing 5.14 :

- ✓ Dans le premier cas, les `nil` portent un type concret, et sont égaux entre eux.
- ✓ Dans le deuxième cas, ils sont du même type statique interface et du même type dynamique, et sont toujours égaux entre eux.
- ✓ Dans le troisième, ils sont toujours du même type statique interface, mais plus du même type dynamique, et ne sont pas égaux entre eux.

5.5.3 `nil` comme récepteur

À la différence de nombre d'autres langages, invoquer une méthode sur une instance `nil` d'un type ne provoque pas intrinsèquement d'erreurs, tant que le code de la méthode invoquée n'effectue pas d'opérations incorrectes avec ce récepteur vierge.

Dans ces langages, l'instance est normalement utilisée pour pointer, directement ou par l'intermédiaire d'une table de méthodes virtuelles, vers le code de la méthode effectivement utilisée, et ce même si elle est portée par un type ancêtre de l'instance. Mais en présence d'un pointeur nul, qui est la représentation standard d'une valeur vierge, le langage ne peut adresser la table de méthodes, et provoque une erreur de compilation ou d'exécution.

Listing 5.14 – `nil` n'est pas toujours égal à lui-même
https://osinet.fr/go/data/play-nil-egal-nil

```go
package main

import "fmt"

type v = interface{}

func main() {
        var c, d *int
        var e *rune

        // Nil == nil sur type concret
        fmt.Printf("c: %v, d: %v, %t %t %t\n",
                c, d, c == nil, d == nil, c == d)

        // Nil == nil sur type interface
        fmt.Printf("c: %v, d: %v, %t %t %t\n",
                c, d, c == nil, d == nil, v(c) == v(d))

        // Nil != nil sur type interface
        fmt.Printf("c: %v, d: %v, %t %t %t\n",
                c, e, c == nil, e == nil, v(c) == v(e))
}

// Résultat
// c: <nil>, d: <nil>, true true true
// c: <nil>, d: <nil>, true true true
// c: <nil>, d: <nil>, true true false
```

Comme détaillé au § 3.4.1, les méthodes Go sont simplement un sucre syntaxique pour rattacher des fonctions à une valeur : elles ne dépendent de ce fait pas de la valeur du récepteur, qui n'est rien de plus qu'un paramètre additionnel déguisé en tête de la liste de paramètres.

Dès lors, un appel de méthode sur une valeur `nil` n'est qu'un appel de fonction ordinaire auquel cet argument `nil` est passé en premier paramètre, comme dans le listing 5.15.

La seule difficulté de ce mécanisme se situe sur les types interfaces : le type statique d'une valeur interface ne comporte que la définition de la liste de méthodes, et non leurs implémentations, qui sont portées par le type dynamique.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 5.15 – Appel de méthodes sur `nil` concret
https://osinet.fr/go/data/play-nil-methodes-concret

```
1  package main
2
3  import (
4          "fmt"
5          "strconv"
6  )
7
8  type N int
9
10 func (n *N) s() string {
11         if n == nil {
12                 return "<nil>"
13         } else {
14                 return strconv.Itoa(int(*n))
15         }
16 }
17
18 func main() {
19         var n N = 3
20         var pn1 *N
21         var pn2 *N = &n
22         fmt.Printf("%#v -> %s\n", pn1, pn1.s())
23         fmt.Printf("%#v -> %s\n", pn2, pn2.s())
24 }
25
26 // Résultat:
27 // (*main.N)(nil) -> <nil>
28 // (*main.N)(0x414020) -> 3
```

Comme détaillé au § 3.5.3, les valeurs interfaces sont représentées par une paire de pointeurs, l'un vers une représentation du type concret, et donc la table de ses méthodes, l'autre vers la donnée.

Lorsqu'une valeur interface est un `nil` typé, les méthodes sont toujours disponibles comme dans le cas des autres types ; la seule impossibilité se situe lorsque le type interface est initialisé avec un `nil` non typé, typiquement comme valeur par défaut : en ce cas, l'invocation de méthode n'est pas possible, aucune adresse de fonction n'étant disponible, comme dans le listing 5.16.

Dans cet exemple, dérivé du précédent, les deux types de `nil` sont présents : celui des lignes 13 et 30 est type comme pointeur sur le type non-interface `N`, tandis que celui des lignes 21 et 32 est un littéral `nil`, de type dynamique également `nil`, sans méthode susceptible d'être invoquée.

Listing 5.16 – Appel de méthodes sur **nil** interface
https://osinet.fr/go/data/play-nil-methodes-concret

```
1  package main
2
3  import (
4          "fmt"
5          "strconv"
6  )
7
8  type I interface{ s() string }
9
10 type N int
11
12 func (n *N) s() string {
13         if n == nil { // Ce nil est typé *N.
14                 return "<nil>"
15         } else {
16                 return strconv.Itoa(int(*n))
17         }
18 }
19
20 func utiliseI(i I) {
21         if i != nil { // Ce nil est non typé.
22                 fmt.Printf("%#v -> %s\n", i, i.s())
23         } else {
24                 fmt.Printf("%#v -> nil non typé\n", i)
25         }
26 }
27
28 func main() {
29         var n N = 3
30         var pn1 *N
31         var pn2 *N = &n
32         var pi I
33         utiliseI(pn1) // (*main.N)(nil) -> <nil>
34         utiliseI(pn2) // (*main.N)(0x414020) -> 3
35         utiliseI(pi)  // <nil> -> nil non typé
36 }
```

Une utilisation pratique de ce fonctionnement est le fait de fournir des valeurs par défaut simplement dans les méthodes : une méthode devant fournir une valeur par défaut pour un type sans constructeur défini peut être implémentée avec un récepteur pointeur et, en testant si ce récepteur est **nil**, renvoyer la valeur prévue par défaut pour ce type.

© Dunod – Toute reproduction non autorisée est un délit.

5.5.4 Une erreur à un milliard de dollars ?

Lors de la présentation de Go, la communauté académique s'est initialement interrogée sur la présence de `nil` dans un langage récent.

En effet, revenant en 2009 [16] sur l'invention des références nulles dans ALGOL W en 1965, et ses conséquences en bogues constatées dans la littérature depuis lors, C.A.R. Hoare avait appelé cette idée son « erreur à un milliard de dollars » (*billion-dollar mistake*), et l'assimilation entre `nil` et le pointeur nul ou la valeur `NULL` traditionnelle avait été source de nombreuses critiques concernant la conception de Go.

Le problème des pointeurs et références nulles est indéniable [17] depuis les références ALGOL ou les pointeurs C jusqu'aux `NullPointerException` de Java, de sorte que l'assimilation était tentante.

Dans les faits, `nil` n'est pas exactement un simple pointeur nul ou une référence nulle, mais une valeur vierge pour cinq natures de types concrets, et aussi la valeur vierge non typée des types interface.

Respectant l'approche visant à donner un sens aux valeurs vierges, `nil` est donc une valeur valide et utile pour chaque type dont il est la valeur vierge : que ce soit pour un récepteur pointeur comme décrit ci-dessus, un canal, une carte, une tranche, comme décrit dans la rubrique les concernant, et même pour une fonction passée comme argument dans un patron Stratégie.

Puisque `nil` ne présente de ce fait pas les défauts des références nulles traditionnelles, ces critiques ne lui sont pas réellement applicables, et il constitue un élément important du style idiomatique de la programmation Go.

Pour aller plus loin sur ce sujet, voir la présentation de Francesc Campoy à Gophercon 2016 : https://osinet.fr/go/data/campoy2016-nil

5.6 FIN DE VIE

La fin de vie des données en Go se déroule en deux temps :

- ✓ la fin d'utilisation et la sortie de visibilité depuis le code utilisateur, correspondant dans divers langages aux destructeurs ;
- ✓ la libération des données et le recyclage éventuel par le ramasse-miettes.

5.6.1 Destructeurs

Tout comme Go ne définit pas de constructeurs comme méthodes sur les types structure, il ne définit pas non plus de destructeurs comme méthodes, à l'inverse de C++, Java, ou PHP.

16. https://osinet.fr/go/data/hoare2009-nil
17. https://osinet.fr/go/data/wiki-null

La situation est comparable à la situation des constructeurs :

- ✓ Il est possible de créer des constructeurs sous la forme de fonctions ordinaires dans le paquet définissant un type, avec une convention de nommage `NewT` / `MakeT` (cf. § 4.4.6).
- ✓ De la même façon, la pratique recommandée [18] pour marquer la fin d'utilisation d'une donnée consiste à utiliser une fonction ordinaire ou une méthode dénommée `Close` ou `close`, sans paramètres, et ne renvoyant aucun résultat, ou uniquement une valeur de type **`error`**.
- ✓ La bibliothèque standard n'a en revanche pas généralisé l'interface unitaire `Closer` ne contenant que cette méthode : elle n'est présente que comme `io.Closer`. Dans tous les autres cas dans la bibliothèque standard, la méthode fait partie d'interfaces plus étendues.

Plus généralement, de la même façon que les types sont dans l'idéal conçus pour qu'une valeur vierge soit directement utilisable sans nécessiter un constructeur, ils doivent dans l'idéal être conçus pour qu'une valeur utile, active, puisse être directement détruite sans passer par une destructeur tel qu'une fonction ou méthode `Close`.

5.6.2 Finaliseurs

Parfois confondus avec des destructeurs par les développeurs venant d'autres langages, les finaliseurs interviennent en réalité plus tard que des destructeurs.

Le finaliseur d'une valeur est une fonction pouvant être attachée à celle-ci, et destinée à être invoquée par le ramasse-miettes (*garbage collector*) au moment où il détermine que la valeur n'est plus utilisée et que son espace mémoire peut être recyclé.

En tant que tels, ils sont invoqués :

- ✓ De façon asynchrone par rapport à l'exécution et, plus généralement, dans un délai arbitraire après la disparition de la donnée de la visibilité du programme en cours, voire jamais avant la fin de l'exécution.
- ✓ Potentiellement plus tôt que souhaité, comme dans le listing 5.17, qui reproduit la situation décrite dans le source de `runtime.SetFinalizer`.
  Comme cette situation de finalisation précoce survient en pratique rarement, elle se manifeste par des bogues ne survenant que de façon difficilement reproductible, et principalement sous forte charge.
  L'appel à `runtime.KeepAlive` permet de l'éviter dans tous les cas.
- ✓ Avec la valeur telle qu'elle était au moment de sa sortie de visibilité par le programme : si elle contient directement ou indirectement des pointeurs vers d'autres données, ceux-ci ont pu devenir invalides dans l'intervalle.

18. 212 occurrences de ce motif dans les sources de Go 1.13.

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ Au mieux pour les valeurs définies dans des portées de fonctions ou intérieures à celles-ci : les valeur définies dans la portée paquet peuvent se trouver allouées à l'édition de liens et non sur le tas, donc hors de portée du ramasse-miettes.
- ✓ En respectant les dépendances résultant des pointeurs internes : si un objet à finaliser pointe vers un autre objet également pourvu d'un finaliseur, le ramasse-miettes va d'abord exécuter le finaliseur du premier, pour éviter le risque que la donnée pointée depuis `X` ne soit rendue invalide par le finaliseur de `Y`.
  Cette analyse de dépendances a pour conséquence que les structures cycliques – comme les listes doublement chaînées – pourvues de finaliseurs sont susceptibles de ne jamais être recouvrées par le ramasse-miettes, puisqu'il n'existe aucun ordre d'invocation des finaliseurs qui pourra respecter cette chaîne de dépendances.

Listing 5.17 – Finaliseur et `runtime.KeepAlive`
https://osinet.fr/go/data/play-finaliseurs

```
package main

import (
        "os"
        "runtime"
        "syscall"
)

type EnveloppeFichier struct { d os.File }

func main() {
        const size = 512
        n, _ := os.Open("/dev/null")
        buf := make([]byte, size)

        p := &EnveloppeFichier{d: *n}
        runtime.SetFinalizer(p, func(e *EnveloppeFichier) {
                e.d.Close()
        })

        // À l'entrée dans ce Write, p devient recyclable
        // sans le KeepAlive final: si le finaliseur se
        // déclenche, il va fermer le descripteur et Write va
        // échouer, voire écrire ailleurs s'il a été réalloué
        // avant que la goroutine ne soit exécutée.
        syscall.Write(int(p.d.Fd()), buf)

        // Cet appel force la référence à demeurer non
        // collectable, évitant le déclenchement du finaliseur.
        runtime.KeepAlive(p)
}
```

Enfin, puisqu'ils sont invoqués par le ramasse-miettes pour le recyclage de la mémoire, il n'y a aucune garantie qu'ils soient invoqués sur une donnée de longueur nulle, puisqu'elle n'occupe aucune mémoire susceptible d'être recyclée.

Pour ce qui est de l'API, elle ne fait pas partie de la spécification du langage, mais uniquement de l'API publique du paquet `runtime`. Elle se limite aux deux fonctions utilisées dans le listing 5.17 :

- ✓ **`runtime.SetFinalizer(p interface{}, f interface{})`** : associe à la valeur `p`, qui doit être l'adresse d'une donnée adressable, une fonction `f` dite *finaliseur*, qui sera invoquée lors de son recyclage.
  `f` est une fonction sans paramètres, qui peut renvoyer un résultat, lequel sera de toute façon ignoré.
  Lors de l'exécution du ramasse-miettes, tous les finaliseurs sont invoqués dans une même goroutine, concurremment avec l'exécution depuis Go 1.5 : si un finaliseur doit être long, il est souhaitable qu'il crée une goroutine pour ne pas ralentir le ramasse-miettes.
- ✓ **`runtime.SetFinalizer(p interface{}, nil)`** : est un cas particulier du précédent, avec pour effet de supprimer un finaliseur déjà associé.
  Elle doit être utilisée lorsqu'un programme veut modifier un finaliseur déjà associée : en effet, l'invoquer avec un finaliseur non-**`nil`** alors qu'un autre finaliseur est déjà associé à la valeur provoque une erreur d'exécution.
- ✓ **`runtime.KeepAlive(x interface{})`** : dans laquelle `x` est un pointeur vers une valeur portant un finaliseur, permet de forcer cette valeur à ne pas sortir de la visibilité de la portée en cours, et donc d'empêcher le ramasse-miettes de la collecter avant la sortie de cette portée.

Avant de les utiliser, il est donc préférable d'aller au-delà de la présente rubrique introductive, en analysant le source du fichier `mfinal.go` [19], et les divers articles publiés sur leurs usages et leurs dangers.

La compréhension du fonctionnement du ramasse-miettes (cf. *Plus en ligne*), dans ses évolutions au fil des versions, est également utile, en particulier pour ce qui concerne la visibilité des données durant une chaîne de finaliseurs.

En dernier lieu, tout comme l'API de réflexion, le mécanisme des finaliseurs n'est pas destiné à un usage général, mais concerne un nombre limité de cas, principalement trouvés dans les couches basse de la bibliothèque d'exécution.

© Dunod – Toute reproduction non autorisée est un délit.

19. https://osinet.fr/go/data/go-mfinal

5.7 RÉFLEXION ET MÉTA-PROGRAMMATION

« *Reflection is never clear.* »

Rob Pike - *Go Proverbs*
https://osinet.fr/go/data/yt-pike2015-reflection

5.7.1 Définition

Les mécanismes de réflexion (*reflection*) dans le sens général font partie des outils de méta-programmation, permettant à un programme écrit dans un langage réflexif d'examiner (introspection) et modifier (intercession) à l'exécution ses propres structures de données par des mécanismes accessibles aux auteurs de programmes et non seulement aux auteurs du langage. Les langages supportant ce mécanisme sont dit réflexifs.

Très répandue jusque dans les programmes applicatifs dans les langages dynamiques utilisés dans le développement web, comme PHP ou JavaScript, elle est souvent recherchée par les développeurs nouvellement venus à Go depuis ces langages.

Dans la pratique Go, elle se révèle beaucoup plus rarement utile, et son utilisation est le plus souvent limitée à trois cas :

- ✓ La manipulation des types interfaces par des assertions et commutateurs de type, utilisant pour ce faire le soutien de réflexion inclus dans la syntaxe du langage, sans exposer l'API sous-jacente : ce mécanisme a été décrit au § 3.5.5.
- ✓ L'examen des étiquettes de champs (cf. § 4.4.4) dans les types structures par des outils comme les (dé-)sérialiseurs, ORMs, utilisant pour ce faire un « parcours » d'API clairement défini.
- ✓ L'utilisation de fonctions à vocation générique fournies par le paquet `reflect`.

En effet, par sa nature totalement dynamique, l'utilisation de la réflexion au sens général va à l'encontre des objectifs recherchés par l'utilisation de Go : elle favorise une programmation faiblement typée, avec des logiques complexes. Comme le rappelle le proverbe en exergue : « la réflexion n'est jamais claire ».

5.7.2 Lecture d'étiquettes de structure

La lecture des étiquettes de structure est la principale motivation pour utiliser l'API de réflexion Go directement.

Pour faciliter autant que possible cette lecture, un format standardisé a été défini pour les étiquettes, de la forme :

```
<champ> <type> `<p1>:"k1:v1,v,..." <p2>:...`
```

Ce format consiste à définir dans une étiquette de champ l'ensemble des informations que pourraient utiliser plusieurs paquets différents utilisant l'API de réflexion :

✓ Définition de l'étiquette comme une chaîne brute (cf. § 3.1.4.2).

✓ Dans cette chaîne, constitution de groupes de symboles séparés par des espaces, et composés chacun de :
- l'identifiant d'un paquet (`<p1>` et `<p2>` ci-dessus) ;
- le séparateur deux-points ;
- une chaîne entre guillemets (`"k1:v1,v"` ci-dessus), constituant la valeur de l'étiquette telle que destinée à un paquet donné.

Ce format n'est pas obligatoire, mais il est nécessaire pour pouvoir utiliser les méthodes optimisées de lecture des étiquettes, plutôt que de devoir traiter avec l'API de réflexion sans le bénéfice de ces optimisations.

Le format interne de la valeur chaîne ainsi encadrée est en revanche libre pour ce qui concerne l'API de réflexion.

En pratique, toutefois, la plupart des auteurs s'alignent sur un format composé de groupes, eux-même constitués d'identifiants (`v` ci-dessus) et de paires clef-valeur (`k1:v1` ci-dessus), comme dans le listing 5.18, tiré de l'ORM GORM.

Listing 5.18 – Étiquettes structurées dans l'ORM GORM (licence MIT)
https://osinet.fr/go/data/gh-gorm-tags

```
65 type CreditCard struct {
66         ID        int8
67         Number    string
68         UserId    sql.NullInt64
69         CreatedAt time.Time  `sql:"not null"`
70         UpdatedAt time.Time
71         DeletedAt *time.Time `sql:"column:deleted_time"`
72 }
73
74 type Email struct {
75         Id        int16
76         UserId    int
77         Email     string `sql:"type:varchar(100);"`
78         CreatedAt time.Time
79         UpdatedAt time.Time
80 }
```

Le listing 5.19 illustre la démarche recommandée pour analyser de telles étiquettes :

1. Obtenir le type de la valeur structure examinée, sous la forme d'une valeur de type `reflect.Type` au moyen de la fonction `reflect.TypeOf`.
   Aucun échec n'est prévu à cette étape.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 5.19 – Lecture d'étiquettes de structure
https://osinet.fr/go/data/play-lecture-tags

```
package main

import "reflect"

type S struct {
        c int `un:"a:x,b"`
}

func main() {
        var e reflect.StructField
        var ok bool
        var s S = S{37}

        // Etape 1: réflexion pour le type.
        var t reflect.Type = reflect.TypeOf(s)

        // Etape 2: réflexion pour le champ.
        e, ok = t.FieldByName("c")
        if !ok {
                println(`champ "c" non trouvé`)
                return
        }

        // Etape 3: réflexion pour l'étiquette.
        var st reflect.StructTag = e.Tag

        // 3.1 simple: Get.
        println(st.Get("un")) // Résultat: a:x,b

        // 3.2 prudent: Lookup.
        adeux, ok := st.Lookup("deux")
        if !ok {
                println(`valeur "deux" non trouvée`)
                return
        }
        println(adeux)
}
```

2. À partir de ce type obtenir la description du champ examiné, comme valeur de type `reflect.StructField`, au moyen de la méthode `Type.FieldByName` – ou une autre méthode du même groupe – sur la valeur de type obtenue à l'étape précédente. Les champs non exportés sont accessibles à cette API.

Cette étape peut échouer, pour toutes ces méthodes `FindXXX`, en cas de champ absent, ou en présence de champs intégrés non exportés. [20]

3. À partir de cette valeur, extraire son étiquette `reflect.StructTag`, en accédant à son champ `Tag`.
4. À ce stade, si l'étiquette du champ respecte le format standard, obtenir la partie souhaitée du contenu de l'étiquette : soit par la méthode prudente `StructTag.Lookup`, qui permet d'obtenir un indicateur de succès ; soit par la méthode simplifiée `StructTag.Get` si le type n'a pas de raison de provoquer un échec.
5. Le résultat obtenu commence par la partie souhaitée de l'étiquette, le nom de la partie et le séparateur deux-points ayant été retirés. L'API de réflexion n'analyse pas cette partie plus avant ; le reste de l'exploitation de l'étiquette est de la logique applicative.

5.7.3 Fonctions génériques

Outre tous les outils dédiés aux types et valeurs appartenant à la réflexion proprement dite, le paquet `reflect` comporte également deux fonctions applicables à tous les types de façon générique :

✓ **`reflect.DeepEqual`** : permet de comparer des valeurs pour lesquelles les opérateurs de comparaison ne sont pas définis, comme les cartes ou les tranches. Elle est applicable même aux structures cycliques.
Dans la plupart des cas, elle est envisageable comme une généralisation plus lente de ces opérateurs, mais elle implique certaines incohérences, comme le fait que des valeurs peuvent être différentes d'elles-même, si elles contiennent à un quelconque niveau des valeurs non comparables comme `NaN` ou des valeurs fonctions, pour lesquelles la comparaison permet seulement de savoir si les deux valeurs comparées sont toutes deux **`nil`** ou si au moins l'une des deux ne l'est pas, comme dans le listing 5.20.

✓ **`reflect.Swapper(tranche) func(i, j int)`** : est une fonction d'ordre supérieur qui, appliquée à une valeur tranche quelconque, renvoie une fonction à deux arguments entiers positifs capable d'échanger les positions respectives des éléments situés à ces indices dans la tranche d'origine, comme dans le listing 5.21.

20. Si un type comporte plusieurs champs non exportés du même nom local provenant de champs intégrés définis dans des paquets différents, la fonction peut renvoyer n'importe laquelle de ces implémentations, ou n'en trouver aucune, comme dans cet exemple documenté dans le paquet `reflect` : https://osinet.fr/go/data/play-reflect-tags

© Dunod – Toute reproduction non autorisée est un délit.

Listing 5.20 – Comparaison par `reflect.DeepEqual`
https://osinet.fr/go/data/play-deepequal

```
package main

import "reflect"

func main() {
        f := func() {}

        // Erreur de compilation.
        // println(f == f)

        // Comparaison par réflexion.
        println(reflect.DeepEqual(f, f)) // false
}
```

Listing 5.21 – Permutation dans une tranche avec `reflect.Swapper`
https://osinet.fr/go/data/play-swapper

```
package main

import (
        "fmt"
        "reflect"
)

func main() {
        f := []byte{10, 20, 30, 40}

        échange := reflect.Swapper(f)

        échange(0, 3)
        fmt.Println(f) // [40 20 30 10]
}
```

5.7.4 API de réflexion

Parfois, des besoins très particuliers peuvent toutfois nécessiter une utilisation plus approfondie des mécanismes de réflexion.

L'API du paquet `reflect` est disponible pour ces cas.

5.7.4.1 Introspection

Le point d'entrée de l'API de réflexion exposée par le paquet `reflect` est la paire de fonctions `TypeOf(i interface Type)` et `ValueOf(i interface Value)`, qui fournissent des descriptions utilisables des deux parties de toute valeur interface : le type dynamique et la valeur courante appartenant à ce type.

Lorsque ces fonctions reçoivent une valeur quelconque comme argument, elles le font dans un paramètre typé **`interface`**`{}`, de sorte que la valeur est convertie en valeur de ce type interface si nécessaire, conversion qui ne peut échouer puisque toute valeur implémente l'interface vide (cf. §3.5.4). Si l'argument est d'un type concret, la valeur interface reçue porte ce type concret comme type dynamique ; tandis que si l'argument est déjà d'un type interface, aucune conversion n'est opérée.

Parmi les multiples méthodes des deux types renvoyés par ces fonctions, les deux plus simples d'utilisation sont celles du type `Value` :

- ✓ **`Kind()`** : permet d'accéder au code numérique du type de la donnée, défini par les valeurs des constantes **`iota`** du type `reflect.Kind` [21], permettant aux programmes d'agir en fonction du type des données par comparaison aux valeurs de cette énumération, en agissant en fonction du type sous-jacent, sans avoir à prendre en compte le type défini qui peut être associé à la valeur, alors que la valeur renvoyée par `TypeOf` renvoie ce dernier.
- ✓ **`String()`** : permet d'obtenir une représentation sous forme de chaîne de toute valeur.

5.7.4.2 Intercession

Une fois l'introspection appliquée à une valeur, il est possible de la manipuler sous sa forme symbolique `Value` au moyen de la méthode `Value.Elem()` et des méthodes du groupe `Value.Set<`**`type`**`>`.

Enfin, miroir du `ValueOf` initial, justifiant le nom de *réflexion* appliqué à ces opérations, le fonctionnement inverse est également disponible, avec la méthode `Value.Interface(v Value)` **`interface`**`{}`, qui reconstitue une valeur d'un type usuel à partir d'une structure de réflexion.

Comme le type de résultat de cette dernière est de nouveau **`interface`**`{}`, il peut contenir tout type concret ou interface, que le code applicatif peut en extraire par une assertion de type.

C'est ce que démontre le listing 5.22, dans lequel une donnée est lue et modifiée sous sa forme réflexive avant d'être renvoyée à la fonction principale pour utilisation comme le type d'origine.

21. https://osinet.fr/go/data/go-kind

© Dunod – Toute reproduction non autorisée est un délit.

Listing 5.22 – Introspection et intercession
https://osinet.fr/go/data/play-intercession

```
package main

import (
        "fmt"
        "reflect"
)

type Local complex64

// Examen de la valeur et de ses types.
func introspecte(c Local) reflect.Value {
        v := reflect.ValueOf(&c)

        t := reflect.TypeOf(c)
        k := v.Kind()

        fmt.Println(t, k)
        return v
}

// Manipulation d'une donnée symbolique.
func intercède(v reflect.Value) interface{} {
        e := v.Elem()
        if e.CanSet() {
                e.SetComplex(1+3i)
        }

        // Restitution d'une valeur hors réflexion.
        i := e.Interface()

        fmt.Printf("%T %T\n", i, v)
        return i
}

func main() {
        var c Local = complex(2, 5)
        v := introspecte(c)
        d := intercède(v)
        fmt.Printf("%T %f\n", d, d)
}

// Résultat
// main.Local ptr
// main.Local reflect.Value
// main.Local (1.000000+3.000000i)
```

5.7.5 Conclusion

La combinaison des trois étapes d'introspection, manipulation et intercession permet donc, comme le montre l'exemple présenté dans cette section, d'atteindre une flexibilité de manipulation des données à la volée aussi étendue que nécessaire.

Est-elle pour autant souhaitable au quotidien ? Non, bien sûr, pour deux principales raisons :

- ✓ elle permet de faire abstraction de la sécurité apportée par le système de types forts créé pour le langage ;
- ✓ elle introduit une complexité inhabituelle dans le code Go.

Pour ces deux raisons, son utilisation dans le cadre des développements courants va à l'encontre de toute la philosophie du langage, qui vise à rendre le code applicatif le plus simple et maintenable possible.

Pour citer Rob Pike, dans la conférence où il a présenté le proverbe qui ouvre cette section, s'adressant à une audience de développeurs Go enthousiastes :

> « Ce n'est pas pour vous. »

Mais alors, pourquoi existe-t-elle ?

L'examen des sources de Go et de sa bibliothèque standard apporte la réponse à cette question : la majorité des usages de l'API de réflexion se trouvent dans le compilateur, l'environnement d'exécution (*runtime*), et le ramasse-miettes, illustrant bien sa « cible » : un usage avancé de manipulation de code, qui concerne en réalité très peu les développeurs d'applications.

Le paquet `reflectlite`

Go 1.13 a introduit un paquet `reflectlite`, comportant des mécanismes de réflexions plus simples d'utilisation et avec moins de dépendances, mais plus limités : il serait donc tentant de les utiliser.

En pratique, ce paquet a été créé pour les besoins des nouveaux mécanismes d'emballage et déballage d'erreurs (cf. § 8.8.4) et il est implémenté comme paquet `internal` (cf. chapitre 9), dans `src/internal/reflectlite`, de sorte qu'il n'est pas accessible hors du projet Go lui-même, qui devrait progressivement voir son utilisation s'étendre, du fait de dépendances moindres que `reflect`.

Expressions

Une expression est un fragment de code donnant lieu à une évaluation à partir d'opérandes, et renvoyant un jeu de résultats.

Ce chapitre détaille :

- ✓ les opérandes utilisés dans les expressions ;
- ✓ les types d'expressions disponibles en Go, indépendamment de leur application à des types particuliers, qui est détaillée sur chaque type impliqué au chapitre 5 ;
- ✓ l'ordre dans lequel les divers composants d'une expression sont évalués pour aboutir aux résultats.

6.1 OPÉRATEURS ET OPÉRANDES

Les opérandes sont les briques élémentaires porteuses d'une valeur au sein d'une expression. L'application des opérateurs (cf. § 2.4) au nombre d'opérandes attendu, constitue le type le plus basique d'expression.

Ces opérandes ne peuvent être que de trois natures :

- ✓ des littéraux (cf. § 6.2) ;
- ✓ des identificateurs (cf. § 2.3.1), qualifiés ou non (cf. § 6.3), à l'exception de l'identificateur vierge ;
- ✓ d'autres expressions.

6.2 LITTÉRAUX

6.2.1 Règles générales

Les littéraux sont l'un des quatre types de jetons définis en Go, avec les identificateurs, les mots-clefs, et les opérateurs, qui sont traités comme la ponctuation.

Pour chaque type de valeur, les littéraux correspondants sont décrits dans la rubrique consacrée au type correspondant du chapitre 5.

Les littéraux de type proprement dits sont simplement les identificateurs des types, ou la première partie des littéraux de valeurs du type correspondant, pour les types qui ne sont définis ni par le langage ni par le code applicatif.

Le tableau 6.1 fournit un résumé de leurs divers formats.

Tableau 6.1 – Littéraux

Nature	Exemples de littéraux
Numériques	`1, 2.3, -4.5E-6, 0755, 0xdeadbeef, -3.2+4.e-3i, 0xEP5`
Runes	`'a', '\U0001D122'`
Chaînes	`"interprétée", `brute``
Fonction	`func(int) string {}`
Tableaux	`[2]int{3, 8}, [...]byte{2, 7}`
Tranches	`[]rune{'a', 'b', 65}`
Cartes	`map[string]int{"un": 1, "deux": 2}`
Structure	`struct { a, b int }{a: 1, b: 2}` `struct { a, b int }{1, 2}`
Type	`int, []float32, map[AdresseIP]Machine`

Pour les littéraux composites, les types des éléments, clefs, et champs apparaissant dans le littéral doivent être tels que les valeurs exprimées soient affectables (cf. §3.6.2) à la partie « littéral de type » du littéral.

La question ne se pose pas pour les littéraux simples, puisqu'ils ne commencent pas par un littéral de type.

6.2.2 Littéraux tableaux, tranches, et cartes

Pour les littéraux tableaux et tranches, le listing 6.1 démontre la plupart des cas :

- ✓ Les éléments n'ont pas obligation d'être du type défini pour le tableau mais doivent être représentables (cf. § 3.6.3) dans ce type, comme le dernier élément de l'exemple, une rune correspondant au point de code U+0021, représentée comme élément `byte` de valeur 33.
- ✓ Les indices explicites sont entièrement optionnels ; en leur absence, les valeurs d'indice utilisé démarrent à 0 et augmentent de 1 à chaque élément.

Listing 6.1 – Littéraux tableaux et tranches
https://osinet.fr/go/expr/play-litteraux-tableaux

```
package main

import "fmt"

func main() {
        // Mêmes règles pour les tranches.
        // Indices:     0   1  5       6  3      4    33
        a := [...]byte{40, 41, 5: 45, 46, 3: 43, 44, '!': '!'}
        fmt.Println(len(a), a)
}

// Résultat.
// 34 [40 41 0 43 44 45 46 0..(26 zéros)..0 33]
```

- ✓ Si un indice est explicite, il doit être représentable comme un entier positif, comme les indices 0, 3, 5, et `'!'` de l'exemple.
- ✓ Les indices explicites n'ont pas besoin d'être en ordre croissant : ils doivent seulement être uniques, comme dans l'exemple, où l'indice 3 est placé après l'indice 5.
- ✓ Après un indice explicite, les indices implicites reprennent leur progression à partir de l'indice suivant au lieu de continuer la progression implicite, comme dans l'exemple, où la valeur 46 est affectée à l'indice 6, tandis que la valeur 44 est affectée à l'indice 4.
- ✓ Les indices implicites doivent aussi être uniques : dans l'exemple, si une valeur d'élément était insérée après l'élément 44 sans clef explicite, elle provoquerait une erreur de clef dupliquée. En effet, la progression implicite donne l'indice 4 pour le 44, donc 5 pour l'élément suivant, mais cette clef a déjà été utilisée pour l'élément 45.
- ✓ Dans les tableaux de longueur implicite initialisés avec une « longueur » trois-points et dans les tranches, l'insertion d'un élément à un indice explicite augmente la longueur du littéral composite en provoquant l'insertion d'autant de valeurs vierges que nécessaire entre la dernière valeur initialisée et la valeur à ajouter. C'est ce que montre la dernière valeur, qui provoque l'insertion de 26 valeurs vierges à 0 et l'allongement du tableau à une longueur 34 bien que seuls sept éléments soient fournis dans le littéral. La longueur des tableaux à longueur explicite est en revanche définie par leur littéral de type.

Les littéraux tranches sont en pratique un raccourci pour l'application d'une tranche complète à un littéral tableau en faisant disparaître la variable intermédiaire nécessaire en pareil cas.

© Dunod – Toute reproduction non autorisée est un délit.

Les éléments des littéraux tableaux, tranches et cartes peuvent eux-mêmes appartenir à des types composites ou leurs pointeurs, eux-mêmes initialisés par des littéraux composites.

En pareil cas, la spécification du type des éléments à chaque élément est facultative, comme dans le listing 6.2, où le premier littéral inclut les types des éléments mais pas le second.

Listing 6.2 – Omission des types d'éléments dans les littéraux composites
https://osinet.fr/go/expr/play-litteraux-composites

```
package main

import "reflect"

type S struct{ champ int }

type T map[string]S

func main() {
        t1 := T{"un": S{1}, "deux": S{champ: 2}}
        t2 := T{"un": {1}, "deux": {champ: 2}}
        println(reflect.DeepEqual(t1, t2)) // true
}
```

6.2.3 Littéraux structures

Pour les littéraux de types structures, les contraintes additionnelles sont :

- ✓ Chaque clef apparaissant dans le littéral doit correspondre à un nom de champ du type.
- ✓ Il n'est pas possible de mélanger dans un même littéral des champs initialisés sans clef et avec clef.
- ✓ Si aucune clef n'est utilisée, toutes les valeurs du littéral doivent correspondre aux champs du type dans leur ordre de déclaration, et il est nécessaire d'inclure des valeurs pour tous les champs, même pour les valeurs vierges.
- ✓ Si des clefs sont utilisées, les initialisations ne sont nécessaires que pour les champs devant porter une valeur autre que la valeur vierge de leur type.
  Par extension, tous les champs peuvent être omis : en ce cas ils sont considérés comme une initialisation avec des clefs toutes absentes, et les champs sont initialisés à leur valeur vierge.
- ✓ Si le type appartient à un autre paquet, et comporte des champs non exportés, il est interdit d'initialiser ces champs.

Comme l'initialisation par une simple liste de valeurs a obligation de fournir des valeurs pour tous les champs, elle conduit à initialiser ces champs non exportés, et n'est donc pas possible.
Un tel littéral doit donc initialiser la structure par noms de champs, en ne fournissant de valeurs que pour les champs exportés différents de la valeur vierge, comme dans le listing 6.3, où tous les champs privés du type `os.Process` sont initialisés à leur valeur vierge, mais où le champ public est initialisé avec une valeur différente.

Listing 6.3 – Littéraux structures et champs non exportés
https://osinet.fr/go/expr/play-litteraux-structures

```
1  package main
2
3  import (
4          "fmt"
5          "os"
6  )
7
8  func main() {
9          var p *os.Process = &os.Process{Pid: 1}
10         fmt.Printf("%#v\n", p)
11 }
12
13 // Résultat: &os.Process{Pid:1, handle:0x0, [...tronqué...]}
```

6.3 QUALIFICATION DES IDENTIFICATEURS

Une expression d'identificateur qualifié se compose :

- ✓ **D'un identificateur de paquet** : autre que l'identificateur vierge. Il provient d'une déclaration **`import`** dans le paquet en cours.
  Le fonctionnement des paquets et de leur import est détaillé au chapitre 9, en particulier § 9.7 et 9.9.
- ✓ **D'un point** : séparant les deux identificateurs.
- ✓ **D'un identificateur** : autre qu'un identificateur de paquet, et qui ne peut pas non plus être l'identificateur vierge.
  Cet identificateur doit avoir été défini par le paquet par lequel il se trouve qualifié, dans sa portée paquet, et doit être exporté, comme défini au § 2.3.2.

© Dunod – Toute reproduction non autorisée est un délit.

La combinaison de ces deux identificateurs permet d'identifier de façon non ambiguë toute donnée manipulable depuis un paquet donné.

Son résultat est un identificateur utilisable de la même façon qu'un identificateur non qualifié.

Exemple : `http.StatusFound` : la constante `StatusFound` du paquet `net/http`.

Morphologiquement, la forme de ces expressions est la même que celle des expressions de sélection. La différence est le type du premier membre, qui est ici un identificateur de paquet, alors que c'est nécessairement une expression primaire dans les expressions de qualification.

6.4 EXPRESSIONS PRIMAIRES

Les expressions dites *primaires* sont celles qui peuvent être combinées comme opérandes d'autres expressions unaires ou binaires.

Cette notion est de peu d'importance pour les développeurs Go en général, et concerne essentiellement les développeurs d'outils manipulant les source Go à suffisamment bas niveau pour devoir se préoccuper de la grammaire.

Le tableau 6.2 récapitule les expressions primaires et les autres.

Tableau 6.2 – Expressions primaires et autres

Expressions primaires (<EP>)	Autres expressions
– opérande – conversion – expression de méthode – <EP>sélecteur – <EP>indexation – <EP>tranchage – <EP>assertion de type – <EP>arguments (invocation)	– opérations générales – comparaison

Les lignes de la forme `<EP>action` correspondent à la concaténation dans le source d'une expression primaire et d'une action. Elles sont décrites dans les paragraphes suivants sous le nom de l'action.

Les opérations primaires sont les seules qui peuvent être utilisées dans un commutateur de type (cf. § 7.3.6) ; le commutateur d'expression (cf. § 7.3.5) n'est pas affecté par cette limitation.

Évolutions de la grammaire

La définition des expressions primaires a évolué au fil des versions :

- ✓ Avant Go 1.10, les expressions de méthode n'étaient pas identifiées comme des expressions primaires en tant que telles, mais l'étaient déjà en tant qu'opérandes. Ticket : 9060.
- ✓ Avant Go 1.4, les appels de fonction intégrés (*builtins*) étaient identifiés comme des expressions primaires en tant que telles dans la spécification mais indifférenciés des appels de fonction en général. Go 1.4 les a regroupés avec les appels de fonctions et a renommé l'expression primaire `appel` de fonction (*call*) en `arguments`.

Dans les deux cas, cela n'a pas affecté la compatibilité pour les sources, mais a modifié le fonctionnement interne de l'analyse syntaxique.

Dans le second cas, l'évolution de spécification de spécification n'a fait que formaliser ce qui était déjà le comportement du compilateur.

6.5 SÉLECTION

Une expression primaire de sélecteur, ou sélection, est composée :

- ✓ **D'une expression primaire** : à l'intérieur de laquelle une partie va être sélectionnée.
  Celle-ci ne peut pas être un identificateur de paquet, à l'inverse des expressions de qualification d'identificateurs.
- ✓ **D'un point** : séparant expression et identificateur.
- ✓ **D'un identificateur** : de champ ou de méthode, qui est dit le *sélecteur*, et ne peut pas être l'identificateur vierge.

Son résultat est une valeur du type du sélecteur. Pour les expressions primaires de types interfaces, la sélection d'une méthode – mais pas d'un champ – renvoie la méthode du type dynamique implémentant l'interface.

Si le sélecteur désigne un champ ou une méthode d'un type intégré du type de l'expression primaire, le champ intégré est implicitement sélectionné, comme dans le listing 6.4.

Comme le montrent les sélections de la deuxième partie du listing, ces sélections peuvent également être explicites et chaînées : le résultat d'une sélection est une expression primaire, à laquelle une nouvelle sélection peut être appliquée.

Le nombre de niveaux de types intégrés entre l'expression et le champ intégré effectivement désigné par la sélection est dit la *profondeur* du champ ou de la méthode concernée.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 6.4 – Sélection et profondeur de champs
https://osinet.fr/go/expr/play-selection-profondeur

```
package main

import (
        "fmt"
        "golang.org/x/net/dns/dnsmessage"
)

type Requête struct {
        dnsmessage.Message
}

func main() {
        var e Requête
        fmt.Printf("%#v\n%#v\n%#v\n%#v\n%#v\n%#v\n%#v\n%#v\n",
                e.Message,  // Profondeur 0
                e.Header,   // Profondeur 1
                e.OpCode,   // Profondeur 2
                e.GoString, // Profondeur 3
                // Sélections de profondeur 0 équivalentes
                e.Message.Header,
                e.Message.Header.OpCode,
                e.Message.Header.OpCode.GoString,
        )
}
```

Cette sélection implicite peut entraîner des ambiguïtés, le même nom de champ ou de méthode pouvant apparaître sur plusieurs chaînes de sélection à des profondeurs différentes, comme dans le listing 6.5, qui illustre les deux règles applicables :

- ✓ **Priorité au moins profond** : la sélection `e.bar` pourrait donner lieu à deux résolutions : `e.I1.bar` et `e.I2.bar`. Comme le montre le résultat, Go donne la priorité à la sélection du champ de plus faible profondeur : en effet `e.I1.bar` est de profondeur 1, tandis que `e.I2.bar` est une sélection implicite de de profondeur 2 : `e.I2.C.bar` ; c'est donc la sélection `e.I1.bar` qui est appliquée.
- ✓ **Pas d'ambiguïté** : la sélection `e.foo` pourrait donner lieu à deux résolutions équivalentes : `e.I1.foo` et `e.I2.foo`. Comme les deux interprétations accèdent à un champ à la même profondeur 1, l'utilisation de `e.foo` provoque une erreur de compilation pour sélecteur ambigu : il est nécessaire d'utiliser une chaîne de sélections explicites comme `e.I1.foo` ou `e.I2.foo` pour lever l'ambiguïté.

La sélection donne lieu à un déréférencement implicite pour certains cas où l'expression primaire est un pointeur, comme dans les listings 6.6 et 6.7 :

Listing 6.5 – Sélecteurs homonymes et ambigus
https://osinet.fr/go/expr/play-selecteurs-homonymes

```
package main

type C struct{ bar complex64 }
type I1 struct {foo int; bar string}
type I2 struct {foo int; C}
type Externe struct {I1; I2}

func main() {
        e := Externe{I1: I1{1, "un"}, I2: I2{2, C{3 + 4i}}}

        // Priorité à la plus faible profondeur
        println(e.bar == e.I1.bar) // true

        println(e.I1.foo, e.I2.foo) // 1 2

        // Erreur: sélecteur ambigu e.foo
        // println(e.foo)
}
```

Listing 6.6 – Sélecteurs sur expressions pointeurs
https://osinet.fr/go/expr/play-selecteurs-pointeurs

```
package main

import "fmt"

type T struct{ i int }
func (t T) f() int  { return t.i }
func (t *T) g() int { return t.i }
type PT *T

func main() {
        t := T{5}
        var ptDéfini PT = &t
        var ptAnonyme = &t
        fmt.Println(
                ptDéfini.i, ptAnonyme.i,       // 5 5
                ptAnonyme.f(), ptAnonyme.g(), // 5 5
                // Erreurs: PT n'a ni champ ni méthode f ou g
                // ptDéfini.f(), ptDéfini.g(),
        )
}
```

✓ **Sélection de champs** : sur une expression pointeur `E`, de type défini ou non, pour laquelle `(*E).x` est valide et sélectionne un champ, `E.x` est valide et compilé comme `(*E).x`. C'est le cas pour `ptDéfini.i` et `ptAnonyme.i`.

© Dunod – Toute reproduction non autorisée est un délit.

✓ **Sélection de méthodes** : sur une expression pointeur `E` pour laquelle `(*E).m` est valide et sélectionne une méthode :
- si le type de l'expression n'est pas défini, comme pour `ptAnonyme` dans l'exemple, `E.m` est valide et compilé comme `(*E).m` ;
- si le type de l'expression est un type pointeur défini, comme pour `ptDéfini` dans l'exemple, `E.m` est une erreur, que le récepteur sur le type amont soit de type adresse ou valeur, comme pour les méthodes `f` et `g` dans l'exemple.

Le traitement des sélections appliquées à une valeur **`nil`** diffère selon le type de celui-ci, et selon que le sélecteur est un champ ou une méthode, comme le montre le listing 6.7. Les règles sont celles du tableau 6.3.

Tableau 6.3 – Sélection sur `nil`

	Interface	Pointeur sur type concret
Champ	erreur de compilation	panique
Méthode	panique	pointeur sur la méthode

L'erreur de compilation correspond au cas d'erreur le plus général dans les expressions de sélection : l'utilisation comme sélecteur d'un identificateur ne pouvant être résolu, puisque les types interfaces ne comportent que des méthodes et pas de champs.

Listing 6.7 – Sélection sur `nil`
https://osinet.fr/go/expr/play-selecteurs-nil

```
1  package main
2
3  import "fmt"
4
5  type I interface{ f() }
6  type C int
7  func (c *C) f() { fmt.Println(c) }
8
9  func main() {
10         var c *C
11         fmt.Println(c, c.f) // <nil> <un pointeur>
12         c.f()               // <nil>
13
14         var i I
15         fmt.Println(i)      // <nil>
16         // Panique: déréférencement de <nil>
17         // fmt.Println(i.f)
18 }
```

La sélection est l'un des types d'expressions utilisable comme destination dans une affectation (cf. § 7.2.5).

6.6 EXPRESSIONS DE MÉTHODES

Une expression de méthode est une expression primaire qui se compose, comme dans le listing 6.8 :

✓ **D'un type** : un identificateur de type défini, ou type pointeur associé : pour un type défini `C`, cela signifie soit `C`, soit `APC` où **`type`** `APC = *C`, soit `(*C)`. Les parenthèses sont indispensables dans le dernier cas du fait des priorités.
Le type alias – `APC` dans l'exemple – vers le pointeur sur le type visé est utilisable alors qu'un type défini basé sur le type visé – `DPC` dans l'exemple – ne l'est pas, puisque les types définis à partir d'un autre type défini n'héritent pas de ses méthodes (cf. § 5.3).
Le type peut être un type interface, comme `I` dans l'exemple.

Listing 6.8 – Expressions de méthodes
https://osinet.fr/go/expr/play-expressions-methodes

```
1  package main
2
3  import "fmt"
4
5  type I interface{ parValeur(string) }
6  type C struct{ n int }
7  type APC = *C // Type alias de pointeur sur C
8  type DPC *C   // Type défini pointeur sur C: inutilisable
9
10 func (c C) parValeur(f string)   { fmt.Printf(f, c.n) }
11 func (c *C) parAdresse(f string) { fmt.Printf(f, c.n) }
12
13 func main() {
14         var c C = C{73}         // Valeur du type concret
15         var i I = c             // Valeur du type interface
16         valC := C.parValeur
17         valI := I.parValeur
18         adr0 := APC.parValeur
19         adr1 := (*C).parAdresse
20         adr2 := APC.parAdresse // DPC n'a pas d'équivalent.
21         valC(c, "%o\n")         // 111
22         valI(i, "%o\n")         // 111
23         adr0(&c, "%o\n")        // 111
24         adr1(&c, "%o\n")        // 111
25         adr2(&c, "%o\n")        // 111
26 }
```

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ **D'un point** : séparant type et identificateur de méthode.
- ✓ **D'un identificateur** : de méthode appartenant au jeu de méthodes du type désigné par le premier identificateur.
  S'il désigne une méthode à récepteur pointeur, le premier identificateur doit désigner le type pointeur.

Le résultat d'une telle expression est une valeur de type fonction. Cette fonction renvoie les mêmes résultats que la méthode, mais l'argument masqué correspondant au récepteur de la méthode est converti en un argument explicite.

Le type de cet argument dépend à la fois du type de récepteur de la méthode originale et de la forme de l'expression, comme illustré dans le listing 6.8 et résumé dans le tableau 6.4.

Tableau 6.4 – Type d'une expression de méthode

Type	Récepteur	Premier paramètre	Exemple
type défini	valeur	valeur	`valC`, `valI`
type pointeur	valeur	pointeur	`adr0`
type pointeur	pointeur	pointeur	`adr1`, `adr2`

Dans tous les cas, le récepteur n'est qu'un paramètre explicite ajouté en tête de la signature de la fonction résultante. Pour obtenir une fonction associée à une valeur de récepteur prédéfinie et conservant la signature apparente de la méthode, l'équivalent est une valeur de méthode, définie au § 6.7.

6.7 VALEURS DE MÉTHODES

Les valeurs de méthodes sont le pendant des expressions de méthode, mais en utilisant la valeur du récepteur au lieu de son type, comme dans le listing 6.9. Elles se composent :

- ✓ **D'une valeur** : une expression d'un type qui dispose d'une méthode du nom utilisé comme second opérande.
  Le type de cette valeur peut être un type interface, comme `valI` dans l'exemple.
- ✓ **D'un point** : séparant valeur et identificateur de méthode.
- ✓ **D'un identificateur** : de méthode appartenant au jeu de méthodes du type de la valeur passée comme premier opérande, directement ou par champ intégré comme le montre l'utilisation des méthodes du type `C` sur la valeur ou l'adresse d'une valeur du type `Enveloppe` qui l'intègre.

Le listing illustre le fait que le premier opérande n'est pas nécessairement un identificateur : toute valeur dont le type porte la méthode demandée est utilisable, et les pointeurs de types alias ou anonymes sont utilisables sans restrictions particulières.

Listing 6.9 – Valeurs de méthodes
https://osinet.fr/go/expr/play-valeurs-methodes

```go
package main

import "fmt"

type I interface{ parValeur(string) }
type C struct{ n int }

func (c C) parValeur(f string)    { fmt.Printf(f, c.n) }
func (c *C) parAdresse(f string) { fmt.Printf(f, c.n) }

type Enveloppe struct{ C }
type APE = *Enveloppe

func main() {
        e := Enveloppe{C{73}}    // Valeur du type concret
        i := e.C                 // Valeur du type interface
        var a APE = &e           // Pointeur du type alias
        var b = &e               // Pointeur de type anonyme
        valE := e.parValeur      // <fonction 1>
        valC := (e.C).parValeur // <fonction 1>
        valI := i.parValeur      // <fonction 1>
        adr0 := a.parValeur      // <fonction 1>
        adr1 := b.parAdresse     // <fonction 2>
        adr2 := a.parAdresse     // <fonction 2>
        fns := []func(string){valC, valE, valI, adr0, adr1, adr2}
        for _, f := range fns {
                f("%o\t") // 111 111 111 111 111 111
        }
        println()
        e.C.n = 146 // Modification du récepteur
        for _, f := range fns {
                f("%o\t") // 111 111 111 111 222 222
        }
}
```

Le résultat de l'expression est une fonction dont la signature correspond à la méthode d'origine : à la différence d'une expression de méthode, la fonction ne prend pas d'argument supplémentaire explicite correspondant au receveur, mais celui-ci est lié à la valeur de méthode par l'expression, comme dans les listings 6.8 et 6.9 :

✓ **expression de méthode** : `valC(<récepteur>, <format>)`

✓ **valeur de méthode** : `valC(<format>)`

La principale utilisation des valeurs de méthodes consiste à les faire suivre d'une liste d'arguments pour réaliser un appel de méthode traditionnel.

© Dunod – Toute reproduction non autorisée est un délit.

Dans l'implémentation, la fonction dispose de la valeur du récepteur tel qu'il était au moment de l'évaluation de la valeur de méthode, valeur qui ne peut plus être modifiée.

Comme le montre le listing 6.9, ceci signifie que :

- ✓ Si le récepteur est passé par valeur, les modifications ultérieures de l'instance sur laquelle est appelée la méthode ne changeront pas la valeur du récepteur dans la fonction.
  C'est le cas de `valC`, `valE`, `valI` et `adr0`.
  Le cas de `adr0` est notable : bien que l'expression soit évaluée à partir du pointeur `a`, c'est bien la valeur qui est passée comme récepteur et non le pointeur.
  En effet, la méthode spécifiant un receveur par valeur, le pointeur est automatiquement déréférencé comme décrit au § 3.4.6.
- ✓ En revanche, si le récepteur est passé par adresse, c'est celle-ci qui est la valeur du récepteur et, de ce fait, les modifications de la valeur de l'instance s'appliquent puisque celle-ci est déréférencée à partir du récepteur.
  C'est le cas de `adr1` et `adr2`.

Des valeurs de méthode différentes peuvent renvoyer la même fonction, comme le montre le listing 6.9 : les quatre premières expressions renvoient une même fonction, et les deux dernières en renvoient une autre, ce qui est vérifiable en comparant leur adresse sous forme numérique.

6.8 INDEXATION

L'indexation, illustrée dans le listing 6.10, est une expression primaire permettant d'accéder à l'un des éléments de certains types construits par agrégation de multiples instances d'un même type élémentaire :

- ✓ **Types composites** : tableaux, tranches, cartes, mais pas les structures.
- ✓ **Pointeurs** : les pointeurs sur tableaux peuvent être indexés et, comme pour les expressions de sélection et les valeurs de méthodes, le pointeur est automatiquement déréférencé, de sorte que pour un pointeur sur tableau `pt`, les expressions `pt[i]` et `(*pt)[i]` sont équivalentes, comme aux lignes 15 et 21 du listing 6.10.
  À la différence de ces dernières, le déréférencement est aussi utilisable pour les types pointeurs définis.
- ✓ **Chaînes** : les chaînes de caractère sont traitées dans ces expressions comme des tranches d'octets `[]byte`.

L'expression d'indexation se compose :

- ✓ **D'une expression** : à l'intérieur de laquelle un élément va être sélectionné.
- ✓ **D'un crochet ouvrant** : séparant expression et index ou clef.
- ✓ **D'un index ou d'une clef** : pour les types cartes, cette partie de l'expression est une clef, d'un type affectable au type de clef de la carte.

Pour les autres cas, c'est un nombre entier. Si cet entier est une constante entière non typée, elle doit être représentable par le type `int`. Pour que l'expression soit valide, ce nombre doit être compris entre 0 inclus et la longueur de la donnée indexée, moins un, intervalle dénommé *étendue* (*range*).

✓ **D'un crochet fermant** : terminant l'expression.

Listing 6.10 – Exemples d'indexation
https://osinet.fr/go/expr/play-index

```
package main

import "fmt"

type Tableau [2]int
type Tranche []int
type Carte map[string]int
type PT *Tableau

func main() {
        var ch string = "@ = Unicode 64"
        var ta Tableau = Tableau{'@', 8}
        var tr Tranche = Tranche{'@', 7}
        var ca Carte = Carte{"zero": '@', "un": 1}
        var pt PT = &ta

        fmt.Println(ch[0])       // 64
        fmt.Println(ta[0])       // 64
        fmt.Println(tr[0])       // 64
        fmt.Println(ca["zero"]) // 64
        fmt.Println(pt[0])       // 64

        x, ok := ca["zero"] // Succès vérifié
        fmt.Println(x, ok)  // 64 true
        x, ok = ca["0"]     // Échec intercepté
        fmt.Println(x, ok)  // 0 false
        x = ca["0"]         // Échec ignoré
        fmt.Println(x)      // 0 false
}
```

En cas de succès, comme dans tout autre langage usuel, elle renvoie l'élément désigné par l'index ou la clef ; cet élément est du type des éléments de l'agrégat, ou de type octet pour les chaînes.

Ce résultat est adressable si la valeur indexée est une tranche, un tableau adressable, ou un pointeur sur un tableau lui-même adressable uniquement.

© Dunod – Toute reproduction non autorisée est un délit.

Si la valeur indexée est une carte, l'expression peut être utilisée pour assigner à deux valeurs au lieu d'une seule : la première reçoit le résultat de l'indexation, et la seconde un booléen indiquant si la clef correspondait à un élément existant. Si cette seconde valeur est fausse, la valeur de la première est la valeur vierge du type d'éléments de la valeur indexée. Ce mécanisme fonctionne même si la carte est `nil`.

Pour tous les autres types indexables, l'utilisation d'un index hors de l'étendue provoque une erreur, qui peut être détectée à la compilation pour les tableaux et les chaînes indexées par des valeurs constantes, ou à l'exécution.

L'indexation est l'un des types d'expressions utilisable comme destination d'affectation (cf. § 7.2.5), aussi bien dans sa forme numérique que dans l'indexation de carte. Certains destinations d'affectation sont néanmoins invalides :

- ✓ Les chaînes de caractères sont constantes, de sorte qu'il est impossible de modifier un de leurs éléments par une affectation vers une indexation.
- ✓ L'utilisation d'un index hors de l'étendue provoque une erreur de sortie d'étendue, pour éviter un écrasement mémoire.
- ✓ L'utilisation d'une carte `nil` est interdite en écriture.

6.9 TRANCHAGE (DÉCOUPAGE)

6.9.1 Définition

Le tranchage [1] (*slicing*) est une expression produisant une valeur de type tranche anonyme à partir d'une suite d'éléments contigus, présents dans certains types construits par agrégation de multiples instances d'un même type élémentaire :

- ✓ **Types composites** : tableaux, tranches, mais ni les cartes, ni structures.
- ✓ **Pointeurs** : les pointeurs sur tableaux peuvent être tranchés et, comme pour les expressions de sélection, les valeurs de méthodes, et l'indexation, le pointeur est automatiquement déréférencé, de sorte que pour un pointeur sur tableau `pt`, les expressions `pt[b:h:m]` et `(*pt)[b:h:m]` sont équivalentes.
  Comme pour l'indexation et à la différence des expressions de sélection et des valeurs de méthodes, le déréférencement est aussi utilisable pour les types pointeurs définis.
- ✓ **Chaînes** : les chaînes de caractère sont traitées dans ces expressions comme des tranches d'octets `[]byte`.

Il est similaire à l'opérateur de tranchage Python, et aux fonctions JavaScript `slice` et PHP `array_slice`.

1. Le terme *découpage* est également employé avec exactement la même signification.

Comme en Python et JavaScript, mais au contraire de PHP, la deuxième valeur de tranche est la borne supérieure exclue de l'intervalle à extraire, et non la longueur souhaitée pour la tranche. Au contraire de ces trois langages, il n'accepte pas l'utilisation d'indices négatifs pour désigner des décalages à partir de la fin de la valeur source plutôt que de son début.

Visuellement proche de l'indexation par l'utilisation d'une notation à base de crochets, il en diffère en ce qu'il retourne toujours une tranche d'éléments et non un élément unique.

6.9.2 Notation

Une expression de tranchage, comme illustré par le listing 6.11, se compose :

- ✓ **D'une expression** : à l'intérieur de laquelle une plage d'éléments contigus va être sélectionnée.
- ✓ **D'un crochet ouvrant** : séparant expression et jeu d'indices.
- ✓ **D'un jeu d'indices** : précisant les bornes de la plage d'éléments utilisés dans la valeur à trancher, et éventuellement la capacité maximale souhaitée pour la tranche résultante.
- ✓ **D'un crochet fermant** : terminant l'expression.

Listing 6.11 – Syntaxe du tranchage
https://osinet.fr/go/expr/play-tranchage

```
package main

import "fmt"

func main() {
        var t = []int{0, 1, 2, 3}
        //  Tranchage Capacité Résultat
        s1 := t[:]      // 4    [0, 1, 2, 3]
        s2 := t[2:]     // 2    [2, 3]
        s3 := t[:2]     // 4    [0, 1]
        s4 := t[1:3]    // 3    [1, 2]
        s5 := t[:2:2]   // 2    [0, 1]
        s6 := t[1:2:2] // 1    [1]
        fmt.Println(cap(s1), s1, cap(s2), s2, cap(s3), s3,
                cap(s4), s4, cap(s5), s5, cap(s6), s6)
}
```

© Dunod – Toute reproduction non autorisée est un délit.

Depuis Go 1.2, le jeu d'indices peut avoir deux formats différents, à base d'intervalles semi-ouverts à droite, tandis que dans les versions antérieures seul le premier format était disponible :

✓ **Format simple** : `bas:haut` pour une tranche [*bas*, *haut*[
- indice bas, inclus dans la plage, et égal à 0 si omis ;
- séparateur deux-points ;
- indice haut, exclu de la plage, égal à la longueur du tableau sous-jacent si omis ;

✓ **Format complet** : `bas:haut:max` pour une tranche [*bas*, *haut*[pointant vers un tableau [*bas*, *max*[
- indice bas, inclus dans la plage, et égal à 0 si omis ;
- séparateur deux-points ;
- indice haut, exclu de la plage, et ne pouvant pas être omis ;
- séparateur deux-points ;
- indice maximal du tableau sous-jacent, exclu, définissant la capacité de la tranche comme `max - bas`, et ne pouvant pas être omis.

Cas particulier découlant de cette notation : pour une valeur tranchable `v`, la notation `v[:]` est valide et désigne une tranche exposant l'intégralité de la donnée tranchée.

Pour le format complet, son annonce d'introduction [2] indique qu'il est possible que des versions futures de Go incluent des valeurs par défaut pour les valeurs actuellement obligatoires, mais la situation est inchangée en Go 1.14.

6.9.3 Contraintes et résultats

L'opérande et les valeurs des indices utilisés dans l'expression sont soumis à de multiples contraintes et affectent le type des résultats :

✓ Si l'opérande est un tableau, il doit être adressable, provoquant une erreur de compilation dans le cas contraire. Ceci interdit en particulier d'appliquer le tranchage au résultat d'une fonction renvoyant un tableau, comme dans le listing 6.12.
Les solutions en pareil cas sont :
- modifier la fonction pour renvoyer l'adresse du tableau, de sorte que le déréférencement automatique du pointeur renvoie sur un tableau adressable, dont l'adresse est précisément la valeur renvoyée par la fonction, permettant un tranchage direct comme pour `Adr` dans l'exemple ;
- passer par une variable intermédiaire qui recevra le résultat tableau et sera adressable pour l'opération de tranchage, comme `tab` dans l'exemple ;
- modifier la fonction pour renvoyer une tranche, qui est toujours adressable, comme `Tra` dans l'exemple.

2. https://osinet.fr/go/expr/go-tranchage-complet

- ✓ Les indices doivent être convertibles en entiers ou, dans le cas de constantes entières non typées, doivent être représentables (cf. § 3.6.3) en entiers.
- ✓ Outre le fait d'être entiers, les indices doivent respecter les inégalités :

  $$0 \leqslant bas \leqslant haut \leqslant max \leqslant longueur$$

  où `longueur` est la longueur de l'opérande pour les tableaux et chaîne, mais la capacité – donc la longueur du tableau sous-jacent – plutôt que la longueur pour les tranches.
  Cette étendue est valable pour les trois indices, et est différente de celle des expressions de sélection, où elle exclut la valeur supérieure.
- ✓ Si l'opérande tranché est une tranche ou une chaîne de caractères typée, le résultat du tranchage est une valeur non constante du même type que l'opérande. S'il est une chaîne de caractères non typée, comme un littéral chaîne, le résultat est une valeur non constante de type chaîne.

Listing 6.12 – Tranchage de tableau et adressabilité
https://osinet.fr/go/expr/play-adressabilite-tranchage

```
package main

import "fmt"

var chiffres = [...]byte{0, 1, 2}

func Val() [3]byte  { return chiffres }
func Adr() *[3]byte { return &chiffres }
func Tra() []byte   { return chiffres[:] }

func main() {
        // invalid operation Foo()[:]
        // (slice of unaddressable value)
        // Val()[:]

        fmt.Println(Adr()[:]) // [0 1 2]
        fmt.Println(Tra()[:]) // [0 1 2]
        tab := Val()          //
        fmt.Println(tab[:])   // [0 1 2]
}
```

6.9.4 Implantation mémoire

La particularité la plus notable du tranchage est le fait qu'il réutilise dans toute la mesure du possible l'agrégat présent dans l'opérande auquel il s'applique, comme le montre le listing 6.13.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 6.13 – Réutilisation mémoire lors du tranchage
https://osinet.fr/go/expr/play-memoire-tranchage

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          a := []int{0, 1, 2}
7          b := a[:]
8          fmt.Printf("%d %v\n", cap(a), a)
9          fmt.Printf("%d %v\n", cap(b), b)
10
11         a[0] = 10               // Même tableau
12         fmt.Println(cap(b), b) // b modifié
13
14         a = append(a, 3)        // Réallocation
15         a[0] = -1               // Plus le même tableau
16         fmt.Println(cap(b), b) // b non remodifié
17 }
```

Lors d'un tranchage à partir d'un tableau ou d'une tranche, la tranche résultat s'appuie sur le tableau d'origine ou le tableau sous-jacent de la tranche d'origine.

De ce fait, la modification des éléments du tableau d'origine modifie toutes les tranches résultant du tranchage du même tableau, ou de tranchages subséquents de l'une d'entre elles, comme le montrent les lignes 11-12 de l'exemple.

Mais ce partage n'est qu'un état à un instant donné, et non un état persistant, comme le montrent les lignes 14-16 de l'exemple : lors d'un `append` sur la donnée tranche d'origine, le tableau sous-jacent est réalloué, et les modifications apportées au tableau d'origine sont dorénavant sans effet sur les tranches qui en ont été extraites.

Plus généralement, le listing 6.14 détaille l'utilisation des tableaux lors de tranchages, avec ou sans capacité explicite. Dans cet exemple, le type `slice` est une copie de l'implémentation interne des tranches dans le runtime Go, copiée depuis `src/runtime/slice.go`, utilisée pour examiner l'implémentation interne des tranches le plus simplement possible au moyen des conversions non protégées permises par le paquet `unsafe` (cf. *Plus en ligne*).

✓ Ligne 18 : `ar` est un tableau des huit premiers entiers.

✓ Lignes 19-22, 33-35, 43-45 : le tranchage à partir de l'index 0 crée une tranche dont le tableau sous-jacent est exactement celui d'origine, comme le montre l'égalité de leurs pointeurs.

✓ Lignes 19, 21, 34, 44 : lors d'un tranchage simple, la tranche résultat initiale pointe sur l'ensemble du tableau d'origine et s'en autorise l'intégralité, commme le montre la capacité égale à 8.

Listing 6.14 – Implantation mémoire du tranchage
https://osinet.fr/go/expr/play-mecanique-tranchage

```go
package main

import (
        "fmt"
        "unsafe"
)

// Copie du type privé runtime.slice.
type slice struct {
        array unsafe.Pointer
        len   int
        cap   int
}

type P = unsafe.Pointer

func main() {
        var ar = [8]int{0, 1, 2, 3, 4, 5, 6, 7}
        tr04x := ar[0:4]   // Tableau interne == origine.
        tr046 := ar[0:4:6] // utilisé avec un décalage 0.
        sl04x := (*slice)(P(&tr04x))
        sl046 := (*slice)(P(&tr046))

        tr48x := ar[4:8]   // Tableau interne == origine,
        tr488 := ar[4:8:8] // utilisé avec un décalage 4.
        sl48x := (*slice)(P(&tr48x))
        sl488 := (*slice)(P(&tr488))

        ar[1] = 11 // Modifier le tableau modifie les tranches
        ar[5] = 15 // Modifier le tableau modifie les tranches

        fmt.Println("               Tableau Lon Cap Contenu")
        fmt.Printf("Tableau %16p\n", &ar)
        fmt.Printf("Tranche 0:4   %v %v\n", sl04x, tr04x)
        fmt.Printf("Tranche 0:4:6 %v %v\n", sl046, tr046)
        fmt.Printf("Tableau[4] %13p\n", &ar[4])
        fmt.Printf("Tranche 4:8   %v %v\n", sl48x, tr48x)
        fmt.Printf("Tranche 4:8:8 %v %v\n", sl488, tr488)

}
// Résultats
//                Tableau Lon Cap Contenu
// Tableau          0x456000
// Tranche 0:4   &{0x456000 4 8} [0 11 2 3]
// Tranche 0:4:6 &{0x456000 4 6} [0 11 2 3]
// Tableau[4]       0x456010
// Tranche 4:8   &{0x456010 4 4} [4 15 6 7]
// Tranche 4:8:8 &{0x456010 4 4} [4 15 6 7]
```

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ Lignes 20, 22, 35, 45 : lors d'un tranchage à capacité explicite, la tranche résultat initiale pointe aussi sur l'ensemble du tableau d'origine, mais ne s'en autorise qu'une longueur limitée à celle requise lors du tranchage, comme le montre la capacité égale à 6 – 0 = 6.
- ✓ Lignes 24-27, 34-38, 46-48 : le tranchage à partir d'un index autre que zéro crée lui aussi une tranche dont le tableau est exactement celui d'origine, mais affecté d'un décalage, comme le montre l'égalité entre l'adresse de du tableau sous-jacent et celle de l'élément 4 du tableau d'origine : il n'y a pas de nouvelle allocation.
- ✓ Lignes 29-30 et tous les affichages : la modification du tableau source modifie bien les tranches résultantes, confirmant l'absence de réallocation.

Cette réutilisation implique un cas particulier, celui des tranches **`nil`**.

Comme décrit au § 4.2.1, la tranche **`nil`** ne comporte pas de tableau sous-jacent valide. De ce fait, sa longueur et sa capacité sont nécessairement nulles, et seul un tranchage `[0:0:0]`, sous toutes ses formes, lui est applicable

Il renvoie la même tranche **`nil`** comme résultat, ainsi que le montre le listing 6.15.

Listing 6.15 – Tranchage de `nil`
https://osinet.fr/go/expr/play-tranchage-nil

```
1  package main
2
3  func main() {
4      var tr []int
5      println(
6          tr, tr == nil,              // [0/0]0x0 true
7          tr[:], tr[:] == nil,        // [0/0]0x0 true
8          tr[0:], tr[0:] == nil,      // [0/0]0x0 true
9          tr[:0], tr[:0] == nil,      // [0/0]0x0 true
10         tr[:0:0], tr[:0] == nil,    // [0/0]0x0 true
11         tr[0:0:0], tr[:0] == nil, // [0/0]0x0 true
12     )
13 }
```

6.10 ASSERTIONS DE TYPES

6.10.1 Définition

L'assertion de type, illustrée dans le listing 6.16, est une expression primaire permettant d'obtenir à partir d'une valeur de type statique interface une valeur portant son type dynamique.

Comme son nom l'indique, elle exprime l'hypothèse que le type dynamique de la valeur interface est celui spécifié dans l'expression, et réussit si cette hypothèse est vérifiée. Cette assertion peut réussir dans deux cas :

- ✓ Si le type choisi est un type interface, l'assertion réussit si le type dynamique présent dans la valeur d'origine implémente cette interface, et la valeur renvoyée est une valeur interface du type statique choisi et dont le type dynamique est celui d'origine.
- ✓ Si le type choisi est un type concret, l'assertion réussit si celui-ci et le type dynamique de la valeur d'origine sont identiques (cf. § 3.6.1), ce qui suppose que le type choisi implémente les méthodes du type interface de la valeur d'origine.

Cas particulier : les types anonymes ne peuvent implémenter de méthodes. La définition précédente implique qu'ils ne peuvent être utilisés avec succès dans une assertion de type que si le type de la valeur d'origine est l'interface vide `interface{}`. C'est ce que montrent les lignes 19, 30, et 31 du listing 6.16.

6.10.2 Notation

Elle est de la forme `<expression>.(<type>)` comme aux lignes 22, 24, 28 et 30 du listing 6.16 :

- ✓ **Une expression** : dont la valeur est d'un type interface.
- ✓ **Un délimiteur ouvrant** : la séquence « `.(` ».
- ✓ **Un type** : le type choisi peut être un type concret ou un autre type interface. Qu'il soit concret ou interface, il peut être défini ou anonyme.
  L'exemple illustre ces quatre cas.
- ✓ **Un délimiteur fermant** : une parenthèse fermante « `)` ».

6.10.3 Résultats

L'expression peut être utilisée comme une valeur unique ou deux valeurs. La première – ou seule – valeur reçoit le résultat de l'indexation, toujours du type choisi, et la seconde un booléen.

- ✓ En cas de succès, la première valeur est celle contenue dans la valeur interface d'origine et, s'il est utilisé, le booléen est vrai.
- ✓ En cas d'échec, si le booléen est utilisé, par exemple dans une affectation à deux variables, la première valeur est la valeur vierge du type spécifié, et le booléen est faux, comme le montrent les lignes 16-17 et 24 du listing 6.17.
- ✓ En cas d'échec, si le booléen n'est pas utilisé, l'expression provoque une panique d'exécution, et ne renvoie pas de résultat, comme le montre les lignes 11-12, 18-20 et 25-26 du listing 6.17, où la valeur réaffectée de la variable `huit` se révèle ne pas avoir été modifiée, la panique survenant avant l'affectation.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 6.16 – Assertions de type
https://osinet.fr/go/expr/play-assertions-type

```go
package main

import (
    "expvar"
    "fmt"
)

type I interface{}        // Type défini interface vide
type A = struct{ N int } // Alias de type anonyme structure
type T A                  // Type défini structure

// T implémente fmt.Stringer et expvar.Var
func (t T) String() string { return fmt.Sprintf("%d", t.N) }

func main() {
    var t T = T{4}        // Valeur de type défini structure
    var v expvar.Var = t // Valeur de type interface
    var a A = t           // Valeur de type anonyme structure
    var i I = a           // Valeur de type interface vide

    // Assertions de types définis
    c, okc := v.(T)
    fmt.Printf("%T %t\n", c, okc)    // main.T true
    str, oks := v.(fmt.Stringer)
    fmt.Printf("%T %t\n", str, oks) // main.T true

    // Assertions de types anonymes
    ai, okai := v.(interface{ String() string })
    fmt.Printf("%T %t\n", ai, okai) // main.T true
    ac, okac := i.(struct{ N int })
    fmt.Printf("%T %t\n", ac, okac) // struct { N int } true
}
```

6.11 APPELS DE FONCTIONS/MÉTHODES

6.11.1 Définition et notation

L'appel – ou invocation – de fonction combine une expression primaire dont la valeur est une fonction avec une autre expression primaire de type « arguments ».

Listing 6.17 – Panique sur assertion de type
https://osinet.fr/go/expr/play-assertions-panique

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var i interface{} = "chaine"
7          var huit int = 8
8
9          defer func() {
10                 if e := recover(); e != nil {
11                         fmt.Println(e)
12                         fmt.Printf("Huit %#v\n", huit)
13                 }
14         }()
15
16         huit, ok := i.(int)    // Pas de panique
17         fmt.Println(huit, ok)
18         huit = 8
19         huit = i.(int)         // Panique: string != uint8
20         fmt.Println(huit)      // Non atteint
21 }
22
23 // Résultat:
24 // 0 false
25 // interface conversion: interface {} is string, not int
26 // Huit 8
```

Les trois formes courantes d'appel, présentées dans le listing 6.18 aux lignes 13 à 15, semblent différentes mais sont en fait une seule et même forme :

✓ Dans l'appel de fonction traditionnel, ligne 13, la première expression est un identificateur de fonction, désignant une valeur non **`nil`**.
Elle peut aussi être n'importe quelle autre forme de valeur fonction, comme dans tout autre langage dans lequel les fonctions sont des objets de première classe, à l'image d'ECMAScript, PHP, ou Python.

✓ Dans l'appel de méthode traditionnel, ligne 14, la première expression est la concaténation d'une valeur, d'un point, et d'un nom de méthode.
Ce n'est rien d'autre qu'un appel de fonction, dans lequel la première expression est une valeur de méthode – `t.f` dans cet exemple – comme défini au § 6.7.
Ceci permet en particulier l'utilisation automatique du pointeur si le premier membre de l'expression valeur de méthode est adressable et si la méthode a un récepteur pointeur sur le type de ce membre.

© Dunod – Toute reproduction non autorisée est un délit.

✓ Dans l'appel d'expression de méthode, ligne 15, qui évoque superficiellement la notation des appels de méthodes statiques – inexistants en Go – en ECMAScript ou PHP, la première expression est une expression de méthode, comme défini au § 6.6.
Cette analogie est toutefois superficielle : dans une méthode statique, la méthode n'est pas invoqué avec un paramètre additionnel par rapport à sa signature.

Listing 6.18 – Quatre formes d'expressions d'appel
https://osinet.fr/go/expr/play-appels

```
1  package main
2
3  import "fmt"
4
5  func itoa(n int) string { return fmt.Sprintf("%d", n) }
6
7  type T int
8  func (t T) f() string { return itoa(int(t)) }
9
10 func main() {
11         t := T(3)
12         fmt.Println(
13                 itoa(2), // Fonction classique: 2
14                 t.f(),   // Valeur de méthode: 3
15                 T.f(4),  // Expression de méthode: 4
16         )
17 }
```

Une expression d'appel dans laquelle la première expression a la valeur **nil** provoque une panique d'exécution.

Fonction nil et instance nil

Pourquoi un appel de fonction **nil** panique-t-il alors que les méthodes Go peuvent être invoquées sur une valeur **nil** ?

L'explication tient à l'évaluation de ces méthodes : dans un appel de méthode de la forme `v.M(args)`, l'appel réellement exécuté est `(v.M)(args)` : de ce fait, la fonction invoquée n'est pas **nil**, mais le résultat de l'expression valeur de méthode `v.M`, qui n'est pas **nil**.

Les valeurs respectives des deux expressions « fonction » et « arguments » sont évaluées de gauche à droite.

6.11.2 Arguments et résultats

Les arguments sont les valeurs passées à une fonction lors d'un appel. Comme indiqué au § 3.3.5 à propos du code des fonctions, ces arguments sont passés à la fonction appelée dans l'ordre de ses paramètres, par valeur.

De même, les valeurs de retour de la fonction constituent les résultats de l'appel.

Pour disposer de plus de flexibilité en matière de paramètres, les fonctions disposent de deux mécanismes pour accepter un nombre variable de paramètre du même type [3] :

- ✓ **Les paramètres de types tranches** : en définissant un paramètre de type `[]T`, une fonction peut accepter un nombre variable de paramètres de type `T`.
  Une partie de la complexité est reportée sur l'appelant qui doit invoquer la fonction en regroupant ses paramètres dans une tranche, même s'il n'en a qu'un seul. L'autre demeure dans la fonction, qui doit accepter le cas des tranches **`nil`** en plus des tranches non-**`nil`**.
- ✓ **Les fonctions variadiques** : si le code d'une fonction définit son dernier paramètre comme de type `...T`, ainsi que décrit au § 3.3.5, la fonction est variadique, et accepte un nombre variables de paramètres de type `T`.

Les deux notations sont illustrées par le listing 6.19, qui montre aussi comment traiter le cas où la fonction requiert que le nombre d'arguments du même type attendu ne soit pas zéro : la solution consiste à requérir les paramètres obligatoires comme autant de paramètres du type `T` avant le paramètre variadique final.

Comme le montre cet exemple, à l'usage, la première solution se montre moins pratique à l'usage : il existe deux notations pour l'absence d'arguments variables, et passer des arguments variables nécessite de construire une tranche, alors que la notation variadique n'a qu'une seule forme pour l'absence d'arguments, et ne nécessite pas de création de tranche annexe.

Enfin, si la tranche préexiste, comme la variable `paire` dans l'exemple, il est aussi possible de la passer directement à une fonction variadique comme dans le dernier appel de l'exemple, en faisant suivre l'argument d'un symbole trois-points.

Le cas où la notation par tranche est nécessaire existe néanmoins : il se présente lorsqu'une fonction doit être appelé avec un nombre variable de paramètres inconnu à la compilation.

6.11.3 Chaînage d'appels

Pour qu'un appel ou l'utilisation de ses résultats soit possible, deux possibilités existent :

- ✓ Soit le nombre d'arguments correspond exactement aux paramètres de la fonction appelée, et toutes leurs valeurs sont de types affectables (cf. § 3.6.2) aux paramètres correspondants.

3. Pour accepter des arguments de types différents, il suffit de définir le type du paramètre attendu comme **`interface`**`{}` ; le problème est alors d'identifier leur type individuel ; par exemple par un paramètre décrivant ces types à l'image des chaînes de format de `fmt.Printf`.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 6.19 – Appel de fonction variadique
https://osinet.fr/go/expr/play-appel-variadique

```
package main

import (
        "fmt"
        "strings"
)

func tranche(s, t string, liste []string) {
        liste = append([]string{s, t}, liste...)
        fmt.Printf("%#v\n", strings.Join(liste, ", "))
}

func variadique(s, t string, liste ...string) {
        liste = append([]string{s, t}, liste...)
        fmt.Printf("%#v\n", strings.Join(liste, ", "))
}

func main() {
        var paire = []string{"a", "b"}
        tranche("x", "y", nil)            // "x, y"
        tranche("x", "y", []string{})     // "x, y"
        tranche("x", "y", []string{"a"}) // "x, y, a"
        tranche("x", "y", paire)          // "x, y, a, b"
        variadique("x", "y")              // "x, y"
        variadique("x", "y", "a")         // "x, y"
        variadique("x", "y", "a", "b")    // "x, y, a, b"
        variadique("x", "y", paire...)    // "x, y, a, b"
}
```

✓ Soit la fonction est appelée avec un seul argument apparent, qui est le retour d'un autre appel, et les résultats de cet autre appel sont directement affectables aux paramètres de la fonction, soit comme paramètres individuels, soit par le mécanisme des fonctions variadiques, comme dans le listing 6.20.

La première notation est de loin le cas le plus courant, correspondant aux pratiques de la plupart des autres langages de programmation.

La seconde notation permet le chaînage de fonctions à retours multiples, de façon similaire aux affectations parallèles de ces retours (cf. § 7.2.5), comme dans l'exemple, où la fonction `deuxParamètres` reçoit les deux arguments d'un appel à la fonction `deuxRésultats` qui renvoie deux résultats de types correspondant aux arguments attendus.

Listing 6.20 – Chaînage de fonctions à retours multiples
https://osinet.fr/go/expr/play-chainage-multiple

```
package main

import (
        "math"
        "strconv"
)

func deuxRésultats(f float64) (int, string) {
        return int(math.Round(f)),
                strconv.FormatFloat(f, 'f', 0, 32)
}

func deuxParamètres(i int, s string) bool {
        return strconv.Itoa(i) == s
}

func main() {
        println(deuxParamètres(deuxRésultats(1.23)))
}
```

6.12 CONVERSIONS

6.12.1 Définition

La conversion transforme le type d'une valeur en le type spécifié dans l'expression, si elle réussit. Pour les types d'origine de nature interface, c'est le type statique qui est modifié, et non le type dynamique : la conversion intervient toujours au moins à la compilation.

Pour ce qui est de son impact sur l'exécution, il existe deux cas :

- ✓ une transformation impliquant des valeurs numériques ou des chaînes de caractères peut donner lieu à des transformations effectives de la représentation de la donnée convertie, réalisées à l'exécution sauf pour les valeurs constantes ;
- ✓ une transformation entre tous les autres types n'affecte pas la représentation mémoire, ni le type dynamique pour les interfaces.

Elle diffère de l'assertion de type (cf. § 6.10) – avec laquelle elle est souvent confondue lors de la découverte de Go – sur trois points :

- ✓ L'assertion ne s'applique qu'à des types source de nature interface, alors que la conversion s'applique entre tous les types de données.

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ L'assertion ne fait que vérifier que le type dynamique d'une donnée interface est du type choisi (pour les types concrets) ou implémente le type interface choisi (pour les types interfaces), sans jamais modifier la donnée elle-même, alors que la conversion peut opérer des modifications de la donnée.
- ✓ L'assertion ne peut pas s'appliquer à des valeurs constantes, puisqu'aucune constante ne peut être d'un type interface ; alors que la conversion s'applique aussi aux valeurs constantes.

Et le fonctionnement interne ?

Les sources de toutes les logiques de conversion entre types sont disponibles dans le répertoire `$GOROOT/src/runtime`, en ligne sur https://osinet.fr/go/expr/gh-go-runtime

Les fonctions concernées sont privées et non documentées, mais leurs noms sont sans ambiguïté, et sont documentés dans `src/cmd/compile/internal/gc/ builtin/runtime.go`.

Pour ce qui est de la terminologie, la documentation Go utilise toujours le terme anglais *conversion*, et jamais *cast*, qui est plus couramment utilisé dans d'autres langages pour la même notion.

Le style idiomatique Go implique de ne pas ajouter de conversions explicites là où des conversions implicites existent.

6.12.2 Notation

Les expressions de conversion peuvent être implicites, créées automatiquement par le compilateur lors d'une affectation ou d'un appel, ou explicites.

Dans le premier cas, elles ne sont pas matérialisées dans le code source ; dans le second elles sont de la forme `T(expression)`, où `T` désigne un type, défini ou non :

- ✓ **Type** : le type choisi peut être un type concret ou un autre type interface.
  Qu'il soit concret ou interface, il peut être défini ou anonyme. Dans le cas de types anonymes, des parenthèses peuvent être nécessaires pour éviter une autre interprétation par le jeu des priorités d'opérateurs :
  - si `T` est un type pointeur comme `*T`, l'expression `*T(expression)` dénote le déréférencement du résultat d'une conversion vers le type `T`.
    Il est donc nécessaire de la noter `(*T)(expression)` pour convertir l'expression vers le type `*T` ;
  - si `T` est un type canal en réception comme `<-chan T`, l'expression non parenthésée `<-chan T(expression)` dénote la réception depuis le résultat d'une conversion vers le type canal bidirectionnel `chan T`.
    Il est donc nécessaire de la noter `(<-chan T)(expression)` pour convertir l'expression vers le type canal monodirectionnel en réception `<-chan T` ;

- si `T` est un type fonction sans résultats [4] comme **`func`**`(<arguments>)`, l'expression non parenthésée **`func`**`(<arguments>)(expression)` dénote une fonction anonyme acceptant les mêmes arguments et renvoyant un résultat du type désigné par l'expression. L'expression ne pouvant s'évaluer comme un type, ceci échouera à la compilation.
  Il est donc nécessaire de la noter `(`**`func`**` (<arguments>))(expression)` pour convertir l'expression vers le type souhaité.

✓ **Délimiteur ouvrant** : une parenthèse ouvrante.

✓ **Expression** : l'expression dont la valeur est à convertir au type cible.

✓ **Virgule** : optionnelle sur une seule ligne, et généralement absente.
Le formatage standard Go retire la virgule si elle est utilisée dans une expression de conversion sur une seule ligne comme pour la variable `b` du listing 6.21.
Elle est en revanche obligatoire pour une expression sur plusieurs lignes, comme pour les variables `c` et `d` du listing.

✓ **Délimiteur fermant** : une parenthèse fermante.

Listing 6.21 – Notation des conversions explicites
https://osinet.fr/go/expr/play-conversions-explicites

```
1  package main
2
3  func main() {
4      var o byte = 'o'
5      a := int64(o)        // Notation usuelle
6      b := int64(o,)       // Possible mais inusité
7      c := int64(          // Utile pour les lignes
8          o,               // longues: format
9      )                    // multilignes courant
10     /*
11     d := int64(          // Erreur de syntaxe
12         o                // Virgule obligatoire
13     )
14     */
15     println(o, a, b, c) // 111 111 111 111
16 }
```

4. Le problème ne se présente pas pour une fonction avec résultats comme **`func`**`(<arguments>) U`. En effet, l'expression non parenthésée **`func`**`(<arguments>) U (expression)` n'est pas ambiguë, et les parenthèses additionnelles sont donc superflues dans ce cas.

© Dunod – Toute reproduction non autorisée est un délit.

6.12.3 Possibilités et limites

Les règles de conversion, visant les données elles-même, s'appliquent au type sous-jacent des données, indépendamment du type visible : dès lors que la conversion n'est pas l'une des versions pouvant modifier la donnée (chaînes ou nombres), son seul rôle est précisément de modifier ce type, comme outil de classification et de sélection des implémentations.

Pour les valeurs non constantes, la conversion est possible syntaxiquement dans de nombreuses situations, même si sa réussite peut dépendre des valeurs, comme dans les exemples du listing 6.22 :

Listing 6.22 – Possibilités et limites de conversion
https://osinet.fr/go/expr/play-conversions-possibles

```
package main

import "fmt"

type T1 struct { n int `foo:"bar"` }
type T2 struct { n int }
type PT1 *T1
type PT2 *T2

func main() {
        var t21 T2 = struct{ n int }{n: 5} // Affectable
        // var t22 T2 = T1{n: 5}          // Non affectable..
        var t23 T2 = T2(T1{n: 5})         // ...mais convertible
        fmt.Println(t21, t23)             // {5} {5}

        var p2 = PT2(&t21)                // Type pointeur défini
        var p21 *T1 = (*T1)(&t21)         // Types anonymes
        // var p22 PT1 = PT1(&t21)        // Interdit: PT1 défini
        // var p23 *T1 = (*T1)(p2)        // Interdit: PT2 défini
        fmt.Println(p2, p21)              // &{5} &{5}

        const c1 complex128 = 3 + 1E-80i
        const c2 complex64 = complex64(c1)
        // var f1 float32 = float32(c1) // Perte de l'imaginaire
        var f2 float32 = float32(c2)    // Imaginaire == 0
        fmt.Println(c1, c2, f2)
}
```

✓ Le critère le plus général est l'affectabilité : si la valeur source est affectable au type cible, elle est aussi convertible dans ce type, comme pour `t23`.

- ✓ La conversion est plus tolérante que l'affectabilité : comme elle ne s'intéresse qu'au type sous-jacent, elle fait en particulier abstraction des étiquettes dans les champs structure, comme pour `T1` et `T2`.
- ✓ Elle est possible entre types pointeurs tous deux anonymes vers le même type de base, comme pour `p21`. Si un seul des types est défini, comme pour `p22` et `p23`, elle demeure impossible.
- ✓ Les nombres réels sont convertibles entre eux, qu'ils soient entiers ou flottants ; de même les nombres complexes le sont entre eux, mais aucune conversion n'est possibles entre réels et complexes, à la différence de la situation pour des valeurs constantes, décrite plus bas.
- ✓ Enfin, les tranches d'octets ou de runes peuvent être converties vers des chaînes, et réciproquement, et les entiers peuvent être convertis en chaînes, mais pas l'inverse.

Pour les valeurs constantes, trois règles seulement s'appliquent :

- ✓ **Représentabilité** : la conversion est possible si la donnée source est représentable (cf. § 3.6.3) dans le type cible, ce qui est la transposition applicable aux constantes du critère d'affectabilité décrit dans le cas général. Ceci permet certaines conversions impossibles pour les valeurs non constantes, comme entre réels et complexes à partie imaginaire nulle, comme `f2` dans le listing 6.22.
- ✓ **Chaînes numériques** : la conversion d'une constante entière en chaîne d'une seule rune de la même valeur, comme pour les entiers non constants.
- ✓ **Type** : à l'issue d'une conversion, la valeur résultante est une constante typée, même si la valeur source était non typée.

Enfin, deux conversions significatives ne sont pas fournies par le langage :

- ✓ **Pointeurs** : la conversion d'entiers en pointeurs et vice-versa n'est pas possible au niveau du langage seul. Elle nécessite, pour les implémentations qui en disposent, l'utilisation des mécanismes du paquet `unsafe` (cf. *Plus en ligne*), qui peuvent effectuer ces conversions dans certains cas.
  Cette restriction, qui peut surprendre puisque les formats de données sous-jacents sont identiques, correspond au choix d'empêcher l'arithmétique de pointeurs dans le cas général, comme décrit au § 1.1.1.2.
- ✓ **Chaînes vers entiers** : une asymétrie existe : alors qu'il est possible de convertir un entier en chaîne, même pour les constantes, l'inverse n'est pas proposé par le langage, mais est facile à réaliser dans l'espace utilisateur, par une fonction comme celle du listing 6.23.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 6.23 – Conversion chaîne vers entier
https://osinet.fr/go/expr/play-conversions-chaines-entiers

```
func StoI(s string) (int, error) {
        if len(s) != 1 {
                return 0, errors.New("Not single-rune")
        }
        return int(s[0]), nil
}
```

6.12.4 Conversions numériques

Les conversions entres valeurs de types numériques peuvent donner lieu à un recalcul, et non simplement à un changement de classification à la compilation.

Pour les entiers, la réduction de taille du type se traduit par une simple troncature à gauche, tandis que deux cas peuvent se présenter lors de l'accroissement de taille, comme illustré par le listing 6.24 :

- ✓ Pour une valeur source non signée comme dans la conversion de `u08`, vers `u16`, les bits additionnels à gauche sont définis à 0.
- ✓ En revanche, pour une valeur source signée comme dans la conversion de `s08` vers `s16`, ils sont tous définis au bit de signe de la valeur signée, comme le montre le décodage binaire de `s16`.

Listing 6.24 – Conversion d'entiers
https://osinet.fr/go/expr/play-conversions-entiers

```
package main

import "fmt"

func main() {
        var f = func(i interface{}) { fmt.Printf("%04x\n", i) }
        var u uint16 = 0x2480
        var u08 uint8 = uint8(u)      // Troncature
        var s08 int8 = int8(u)        // Troncature
        var u16 uint16 = uint16(u08) // Extension par des 0
        var s16 int16 = int16(s08)    // Extension du signe

        f(u)    // 0010 0100 1000 0000:  0x2480
        f(u08) //           1000 0000:  0x0080
        f(u16) // 0000 0000 1000 0000:  0xff80
        f(s08) //           1000 0000: -0x0080
        f(s16) // 1111 1111 1000 0000: -0x0080
}
```

Lors des conversions de valeurs flottantes réelles vers des entiers, la partie décimale est simplement tronquée, résultant en un arrondi vers 0, aussi bien pour les valeurs positives que négatives.

Inversement, lors des conversions de valeurs entières ou flottantes vers des flottants – par exemple `float32` vers `float64` – le résultat de la conversion est arrondi à la précision du type cible.

Conversions dangereuses

Aucune de ces conversions ne déclenche d'alerte de dépassement ou soupassement arithmétique.

Dans le cas des nombres flottants – réels ou complexes – un échec au stockage du résultat de la conversion ne provoque pas d'erreur mais produit une valeur non spécifiée, dépendante de l'implémentation Go utilisée.

Les conversions numériques doivent donc être protégées par un contrôle des valeurs utilisées, pour garantir des valeurs numériquement correctes.

6.12.5 Conversions de chaînes

Les conversions impliquant un transfert entre chaînes et autres natures de données peuvent donner lieu à une reconstruction des données, et non simplement à un changement de type, selon les règles suivantes, illustrées par le listing 6.25 :

- ✓ **Entier vers chaîne** : l'entier est converti en `rune`, et la résultat est une chaîne de longueur 1. Si la rune correspond à un point de code Unicode valide, la chaîne est composée de cette seule rune ; sinon elle est composée du caractère de point de code `U+FFFD`, la *rune de remplacement* Unicode représentant un caractère invalide. C'est donc toujours une chaîne UTF-8 bien bien formée.
- ✓ **Chaîne vers entier** : pas de conversion directe, nécessite une fonction spécifique comme décrit au § 6.12.3.
- ✓ **Chaîne vers tranche `[]byte`** : les octets de la tranche résultante sont ceux de la chaîne d'origine, l'encodage UTF-8 étant préservé.
- ✓ **Tranche `[]byte` vers chaîne** : la chaîne résultante est composée des octets de la tranche d'origine, sans vérification de validité de l'encodage UTF-8. Il est préférable de traiter la chaîne résultante avec `strings.ToValidUTF8` [5] pour obtenir une chaîne UTF-8 valide.
- ✓ **Chaîne vers tranche `[]rune`** : la chaîne est décodée en runes, qui sont ensuite copiées dans la nouvelle tranche : l'encodage UTF-8 est donc supprimé.
- ✓ **Tranche `[]rune` vers chaîne** : les runes de la tranche sont converties en chaînes comme au premier cas de cette liste, et ces chaînes sont concaténées. Le résultat est donc une chaîne UTF-8 bien formée.

5. Disponible depuis Go 1.13 uniquement.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 6.25 – Conversion de chaînes
https://osinet.fr/go/expr/play-conversions-chaines

```
package main

import "fmt"

func main() {
        var f = "%v\t%d %d\n"
        var s = "thé" // 3 runes, 4 octets
        var to []byte = []byte(s)
        var tr []rune = []rune(s)
        var ov []byte = []byte("")
        fmt.Printf(f, to, len(to), cap(to))
        fmt.Printf(f, tr, len(tr), cap(tr))
        var fr = []rune{ 0x1F1EB, 0x1F1F7 }
        fmt.Printf(f, ov, len(ov), cap(ov))
        fmt.Printf(string(fr))
}

// Résultat
// [116 104 195 169] 4 8: octets UTF-8
// [116 104 233]     3 4: runes
// []                0 0: chaîne vide
// <emoji drapeau français>
```

Comme l'illustre le listing 6.25, en Go 1.13 :

- ✓ Pour les chaînes non vides, dans les deux cas de conversions de chaîne vers tranche, le tableau sous-jacent à la tranche est alloué selon la stratégie par défaut de surallocation, et non à la taille exacte de la conversion.
- ✓ Pour les chaînes vides, le résultat n'est pas une tranche **`nil`**, mais une tranche non-**`nil`** basée sur un tableau vide, d'adresse 0.

Ces derniers aspects ne sont toutefois pas spécifiés par la définition du langage.

6.12.6 Conversions d'interfaces

Dans la plupart des cas, le passage d'une interface à une autre est opéré par une *assertion de type* (cf. § 6.10), et non par une conversion, et la spécification Go ne décrit pas ce point explicitement.

Or la conversion d'une valeur de type interface vers un autre type interface est bien disponible, comme dans le listing 6.26.

Comme le montre cet exemple, `fmt.Stringer(i)` est une expression de conversion transformant une valeur de type interface en une autre valeur de type interface, affectée à la variable `s2`. Pourquoi n'est-ce pas un point abordé par la spécification ?

Listing 6.26 – Expression de conversion entre interfaces
https://osinet.fr/go/expr/play-conversions-interfaces

```
1  package main
2
3  import (
4          "fmt"
5          "strconv"
6  )
7
8  type I interface {
9          EnChaîne() string
10         fmt.Stringer
11 }
12
13 type T int
14
15 func (t T) EnChaîne() string { return t.String() }
16 func (t T) String() string   { return strconv.Itoa(int(t)) }
17
18 func main() {
19         var i I = T(5)
20         var s1, s2, s3 fmt.Stringer
21         s1 = i.(fmt.Stringer)       // Assertion de type
22         s2 = fmt.Stringer(i)        // Conversion explicite
23         s3 = i                      // Conversion implicite
24         fmt.Println(i, s1, s2, s3) // 5 5 5 5
25 }
```

Un indice est fourni par l'affectation de `i` à `s3` à la ligne suivante sans conversion explicite : la valeur de `i` est affectable au type de `s3`, et la spécification précise qu'une expression de conversion est possible lorsque la valeur d'une expression non constante à convertir est affectable au type cible.

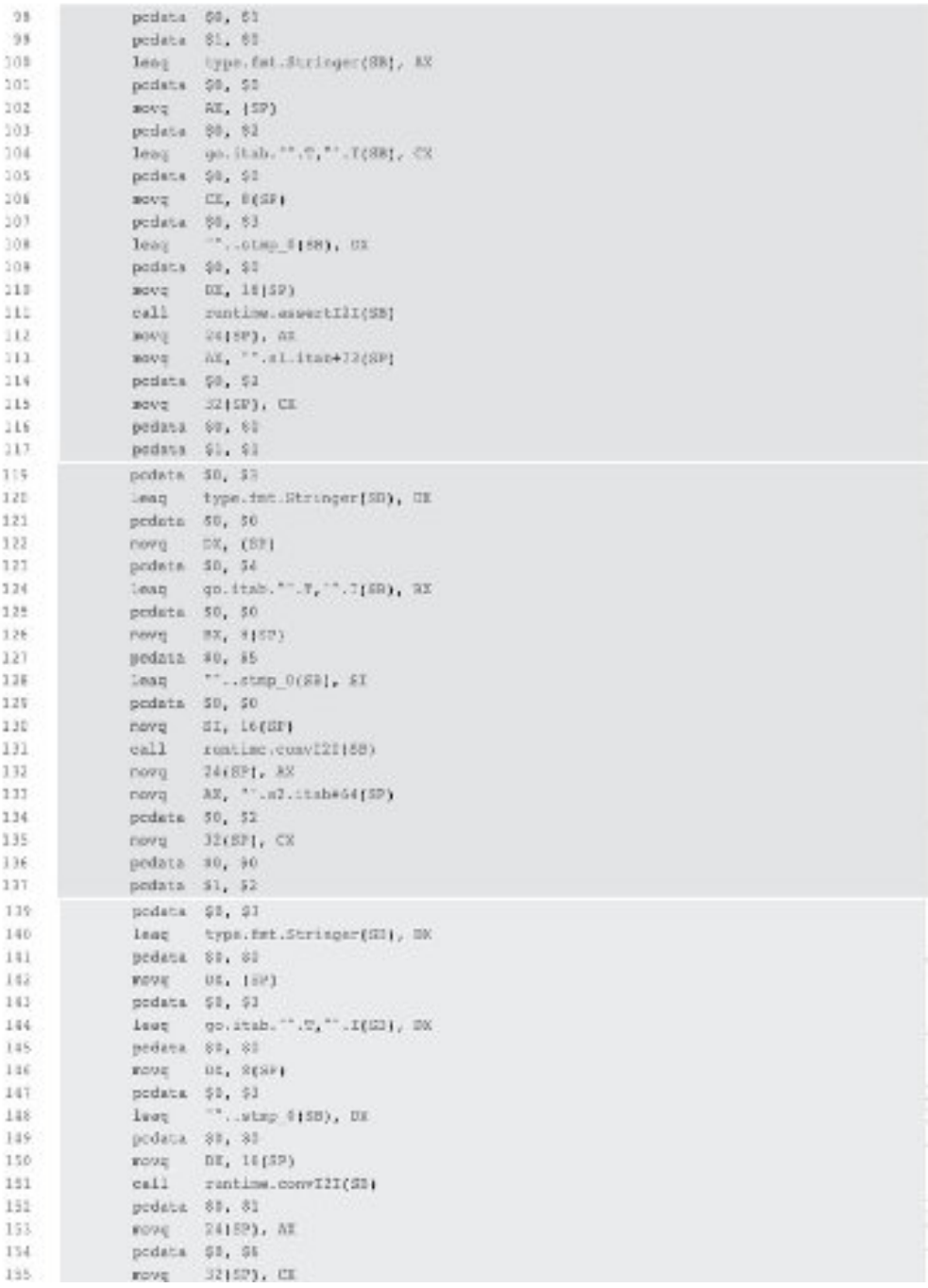
L'examen du code assembleur généré en Go 1.13 en pareil cas, visible à l'adresse https://osinet.fr/go/expr/asm-conversions et présenté par les fragments de code repris sur la figure 6.1, illustre aux lignes 111, 131, et 151 le traitement par Go de ces trois affectations.

Comme le montre cet exemple, l'assertion de type (premier fragment) est réalisée par la fonction `runtime.assertI2I`, alors que l'expression de conversion explicite (deuxième fragment), tout comme la conversion implicite (troisième fragment) causée par l'affectation directe à `s3`, est réalisée par la fonction `runtime.convI2I`.

Les bonnes pratiques Go recommandant de ne pas ajouter de conversions explicites lorsqu'elles sont inutiles, la version recommandée dans un cas comme celui de cet exemple sera de recourir à l'affectation directe. C'est là la raison pour laquelle

© Dunod – Toute reproduction non autorisée est un délit.

Figure 6.1 – Affectation d'interface vers interface : assertion, conversions explicite et implicite



ces expressions de conversion entre interfaces sont la plupart du temps absentes des programmes Go, et rarement décrites : elles ne sont nécessaires que dans un nombre très restreint de cas – les outils d'analyse de code – et le besoin réel est presque toujours couvert par une assertion de type ou une conversion implicite.

6.13 RÈGLES COMMUNES

6.13.1 Expressions constantes

Les expressions constantes sont composées uniquement d'opérandes constants.

Contrepartie de cette restriction, elle sont évaluées à la compilation, ne consomment donc pas de temps d'exécution, et bénéficient d'une précision étendue, calculant avec des valeurs exactes pour les valeurs entières [6], comme détaillé au § 5.2.4.

Les opérandes constants qui les composent sont utilisables partout où une valeur de la même nature (booléen, numérique, ou chaîne) peut l'être, et les expressions résultantes renvoient une valeur constante, qui peut être typée ou non :

- ✓ **Décalages** : si l'opérande de gauche, nécessairement entier, est une constante typée, le résultat est du même type, sinon il est non typé.
- ✓ **Comparaisons** : le résultat est une constante booléenne non typée
- ✓ **Autres expressions, même nature** : si les opérandes sont tous deux non typés mais de même nature, le résultat l'est également et est de la même nature que les deux opérandes.
- ✓ **Autres expressions, natures différentes** : si les opérandes d'une expression sont de natures différentes mais compatibles comme opérandes, ils sont nécessairement numériques. La nature du résultat est alors déterminée par une priorité définie entre les natures de données numériques, comme dans le listing 6.27 :
  - les complexes l'emportent sur les flottants ;
  - les flottants l'emportent sur les runes ;
  - les runes l'emportent sur les entiers.

  Cette priorité des natures entre elles est le seul critère définissant la nature du résultat : l'ordre des opérandes est indifférent.

Le type de ces expressions, lorsqu'il a besoin d'être évalué, est le type par défaut pour la nature de données concernée :

- ✓ `int` pour les entiers autres que les runes ;
- ✓ `int32` pour les runes, le type `rune` étant un alias de `int32` ;
- ✓ `float64` pour les flottants – c'est aussi le retour de la fonction intégrée `complex` ;
- ✓ `complex128` pour les complexes – c'est aussi le retour des fonctions intégrées `real` et `imag`.

6. La spécification du langage ne contraint pas les implémentations à effectuer les calculs en précision étendue sur les expressions flottantes réelles ou complexes, ce qui permet d'utiliser pour leur évaluation la logique IEEE-754 fournie par le matériel sur lequel la compilation est exécutée, avec un risque d'imprécision lors de la représentation de grandes valeurs flottantes, qui n'existerait pas si la précision était aussi conservée dans ces cas.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 6.27 – Types des expressions à opérandes hétérogènes
https://osinet.fr/go/expr/play-types-heterogenes

```
package main

import "fmt"

func main() {
        var f = func(x interface{}) {
                fmt.Printf("%T %v\n", x, x)
        }

        // Expression   Règle appliquée       Type         Valeur
        f(3 + 2)      // entier == entier     int          5
        f(3 + 'a')    // entier < rune        int32        100
        f(2 * 3.5)    // entier < flottant    float64      7
        f(4 * 2i)     // entier < complexe    complex128   0+8i
        f('A' + '0') // rune == rune          int32        113
        f('a' + 2.0) // rune < flottant       float64      99
        f('0' + 2i)  // rune < complexe       complex128   48+2i
        f(3.5 / 0.5) // flottant == flottant float64      7
        f(3.5 + 2i)  // flottant < complexe   complex128   3.5+2i
        f(2i / 2i)    // complexe + complexe  complex128   1+0i
}
```

Lors de la combinaison d'opérandes constants typés et non typés compatibles dans une expression, en revanche, le type du résultat est défini par l'opérande typé : le résultat doit être représentable dans ce type. Ceci peut conduire à des erreurs de dépassement de capacité, comme avec l'expression :

```
const k = byte(1) << 10
```

… dans laquelle la constante non typée 1 devient typée par la conversion en octet, de sorte que le résultat du décalage, soit 2^{10} = 1024, dépasse la capacité du type du résultat, contraint d'être un octet puisque l'opérande de gauche du décalage est une constante typée octet.

6.13.2 Ordre d'évaluation

L'ordre d'évaluation des expressions n'est que partiellement défini par la spécification de Go :

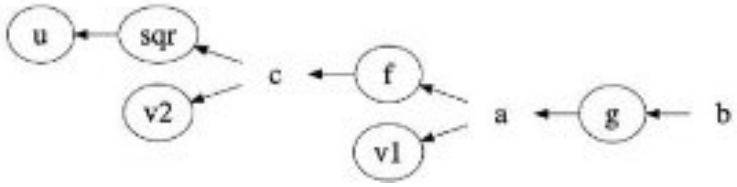
✓ Dans la portée paquet, qui n'est pas définie comme imposant un ordre séquentiel défini par les lignes successives :
- les expressions utilisées pour l'initialisation des variables de portée paquet sont calculées dans l'ordre des dépendances afin de permettre la résolution ;

- entre expressions indépendantes, l'ordre est celui d'apparition dans le code source, de gauche à droite, comme illustré par la figure 6.2, résolvant un exemple de la spécification Go.

Figure 6.2 – Ordre d'évaluation dans la portée paquet
©2019 Contributeurs Go - sous licence Creative Commons BY-3.0

```
var a, b, c = f() + v(), g(), sqr(u()) + v()

func f() int { return c }; func g() int { return a }
func u() int { return 0 }; func v() int { return 1 }
       func sqr(x int) int { return x*x }
```



Comme le représente le graphe des dépendances, les fonctions sont évaluées dans l'ordre suivant : `u`, `sqr`, `v` (2ème apparition), `f`, `v` (1ère apparition), puis `g`. Ces évaluations initialisent successivement `c`, `a`, et enfin `b`.

✓ Dans les autres portées, l'ordre est partiellement défini : lors de l'évaluation des opérandes d'une expression ou d'une instruction d'affectation ou de retour de fonction :
- l'ordre lexical de gauche à droite s'applique aux appels de fonctions/méthodes et à la réception depuis des canaux ;
- en revanche, leur ordre relatif par rapport aux expressions d'indexation n'est pas précisé par la spécification du langage.

✓ Au sein d'une même expression opérant sur des valeurs flottantes, l'ordre des opérations flottantes est défini par l'associativité des opérateurs, qui peut être précisée par l'utilisation de parenthèses. Dans cet exemple à trois variables flottantes réelles ou complexes :
- `(a + b) + c` évalue d'abord `a + b`, puis lui ajoute `c` ;
- `a + (b + c)` force l'évaluation à commencer par `b + c`, puis ajoute `a` au résultat.

Bien que les résultats soient identiques arithmétiquement, le jeu des imprécisions propre aux valeurs flottantes peut conduire ces deux expressions à aboutir à des résultats différents.

Cette latitude importante laissée aux diverses implémentations du langage est un problème pour tout projet à grande échelle, susceptible d'être compilé sur plusieurs compilateurs et pour plusieurs architectures cibles au fil du temps.

© Dunod – Toute reproduction non autorisée est un délit.

Pour éviter tous ces effets, deux pratiques permettent d'éliminer les ambiguïtés :

- ✓ L'utilisation de fonctions pures, qui ne dépendent que de leurs paramètres. Dans l'exemple de la figure 6.2, les fonctions `f` et `g` sont impures, dépendant de variables extérieures : en l'absence de telles dépendances, la même instruction serait évaluée de gauche à droite.
- ✓ L'utilisation systématique de parenthèses lorsque la priorité des opérateurs ne suffit pas à déterminer l'ordre d'évaluation pour les valeurs flottantes.

7 Instructions

7.1 ORGANISATION DE L'EXÉCUTION

Les instructions sont le cœur des programmes : c'est leur enchaînement qui en crée la fonctionnalité.

Go différencie deux types d'instructions : les instructions *simples* (*simple instructions*) et les autres.

Séparément, certaines instructions peuvent – intrinsèquement ou selon leur contenu – posséder la propriété d'être *terminales*.

Dans ce chapitre, les instructions sont présentées dans la séquence suivante :

- ✓ Les notions générales comme celles d'instruction simple ou terminale.
- ✓ Les instructions simples : vide, expressions, émission, incrémentation, affectation, et déclaration courte.
- ✓ Les instructions générales : déclaration non courte, instructions étiquetées, si/alors, commutateurs, boucles, avec les instructions associées, et `goto`.
- ✓ Les instructions de contrôle liées aux fonctions : le retour de fonction, et l'exécution de fonctions asynchrones ou terminales.

7.1.1 Instructions terminales

Une instruction est dite *terminale* lorsque l'exécution se poursuit après son exécution ailleurs qu'à l'instruction suivante. Les cas où ceci se produit sont expressément limités ; tous les cas non listés ci-dessous sont spécifiés comme non terminaux :

- ✓ **`goto` et `return`** : ces deux instructions sont toujours terminales. Elles définissent l'une comme l'autre une destination explicite : une étiquette pour **`goto`**, et la sortie de la fonction en cours pour **`return`**.

- ✓ **panic** : une instruction d'appel à la fonction intégrée `panic` est toujours terminale : à moins d'être interceptée, elle met fin au programme entier.
- ✓ **blocs** : un bloc est une instruction terminale si la liste de ses instructions se termine par une instruction terminale. Voir l'encadré « Code inatteignable » à ce sujet.
- ✓ **if** : une instruction **if** est terminale si elle comporte une branche **else** et si les deux branches sont elles-mêmes des blocs terminaux.
- ✓ **for** : une instruction **for** est terminale si elle ne comporte pas de condition de boucle et si elle n'inclut pas d'instruction **break** relative au **for** ou plus généralement si son bloc d'instructions est terminal.
- ✓ **switch** : une instruction **switch**, dans ses deux formes, est terminale si les trois conditions suivantes sont respectées :
  - elle n'inclut pas d'instruction **break** relative au **switch** ;
  - elle inclut une clause **default** : en son absence, l'exécution se poursuit sans erreur après le **switch** en cas de valeur non acceptée ;
  - chacune des listes d'instructions, **default** compris, est terminale, ou est une instruction **fallthrough** : dans ce dernier cas, l'exécution du cas actif se poursuit au cas suivant, qui est récursivement dans la même situation.
- ✓ **select** : une instruction **select** est terminale si elle n'inclut pas d'instruction **break** relative au **select** ; et si chacune des listes d'instructions, **default** compris, est terminale.
- ✓ **instruction étiquetée** : une instruction étiquetée est terminale si l'instruction désignée par l'étiquette est elle-même terminale.

Au-delà des instructions elles-même, une liste d'instructions est considérée comme se terminant par une instruction terminale si :

- ✓ la liste n'est pas vide ;
- ✓ sa dernière instruction exécutée non vide est terminale. Par le jeu des boucles et **goto**, la dernière instruction exécutée n'est pas nécessairement la dernière ligne de code source de la liste d'instructions, comme dans le listing 7.1, où la dernière instruction de la fonction `f` est le **return**.

Code inatteignable

L'appréciation de la dernière instruction d'une liste ne prend pas en compte les instructions inatteignables, car situées après une instruction terminale.

La présence de telles instructions – qui dénote une erreur de conception – ne constitue pas une erreur de syntaxe, mais est signalée par les outils d'analyse de code comme `go vet`.

Listing 7.1 – Instruction terminale d'une liste
https://osinet.fr/go/instr/play-terminales

```
package main

func f() (n int) {
        n = 0
        goto un
deux:
        n = 2
        return // Dernière instruction, terminale.
un:
        n = 1
        goto deux // Dernière ligne, pas dernière instruction.
}

func main() {
        println(f()) // 2
}
```

7.1.2 Instructions simples

À la différence du caractère *terminal*, qui ne découle pas toujours uniquement de l'instruction elle-même, le caractère d'instruction *simple* (*simple instruction*) est une propriété intrinsèque de certaines instructions, qui seront les premières décrites dans ce chapitre :

- ✓ l'instruction vide ;
- ✓ les instructions d'expressions ;
- ✓ les instructions d'incrémentation/décrémentation ;
- ✓ les instructions d'affectation ;
- ✓ les instructions d'émission sur un canal ;
- ✓ les instructions de déclaration courte de variable (cf. § 5.1.2).

Ces instructions simples ont pour particularité de pouvoir être incluses à l'intérieur de certaines autres instructions :

- ✓ l'instruction de test **if** ;
- ✓ la boucle **for** ;
- ✓ les commutateurs d'expression et de type **switch**.

© Dunod – Toute reproduction non autorisée est un délit.

7.2 INSTRUCTIONS SIMPLES

7.2.1 L'instruction vide

La plus simple de toutes les instructions simples est l'instruction vide, qui n'est composée de rien, et ne fait rien. Elle est principalement rencontrée dans trois situations, illustrées dans le listing 7.2 par des commentaires `/* vide */` :

- ✓ Instruction courte non explicitée dans les instructions qui en admettent une, comme dans la première partie de la boucle **`for`**, ligne 5. C'est le cas le plus courant.
- ✓ Point-virgule isolé, comme à la ligne 13.
- ✓ Instruction sous-jacente incluse dans une instruction étiquetée terminale, comme après l'étiquette `fin` ligne 16.

Listing 7.2 – Instructions vides
https://osinet.fr/go/instr/play-vide

```
1  package main
2
3  func main() {
4          var i int
5          for /* vide */ ; i < 10; i += 3 {
6                  println(i)
7                  if i == 5 {
8                          goto fin
9                  }
10         }
11         println(i)
12
13         /* vide */ ;
14
15 fin:
16         /* vide */
17 }
```

7.2.2 Expressions

Deux types d'expressions peuvent être utilisés comme instructions simples en ignorant leurs résultats, comme dans le listing 7.3 : l'appel de fonction ou méthode, à l'exception de certaines fonctions intégrées listées dans le tableau 7.1 ; et la réception depuis un canal.

Les fonctions intégrées interdites dans cette liste sont celles dont le rôle principal, voire unique, est précisément de renvoyer un résultat : initialisation mémoire, ou calcul de résultats.

Listing 7.3 – Instructions expression
https://osinet.fr/go/instr/play-expression

```
1  package main
2
3  import "fmt"
4
5  func f(n int) string {
6          s := fmt.Sprintf("Reçu: %d\n", n)
7          print(s)
8          return s
9  }
10
11 func main() {
12         var c chan int = make(chan int, 2)
13         c <- 1
14         c <- 2
15         <-c       // Instruction expression: réception
16         x := <-c
17         f(x)      // Instruction expression: appel
18 }
```

Tableau 7.1 – Fonctions intégrées interdites comme instructions

Intégrées	Paquet `unsafe`
`append`	`unsafe.Alignof`
`cap`, `len`	`unsafe.Offsetof`
`complex`, `imag`, `real`	`unsafe.Sizeof`
`make`, `new`	

Celles qui renvoient un résultat, mais présentent un effet de bord justifiant leur utilisation indépendamment du résultat renvoyé, à l'instar de `copy` (cf. § 8.5), sont utilisables comme instructions.

De même, une expression de réception dont le résultat est ignoré comme à la ligne 15 de l'exemple, présente elle aussi un effet de bord en retirant un élément du canal, sans le stocker dans une variable. Ce format est en particulier employé lorsque le type d'éléments du canal est de longueur zéro, donc avec des valeurs intrinsèquement sans intérêt ; de tels canaux sont utilisés comme mécanismes de signalisation / synchronisation, et décrits au chapitre 10, notamment à propos des contextes (cf. § 10.3) et `sync.Cond` (cf. § 10.4.6).

© Dunod – Toute reproduction non autorisée est un délit.

7.2.3 Émission sur un canal

Complément de l'expression de réception et de sa possibilité d'utilisation comme instruction simple, l'émission sur un canal est une autre instruction simple.

L'émission permet de soumettre une valeur d'un type donné à un canal d'éléments de ce type.

Pour ce qui est de la notation :

✓ Elle utilise le même symbole `<-` que la déclaration de types canaux unidirectionnels (cf. § 4.5.2) et l'opérateur de réception (cf. § 2.4.9).

✓ Pour savoir où placer l'opérateur, il existe un moyen mnémotechnique : l'opérateur dessine une flèche, et le sens de cette flèche décrit le flux de la donnée. Avec une variable canal `canal` et une valeur élément `élément`, l'émission est notée :

```
canal <- élément
```

La « flèche » se dirige vers le canal, montrant que la donnée est émise vers le canal. Les espaces de chaque côté de la flèche font partie du format standard Go, mais leur absence ne provoque pas d'erreur.

L'émission est une opération bloquante, qui est tentée après l'évaluation des deux expressions qui la composent : le canal et l'élément.

Son comportement est en fonction de l'état du canal est décrit par le tableau 7.2.

Tableau 7.2 – Instruction d'émission

État du canal	Tampon	Comportement
`nil`, forcément ouvert	forcément non tamponné	blocage indéfini
non-**`nil`**, ouvert	non tamponné	blocage jusqu'à réception
	tamponné, plein	
	tamponné, non plein	retour immédiat
non-**`nil`**, fermé	indifférent	panique d'exécution

Le blocage jusqu'à réception sur un canal peut donner lieu à un retour immédiat si une goroutine est déjà en attente de réception dessus : en ce cas, son exécution est débloquée et les deux goroutines peuvent poursuivre leur exécution concurrente.

La situation est différente pour les canaux **`nil`** : même si une goroutine se place en attente de réception sur le canal avant la tentative d'émission, elle demeure bloquée, le passage direct de l'émetteur au récepteur n'étant pas possible en pareil cas.

7.2.4 Incrémentation / décrémentation

L'instruction simple d'incrémentation, qui recouvre à la fois l'opération d'incrémentation et celle de décrémentation, a pour effet d'ajouter ou retirer une valeur constante non typée égale à 1 d'une expression adressable.

La notation se compose :

✓ **D'une expression** : qui doit être adressable ou être une expression d'indexation de carte, pour que l'instruction puisse en modifier la valeur.

✓ **D'un opérateur** :
- `++` pour l'incrémentation ;
- `--` pour la décrémentation.

Cette instruction est équivalente à une affectation (cf. § 7.2.5) modificatrice. Ainsi, pour une variable numérique `n` :

✓ `n++` est équivalent à `n += 1`

✓ `n--` est équivalent à `n -= 1`

À la différence de nombre de langages dérivés de C :

✓ La valeur de l'expression modifiée n'a pas besoin d'être d'un type entier ; il suffit qu'elle soit d'un type numérique : comme la valeur d'incrément est une constante non typée, elle est représentable (cf. § 3.6.3) dans le type de l'expression et peut donc lui être ajoutée.

✓ L'opérateur est toujours placé en suffixe de l'expression (`n++`) et non en préfixe (`++n`)

La raison d'être de ces deux formes préfixe et suffixe est, dans les langages où cette opération est une expression, de choisir la valeur renvoyée :

✓ La valeur d'origine avec la notation suffixe ;

✓ Ou la valeur modifiée avec la notation préfixe.

Comme cette instruction n'est pas une expression en Go, il n'y a pas de valeur de retour à choisir et, par le fait, pas de raison d'avoir deux formes syntaxiques différentes.

7.2.5 Affectation

7.2.5.1 Définition

L'instruction simple d'affectation permet de définit ou modifier les valeurs des expressions d'une liste par celles d'une autre.

À la différence de nombre d'autres langages, elle recouvre à la fois :

✓ l'affectation traditionnelle de la valeur d'une expression de droite à une expression de gauche ;

✓ la modification d'une expression de gauche par la combinaison de sa valeur avec une expression de droite au moyen d'un opérateur ;

✓ l'affectation parallèle de plusieurs expressions de droite à un même nombre d'autres expressions de gauche, comme en Python ;

✓ l'affectation à plusieurs expressions de gauche d'une seule expression de droite à valeurs multiples (un *tuple* ou *n-uplet*), comme un appel de fonction, une réception, une indexation de carte, ou une assertion de type.

© Dunod – Toute reproduction non autorisée est un délit.

Le traitement des n-uplets est différent dans l'instruction d'affectation de ce qu'il est dans l'appel de fonction :

✓ Si le n-uplet a une longueur fixe, comme c'est le cas pour un résultat de fonction, il est utilisable comme liste d'arguments dans une expression d'appel ; et comme une liste d'expressions de droite dans une instruction d'affectation parallèle.

✓ En revanche, s'il a une longueur variable comme dans le cas de la réception, de l'indexation de carte, ou l'assertion de type, il est traité comme une valeur unique – et donc un argument unique – lors d'un appel, la seconde valeur étant ignorée ; mais comme deux expressions dans une instruction d'affectation.

7.2.5.2 Notation

La notation, illustrée par le listing 7.4, se compose :

✓ **D'une liste d'expressions** : cibles. Comme leurs valeurs doivent pouvoir être assignées par l'instruction, elles doivent être adressables, ou être des expressions d'indexation de carte.
Si l'opérateur n'est pas l'affectation basique =, l'instruction doit également pouvoir évaluer l'expression de gauche, qui ne peut donc être l'opérateur vierge _.

✓ **D'un opérateur** : qui est :
- soit le symbole d'affectation =, qui n'est pas strictement parlant un opérateur (cf. § 2.4.1) ;

Listing 7.4 – Instructions d'affectation
https://osinet.fr/go/instr/play-affectation

```
1  package main
2
3  func multi() (x, y int) { return 1, 3 }
4
5  func main() {
6          var a, b string
7          var n, o int
8
9          a = "x"          // Affectation unique
10         n, o = multi()   // Affectation parallèle de n-uplet
11         a, b = "x", "y" // Affectation parallèle d'expressions
12         a, _ = "x", "y" // Affectation parallèle avec id. vierge
13
14         a += b           // Op. d'affectation additive
15         println(a)       // "xy"
16         n <<= o          // Op. d'affectation multiplicative
17         println(n)       // 8
18 }
```

- soit un des cinq opérateurs additifs : concaténation, addition, soustraction, `OR` bit à bit, et `XOR` bit à bit (cf. § 2.4.4) ;
- soit un des sept opérateurs multiplicatifs : multiplication, division, modulo, décalages à gauche et à droites, `AND` bit à bit, et `AND NOT` bit à bit (cf. § 2.4.3).

Dans les deux derniers cas l'instruction est dite *opération d'affectation* (*assignment operation*), pour la différencier de l'affectation basique.

✓ **D'une liste d'expressions** : sources, qui peut être :
- soit une liste d'epxressions à valeur unique, le nombre d'expressions correspondant à celui de la première liste ;
- soit une expression unique à valeurs multiples (n-uplet), le nombre de valeurs correspondant à celui de la première liste.

Dans le cas des *opérations d'affectation*, les deux listes sont limitées à une seule expression mono-valuée chacune.

7.2.5.3 Ordre d'exécution

Du fait des valeurs multiples pouvant être en jeu, et en particulier par la réutilisation des mêmes variables dans plusieurs expressions ou sous-expressions de l'instruction, celle-ci se déroule en deux grandes parties :

✓ **Préparation** : évaluation préliminaire pour éviter les écrasements :
- évaluation des opérandes appartenant aux expressions d'indexation et aux déréférencements – y compris implicites comme dans les sélections – des expressions de gauche afin d'obtenir des valeurs figées pour les opérations suivantes ;
- évaluation des expressions de droite. Dans le cas des *opérations d'affectation*, cette évaluation procède à l'évaluation de l'unique expression de gauche pour pouvoir la combiner avec le résultat de l'évaluation de l'expression de droite.

La spécification regroupe ces deux opérations en une seule phase, n'excluant pas une évaluation parallèle ou entrelacée.

✓ **Copie** : affectation des valeurs résultant de l'évaluation des expressions de droite aux destinations définies par les expressions de gauche en utilisant pour leurs sous-expressions d'indexation et autres les valeurs calculées lors de la préparation. Celle-ci de déroule de gauche à droite, sans écrasement inopiné, comme dans le listing 7.5, à l'image de l'équivalent Python, sans nécessiter le passage par des notations spécifiques comme la déstructuration d'ECMAScript 6 ou PHP [1].

1. Versions comparables à l'échange de variables du listing 7.5 :

✓ **ECMAScript 6** : `let a = 1; let b = 2; [a, b] = [b, a]; console.log(a, b);`

✓ **PHP 7.1** : `$a = 1; $b = 2; [$a, $b] = [$b, $a]; echo "$a $b";`

© Dunod – Toute reproduction non autorisée est un délit.

Listing 7.5 – Échange de valeurs
https://osinet.fr/go/instr/play-echange

```
1  package main
2
3  func main() {
4          var a, b int = 1, 2
5          a, b = b, a         // Échange de variables
6          println(a, b)       // 2 1
7
8          a, a, a = 1, 2, 3 // De gauche à droite
9          println(a)          // 3
10 }
```

Un écrasement peut survenir si plusieurs expressions de gauche désignent la même donnée, comme dans la deuxième partie de l'exemple : les valeurs des expressions de droite sont successivement affectées aux expressions de gauche de gauche à droite, de sorte que la variable `a` reçoit successivement 1, 2, et enfin 3, qui est sa valeur à la fin de l'instruction.

Durant la dernière étape de copie des valeurs, l'affectabilité des valeurs au type des expressions de destination correspondantes est vérifiée. Certaines affectations sont toutefois possibles même lorsque ce critère n'est pas respecté :

- ✓ Toute valeur typée peut être affectée à l'identificateur vierge, qui n'a aucun besoin de la stocker ou de la traiter.
- ✓ Si la valeur est une constante non typée, et que l'expression de destination ne précise pas de type concret, la valeur reçoit le type par défaut de la nature de données correspondante, par exemple `int` pour une constante entière, comme décrit au § 5.2.3.
  Ceci survient dans le cas de l'affectation à l'identificateur vierge, ou à une variable de type interface, qui a besoin d'un type dynamique concret pour pouvoir stocker la valeur reçue.
- ✓ La même conversion est opérée pour les valeurs booléennes non typées même si elles ne sont pas des constantes.

7.2.6 Déclaration courte de variable

La dernière des instructions simples est aussi la moins simple : la déclaration courte de variable fournit en effet en une seule notation compacte la combinaison de deux actions :

- ✓ une déclaration de variable (cf. § 5.1.2) de type implicite ;
- ✓ une instruction d'affectation (cf. § 7.2.5) définissant la valeur initiale de cette variable.

À la différence aussi bien des déclarations normales introduites par le mot-clef **`var`** que des affectations, elle ne peut être utilisée qu'à l'intérieur d'une fonction/méthode, et non dans la portée globale d'un paquet.

Elle est fonctionnellement équivalente à une déclaration multiple à type implicite comme **`var`** `uneVariable = uneValeur`, et sa notation se compose :

- ✓ **D'une liste d'identificateurs** : dont l'un au moins n'a pas encore fait l'objet d'une déclaration dans la même portée.
  Comme pour une affectation, certains de ces identificateurs peuvent être l'identificateur vierge, tant qu'il y a au moins une nouvelle variable définie dans la liste.
  Si une variable du même nom est définie dans une portée englobant la portée en cours, une nouvelle variable est définie est initialisée, opérant un masquage (*shadowing*) de la variable homonyme de la portée englobante.
- ✓ **D'un symbole `:=`** : séparant les deux listes.
- ✓ **D'une liste d'expressions** : utilisées pour initialiser les variables désignées par les identificateurs de la première liste.

Si certains identificateurs présents dans la liste de gauche correspondent à des variables ou quasi-variables (cf. § 5.1.4) déjà définies, il n'y a pas de redéclaration, et seule la partie « affectation » de l'instruction est exécutée pour ce qui les concerne.

C'est ce qu'illustre le listing 7.6.

Listing 7.6 – Redéclarations courtes parallèles
https://osinet.fr/go/instr/play-redeclarations-parallelles

```
1  package main
2
3  func main() {
4          // Déclaration courte parallèle de deux variables.
5          a, b := 1, "B"
6
7          // Correct: redéclaration avec une nouvelle variable.
8          a, c := 2, "C"
9
10         // Incorrect: redéclaration sans nouvelle variable.
11         // a, b := 3, "b"
12
13         // Incorrect: redéclaration avec un autre type.
14         // a, d := "A", 4
15
16         println(a, b, c)
17 }
18
19 // Résultat: 2 B C
```

© Dunod – Toute reproduction non autorisée est un délit.

En tant qu'instruction simple, la déclaration courte est particulièrement pratique pour définir des variables limitées à la portée d'une instruction complexe.

Ceci permet de ne pas avoir à se préoccuper du reste de la portée englobant l'instruction complexe, comme dans le listing 7.7, où la variable `i` définie par la déclaration courte à la ligne 9 masque la variable homonyme définie dans la portée de la fonction `main`.

Listing 7.7 – Variables temporaires
https://osinet.fr/go/instr/play-variables-temporaires

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var i int = 5
7
8          fmt.Printf("Avant la boucle, %d\n", i) // 5
9          for i := 0; i < 1; i++ {
10                 fmt.Printf("Boucle: %d\n", i)   // 0
11         }
12         fmt.Printf("Après la boucle, %d\n", i) // 5
13 }
```

Un cas particulier d'utilisation est régulièrement utile : celui de la redéclaration courte dans une boucle lançant des tâches en arrière-plan, comme dans le listing 7.8.

Dans cet exemple, la boucle naïve de la première version lance les tâches en arrière plan en leur passant directement la variable de boucle ; et par le jeu de l'ordonnancement, le résultat reçoit toujours la dernière valeur de la boucle : les goroutines sont en effet lancées à la première occasion, qui se trouve être l'appel à `time.Sleep`, et avant cette commutation de tâches, la variable de boucle poursuit sa progression.

La correction idiomatique utilise une déclaration courte avec le même nom de variable, comme à la ligne 14 [2] : dans cette version, par le jeu de l'ordre d'évaluation des affectations (cf. § 7.2.5.3), la déclaration courte alloue une nouvelle variable dans la portée du bloc d'itération, et lui assigne la valeur visible, qui est celle de la variable de boucle. Lorsque les goroutines lancées en arrière-plan sont exécutées, leur référence à `val` concerne la variable locale créée par la déclaration courte, avec cette valeur appropriée, et non la variable de boucle.

2. Cette technique est décrite en détail dans la FAQ Go : https://osinet.fr/go/instr/go-boucle-goroutine

Listing 7.8 – Redéclaration dans une boucle de goroutines
https://osinet.fr/go/instr/play-boucle-goroutine

```
package main

import "time"

func main() {
        values := []int{0, 1, 2, 3, 4, 5}
        p := func(n int) { print(n, "\t") }
        for _, val := range values {
                go func() { p(val) }() // 5 5 5 5 5 5
        }
        time.Sleep(time.Second)
        println()
        for _, val := range values {
                val := val
                go func() { p(val) }() // 0 1 2 3 4 5
        }
        time.Sleep(time.Second)
}
```

7.3 INSTRUCTIONS GÉNÉRALES

7.3.1 Déclaration

Suite logique de l'instruction simple de déclaration courte, la première instruction non simple est l'instruction de déclaration dans le cas général.

Comme sa version courte, la déclaration permet d'associer « quelque chose » à un identificateur, pour toute la portée d'un bloc. La hiérarchie des blocs est décrite au § 2.2 page 21.

À la différence de la déclaration courte, limitée aux variables déclarées dans les blocs de fonction et inférieurs, la déclaration ordinaire :

- ✓ ne se limite pas à déclarer des variables, mais permet aussi de définir des constantes (§ 5.2.5), des types (chapitres 3 et 4), des fonctions (§ 3.3.3), des étiquettes d'instructions (§ 7.3.2), et des paquets (§ 9.7) ;
- ✓ n'est pas limitée aux blocs de fonctions et inférieurs.

Compte tenu de la diversité de ses formes, chaque version de déclaration est détaillée dans la section consacrée à la donnée déclarée, listée ci-dessus.

© Dunod – Toute reproduction non autorisée est un délit.

La hiérarchie des blocs donne lieu à quatre groupes de déclarations formellement différenciés :

- ✓ Les « déclarations » au sens usuel, qui permettent de déclarer types, constantes, et variables, et peuvent se rencontrer uniquement dans les blocs de fonctions et inférieurs.
- ✓ Les déclarations de premier niveau (*top-level declarations*) qui, comme leur nom l'indique, peuvent être présentes dans les blocs de paquet et de fichier. Outre les déclarations au sens usuel, les déclarations de premier niveau incluent les déclarations de fonctions et de méthodes.
- ✓ Les déclarations de paquet, désignées comme *déclarations d'import* dans la spécification du langage, qui sont définies dans le bloc fichier d'un paquet et ont une portée limitée à ce bloc, à la différence des autres identificateurs définis dans un bloc fichier, dont la portée est celle du paquet (cf. § 9.7). Ne pas les confondre avec les *clauses paquet* (*package clause*), qui ne sont pas des déclarations, comme détaillé au chapitre 9.
- ✓ Les déclarations d'étiquette, qui sont réalisées par la création d'une instruction étiquetée, décrite au § 7.3.2.

Hormis deux cas particuliers, l'identificateur vierge _ et l'identificateur `init`, aucun identificateur ne peut être déclaré deux fois dans le même bloc, ni figurer à la fois dans un bloc paquet et un bloc fichier du même paquet.

Ces deux identificateurs présentent des particularités pour ce qui est des déclarations, illustrées par le listing 7.9 :

- ✓ **L'identificateur vierge _** : peut être utilisé dans toute déclaration, mais aucun artefact ne lui est associé par ces déclarations, ce qui permet de le faire participer à autant de déclarations incompatibles entre elles que souhaité dans une même portée.
  Le fait qu'il soit ensuite impossible d'accéder à l'artefact ainsi « déclaré » limite considérablement l'utilité de cette faculté.
- ✓ **L'identificateur de fonction `init`** : peut être utilisé dans les portées fichier et paquet autant de fois que souhaité, et crée autant de (pseudo-)fonctions `init` [3] que de déclarations.
  Cet identificateur peut aussi être utilisé pour déclarer d'autres natures d'artefacts, par exemple des variables ou des constantes, comme aux lignes 15 et 20 de l'exemple. Dans ce cas, toutefois, il est traité comme un identificateur normal, soumis à la contrainte d'unicité dans une portée, même s'il définit une fonction comme à la ligne 15.

3. Les « fonctions » `init` sont très particulières, étant obligatoirement du type `func()`, sans paramètres ni résultats, et ne pouvant être utilisées dans une expression d'appel de fonction. Elles sont invoquées par le runtime Go – et non par le code utilisateur – lors de l'initialisation des paquets selon le processus décrit au §9.8.

Listing 7.9 – Les identificateurs « _ » et « `init` »
https://osinet.fr/go/instr/play-vierge-init

```
package main

import _ "database/sql"      // Id. vierge pour un paquet
import _ "time"              // ...et un autre

type _ int                   // Id vierge pour un type
type _ []bool                // ...et un autre

func _() int { return 1 }    // Id vierge pour une fonction
func _(s string) string {    // ...et une autre d'un autre type
    return s
}

func init() {                // init pour une fonction
    var init = func() {}     // init pour une variable
    println("init ", init)
}

func init() {                // init pour une autre
    const init = 2           // init pour une constante
    println("init ", init)
}

func main() {
    const _ = 3i             // Id. vierge pour une constante
    const _ = "chaîne"       // et une autre d'un autre type

    var _ int = 5            // Id. vierge pour une variable
    var _ float32 = 2.71828  // et une autre d'un autre type

_:  println("étiquette 1")   // Id. vierge pour une étiquette
_:  println("étiquette 2")   // ...et une autre
}
```

La portée des identificateurs dépend de la nature de l'artefact déclaré, selon les règles du tableau 7.3. Un point qui peut surprendre dans ce tableau est l'absence de portée pour les méthodes et les étiquettes :

✓ Pour les méthodes, cela reflète le fait qu'il n'est pas possible d'accéder directement à une méthode par son seul identificateur, mais uniquement à partir du type auquel elle est rattachée – dans une expression de méthode – ou d'une valeur de ce type.

© Dunod – Toute reproduction non autorisée est un délit.

Tableau 7.3 – Portée des déclarations

Déclaration	Portée
identificateurs prédéclarés	bloc univers
constante, fonction, type, ou variable de premier niveau	bloc paquet
paquet	bloc fichier de l'import
récepteur de méthode, paramètre ou résultat de fonction	bloc corps de la fonction
constante ou variable dans un bloc fonction ou inférieur	de la fin de la liste d'identificateurs à la fin du bloc
type dans un bloc fonction ou inférieur	de l'identificateur à la fin du bloc
étiquette d'instruction	bloc de la fonction englobante

✓ Pour les étiquettes, celles-ci suivent des règles différentes des autres identificateurs : elles n'entrent pas en conflit dans une même portée avec d'autres identificateurs du même nom, tant qu'ils ne désignent pas eux aussi des étiquettes.
Leur portée est celle de l'ensemble de la fonction où elles sont définies, y compris avant leur déclaration (*hissage* ou *hoisting*), et exclut les blocs internes à cette fonction : elles n'ont donc pas de portée bloc, à la différence de tous les autres identificateurs.

La différence de traitement entre constantes et variables d'une part, et types d'autre part, lors de la définition dans un bloc de fonction ou inférieur, reflète le fait que les déclarations des premières peuvent être groupées en déclaration avec initialisation parallèle, alors qu'une déclaration de type est toujours isolée.

Dans les deux cas, toutefois, le point important est l'absence de hissage, à la différence des étiquettes.

7.3.2 Instruction étiquetée

Une instruction étiquetée est créée par la déclaration d'une étiquette devant une instruction, et se compose :

✓ **D'une étiquette (*label*)** : un identificateur.

✓ **D'un séparateur deux-points** : pour séparer l'identificateur de l'instruction.

✓ **D'une instruction** : une instruction quelconque, même vide.

Bien qu'il soit d'usage se laisser l'étiquette seule sur sa ligne, ce n'est pas une obligation, comme l'illustrent les lignes 31 et 32 du listing 7.9.

Le formatage standard applique une indentation négative d'un niveau pour l'étiquette par rapport à l'instruction qu'elle cible, comme le montrent les listings 7.16 page 256 et 7.20 page 264.

L'identificateur devra être le seul label du même nom dans la fonction où il est déclaré, et être utilisé, à l'instar de tout autre identificateur. En revanche, comme mentionné à propos des instructions de déclaration, il pourra porter le même nom qu'un identificateur de toute autre nature.

Comme tout identificateur, une étiquette a l'obligation être utilisée pour ne pas provoquer d'erreur de compilation. Cette utilisation est réalisée par les instructions de déroutement **`break`** (§ 7.3.9), **`continue`** (§ 7.3.10), et **`goto`** (§ 7.3.11). Elle ne définit pas de valeur associée à l'exécution, et ne peut donc pas être utilisée dans une autre instruction que celles-ci.

7.3.3 Bloc

Comme en C, une suite d'instructions groupées entre deux accolades constitue une instruction unique [4], qui est l'instruction de type bloc (*block*).

Ce n'est pas une *instruction simple*.

En complément des blocs définis par ces instructions existent des blocs implicites : blocs d'univers, de paquet, de fichier, d'instruction, et de clause.

Les blocs sont imbriqués et influencent la portée des identificateurs.

Leur organisation des blocs est détaillée au § 2.2, et les règles de portée des identificateurs qui y sont déclarés sont détaillées à propos de l'instruction de déclaration au § 7.3.1.

7.3.4 Si - alors - sinon : `if` - `else`

7.3.4.1 Notation

L'instruction **`if`** fournit la traditionnelle condition à deux branches si-alors-sinon. Le langage n'inclut pas de condition ternaire avec la notation

```
condition ? valeur_si : valeur_sinon
```

à la différence de la plupart des autres langages dérivés de C.

Elle tire parti des instructions simples pour permettre la déclaration de variables temporaires, et est notée :

- ✓ **Mot-clef `if`** : le mot clef déterminant l'instruction.
- ✓ **Instruction simple** : optionnelle. Si elle est présente, ce sera le plus souvent une déclaration courte, introduisant une variable temporaire de portée limitée aux deux blocs de l'instruction **`if`**, comme à la ligne 22 du listing 7.10.

4. La grammaire C ANSI utilise le nom `compound_statement` plutôt que `block` pour cette notion. Ces règles sont définies sur https://osinet.fr/go/instr/ansi-c-composite

© Dunod – Toute reproduction non autorisée est un délit.

Listing 7.10 – Instructions `if`, `else` et ternaires
https://osinet.fr/go/instr/play-if-else-ternaire

```
package main

func condFunc(b bool, t, f interface{}) interface{} {
    if b {
        return t
    }
    return f
}

func main() {
    color := 15

    res := map[bool]string{
        true:  "blue",
        false: "red",
    }[color > 10]
    println(res)                // red

    res = condFunc(color > 10, "blue", "red").(string)
    println(res)                // blue

    if color := 5; color > 10 {
        println("blue", color)
    } else {
        println("red", color)   // red 5
    }
    println(color)              // 15
}
```

- ✓ **Terminateur point-virgule** : optionnel, mais obligatoire si une instruction simple est présente, pour la séparer de la condition.
- ✓ **Condition** : une expression quelconque dont la valeur est un booléen.
- ✓ **Bloc « si »** : une instruction « bloc » exécuté si la condition est vérifiée.
- ✓ **Mot-clef `else`** : optionnel, le mot clef signalant la présence d'une branche « sinon ». Il doit être présent sur la même ligne que l'accolade de fin du bloc « si », pour éviter que l'instruction `if` ne se termine et que le `else` ne se retrouve isolé.
- ✓ **Bloc « sinon » ou autre instruction `if`** : optionnel, mais obligatoire en présence du mot-clef `else`. Il est exécuté si la condition n'est pas vérifiée.

Le fait que les deux blocs « si » et « sinon » soient des instructions de type bloc implique que les accolades sont obligatoires, même pour une instruction unique, suivant les recommandations en vigueur dans d'autres langages où elles ne sont pas obligatoires, comme PHP [5] ou JavaScript et les langages dérivés, pour éviter les ambiguïtés pouvant résulter de leur absence, comme évoqué au § 2.2.

7.3.4.2 Utilisation

Le style Go encourage une utilisation modérée des clauses « sinon », privilégiant l'utilisation de clauses de garde (*guard clause*) afin de minimiser la profondeur d'imbrication [6], comme dans la fonction `condFunc` du listing 7.10 : le principe consiste à éliminer les conditions non souhaitées par des **`if`** portant sur les conditions permettant un retour rapide, avant d'exécuter en fin de fonction la logique correspondant au cas d'utilisation nominal de celle-ci. Dans cet exemple, le traitement est le renvoi de `f`, qui se trouve directement dans le bloc fonction, alors qu'il serait dans un bloc imbriqué à un niveau si une clause **`else`** était utilisée.

Ce listing illustre aussi deux approches possibles pour les développeurs souhaitant, par habitude dans d'autres langages, avoir un équivalent de l'instruction ternaire :

- ✓ L'utilisation d'une fonction comme `condFunc`, prenant un paramètre booléen pour la condition, et deux paramètres quelconques, dont elle renvoie celui des deux qui correspond à la valeur de la condition. Elle est définie aux lignes 3-8 et utilisée aux lignes 19-20.
  Cette approche est notamment utilisée dans le moteur de blogue Hugo pour fournir la fonction `cond` dans les modèles (*templates*) [7].
  Utilisée dans un contexte autre que les modèles, elle présente comme inconvénient la nécessité d'une assertion de type sur le résultat comme à la ligne 19, ou l'utilisation de fonctions spécialisées par type, en l'absence de fonctions génériques dans Go 1.x.
- ✓ L'utilisation d'une carte comme aux lignes 13-16 permet de laisser l'instruction « en ligne » sans nécessiter de déclaration additionnelle.
  Son principal inconvénient est son manque de lisibilité.

Bien que ces deux mécanismes soient utilisables, ils n'appartiennent pas au style idiomatique Go et il est généralement préférable de s'en tenir à la formulation basique à deux branches si nécessaire, comme aux lignes 22-26, ou – mieux – à une formulation sans branche **`else`**, comme aux lignes 4-7.

5. PSR-12 : https://osinet.fr/go/instr/psr-12-if
6. Effective Go : https://osinet.fr/go/instr/go-if
7. Documentation Hugo : https://osinet.fr/go/instr/hugo-cond

© Dunod – Toute reproduction non autorisée est un délit.

7.3.5 Commutateur d'expression `switch` v

Comme dans la plupart des langages dérivés de C, Go comporte une généralisation du si-alors-sinon, sous la forme d'une expression de commutation sur la valeur d'une expression, dénommée **`switch`**. À la différence de ces autres langages, l'instruction **`switch`** existe aussi sous une autre forme, le commutateur de type, décrit au § 7.3.6.

7.3.5.1 Notation

La notation du commutateur d'expression est similaire aux autres langages de la même lignée, comme illustré par le listing 7.11 :

Listing 7.11 – L'instruction **`switch`** d'expression
https://osinet.fr/go/instr/play-switch-expression

```
1  package main
2
3  func un() float64 {
4          return 1.0
5  }
6
7  func commute(x interface{}) {
8          switch x {
9          case true:
10                 println("true 1")
11         case true, false:
12                 println("true ou false")
13         case "s":
14                 println("chaîne")
15         case 2.0, un():
16                 println("float", x.(float64))
17         default:
18                 println("default")
19         }
20 }
21
22 func main() {
23         commute(true)   // true 1
24         commute(false)  // true ou false
25         commute(1.)     // float +1.00000e+000
26         commute(2.0)    // float +2.00000e+000
27         commute(-3.0)   // default
28         commute("s")    // chaîne
29 }
```

✓ **Mot-clef `switch`** : le mot-clef initialisant l'instruction.

✓ **Instruction simple** : optionnelle. Si elle est présente, ce sera le plus souvent une déclaration courte, introduisant une variable temporaire de portée limitée aux blocs de clauses de l'instruction **`switch`**.

✓ **Terminateur point-virgule** : optionnel, mais obligatoire si une instruction simple est présente, pour la séparer de l'expression.

✓ **Expression** : optionnelle : une expression quelconque – sauf **`nil`** non typé – à la valeur de laquelle les clauses de cas seront appliquées pour comparaison.
Si l'expression est une constante non typée, elle est convertie au type par défaut de sa nature de données (cf. § 5.3 page 155).
Enfin, si elle est absente, elle est évaluée par défaut à `true`.

✓ **Accolade ouvrante `{`** : débutant la liste de clauses de cas.

✓ **Clauses de cas** : une liste de clauses de cas, éventuellement vide. Chaque clause se compose de :
- **description de cas** : peut être de l'une de deux formes :
  - le mot-clef `case`, suivi d'une liste d'expressions, qui n'ont pas besoin d'être constantes, comme l'illustre la ligne 15 de l'exemple,
  - le mot-clef `default`, qui ne peut être présent qu'une fois au maximum dans la liste. Bien que ce cas puisse figurer n'importe où dans la liste, l'usage veut qu'il soit placé en dernier, pour maximiser la lisibilité ;
- **séparateur deux-points** : pour séparer la description de cas de la liste d'instructions ;
- **liste d'instructions** : exécutées si le cas décrit est vérifié.

✓ **Accolade fermante `}`** : terminant la liste de clauses de cas.

Au niveau syntaxe, la similitude superficielle présentée par les accolades après l'expression dans le **`if`** et dans les commutateurs **`switch`** ne doit pas masquer le fait qu'elles correspondent à deux traitements distincts :

✓ Dans le **`if`**, les accolades font partie d'une unique instruction de nature bloc.

✓ Dans les deux formes de **`switch`**, ce sont des délimiteurs qui entourent une liste de clauses, qui ne contiennent elles-même pas nécessairement des blocs, mais seulement des listes d'instructions.

© Dunod – Toute reproduction non autorisée est un délit.

Le cas des cas répétés

Le traitement des valeurs répétées dans la liste de cas est inhabituel par rapport aux autres langages de référence :

- ✓ En C/C++/C# ou Java, la répétition d'une valeur de cas est interdite, et provoque une erreur de compilation, reflétant l'implémentation possible comme une table de branchements, qui peut introduire des limitations sur les valeurs utilisations dans une telle expression, à l'image de la limitation aux entiers et énumérations en C.
- ✓ En ECMAScript ou PHP, elle est autorisée et sans limitation de type, reflétant l'implémentation comme une suite d'instructions **`if`**.

La spécification Go adopte un choix plus particulier :

- ✓ Elle dispose que les compilateurs peuvent l'interdire ou non, pour les types entiers, flottants (réels ou complexes) et les chaînes.
- ✓ Elle précise que les compilateurs fournis par le projet Go appliquent cette interdiction.
- ✓ Mais elle n'édicte pas d'interdiction pour les cas booléens ; et en pratique les compilateurs standard l'autorisent effectivement, comme montré par les deux cas `true` dans le listing 7.11.

7.3.5.2 Exécution

L'exécution se déroule en séquence :

1. Si applicable, exécution de l'instruction simple.
2. Évaluation de l'expression de commutation, conservée comme le serait une variable temporaire créée implicitement par une déclaration courte, `V`. Si l'instruction est absente, cette valeur vaut **`true`**.
3. Évaluation de chacune des clauses, de gauche à droite et de haut en bas. Pour chaque expression d'une clause non-`default` :
   - si elle n'est pas typée, sa valeur `C` est convertie au type de `V` ;
   - évaluation de la comparaison `C == V`.
4. Si l'une de ces comparaisons réussit, la liste d'instructions de la clause correspondante est exécutée.
   Une fois cette liste terminée, aucune autre comparaison n'est opérée.
   Ce déroulement peut être modifié par les instructions **`break`** (cf. § 7.3.9) et **`fallthrough`** (cf. § 7.3.5.3).
5. Si aucune comparaison ne réussit, et si une clause **`default`** est présente, c'est sa liste d'instructions qui est exécutée. Le fait qu'aucune des comparaisons ne réussisse ne provoque pas d'erreur, même en l'absence de clause **`default`**.
6. Une fois terminée l'exécution de la liste d'instructions sélectionnée, si elle existe, l'exécution se poursuit à l'instruction suivant le **`switch`**.

Le point marquant de cette approche est le fait qu'aucune comparaison ne soit effectuée après un premier succès, comme en Swift et à la différence de C, ECMAScript, PHP, et nombre d'autres langages, alors que dans ces autres langages, chaque liste

d'instructions d'une clause de cas doit se terminer par un **break** pour éviter de poursuivre l'exécution à la clause suivante : le comportement de Go est exactement inverse, dans la mesure où la poursuite des comparaisons doit être explicitement déclenchée par une instruction **fallthrough**.

Le fait que les expressions évaluées dans les cas soient quelconques et ne soient pas limitées à des valeurs constantes signifie que ce commutateur d'expression est équivalent à une succession de **if – else**, et que son utilisation pour remplacer une telle succession fait partie du style idiomatique Go.

7.3.5.3 Continuité d'exécution avec `fallthrough`

L'instruction **fallthrough** permet de modifier le fonctionnement de l'instruction de commutateur d'expression.

Elle ne peut y être utilisée que sous deux conditions :

- ✓ Figurer dans la liste d'instruction d'une clause de cas qui n'est pas la dernière du commutateur.
- ✓ Être la dernière instruction non vide de cette liste d'instructions.

Lorsqu'elle est insérée en fin de la liste d'instruction d'une clause de cas exécutée, l'exécution ne se poursuit pas après l'instruction **switch**, mais se poursuit à la première instruction de la liste d'instructions du cas suivant, en ignorant la liste de comparaisons de celui-ci, comme l'illlustre le listing 7.12, qui affiche successivement :

- ✓ **3 < 5** : en application de la comparaison de `n < 5` avec l'expression vide évaluée à `true` par défaut.
- ✓ **3 > 5** : en raison du **fallthrough** ligne 10, qui passe outre la comparaison `n > 5` ligne 11.
- ✓ **3 = 5** : en raison du **fallthrough** ligne 13, qui passe outre la sélection du cas par défaut ligne 14.

Exercice Qu'affiche le même fragment avec `n = 5` ? Et avec `n = 7` ?

Solution

- ✓ https://osinet.fr/go/instr/play-fallthrough-5
- ✓ https://osinet.fr/go/instr/play-fallthrough-7

Cette instruction se révèle à l'usage compliquée à utiliser correctement et est de ce fait peu utilisée. Ainsi, dans la bibliothèque standard de Go 1.13.1, elle n'est employée que 255 fois pour 1.7 million de lignes hors blancs et commentaires.

On l'y trouve en particulier utilisée pour regrouper une logique entre des cas successifs dont celui situé après le **fallthrough** est plus général que celui qui le porte, comme dans l'exemple du listing 7.13.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 7.12 – Instruction `fallthrough`
https://osinet.fr/go/instr/play-fallthrough

```
1  package main
2
3  func main() {
4          const n = 3
5
6          // Pas d'expression: comparaisons à true.
7          switch {
8          case n < 5:
9                  print(n, " < 5\n") // 3 < 5
10                 fallthrough
11         case n > 5:
12                 print(n, " > 5\n") // 3 > 5
13                 fallthrough
14         default:
15                 print(n, " = 5\n") // 3 = 5
16         }
17 }
```

7.3.6 Commutateur de type `switch v.(type)`

De la même façon que l'instruction **`if`** est généralisée par le commutateur d'expression **`switch`**, l'expression d'assertion de type (cf. § 6.10) est généralisée par le commutateur de type, en réutilisant une partie de la syntaxe de **`switch`** pour fournir une comparaison de types au lieu d'une comparaison de valeurs.

Comme l'assertion de type qu'il généralise, le commutateur de type s'applique uniquement à des données de types interfaces, pour manipuler leur type dynamique.

7.3.6.1 Notation

La notation du commutateur de type est inspirée de celle du commutateur d'expression, comme illustré par le listing 7.14 :

✓ **Mot-clef `switch`** : le mot-clef initialisant l'instruction.

✓ **Instruction simple** : optionnelle, rarement présente : une seule utilisation hors des données de test du compilateur, dans l'ensemble des sources Go 1.13.1, pour calculer la valeur interface à soumettre au commutateur [8].

✓ **Terminateur point-virgule** : optionnel, mais obligatoire si une instruction simple est présente, pour la séparer de la garde de commutateur.

8. https://osinet.fr/go/instr/gh-go-switch-type-simple

Listing 7.13 – Utilisation pratique de `fallthrough`
https://osinet.fr/go/instr/play-fallthrough-pratique

```
1  package main
2
3  type Autorisation int
4
5  const (
6          Indéterminé Autorisation = iota
7          Interdit
8          Autorisé
9  )
10
11 func accès(état Autorisation) {
12         var message = "Accès:"
13         switch état {
14         case Autorisé:
15                 message += " accordé."
16         case Indéterminé:
17                 message += " code non reçu."
18                 fallthrough
19         default:
20                 message += " Refusé."
21         }
22         println(message)
23 }
24
25 func main() {
26         accès(Autorisé)     // Accès: accordé.
27         accès(Indéterminé) // Accès: code non reçu. Refusé.
28         accès(Interdit)     // Accès: Refusé.
29 }
```

✓ **Garde de commutateur** : description de la commutation à effectuer :

- **Déclaration courte** : de variable, optionnelle. Si elle est présente, comme à la ligne 9 de l'exemple, la variable temporaire qu'elle déclare est créée dans le bloc implicite défini par chaque clause de cas.
  Son type dépend du format de la clause de cas : si celle-ci ne spécifie qu'un seul type, la variable sera de ce type ; en revanche, si elle en spécifie plusieurs, la variable est du type de l'expression de la garde de commutateur.
- **Expression** : qu'elle soit utilisée comme partie de la déclaration courte optionnelle, ou isolée, cette expression doit avoir une valeur de type interface.
- **Chaîne `.(type)`** : cette chaîne réservée du langage ressemble à une assertion de type dans laquelle le nom du type serait remplacé par le mot-clef `type`.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 7.14 – L'instruction `switch` de type
https://osinet.fr/go/instr/play-switch-type

```
package main

import (
        "errors"
        "fmt"
)

func voir(i interface{}) {
        switch t := fmt.Sprintf("%-16T", i); c := i.(type) {
        case byte, int, rune:                   // Cas multiples
                fmt.Printf("%s\t%d\n", t, c)
        case error:                             // Cas interface
                fmt.Printf("%s\t%s\n", t, c.Error())
        case nil:
                fmt.Printf("%s\tnil\n", t)
        default:                                // Cas par défaut
                fmt.Printf("%s\t%#v\n", t, c)
        }
}

func main() {
        voir(2)                     // entiers              int 2
        voir(complex(4, 5))         // complex128           (4+5i)
        voir(errors.New("erreur")) // *errors.errorString erreur
        voir(nil)                   // <nil>                nil
}
```

- ✓ **Accolade ouvrante `{`** : débutant la liste de clauses de cas.
- ✓ **Clauses de cas** : une liste de clauses de cas, éventuellement vide. Chaque clause se compose de :
  - **Description de cas** : peut être de l'une de deux formes :
    - Le mot-clef `case`, suivi d'une liste de types, définis ou non, et qui peuvent être des types interfaces.
      Par extension, le mot-clef **`nil`** est également utilisable comme un type, pour accepter une interface **`nil`**. Une telle clause ne peut apparaître qu'une seule fois dans un même commutateur de type.
    - Le mot-clef `default`, qui ne peut être présent qu'une fois au maximum dans la liste. Bien que ce cas puisse figurer n'importe où dans la liste, l'usage veut qu'il soit placé en dernier, pour maximiser la lisibilité.
  - **Séparateur deux-points** : pour séparer la description de cas de la liste d'instructions.
  - **Liste d'instructions** : exécutées si le cas décrit est vérifié.
- ✓ **Accolade fermante `}`** : terminant la liste de clauses de cas.

Lorsque le type spécifié dans une clause de cas est un type interface, l'éventuelle variable définie par la déclaration courte optionnelle porte ce type comme type statique, et comme type dynamique celui de la valeur interface d'origine.

À la différence du commutateur d'expression, le commutateur de type ne peut pas comporter d'instruction **`fallthrough`**.

7.3.6.2 Exécution

Le commutateur de type est exécuté comme une séquence d'affectations du résultat d'assertions de type et d'évaluation de leurs résultats :

1. Évaluation de l'expression, stockage dans une variable temporaire `v`.
2. Pour chaque clause de cas, comparaison au(x) type(s) spécifiés :
   - pour la clause de cas **`nil`** si elle existe, comparaison `v == nil` ;
   - pour une clause de type `T`, exécution de `c, ok := v.(T)`, puis comparaison `ok == true` ;
   - pour une clause de types multiples, évaluation des assertions de type pour chacun d'entre eux, et affectation `c = v` en cas de succès de l'une d'entre elles ;
   - pour une clause **`default`**, affectation `c = v`.
3. Si la clause réussit, exécution de sa liste d'instructions, sinon, passage à la clause suivante.

   Durant l'exécution, si une variable temporaire a été initialisée du fait de la présence de la déclaration courte optionnelle, celle-ci est disponible dans la portée des instructions de la liste. Du fait du mécanisme décrit à l'étape précédent, son type est celui de la clause de cas pour les clauses à type unique, ou celui de l'expression pour les clauses à type multiples : des assertions de type supplémentaires peuvent alors être nécessaires pour obtenir une valeur utilisable.
4. À la fin de l'exécution des instructions d'une clause de cas, l'exécution se poursuit après l'instruction **`switch`**.

Ce déroulement peut être modifié par l'instruction **`break`** (cf. § 7.3.9).

7.3.7 Sélecteur de communication `select`

L'instruction **`select`** est, comme les deux types de **`switch`**, une instruction d'organisation de l'exécution vers des voies multiples. Elle se différencie des deux **`switch`** en portant, non sur les valeurs d'expressions ou sur les types, mais sur la capacité d'effectuer des opérations d'émission et réception sur des canaux.

Son nom évoque la fonction la fonction `select` de l'API UNIX [9], au rôle similaire.

9. https://osinet.fr/go/instr/wiki-select

© Dunod – Toute reproduction non autorisée est un délit.

7.3.7.1 Notation

L'instruction **`select`** rappelle le format de **`switch`**, avec un mot clef introductif et une liste de clauses de cas exécutés selon la situation, comme illustré par le listing 7.15, dont l'exécution reflète les quatre opérations entrelacées, de façon pseudo-aléatoire.

Elle en diffère notamment en ne permettant pas l'insertion d'une instruction simple préliminaire, se composant :

- ✓ **Du mot-clef `select`** : le mot-clef initialisant l'instruction.
- ✓ **D'une accolade ouvrante `{`** : débutant la liste de clauses de cas de communication.
- ✓ **De clauses de cas** : une liste de clauses de cas de communication, éventuellement vide. Chaque clause se compose :
  - ✓ **D'une description de cas** : elle peut être de l'une de trois formes :
    - Le mot-clef `case`, suivi d'une instruction d'émission, comme à la ligne 24 de l'exemple.
    - Le mot-clef `case`, suivi d'une instruction décrivant une réception, qui ne peut être que de l'une de trois formes utilisant une expression de réception :
      - une instruction de déclaration courte, comme à la ligne 26,
      - une expression d'affectation, comme à la ligne 29,
      - une expression de réception seule, comme à la ligne 35.
    - Le mot-clef `default`, qui ne peut être présent qu'une fois au maximum dans la liste. Bien que ce cas puisse figurer n'importe où dans la liste, l'usage veut qu'il soit placé en dernier, pour maximiser la lisibilité.
  - ✓ **D'un séparateur deux-points `:`** : pour séparer la description de cas de la liste d'instructions.
  - ✓ **D'une liste d'instructions** : exécutées si la communication décrite par le cas est possible.
- ✓ **D'une accolade fermante `}`** : terminant la liste de clauses de cas.

Les clauses de cas de type réception peuvent initialiser une ou deux valeurs, reflétant le fonctionnement de l'opérateur de réception (cf. § 2.4.9) : dans le cas du format à deux valeurs, la première est celle reçue du canal, et la seconde reflète le succès de la réception. Ce format peut être utile si les canaux utilisés en réception peuvent être fermés, puisqu'en ce cas les réceptions ne sont plus bloquantes et renvoient indéfiniment la valeur vierge du type d'élément du canal.

À la différence des instructions **`switch`** :

- ✓ il n'est pas possible de regrouper plusieurs opérations dans une même clause **`case`** : chaque clause ne peut concerner qu'une seule opération ;

Listing 7.15 – L'instruction **select**
https://osinet.fr/go/instr/play-select

```
package main

import (
        "fmt"
        "time"
)

func emet(ch chan int, base int) {
        for i := 0; i < 3; i++ {
                ch <- i + base
        }
}

func main() {
        canal := func() chan int { return make(chan int, 2) }
        c1, c2, c3, c4 := canal(), canal(), canal(), canal()
        go emet(c1, 1)   // Émission asynchrone sur c1
        go emet(c2, 101) // Émission asynchrone sur c2
        go emet(c3, 201) // Émission asynchrone sur c3
        println("Passe\t<-C1\t<-C2\t<-C3\tC3<-\tBlocages")
        var v int
        for bloqué, n := 0, 0; bloqué < 3 && n < 20; n++ {
                select {
                case c4<- n:    // Émission
                        fmt.Printf("%2d%30cémet\n", n, ' ')
                case i := <-c1: // Déclaration courte
                        fmt.Printf("%2d\t%3d\n", n, i)
                        bloqué = 0
                case v = <-c2:  // Expression affectable
                        fmt.Printf("%2d%17d\n", n, v)
                        bloqué = 0
                case v = <-c2:  // Expression affectable
                        fmt.Printf("%2d%17d\n", n, v)
                        bloqué = 0
                case <-c3:      // Réception seule
                        fmt.Printf("%2d%22cignore\n", n, ' ')
                        bloqué = 0
                default:
                        bloqué++
                        fmt.Printf("%2d%39d\n", n, bloqué)
                        time.Sleep(10 * time.Millisecond)
                }
        }
}
```

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ il n'est pas possible d'utiliser **`fallthrough`** pour enchaîner l'exécution des listes d'instructions de deux cas consécutifs ;
- ✓ plusieurs clauses peuvent être identiques, ou plus généralement opérer sur le même canal, comme aux lignes 29 et 32.

Toute clause impliquant un canal **`nil`** est bloquante, de sorte que si un **`select`** ne contient que des cas de communication sur de tels canaux, et pas de clause **`default`**, il est bloquant indéfiniment, de même qu'un **`select`** ne comportant aucun cas.

Le nombre de clauses n'est pas limité par la spécification du langage.

Comme pour les instructions **`switch`**, à la fin de l'exécution des instructions d'une clause de cas, l'exécution se poursuit après l'instruction **`select`**, sans boucler sur d'autres communications éventuellement possibles. Pour cette raison, cette instruction est généralement placée dans une boucle, le plus souvent infinie.

7.3.7.2 Exécution

Comme pour les commutateurs **`switch`**, l'exécution de **`select`** comporte plusieurs étapes [10] :

1. Évaluation de tous les opérandes canaux présents dans les clauses en réception, et des expressions utilisées dans les clauses en émission – mais pas de celles utilisées comme destination des instructions de réception. Chaque opérande est évalué une seule fois, dans l'ordre du code source.
   L'exécution de cette étape n'est pas conditionnée par la possibilité d'effectuer l'opération de communication concernée, de sorte que si cette évaluation provoque un effet de bord sur une expression, celui-ci aura lieu à chaque passage dans l'instruction, et non seulement lorsque la clause correspondante sera utilisée.
   Le résultat de l'évaluation est une liste de canaux utilisables et des valeurs qui peuvent être émises sur ceux visés par une clause de cas en émission [11]. L'implémentation standard identifie les canaux **`nil`** pour simplifier leur traitement toujours bloquant.
2. Trois cas peuvent se présenter :
   - si au moins une opération est réalisable, le runtime opère une sélection pseudo-aléatoire parmi ces opérations [12], et sélectionne le cas auquel elle appartient ;
   - si aucune opération n'est réalisable, et que l'instruction comporte un cas **`default`**, celui-ci est sélectionné ;

10. L'implémentation de l'instruction se trouve dans la fonction `runtime.selectgo`.
11. Cette donnée est représentée par la structure `runtime.scase`.
12. Dans l'implémentation standard, ce tirage utilise un générateur pseudo-aléatoire optimisé pour la vitesse plus que la distribution optimale, mais suffisant pour passer le test SmallCrush de la suite TestU01 (https://osinet.fr/go/instr/wiki-testu01), offrant une garantie suffisante d'équité du tirage pour la plupart des usages.

- si aucune opération n'est réalisable, et que l'instruction ne comporte pas de cas **`default`**, l'instruction bloque et l'exécution est transférée à la prochaine goroutine disponible.

3. Si le cas choisi n'est pas **`default`** :
   - l'opération de communication qu'il décrit est exécutée ;
   - si le cas est une instruction d'affectation ou déclaration courte à partir d'une réception, cette instruction est effectuée.
4. La liste d'instructions du cas est exécutée à son tour.

7.3.7.3 Utilisation : temporisation (timeout)

L'une des fonctionnalités présentes dans l'API UNIX `select` et absente de l'instruction **`select`** est la capacité à bloquer uniquement pour un délai limité, et sortir à son expiration si aucune entrée/sortie n'a été possible.

En effet, dans cette instruction, si aucun des cas de communication n'est utilisable, l'exécution se poursuit immédiatement dans le cas **`default`** et, si celui-ci n'interrompt pas l'exécution par un moyen comme un **`break`**, un **`goto`** externe, ou un **`return`** et que l'instruction se trouve dans une boucle infinie, le **`select`** sera de nouveau exécuté immédiatement, réalisant une attente active coûteuse en énergie.

L'approche la plus simple pour éviter ce problème est celle utilisée dans le listing 7.15 : utiliser un appel à `time.Sleep` dans le cas **`default`**, mais cette approche naïve – et très lisible – n'est pas optimale.

En effet, si une communication devient possible pendant la durée de sommeil, l'exécution demeurera bloquée jusqu'à la fin de cette appel, et la communication disponible aura dû attendre inutilement ; et si, pour limiter cette attente inutile, la durée de sommeil est très courte, le problème énergétique de la boucle serrée réapparaît.

Une meilleur solution est fournie directement par la bibliothèque standard, sous la forme de canaux prédéfinis dédiés à la temporisation, comme dans le listing 7.16.

Dans cette technique, le cas **`default`** des boucles infinies utilisées pour l'exemple est remplacé par une clause de cas en réception sur un canal prédéfini fourni par la fonction `time.After(d time.Duration) chan time.Time` [13]. À l'exécution, cette fonction renvoie un canal en réception, sur lequel l'instant courant est émis après le délai passé comme argument.

De la sorte, en situation réelle avec de multiples cas autres que celui dédié à la temporisation, la boucle bénéficie bien d'une temporisation qui peut être longue pour ne pas impacter l'efficacité énergétique de l'attente, tout en étant en mesure de réagir immédiatement si une communication vient à être disponible sur l'un des canaux observés.

13. Documentation : https://osinet.fr/go/instr/go-time-after

© Dunod – Toute reproduction non autorisée est un délit.

Listing 7.16 – Temporisation avec `time.After`
https://osinet.fr/go/instr/play-time-after

```
5  func sortieParBreak() {
6  boucle:
7          for { // Boucle infinie
8                  select {
9                  case <-time.After(1 * time.Second):
10                         break boucle // Sortie du for
11                 }
12         }
13         println("Sortie par break")
14 }
15
16 func sortieParGoto() {
17         for { // Boucle infinie
18                 select {
19                 case <-time.After(1 * time.Second):
20                         goto suite // Sortie du for
21                 }
22         }
23 suite:
24         println("Sortie par goto")
25 }
26
27 func sortieParReturn() {
28         for { // Boucle infinie
29                 select {
30                 case <-time.After(1 * time.Second):
31                         println("Sortie par return")
32                         return // Sortie de la fonction
33                 }
34         }
35 }
```

Des mécanismes plus élaborés sont disponibles avec :

- ✓ des canaux de temporisation réutilisables : `time.NewTimer`
  Documentation : https://osinet.fr/go/instr/go-newtimer
- ✓ des canaux émettant un information temporelle à intervalle fixe au lieu d'une seule fois : `time.NewTicker`
  Documentation : https://osinet.fr/go/instr/go-newticker

Plus généralement, l'instruction **`select`** est au cœur de l'utilisation des canaux et goroutines, et il est essentiel de la maîtriser pour la programmation concurrente (cf. § 10).

7.3.8 `for`, la boucle à tout faire

Là où la plupart des langages proposent une multiplicité de structures de boucles (`for` de style C, `foreach`, `for..in`, `for..of`, `do..while`, `repeat..until`, etc), l'approche simplificatrice qui a guidé la conception de Go a conduit ses concepteurs à n'inclure qu'une seule instruction de boucle : **`for`**.

Pour permettre de conserver une flexibilité équivalente aux autres langages courants, cette boucle dispose de deux variations syntaxiques :

✓ **`for` de condition** : cette variante boucle sur une condition, à l'image d'un `while` traditionnel.
Elle peut être complétée d'une instruction simple de pré-traitement et d'une autre de post-traitement pour constituer un **`for`** avec *clause for*, comparable au **`for`** de C et ses dérivés.

✓ **`for` d'étendue** : cette variante parcourt un itérateur par une instruction **`range`**, à l'image d'un `foreach` PHP ou d'un `for..of` Javascript.

7.3.8.1 Notation

La notation est la suivante, illustrée dans ses principales variations par le listing 7.17 qui utilise quatre formes différentes de la boucle **`for`** pour exécuter dix fois une action donnée :

✓ **Le mot-clef `for`** : le mot-clef initialisant l'instruction.

✓ **Un sélecteur de syntaxe** : l'une des quatre formes possibles de l'instruction :
- **Vide** : il est possible de ne rien préciser entre le mot-clef et son instruction bloc, comme dans la fonction `forEver` de l'exemple, ligne 5. Ce format est souvent informellement appelé *forever*, indiquant que la boucle ne s'arrête pas d'elle-même.
- **Condition** : une expression de type booléen peut être utilisée, comme dans la fonction `forWhile`, ligne 16.
- **Clause `for`** : la condition peut être encadrée de deux instructions simples, comme dans la fonction `forClause`, ligne 23. La délimitation des instructions et de la condition est réalisée par des points-virgules.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 7.17 – Quatre `for` de 0 à 9
https://osinet.fr/go/instr/play-for

```
func forEver() {    // Comme un while (true) {...}
        i := 0
        for {
                print(i, " ")
                i++
                if !(i < 10) {
                        return
                }
        }
}

func forWhile() {   // Comme un while (i < 10) {...}
        i := 0
        for i < 10 {
                print(i, " ")
                i++
        }
}

func forClause() { // Comme un for en C
        for i := 0; i < 10; i++ {
                print(i, " ")
        }
}

func forRange() {   // Comme un for ([k,v] of map) {...}
        tr := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
        for k, v := range tr {
                print(k, ":", v, " ")
        }
}
```

L'utilisation d'une affectation ou d'une déclaration courte pour l'instruction de gauche et d'une incrémentation ou décrémentation pour celle de droite est le cas le plus typique, imitant la syntaxe C ; mais presque toutes les instructions simple peuvent y être utilisées, comme l'illustre le listing 7.18, dans lequel aucune des deux instructions simples n'est de la forme habituelle.
La seule instruction simple interdite est la déclaration courte, et ce uniquement dans l'instruction de droite.

Listing 7.18 – Instructions simples de clause `for`
https://osinet.fr/go/instr/play-for-simple

```
3  func main() {
4          i, canal := 5, make(chan int, 1)
5          for i-- ; i > 0; i = <-canal {
6                  print(i, " ")
7                  canal <- i - 1
8          }
9  }
10
11 // Résultat: 4 3 2 1
```

- **Clause d'étendue `range`** : comme dans la fonction `forRange`, ligne 30, composée :
  - **D'une destination** : optionnelle. Elle se compose d'une affectation ou d'une déclaration courte à une ou deux cibles. Selon la nature de l'expression, elle peut initialiser une ou deux valeurs.
  - **Du mot-clef `range`** : le mot-clef permet d'identifier ce format de clause.
  - **D'une expression** : elle fournit les valeurs sur lesquelles l'itération va se dérouler. Le résultat de cette expression doit appartenir à un nombre limité de natures de données : tableaux, tranches, pointeurs sur tableaux, chaînes, cartes, et canaux en réception.
    Si l'expression est de type canal, une seule valeur sera disponible pour la destination, sinon celle-ci pourra recevoir une valeur ou deux.

✓ **Un bloc** : une instruction de type bloc comprenant la liste des instructions à exécuter à chaque itération.

7.3.8.2 *Exécution de `for..range`*

Dans la forme à clause **`range`**, des valeurs sont définies par la boucle.

Celle-ci itère sur toutes les entrées disponibles dans l'expression suivant le mot-clef **`range`**, produisant des *valeurs d'itération* (*iteration values*), qui pourront être utilisées pour initialiser les valeurs des *variables d'itération* (*iteration variables*), si une destination est présente.

Comme pour les affectations et déclarations courtes dont elle emprunte la syntaxe, la destination doit être adressable ou être une expression d'index de carte.

© Dunod – Toute reproduction non autorisée est un délit.

Dans le cas où la longueur de l'expression est constante et où la clause ne définit qu'une seule variable d'itération, ou aucune, l'expression elle-même n'est pas évaluée, n'étant pas nécessaire pour produire la séquence des indices qui compose les valeurs d'itération successive. Ceci correspond au cas des tableaux et pointeurs sur tableaux, mais pas des chaînes dont la longueur ne suffit pas à connaître les valeurs successives de l'itération, en raison du décodage des runes.

Dans tous les autres cas, l'expression est évaluée une seule fois à la première itération. Les éventuelles fonctions présentes dans la destination sont en revanche évaluées à chaque itération.

Les variables d'itération comportent les valeurs présentées dans le tableau 7.4 :

Tableau 7.4 – Variables d'itération dans un `for .. range`

Expression	Variable 1	Variable 2
chaîne	indice de rune (`int`)	rune
tableau	indice	élément
pointeur sur tableau		
tranche		
carte	clef	
canal	élément	non applicable

- ✓ Les indices d'itération sur les chaînes, tableaux et leurs pointeurs, et cartes, sont produits en séquence croissante.
  Dans le cas des chaînes, ils indiquent la position du premier octet de chaque rune, de sorte que la séquence n'est pas continue si une chaîne contient des caractères non-ASCII, qui donnent lieu à un encodage UTF-8 sur plusieurs octets.
  Si une chaîne contient des séquences UTF-8 invalides, elles sont remplacées par la rune de remplacement Unicode `\U+FFFD`.
  Ceci permet d'itérer simplement sur la notion naturelle de caractères plutôt que sur des décomptes d'octets, en retrouvant des runes correctes même en présence de séquences UTF-8 invalides.
- ✓ Dans le cas des cartes, en revanche, l'ordre n'est pas spécifié et peut varier d'une exécution à l'autre.
- ✓ Dans le cas des canaux, les valeurs reçues sont celles qui ont été émises vers le canal, jusqu'à sa fermeture.
  Une conséquence importante est que la boucle est bloquante si le canal n'est pas fermé, notamment dans le cas d'un canal `nil`.
  Une précaution pour éviter les situations de blocage dans les boucles `range` sur un canal dont la fermeture n'est pas maîtrisée consiste à utiliser plutôt une boucle non-`range` dont le bloc d'instructions effectue la réception au moyen d'un `select` qui permettra d'éviter les blocages, comme décrit au § 7.3.7.3.

Lors de l'itération **range** sur des tableaux, leurs pointeurs, et les tranches, l'évaluation initiale de l'expression suffit à produire toutes les valeurs en les protégeant contre les mises à jour, comme le montrent les deux premières boucles du listing 7.19.

En revanche, lors de l'itération sur les cartes, les modifications se reflètent sur la carte, comme le montre la troisième boucle du même listing :

- ✓ Si une entrée non encore parcourue est supprimée par `delete`, elle ne sera pas listée.
- ✓ Si une entrée est ajoutée durant la boucle, elle peut ou non apparaître en fonction de la valeur de sa clef par rapport à l'itérateur, de sorte que ce comportement est imprévisible d'une clef ajoutée à l'autre, et d'une exécution à l'autre.
- ✓ Si une donnée non encore parcourue est modifiée, c'est la valeur modifiée qui sera affectée à la variable d'itération.

7.3.8.3 Exécution des autres cas

Les trois autres formats se résument à l'exécution de la clause **for** :

1. L'instruction simple de gauche – dite *init statement* dans la référence Go – est exécutée une seule fois au début de l'instruction **for**. Les autres étapes sont présentes à chaque itération.
2. La condition est évaluée. Si sa valeur est **false**, la boucle se termine et l'exécution reprend à l'instruction suivant le **for**.
3. L'instruction de type bloc est exécutée.
4. L'instruction simple de droite – dite *post statement* dans la référence Go – est exécutée.
5. L'itération suivante reprend à l'évaluation de la condition.

La version avec une condition seule est équivalente à une clause **for** avec les instructions de gauche et droite vides, comme **for** `;condition; { /*...*/ }`.

La version vide (*forever*) est équivalente au cas précédent, avec pour condition la valeur constante **true**.

7.3.8.4 Précaution d'emploi avec forever

À la différence de **select** et des opérations et fonctions impliquant des canaux, la boucle **for** ne provoque pas intrinsèquement de commutation de goroutine à chaque itération.

De ce fait, un *forever* vide (**for** `{}`) boucle indéfiniment, sans jamais céder l'exécution à une autre goroutine. Si la logique d'un programme nécessite un *forever*, il est donc prudent que son bloc d'instructions contienne une opération permettant cette commutation.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 7.19 – Modification d'une expression de boucle `for`
https://osinet.fr/go/instr/play-for-modification

```
package main

import "fmt"

func main() {
        // Boucle non affectée par les modifications
        ta := [...]int{0, 1}
        for k, v := range ta {
                fmt.Printf("%3d %3d %v\n", k, v, ta)
                if k == 0 {
                        ta[1] = 101
                }
        }
        tr := []int{0, 1}
        for k, v := range tr {
                fmt.Printf("%3d %3d %v\n", k, v, tr)
                if k == 0 {
                        tr = append(tr, 2, 3)
                        tr[1] = 101
                }
        }
        // Boucle affectée par les modifications
        ca := map[int]int{0: 0, 1: 1}
        premier := true
        for k, v := range ca {
                fmt.Printf("%3d %3d %v\n", k, v, ca)
                if premier {
                        premier = false
                        for i := 0; i < len(ca); i++ {
                                ca[i] = i + 100
                        }
                        ca[2] = 2
                }
        }
}

// Résultat
//   0   0 [0 1]
//   1   1 [0 101]
//   0   0 [0 1]
//   1   1 [0 101 2 3] tranche modifiée, itération stable
//   0   0 map[0:0 1:1 2:2]
//   1 101 map[0:100 1:101 2:2]
//   2   2 map[0:100 1:101 2:2]
```

Cette limitation est à l'origine d'une évolution de l'implémentation standard [14] qui a introduit une préemption non coopérative des goroutines, partiellement mise en oeuvre dans Go 1.14 sur la plupart des plateformes cibles, les exceptions étant WebAssembly et les cibles plus exotiques `windows/arm`, `darwin/arm` et `plan9/*`. En tout état de cause, cette évolution ne modifie pas le langage lui-même, de sorte que même en sa présence, une implémentation conforme autre que la version standard de Go pourrait ne pas en bénéficier, et la prudence demeure de mise.

7.3.9 `break`, l'interrupteur

L'instruction **`break`**, commune à nombre de langages dérivés de C, existe aussi en Go, où elle sert permet de sortir d'une instruction complexe : les deux commutateurs **`switch`**, le sélecteur **`select`**, et toutes les formes de boucle **`for`**.

En termes de notation :

- ✓ Elle est similaire au **`break`** d'ECMAScript et ses dérivés, ou de Swift, acceptant en argument une étiquette d'instruction optionnelle.
- ✓ Elle diffère du **`break`** PHP, qui accepte un argument numérique optionnel spécifiant un nombre de niveaux d'imbrication à interrompre.
- ✓ Elle diffère aussi du **`break`** historique de C, C++, C# et Python, qui n'accepte aucun argument optionnel.

Utilisée sans argument, elle termine l'exécution de l'instruction complexe la plus interne à laquelle elle appartient, et l'exécution se poursuit à l'instruction suivant l'instruction complexe.

Lorsqu'elle est utilisée avec un argument étiquette, celui-ci désigne l'instruction étiquetée à interrompre, qui doit être l'une des instructions complexes dans la portée de laquelle se trouve le **`break`**, comme dans le listing 7.20, où les deux **`break`** provoquent respectivement une sortie du **`for`** `j` et du **`for`** `i` bien qu'ils soient tous les deux dans la boucle la plus interne.

7.3.10 `continue`, le raccourci

L'instruction **`continue`** est, comme dans nombre de langages dérivés de C, le complément de **`break`**.

Là où **`break`** permet de terminer une boucle **`for`**, **`continue`** est elle aussi typiquement placée quelque part au milieu des instructions du bloc de la boucle, et permet elle aussi de ne pas exécuter la fin de cette liste d'instructions.

En revanche, à la différence de **`break`**, l'exécution après **`continue`** prend un raccourci omettant les instructions suivantes du bloc, et reprend à l'itération suivante du **`for`** au lieu de terminer l'exécution de la boucle visée.

14. https://osinet.fr/go/instr/go-preemption

© Dunod – Toute reproduction non autorisée est un délit.

Listing 7.20 – Interruption avec **break**
https://osinet.fr/go/instr/play-break

```
package main

func main() {

        i := 0
        j := 0
Un:
        for ; ; i++ {
        Deux:
                for j = 0; ; j++ {
                        print(i, j, "\t")
                        if j > 1 {
                                break Deux
                        }
                        if i > 1 {
                                break Un
                        }
                }

        }
}

// Résultat: 00 01 02 10 11 12 20
```

En termes de notation, comme **break** :

- ✓ Elle est similaire au **continue** d'ECMAScript et ses dérivés, ou de Swift, acceptant en argument une étiquette d'instruction optionnelle.
- ✓ Elle diffère du **continue** PHP, qui accepte un argument numérique optionnel spécifiant un nombre de niveaux d'imbrication à parcourir pour trouver la boucle dont il faut passer à l'itération suivante.
- ✓ Elle diffère aussi du **continue** historique de C, C++, C# et Python, qui n'accepte aucun argument optionnel.

Utilisée sans argument, elle passe à l'itération suivante de l'instruction **for** la plus interne à laquelle elle appartient.

Lorsqu'elle est utilisée avec un argument étiquette, celui-ci désigne l'instruction **for** étiquetée sur laquelle itérer, qui doit être l'une de celles dans la portée de laquelle se trouve le **continue**, comme dans le listing 7.21, où les deux **continue** provoquent respectivement une itération du **for** `j` et du **for** `i`, bien qu'ils soient tous les deux dans la boucle la plus interne.

Listing 7.21 – Itération anticipée avec `continue`
https://osinet.fr/go/instr/play-continue

```
package main

func main() {

Un:
        for i := 0; i < 4; i++ {
        Deux:
                for j := 0; j < 4; j++ {
                        print(i, j, "\t")
                        if j < 1 {
                                continue Deux
                        }
                        j++
                        if i > 1 {
                                continue Un
                        }
                        i++
                }

        }
}

// Résultat: 00 01 13 30 31
```

7.3.11 Déroutement `goto`

7.3.11.1 Notation

Comme ses homonymes dans d'autres langages, l'instruction **`goto`** transfère l'exécution à une autre instruction portant l'étiquette utilisée comme argument du **`goto`**.

Sa notation est simplement **`goto`** `X`, où `X` est l'étiquette d'une instruction étiquetée, comme dans l'exemple du listing 7.22, qui illustre la même boucle, réalisée une fois avec l'instruction **`for`**, et l'autre fois à base de **`goto`** [15].

À la différence des implémentations de **`goto`** dans les langages historiques antérieurs à la programmation structurée, comme BASIC, C, Cobol ou Fortran, les destinations vers lesquelles un **`goto`** peut rediriger en Go sont strictement limitées :

✓ L'instruction ne peut être présente que dans une fonction, et ne peut viser qu'une instruction étiquetée de la même fonction.

15. Cette version représente une transcription en Go de la boucle **`for`** telle qu'elle est compilée en Go 1.13.1, comme le montre la compilation en cible assembleur sur https://osinet.fr/go/instr/asm-goto

© Dunod – Toute reproduction non autorisée est un délit.

Listing 7.22 – Émulation d'une boucle `for` avec `goto`
https://osinet.fr/go/instr/play-for-par-goto

```
func forÉmulé() {
        i := 0
        goto condition
bloc:
        print(i, " ")
        i++
condition:
        if i < 5 {
                goto bloc
        }
}

func forVéritable() {
        for i := 0; i < 5; i++ {
                print(i, " ")
        }
}
```

- ✓ Le saut du `goto` à l'instruction visée ne peut pas contenir d'instructions qui provoqueraient l'apparition de nouvelles variables dans la portée courante, comme des déclarations courtes ou normales.
- ✓ Un `goto` situé à l'extérieur d'un bloc contenant une instruction étiquetée ne peut pas la cibler ; seuls sont accessibles les instructions du même bloc ou de blocs extérieurs, dans les limites des deux clauses précédentes.

7.3.11.2 Saut non local

Bien que, comme le montrent les restrictions ci-dessus, l'instruction `goto` soit conçue comme l'antithèse absolue d'un saut global tel que le couple `setjmp`/`longjmp` familier aux développeurs C/C++, il est tout de même possible de construire des programmes Go utilisant ce mécanisme de saut non local comme alternative à `goto`, à deux conditions :

- ✓ compiler pour CGO et non pour la chaîne de compilation standard ;
- ✓ utiliser `setjmp` et `longjmp` dans des portions C invoquées depuis le code Go [16].

16. Il existe une proposition d'évolution (26058) de ce critère, qui autoriserait les sauts introduisant de nouvelles variables dans la portée, à condition que la variable ne soit pas utilisée après l'étiquette visée. Cette proposition est encore en discussion durant le développement de Go 1.15 et, comme elle modifierait un élément de la syntaxe Go, ne concerne que Go 2.

Ce mécanisme est connu et – au moins jusqu'en Go 1.13 – comporte des bogues d'exécution avec une stratégie de résolution non déterminée par le projet Go, en particulier pour les cibles Windows et/ou en présence d'exceptions.

L'utilisation des variantes `__builtin_setjmp` et `__builtin_longjmp`, propres aux compilateurs C GNU, est documentée comme moins sujette à problèmes que l'appel de ces deux fonctions génériques. Voir en particulier les tickets `golang/go` 13672 et 30900 pour plus de détails.

7.3.11.3 Pourquoi `goto` ?

Une connaissance superficielle de l'histoire des langages a amené nombre de commentateurs, lors de la présentation de Go, à critiquer le présence de cette instruction dans un nouveau langage, se fondant souvent sur la seule connaissance de l'existence de la célèbre « lettre à l'éditeur » de E.W. Dijkstra publiée dans Communications of the ACM en 1968 et titrée – par ledit éditeur, en l'occurrence Niklaus Wirth – *Go To Statement Considered Harmful* [17] ou sur le théorème de Böhm–Jacopini [18].

L'idée sous-jacente est à rechercher dans l'état des langages disponibles à l'époque, comme BASIC, Cobol, Fortran ou Lisp, qui ne possédaient pas encore les structures de boucles qu'allaient apporter les années suivantes, leurs versions de cette époque ne comportant pas même de clause `else` dans les instructions `if`, ce qui conduisait les programmeurs à produire du code peu lisible à base de `goto` pour les simuler, comme dans l'exemple du listing 7.22, dit « code spaghetti » [19].

L'apport essentiel de l'article de Dijkstra, qui allait être l'un des piliers de la révolution de la programmation structurée dans les années 1970, tient en cette phrase :

> « […] *whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained* […] »
>
> E.W. Dijkstra - *Go To statement considered harmful*
> *Communications of the ACM*, Vol. 11, n° 3, mars 1968

Ce « système de coordonnées » qu'évoque l'article est en substance l'état des variables disponibles à l'exécution, qui ne doit pas changer du **`goto`** à sa destination.

Il est traduit en Go par les trois critères décrits au § 7.3.11.1, qui interdisent les trois situations où de tels changements pourraient survenir : à la sortie d'une fonction, en passant par dessus une création de variable sans hissage (*hoisting*), ou en passant d'un bloc externe interne à un bloc interne, susceptible de déclarer de nouvelles variables.

17. Une version annotée de cette lettre est présentée sur https://osinet.fr/go/instr/goto-harmful
18. https://osinet.fr/go/instr/wiki-bohm-jacopini
19. Exemples en BASIC et Fortran sur https://osinet.fr/go/instr/wiki-spaghetti

© Dunod – Toute reproduction non autorisée est un délit.

La même citation mentionne la principale utilisation de cette instruction : les clauses d'abandon (*abortion clauses*), en d'autres termes une utilisation comme destination en fin d'une fonction pour sortir d'une suite de tests et conditions imbriquées et revenir au premier niveau pour mettre fin à l'exécution d'une fonction de manière ordonnée

La lecture du code source de la bibliothèque standard Go et de son outillage montre que ces clauses sont bien le principal cas d'utilisation de `goto`, comme dans l'exemple de la fonction Gamma d'Euler fourni par le paquet `math` :

https://osinet.fr/go/instr/go-gamma-euler

7.4 INSTRUCTIONS LIÉES AUX FONCTIONS

7.4.1 Retour de fonction avec `return`

7.4.1.1 Définition

L'instruction `return` exécute trois tâches relativement indépendantes les unes des autres :

- ✓ Déclencher l'exécution du code différé en fin de fonction par les instructions `defer` exécutées précédemment.
- ✓ Définir les résultats renvoyés par la fonction en cours.
- ✓ Terminer l'exécution de la fonction en cours, pour que l'instruction suivante soit l'instruction suivant l'appel de la fonction (cf. § 6.11).

7.4.1.2 Notation

La notation, illustrée par le listing 7.23, se compose de :

- ✓ **Le mot-clef `return`** : le nom de l'instruction.
- ✓ **Les résultats** : une liste de valeurs à renvoyer de la fonction, qui peut avoir l'une des trois formes suivantes :
  - Une liste d'expressions non vide, dont le nombre et les types correspondent au nombre et aux types des résultats déclarés pour la fonction.
  - Une expression d'appel à une fonction/méthode, dont la liste de résultats, non vide, correspond au nombre et aux types des résultats déclarés pour la fonction. Les résultats de cette fonction sont traités comme autant de résultats d'expressions à une seule valeur, constituant une liste d'expressions de la forme précédente.
  - Une liste vide : un simple `return` déclenche le retour sans modifier les résultats de la fonction. Il n'est possible que si la liste de résultats de la fonction est vide, ou composée de résultats nommés (cf. § 3.3.2 et 3.3.3).

Listing 7.23 – Les quatre formes de **return**
https://osinet.fr/go/instr/play-go

```go
package main

import "fmt"

func sans() { return }          // Return nu

func deux() (int, int) {
    return 2, 20                // Liste
}

func multi() (int, int) {
    return deux()               // Résultat multi-valué
}

func nommés() (x, y int) {
    x, y = 3, 30
    return                      // Obligatoire
}

func surchargés() (x, y int) {
    x, y = 3, 30                // Résultat pour return nu
    return 4, 40                // Surchargé par explicite
}

func main() {
    sans()                      // <rien>
    fmt.Println(deux())         // 2 20
    fmt.Println(multi())        // 2 20
    fmt.Println(nommés())       // 3 30
    fmt.Println(surchargés()) // 4 40
}
```

7.4.1.3 Fonctionnement

Lors de l'entrée de l'exécution dans une fonction, comme vu au § 3.3, la mémoire nécessaire aux résultats déclarés dans la signature de la fonction est allouée, et la valeur des résultats est initialisée à la valeur vierge de leur type.

Si les résultats sont nommés, ces résultats sont des variables ordinaires, sinon ce sont des quasi-variables.

La seule différence entre les deux est que, en présence de résultats nommés, l'existence de la variable permet aux instructions de la fonction en cours de modifier

© Dunod – Toute reproduction non autorisée est un délit.

les valeurs associées, comme aux lignes 16 et 21 de l'exemple. S'agissant de variables et non de constantes, elles peuvent être modifiées autant de fois que souhaité.

Lors de l'exécution du return, deux cas peuvent se présenter :

- ✓ Si le **`return`** est nu, comme à la ligne 17, l'instruction n'affecte pas de valeurs à ces variables, qui constituent donc le résultat de la fonction.
- ✓ Si le **`return`** reçoit une liste de valeurs ou une expression multi-valuée compatible, comme aux lignes 8, 12 et 22, les valeurs correspondantes sont affectées aux variables ou quasi-variables constituant le retour de la fonction. Dans le cas des retours nommés, ceci surcharge les valeurs définies avant l'exécution du retour.

L'instruction **`return`** effectue alors sa deuxième mission, le renvoi des valeurs de la fonction, puis la dernière, la transfert de l'exécution à l'appelant.

En présence de retours nommés, la fin d'exécution implicite en fin d'exécution d'une fonction, comme dans la fonction `sans` ligne 5, ne provoque pas la deuxième étape du renvoi des résultats, de sorte que la fonction ne renverrait pas les résultats attendus avant de transférer l'exécution à l'appelant.

Ceci provoque donc une erreur de compilation, rappelant l'obligation d'inclure une instruction **`return`** – même sans liste de valeurs – dans une fonction devant renvoyer des résultats.

Pour la même raison, dans une fonction à résultats anonymes, les quasi-variables allouées pour les résultats ne peuvent pas être affectées autrement que par un retour explicite, qui ne peut dans ce cas pas être un **`return`** nu, ne disposant pas de variables décrivant les valeurs à renvoyer.

7.4.2 Exécution concurrente avec `go`

7.4.2.1 Définition

L'instruction **`go`** permet de lancer l'exécution d'un appel de fonction / méthode concurremment avec la fonction en cours, mais dans une unité d'exécution indépendante, partageant le même espace d'adressage que la fonction appelante. L'identification de la fonction et de ses paramètres suit les règles normales de l'expression d'appel de fonction (cf. § 6.11).

Cette unité d'exécution est dite *goroutine*.

Après l'instruction, l'exécution continue dans la goroutine appelante et, concurremment, dans la nouvelle goroutine. Comme l'illustre l'exemple du listing 7.24, la goroutine appelante n'attend pas le retour de la fonction appelée, qui est ignoré.

Comme le résultat d'une goroutine lancée par **`go`** est ignoré, une goroutine ne peut pas renvoyer de résultats à son appelant, et doit donc utiliser un autre moyen pour les communiquer ; ce moyen est le plus souvent l'émission sur un canal de résultats,

généralement passé comme argument à la nouvelle goroutine lors de l'instruction **go** : cela permet de fournir une synchronisation simple au moyen d'expressions de réception, notamment groupées dans des sélecteurs de communication **select**.

Les goroutines, canaux et autres mécanismes liés à la programmation concurrente, sont décrits en détail au chapitre 10.

Listing 7.24 – Exécution concurrente avec **go**
https://osinet.fr/go/instr/play-go-execution

```
1  package main
2
3  import "time"
4
5  type T string
6  func (t T) Show() { println(t) }
7
8  func main() {
9          t := T("Goroutine")
10         println("Avant")        // Première ligne
11         go t.Show()             // Troisième ligne
12         println("Juste après") // Deuxième ligne
13         time.Sleep(100 * time.Millisecond)
14         println("Plus tard")    // Quatrième ligne
15 }
```

7.4.2.2 Notation

La notation, illustrée par le listing 7.24 ligne 11, se compose :

- ✓ **du mot-clef go** : le nom de l'instruction ;
- ✓ **d'une expression d'appel** : de méthode ou de fonction, nommée ou anonyme. Cette expression ne peut pas être parenthésée.

La fonction appelée peut être toute fonction du programme, ou l'une des fonctions intégrées ne faisant pas partie de la liste des fonctions interdites dans les instructions d'expression, listées dans le tableau 7.1, page 229.

La fonction peut renvoyer tout type de résultat, qui sera toujours ignoré.

7.4.2.3 Fonctionnement interne

Une description simplifiée consiste à dire qu'un appel de goroutine par **go** est *presque* implémenté comme un appel de fonction, modifié principalement par le fait qu'il est réalisé par une fonction.

© Dunod – Toute reproduction non autorisée est un délit.

Dans un appel de fonction, la séquence est, en version simplifiée :

✓ dépose des arguments sur la pile ;

✓ appel de la fonction.

Dans un appel de gouroutine, la séquence est modifiée de la façon suivante :

✓ dépose des arguments sur la pile ;

✓ dépose de deux arguments supplémentaires : l'adresse de la fonction à appeler, et la longueur en octets de la liste d'arguments sur la pile ;

✓ appel de la fonction `runtime.newproc` [20].

La fonction `runtime.newproc` obtient ensuite une goroutine, copie la pile reçue sur la longueur correspondant à l'argument longueur, et invoque la fonction reçue comme argument pointeur de fonction.

Ce mécanisme est décrit plus en détail dans l'article de Russ Cox sur l'utilisation de la pile : https://osinet.fr/go/instr/cox-pile

La description qu'il en donne concerne l'implémentation de **`go`** telle qu'elle était dans la version `weekly.2010-03-30`, juste avant qu'il ne la modifie pour ajouter les fonctions intégrées `panic` et `recover` (cf. § 8.9) introduites dans la version `weekly.2010-04-13`. L'implémentation courante dans Go 1.14 est décrite par les structures `runtime.g`, `runtime._defer` et `runtime._panic`, dans le fichier `src/runtime/runtime2.go`. Au fil des versions l'implémentation a évolué par rapport à l'article, en particulier avec le changement de format de la pile, et lors de la conversion du compilateur de C à Go, mais en conservant la même logique générale, le principal changement étant l'ajout de champs supplémentaires à la structure `Defer` (puis `_defer`) pour permettre le fonctionnement de `panic` et `recover`.

7.4.3 Code terminal avec `defer`

7.4.3.1 Définition

L'instruction **`defer`** programme l'exécution d'un appel de fonction ou méthode pour la fin de l'exécution de la fonction dans laquelle **`defer`** est invoqué.

Les paramètres, et pour les méthodes la valeur du récepteur, sont évalués au moment de l'exécution de l'instruction **`defer`**, tandis que l'appel proprement dit, avec transfert de l'exécution à la fonction ainsi différée, survient après la dernière instruction dans la séquence d'exécution normale de la fonction, comme dans l'exemple du listing 7.25.

Comme dans les goroutines créées par l'instruction **`go`**, le résultat des appels ainsi différés est ignoré. En revanche, comme elles s'exécutent dans la même goroutine que l'appelant, elles n'introduisent pas de problèmes de situation de compétition entre la fonction appelante et les fonctions différées appelées.

20. https://osinet.fr/go/instr/play-newproc

Listing 7.25 – Code terminal avec **defer**
https://osinet.fr/go/instr/play-defer

```
1  package main
2
3  func identité(t T) T { println("Identité"); return t }
4
5  type T string
6
7  func (t T) ShowV()  { println("V", t) }
8  func (t *T) ShowP() { println("P", *t) }
9
10 func diffère() T {
11     var t T = "Initial"
12     var v *T = &t
13     println("Début de fonction") // 1ère ligne
14     defer v.ShowP()              // 4ème: "V initial"
15     defer v.ShowV()              // 3ème: "P modifié"
16     t = T("Modifié")             //
17     return identité("Résultat")  // 2ème: "identité"
18 }
19
20 func main() {
21     résultat := diffère()
22     println(résultat)             // 5ème: "résultat"
23 }
```

Particularités :

- ✓ Lorsque plusieurs instructions **defer** sont présentes dans une même fonction, les appels différés sont empilés dans l'ordre d'exécution des instructions **defer**, et dépilés à la fin de la fonction, de sorte que les appels de fonction programmés par les derniers **defer** exécutés sont les premiers exécutés, comme le montrent les lignes 14 et 15 de l'exemple : l'appel à `t.ShowP` est exécuté après l'appel à `t.ShowV`.
- ✓ Dans le cas des appels de méthodes, le récepteur est évalué lors du **defer** et non lors de l'appel, comme le montrent les résultats des lignes 14 et 15 : bien que le récepteur ait été modifié après les **defer**, l'appel de la méthode à récepteur valeur utilise la valeur initiale, tandis que l'appel de la méthode à récepteur pointeur utilise la valeur initiale du pointeur, et la déréférence, obtenant la valeur courante du récepteur, donc la valeur modifiée.
- ✓ Lorsque la fonction appelante contient une instruction **return**, l'évaluation des valeurs de la liste d'expressions argument de **return** survient avant l'exécution du code différé, comme à la ligne 17.

© Dunod – Toute reproduction non autorisée est un délit.

✓ Les fonctions différées ne peuvent pas accéder à ces valeurs devant être renvoyées. En revanche, si la fonction utilise des résultats nommés, la portée de ceux-ci est la fonction, y compris les éventuelles fonctions anonymes qui y seraient déclarées et utilisées comme code terminal avec **defer**.
Ces fonctions peuvent donc accéder aux valeurs de retour, y compris les valeurs explicites passées à l'instruction **return**, comme dans le listing 7.26.

Listing 7.26 – Accès aux résultats de fonction dans une fonction différée
https://osinet.fr/go/instr/play-defer-resultats

```
package main

func accessible() (n int) {
        n = 2
        defer func() {
                print(n)
                n = 4
        }()
        return
}

func surchargé() (n int) {
        n = 2
        defer func() {
                print(n)
                n = 4
        }()
        return 3
}

func main() {
        println("\t", accessible()) // 2 4
        println("\t", surchargé())  // 3 4
}
```

7.4.3.2 Fonctionnement

En interne, le fonctionnement de **defer** est jusqu'en Go 1.13 similaire à celui de **go** décrit au § 7.4.2.3, en ce qu'il fonctionne par un mécanisme d'appel indirect, mais en invoquant la fonction `runtime.deferproc` au lieu de `runtime.newproc`, avec la nécessité de nettoyer la pile au retour. L'article déjà cité pour **go** fournit le même niveau de détail pour **defer**.

À partir de Go 1.14, le compilateur optimise certains de ces cas en insérant des appels de fonction en ligne, supprimant la nécessité des manipulations de pile,comme détaillé par Jaana B. Dogan sur https://osinet.fr/go/instr/dogan-defer

8

Builtins : fonctions intégrées et erreurs

Les *builtins*, ou fonctions intégrées, sont des pseudo-fonctions fournies par le runtime Go et connues du compilateur :

- ✓ Elles s'utilisent avec la syntaxe des fonctions, prenant des arguments et renvoyant un résultat.
- ✓ Elles diffèrent des fonctions en ce que certaines d'entre elles, dont le seul rôle est de renvoyer des résultats, ne peuvent pas être utilisées dans les instructions simples d'expression ou l'instruction `go`.
- ✓ Elles sont connues du compilateur et, de ce fait, leur invocation peut donner lieu à une génération de code différente de celle d'un appel de fonction normal, comme le montre le listing 8.1 page 276.
- ✓ Elles peuvent recevoir comme arguments des types, ce qui n'est pas possible pour une véritable fonction, et conduit là aussi à une génération de code différente d'un appel de fonction normal.

Par extension, ce chapitre inclut aussi la description du seul type interface intégré au langage, le type `error`, et des mécanismes fournis par Go pour l'exploiter.

Tous ces identificateurs, comme ceux des types simples également fournis par le runtime, ont pour portée le bloc univers, et sont accessibles sans les qualifier par un paquet, alors même qu'ils ne commencent pas par une majuscule, ce qui les rendrait non exportés s'ils étaient fournis par du code utilisateur.

Ils sont définis à titre de documentation interne du projet, dans le fichier `src/builtin/builtin.go` de la distribution Go [1].

1. Dans `src/pkg/builtin/builtin.go` de Go 1 à Go 1.3.

8.1 AFFICHAGE : `print` ET `println`

Les fonctions intégrées d'affichage sont les mécanismes d'affichage créés durant la création des premières versions du compilateur Go (*bootstrapping*).

Bien qu'elles fassent partie des éléments de la spécification Go, elles sont exclues de la promesse de compatibilité Go 1, et leur spécification de conformité entre implémentations est limitée :

- ✓ une implémentation Go conforme n'a pas l'obligation de pouvoir afficher tous les types d'arguments au moyen de ces fonctions ;
- ✓ les seuls affichages garantis disponibles sont ceux des booléens, chaînes, et nombres.

En termes d'implémentation, le listing 8.1 montre la différence de traitement entre un appel à `println` et à une fonction normale, illustrant son caractère de fonction intégrée connue par le compilateur.

Comme le montre cet exemple, son utilisation produit un découpage des opérations d'impression en opérations d'affichage spécialisées, argument par argument, et pour le `U+000A` final.

Listing 8.1 – Différence entre une fonction normale et une fonction intégrée
Compiler avec « `-gcflags -l` » pour éviter l'inlining de la fonction `affiche`

```
1  package main
2
3  func affiche(n int) { println(n) }
4
5  func main() {
6          const n int = 25
7          affiche(n)
8          println(n)
9  }
```

```
9     affiche(n)
10            MOVQ $0x19, 0(SP)
11            CALL main.affiche(SB)
12    println(n)
13            CALL runtime.printlock(SB)
14            MOVQ $0x19, 0(SP)
15            CALL runtime.printint(SB)
16            CALL runtime.printnl(SB)
17            CALL runtime.printunlock(SB)
```

Fonctionnellement, ces fonctions sont beaucoup plus limitées que celles du paquet `fmt`, comme `fmt.Println` ou `fmt.Printf` :

- ✓ Comme précisé par la spécification, leur fonctionnalité n'est pas garantie entre implémentations du compilateur, au-delà des types les plus simples cités ci-dessus.
- ✓ L'implémentation dans le compilateur standard est limitée pour les types absents de cette liste, se limitant généralement à afficher des pointeurs sous forme hexadécimale pour les autres types de valeurs.
- ✓ La continuité de leur existence est explicitement non garantie dans les futures versions de Go.
- ✓ Elles n'ont pas d'équivalent pour les fonctions comme `fmt.Fprintf`, ou `fmt. Sprintf`, permettant d'écrire du code capable d'écrire ailleurs que sur la sortie standard, ce qui limite la testabilité du code qui viendrait à les utiliser.

Pour toutes ces raisons, il n'est pas recommandé de les utiliser dans le cas général.

Leur principal cas d'utilisation est celui des exemples du présent ouvrage : sur des fragments de code à usage temporaire, pour éviter la saisie d'une clause d'import, donc essentiellement dans des exemples et non dans du code réel.

8.2 CONVERTISSEURS COMPLEXES

Trois fonctions intégrées sont disponibles pour manipuler les nombres flottants complexes. Elles sont illustrées par le listing 8.2 :

- ✓ **`complex`** : accepte deux arguments flottants du même type réel, et renvoie la valeur complexe construite en les utilisant respectivement comme partie réelle et partie imaginaire.
- ✓ **`real`** : accepte un argument complexe et renvoie sa partie réelle.
- ✓ **`imag`** : accepte un argument complexe et renvoie sa partie imaginaire.

Ces pseudo-fonctions présentent une particularité qui empêchait de les réaliser comme de véritables fonctions : les types de leurs arguments et de leurs résultats sont variables, mais entièrement définis :

- ✓ `complex` renvoie une valeur de type **`complex64`** ou **`complex128`** selon le type de ses arguments.
- ✓ `real` et `imag` renvoient une valeur de type **`float32`** ou **`float64`** selon le type de son argument.

La syntaxe de Go ne permettant pas qu'une fonction renvoie un tel type variable, ces pseudo-fonctions sont documentées avec une signature utilisant des types génériques fictifs, respectivement `ComplexType` et `FloatType`.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 8.2 – Conversions entre valeurs complexes et réelles
https://osinet.fr/go/builtins/play-complexes

```
1  package main
2
3  import "fmt"
4
5  func p(f ...interface{}) {
6          for _, v := range f { fmt.Printf("%T %v\t", v, v) }
7          fmt.Println()
8  }
9
10 func main() {
11         var f31, f32 float32 = 1, 2
12         var f61, f62 float64 = 3, 4
13         var c64 = complex(f31, f32)
14         var c128 = complex(f61, f62)
15         p(c64)                          // complex64   (1+2i)
16         p(c128)                         // complex128  (3+4i)
17
18         r3, i3 := real(c64), imag(c64)
19         r6, i6 := real(c128), imag(c128)
20         p(r3, i3)                       // float32 1 float32 2
21         p(r6, i6)                       // float64 3 float64 4
22 }
```

8.3 ALLOCATION : `make` ET `new`

Outre l'initialisation par des littéraux, Go dispose de deux primitives pour créer des valeurs : les pseudo-fonctions `make` et `new`, illustrées par le listing 8.3.

✓ **`new`** : accepte en argument un type, défini ou anonyme. Elle alloue l'espace nécessaire au stockage d'une valeur de ce type par une variable, initialise cet espace, et renvoie un pointeur égal à sa première adresse en mémoire. Elle est utilisable avec tout type, y compris les types simples, et renvoie la valeur vierge correspondante.

✓ **`make`** : est dédiée à trois types particuliers : les canaux, cartes, et tranches. En effet, comme le montre l'exemple du listing 8.3, le résultat de `new` n'est pas toujours utile tel quel :

- pour un type interface comme **`error`**, le type dynamique de la valeur est **`nil`**, de peu d'utilité pratique puisque ne fournissant pas d'implémentations des méthodes du type interface statique, comme décrit au § 5.5.2 ;

Listing 8.3 – Allocation et initialisation par `new` et `make`
https://osinet.fr/go/builtins/play-new-make

```go
package main

import "fmt"

func p(p ...interface{}) {
        for _, v := range p { fmt.Printf("%#v ", v) }
        fmt.Println()
}

type C chan int
type M map[string]int
type S struct{ int }
type T []int

func main() {
        // New renvoie des pointeurs
        p(new(rune)) // (*int32)(0x414040) 0
        p(new(S))    // &main.S{int:0}

        // ...pas forcément prêts à l'emploi
        p(*new(error)) // <nil>
        p(*new(C))     // (main.C)(nil)
        p(*new(M))     // main.M(nil)
        p(*new(T))     // main.T(nil)

        // Make renvoie des valeurs utilisables.
        p(make(C))    // (main.C)(0x430080)
        p(make(M))    // main.M{}
        p(make(T, 3)) // main.T{0, 0, 0}
}
```

- pour un type canal, le canal pointé par le résultat de `new` est un canal **`nil`** qui n'est utilisable ni en émission ni en réception, et toujours bloquant, comme décrit au § 4.5.4 ;
- pour un type carte, la carte pointée par le résultat de `new` est une carte **`nil`**, dans laquelle il est impossible de stocker des résultats, comme décrit au § 4.3.4 ;
- pour un type tranche, la tranche pointée par le résultat de `new` est une tranche **`nil`**. Bien qu'elle soit valide, toute écriture nécessitera l'allocation d'un tableau sous-jacent, qu'il aurait été plus efficace d'allouer dès le départ à une taille appropriée, comme décrit au § 4.2.4.

© Dunod – Toute reproduction non autorisée est un délit.

`make` va donc un cran au-delà de `new` et complète l'allocation en procédant aux initialisations nécessaires pour renvoyer une valeur prête à l'emploi.

Elle peut utiliser pour cela des paramètres additionnels au-delà du type de donnée à créer, en nombre variable selon le type de donnée à créer. Ces paramètres sont décrits à propos des types correspondant aux paragraphes indiqués ci-dessus : taille de tampon pour les canaux, taille de fenêtre et de tableau pour les tranches, nombre initial approximatif [2] d'éléments pouvant être insérés avant réallocation pour les cartes.

Les arguments passés à ces paramètres additionels doivent être entiers et positifs, ou être des constantes non typées représentables par des entiers positifs.

Ces deux pseudo-fonctions présentent des particularités qui empêchent leur réalisation comme de véritables fonctions :

✓ leur premier argument est un type ;

✓ leurs arguments et résultats sont de types variables mais définis.

La syntaxe de Go ne permettant pas qu'une fonction accepte des types, ou renvoie des résultats de types variables, elles sont documentées avec une signature utilisant le type générique fictif `Type`.

8.4 TAILLE ET CAPACITÉ : `len`, `cap`

Une fois initialisées, les données de certains types peuvent avoir une taille variable et, pour certaines d'entre elles, une capacité de stockage variable.

Les fonctions intégrées `len` et `cap`, renvoient ces deux valeurs. Le tableau 8.1 précise la nature de ces résultats en fonction des types de données

Tableau 8.1 – Résultats de `len` et `cap` par type de données

Nature de type	`len`	`cap`	Unité
canaux	éléments stockés	taille du tampon	éléments
cartes	éléments stockés	non applicable	éléments
chaînes	nombre d'octets	non applicable	octets
tableaux et leurs pointeurs	allocation	allocation	éléments
tranches	éléments accessibles	capacité	éléments

Les détails de l'utilisation de ces fonctions sont précisés type par type dans la section dédiée à chaque type, aux chapitres 3 et 4.

2. L'algorithme utilisé par la structure de cartes ne faisant pas partie de la définition du langage, la taille effectivement allouée dépend de l'implémentation, de même que sa capacité intrinsèque : à titre d'exemple, la capacité directe d'une implémentation basée sur une table de hachage est supérieure ou égale au nombre d'entrées qui pourront y être stockées directement, en raison des éventuelles collisions d'index.

Deux points sont toutefois particulièrement notables, comme source d'erreurs courantes :

✓ La taille des chaînes est exprimée en nombre d'octets, de façon équivalente à un tableau d'octets : elle ne représente pas dans le cas général le nombre de caractères (runes).

✓ La capacité d'une tranche n'est pas la taille du tableau sous-jacent, comme le montre le listing 8.4 : la longueur du tableau est son maximum, mais sa valeur est définie conjointement par :

- l'adresse du pointeur de tableau de la tranche, qui n'est pas nécessairement l'adresse de base du tableau, comme le montre l'exemple aux lignes 23 et 26 ;
- la taille du tableau, qui fournit la limite haute ;
- la longueur de la fenêtre utilisée sur le tableau, comme le montre l'exemple à la ligne 26 : bien que le tableau soit de quatre éléments, la capacité de la dernière tranche est limitée par la valeur de sa longueur.

Cette valeur représente la longueur maximale que la tranche peut atteindre lors d'un tranchage.

Dans certains cas, comme les tableaux et les chaînes constantes, le résultat de ces deux pseudo-fonctions peut être une constante.

Listing 8.4 – Longueur et capacité des tranches
https://osinet.fr/go/builtins/play-len-cap

```
8  // Copie du type privé runtime.slice.
9  type slice struct {
10         array unsafe.Pointer
11         len   int
12         cap   int
13 }
14
15 type P = unsafe.Pointer
16
17 func main() {
18         tableau := [...]int{0, 1, 2, 3}
19         var tranche []int = tableau[:]
20         fmt.Println((*slice)(P(&tranche))) // &{0x414020 4 4}
21
22         tranche = tableau[0:2]
23         fmt.Println((*slice)(P(&tranche))) // &{0x414020 2 4}
24
25         tranche = tableau[0:2:3]
26         fmt.Println((*slice)(P(&tranche))) // &{0x414020 2 3}
27 }
```

© Dunod – Toute reproduction non autorisée est un délit.

8.5 MANIPULATION DES TRANCHES : append, copy

Les deux fonctions intégrées `copy` et `append` permettent respectivement :

- ✓ de copier une tranche vers une autre de façon optimisée et en tenant compte des éventuels recouvrements et limites de taille ;
- ✓ d'ajouter des éléments à une tranche existante.

Ces deux fonctions ne concernant que les tranches, leur description se trouve dans la section sur la manipulation des tranches, § 4.2.4, page 118 et suivantes.

8.6 DESTRUCTION D'ÉLÉMENTS DE CARTE : delete

La pseudo-fonction `delete` permet de supprimer d'une carte une entrée spécifiée par son index, sous la forme `delete(uneCarte, uneClef)`.

Cette pseudo-fonction ne concernant que les cartes, sa description détaillée se trouve dans la section sur leur manipulation, § 4.3.4, page 124.

Il n'existe pas de moyen intégré équivalent à `delete` pour supprimer des entrées de tableaux ou tranches.

Pour un effet équivalent, utiliser deux tranchages combinés autour de la position à supprimer, comme dans le listing 8.5 : l'utilisation d'une boucle peut être à la fois moins efficace et moins lisible.

Listing 8.5 – Équivalent de `delete` pour tranches et tableaux
https://osinet.fr/go/builtins/play-delete

```
38 func tranchage(tr []int, index int) []int {
39         if index >= len(tr) || index < 0 {
40                 return tr
41         }
42         return append(tr[:index], tr[index+1:]...)
43 }
```

8.7 FERMETURE DE CANAUX PAR close

La pseudo-fonction `close` permet de basculer un canal de l'état ouvert, qui est l'état initial des canaux non-`nil`, à l'état fermé, sous la forme `close(unCanal)`.

Cette pseudo-fonction ne concernant que les canaux, sa description détaillée se trouve dans la section sur leur manipulation, § 4.5.4, page 142.

8.8 TRAITEMENT DES ERREURS AVEC L'INTERFACE `error`

8.8.1 Définition

Le type **`error`** est le seul type interface fourni par le runtime Go, et ayant donc pour portée le bloc univers. C'est une interface unitaire, définie par la seule méthode `Error` :

```
type error interface { Error() string }
```

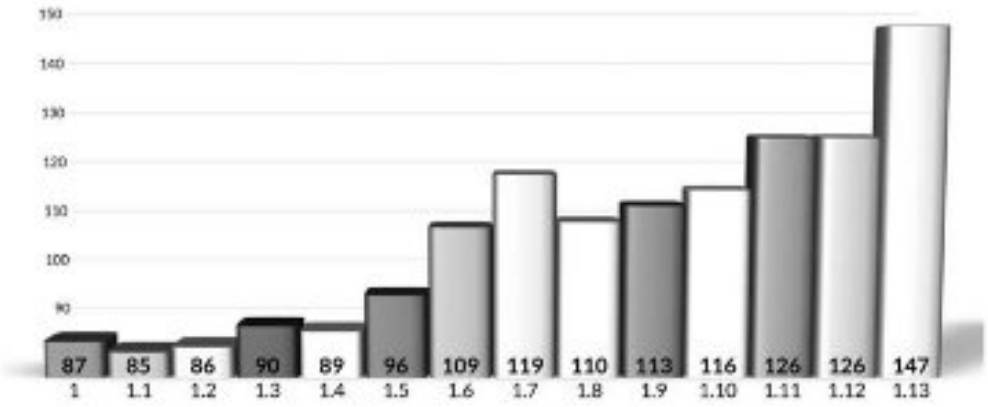
Par convention, la valeur **`nil`** de ce type représente l'absence d'erreur.

La simplicité de cette interface permet de l'implémenter dans autant de types concrets que nécessaire, afin de disposer, à la fois :

- ✓ d'une caractérisation unique des erreurs au-travers du type statique **`error`**,
- ✓ de valeurs d'erreur significatives capables de contenir autant d'information que nécessaire, du fait de la liberté de type dynamique.

La bibliothèque standard démontre largement son utilisation, avec un nombre d'implémentations qui a presque doublé de Go 1 à Go 1.13, comme illustré par la figure 8.1.

Figure 8.1 – Implémentations de l'interface **`error`** par version Go



8.8.2 Le modèle Go de gestion des erreurs

« *Errors are values. Don't just check errors, handle them gracefully.* »

Rob Pike - *Go Proverbs*
https://osinet.fr/go/builtins/yt-pike2015-errors

La gestion des erreurs est l'une des originalités de Go, en faisant le choix de ne pas implémenter un modèle à base d'exceptions comme dans la majorité des langages

© Dunod – Toute reproduction non autorisée est un délit.

usuels, mais de revenir à une approche où l'erreur, dans la majorité des cas, n'est pas une occurrence exceptionnelle, mais une valeur comme une autre, qu'une simple interface suffit à caractériser comme telle.

Dans l'immense majorité des cas, un traitement pouvant résulter en une erreur se présentera comme une affectation parallèle d'une expression multivaluée d'appel de fonction ou méthode :

```
résultat1, ..., résultatN, err := traitement(paramètres)
```

Les conventions concernant ces résultats sont les suivantes :

- ✓ Les valeurs sauf la dernière sont des types de résultats souhaités à l'issue du traitement.
- ✓ La dernière est une valeur de type **`error`**.
- ✓ Si la dernière est **`nil`**, le traitement a réussi et les autres valeurs sont prêtes à l'emploi.
- ✓ Si la dernière est différente de **`nil`**, aucune des autres valeurs n'est significative. Elle seront dans la plupart des cas la valeur vierge des types de résultats, mais ce n'est pas nécessairement le cas, et elle ne doivent pas être interprétées, y compris pour diagnostiquer l'erreur : ce rôle est dévolu à la valeur d'erreur.

En appliquant ces conventions, la gestion des erreurs se développe sur deux axes :

- ✓ Prioriser les cas d'erreurs. Le style idiomatique Go privilégie le traitement des situations d'erreur en premier lieu. Ce traitement permet de ressortir des fonctions dès leur début en présence d'erreurs, mécanisme dénommé « clauses de garde » (*guard clauses*).
- ✓ Utiliser des véritables valeurs d'erreur, comparables et porteuses de sens, typiquement sous forme de structures ; et leur apporter un traitement approprié utilisant ces valeurs au lieu de simplement les comparer à **`nil`**.

Une première approche naïve des clauses de garde consiste à les rédiger au plus simple, sans tirer parti des valeurs d'erreur, à la façon de :

```
if err != nil { return err }
```

Cette approche est généralement le résultat de la transposition d'habitudes acquises par la programmation de langages encourageant l'utilisation des exceptions, et de mécanismes de type `try/catch`, remplaçant l'interception de l'exception par la comparaison `err != nil`.

Une meilleure approche consiste à émettre des valeurs d'erreur représentant des conditions significatives, et à mettre en place des traitements pour ces situations, permettant, dans des fonctions utilisatrices, de ne pas récupérer les erreurs de bas niveau – qui remonteraient isolées et inexploitables par manque de contexte – dans

l'utilisation naïve, mais d'effectuer dans les niveaux intermédiaires les interceptions et traitements appropriés afin que les fonctions plus abstraites puissent remonter des résultats et des erreurs, ne concernant que leur niveau d'abstraction, et pas les niveaux inférieurs [3].

Pour cela, deux grandes approches :

- ✓ Comparer directement les valeurs d'erreur à des valeurs prédéfinies par le paquet émetteur, idéalement des constantes comme le type `Erreur` dans le listing 8.6.
- ✓ Pour réagir à des types d'erreur plutôt qu'à des valeurs d'erreur en particulier, utiliser une assertion de type au lieu d'une comparaison.

Lorsque l'erreur reçue n'est pas d'un type que la fonction en cours peut traiter localement, et seulement dans ce cas, renvoyer l'erreur en l'emballant dans des informations additionnelles permettant de préciser le contexte sans avoir besoin d'une pile d'exécution complète, souvent difficile à exploiter et extrêmement volumineuse. C'est ce qu'illustre la ligne 34 de l'exemple.

Les bonnes pratiques d'emballage (*wrapping*) sont décrites au § 8.8.4.

8.8.3 Création d'erreurs spécifiques

Outre le cas des erreurs de types simples se prêtant bien au mécanisme illustré par le listing 8.6, les mécanismes les plus utiles pour créer de telles erreurs sont :

- ✓ **`errors.New(text string) error`** : cette fonction renvoie une **`error`** dont le type dynamique est un type structure privée contenant un seul champ chaîne, qui contiendra l'argument `text` passé comme message d'erreur.
- ✓ **`fmt.Errorf(format string, a ...interface{}) error`** : est une fonction similaire à `fmt.Printf`, mais renvoyant une **`error`** au lieu d'une chaîne, dont le type dynamique sera soit la même structure privée, soit une autre structure privée emballant une erreur passée parmi ses paramètres et formatée avec la spécification de format `%w` [4].

Pour des résultats encore plus contextualisés, il est aussi possible de créer des types spécifiques, sur lesquels les fonctions utilisatrices pourront faire des assertions de type, ou utiliser les mécanismes de déballage décrits au § 8.8.4.

Il est d'usage que le nom des variables ou constantes d'erreur commence par `Err`, comme dans celles du listing 8.6.

3. Voir à ce sujet la présentation de Dave Cheney à Qcon China 2016, qui décrit en détail comment consommer les erreurs de couches inférieures et réduire ainsi le « bruit » visuel du traitement des erreurs : https://osinet.fr/go/builtins/cheney-errors
4. Ce format et la structure d'emballage sont apparues en Go 1.13. Dans les versions antérieures, ce format n'existait pas, et le résultat était toujours du même type que celui de `errors.New`.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 8.6 – Erreurs constantes
https://osinet.fr/go/builtins/play-erreurs-constantes

```
// Ce sont des chaînes: immuables et comparables.
type Erreur string

func (e Erreur) Error() string { return string(e) }

const (
        ErrPremière = Erreur("une")
        ErrAutre    = Erreur("autre")
)

// Abstraction d'un traitement renvoyant une erreur.
func traitement() (int, Erreur) {
        return 0, ErrPremière
}

func utilisateur() error {
        var err error          // Prédéfinir le type...
        _, err = traitement() // ...évite que ce soit Erreur.
        switch err {
        case nil: // nil est valide pour error, pas pour Erreur.
                fmt.Println("Pas d'erreur")
        case ErrPremière:
                fmt.Println("Erreur traitée 1")
                return nil
        case ErrAutre:
                fmt.Println("Erreur traitée 2")
                return nil
        default:
                // Inconnue: emballer et renvoyer.
                return fmt.Errorf("utilisateur: %w", err)
        }
        return nil
}
```

8.8.4 Emballage et déballage d'erreurs

Pour exploiter au mieux ces valeurs d'erreurs, l'approche décrite dans les rubriques précédentes amène généralement à définir à chaque niveau de traitement pouvant recevoir une valeur d'erreur une structure plus riche, qui contextualise l'erreur plus simple reçue du niveau de traitement inférieur, tout en conservant dans un champ la valeur d'erreur d'origine.

Ce mécanisme est dit *emballage* des erreurs (*error wrapping*), ou parfois *annotation* des erreurs.

Sa généralisation constatée par les grands projets Go à code ouvert au cours des dix premières années d'utilisation de Go a conduit diverses équipes à proposer des solutions sous forme de paquets contribués pour uniformiser ce mécanisme.

Face à l'identification de ce modèle général de traitement et à la divergence des pratiques dans les détails, le projet Go a élaboré dans le cadre de l'initiative Go 2 de nouvelles fonctionnalités, aboutissant à une standardisation en Go 1.13 sur un mécanisme d'emballage et, surtout, de déballage des erreurs (*error unwrapping*).

Le mécanisme introduit en Go 1.13 repose sur la standardisation d'une méthode `Unwrap() error`, qui sous-tend le fonctionnement de trois fonctions du paquet `errors` :

- ✓ **`errors.Unwrap()`** : lors de la création d'un type d'erreur personnalisé destiné à emballer une erreur reçue, si le type utilisé définit cette méthode, il sera reconnu comme implémentant une interface anonyme.
  La fonction `errors.Unwrap()` pourra lui être appliquée, et renverra l'erreur emballée à partir de l'erreur emballage.
  Sur une valeur **`error`** n'implémentant pas cette méthode, la fonction renvoie **`nil`**.
- ✓ **`errors.Is(err, target error) bool`** : prend en paramètre une valeur d'erreur potentiellement emballée, et une valeur d'erreur à laquelle la comparer.
  Elle itère sur `errors.Unwrap` jusqu'à **`nil`**, indiquant au retour si, à un quelconque niveau de déballage, une des erreurs emballées était celle du second argument : ceci permet de généraliser aux erreurs emballées la comparaison faite jusqu'alors sur les valeurs brutes, comme décrit au § 8.8.2 à la façon du listing 8.6.
  Dès lors que le code cible Go 1.13 ou plus, il est généralement recommandé de remplacer toute comparaison d'erreur utilisant une simple égalité `==` par l'utilisation de cette fonction.
- ✓ **`errors.As(err error, target interface{}) bool`** : fonctionne de façon similaire à `errors.Is`, mais vérifie qu'à un quelconque niveau de déballage le premier argument **`error`** peut être affecté au type de la valeur passée en second argument.
  En cas de succès, elle renvoie **`true`** et procède à cette affectation ; en cas d'échec elle renvoie **`false`** et ne procède pas à l'affectation.
  Dès lors que le code cible Go 1.13 ou plus, il est généralement recommandé de remplacer toute assertion de type d'erreur simple par l'utilisation de cette fonction afin de permettre l'examen à tous les niveaux de déballage.

La spécification de formatage `%w` a été ajoutée en Go 1.13 aux chaînes de format, modifiant le type de résultat comme décrit au § 8.8.3, de sorte que le résultat de `fmt.Errorf` implémente la méthode `Unwrap`, autorisant donc l'utilisation des erreurs ainsi créées avec les fonctions `errors.As` et `errors.Is`.

© Dunod – Toute reproduction non autorisée est un délit.

Évolution annoncée de la gestion des erreurs

Lors de l'adoption de ces nouveaux mécanismes, le projet Go a indiqué que d'autres fonctionnalités seraient entreprises à partir de Go 1.14, notamment pour ce qui est du formatage des erreurs, et de la possibilité d'ajout automatique d'autres informations comme la ligne source de l'emballage d'une erreur.

La décision, visible sur https://osinet.fr/go/builtins/go-formatage-erreurs laisse la porte ouverte à l'introduction future d'un affichage de la pile d'appels (*stack trace*) utilisant le type `errors.Frame`, utilisé un temps durant la phase d'expérimentation avant d'être retiré de la version 1.13.

Parallèlement, la Foire aux Questions sur les valeurs d'erreur, disponible sur https://osinet.fr/go/builtins/go-faq-erreurs, décrit les choix à opérer au regard des nouveaux mécanismes en présence de masses de code existantes.

De même, la possibilité d'un type formalisant l'interface anonyme composée de `Unwrap()`, discutée dans le même ticket 29934, a été écartée pour Go 1.13 faute de trancher sur son nom entre `Wrapper` et `Unwrapper`, en laissant ouverte la possibilité de l'ajouter dans une version ultérieure, si sa version existante dans `golang.org/x/xerrors`, le type `xerrors.Wrapper`, venait à se révéler à l'usage largement utilisée.

Go 1.14 n'inclut aucun de ces changements, et des propositions alternatives sont débattues dans le cadre de Go 2.

8.8.5 L'expérimentation `check` / `handle` / `try`

Durant la période de maturation qui a introduit les nouvelles fonctionnalités décrites ci-dessus, d'autres mécanismes plus intrusifs ont été expérimentés, en particulier :

- ✓ **`check` et `handle`** : introduit en 2018, `check` visait à éliminer la nécessité des vérifications `err == nil`, en revenant directement de la fonction qui le contenait en cas d'erreur, en passant par un traitement défini dans `handle`, similaire à **`defer`** mais dédiée aux résultats de `check`.
  Considéré comme alourdissant trop la syntaxe et quelque peu confus quant à la double fonctionnalité de `handle` et **`defer`**, ce mécanisme a été abandonné.
- ✓ **`try` et `defer`** : version 2019 de la réflexion sur le même thème, le nouveau mot-clef `try` fonctionnait de façon similaire, mais permettait de ne plus requérir un mot-clef supplémentaire, en unifiant les traitements avant retour dans **`defer`**.
  Les retours de la communauté des utilisateurs Go ayant été globalement négatifs, cette approche a été finalement abandonnée au profit des mécanismes décrits dans ces pages, considérés comme plus idiomatiques et plus propres à préserver la simplicité du code.

Une conférence de Russ Cox à Gophercon 2019, disponible à l'adresse https://osinet.fr/go/builtins/yt-cox2019-erreurs, décrit en détail le cheminement qui a mené de ces essais à l'API Go 1.13, couvrant à la fois ces mots-clefs abandonnés et l'utilisation pratique du mécanisme d'emballage / déballage des erreurs.

8.9 SITUATIONS DE PANIQUE : `panic` ET `recover`

8.9.1 Définition

Comme l'a détaillé la rubrique consacrée au type **`error`**, dans la plupart des cas, les résultats anormaux sont traités en Go par un résultat secondaire de type **`error`**, et non par l'émission d'une exception, cette notion n'existant pas en Go.

Toutefois, certaines situations d'erreur ne proviennent pas d'appels de fonction aux résultats contrôlés, comme une division entière par zéro, une expression d'indexation, ou tout bogue conduisant à une véritable anomalie d'exécution.

Pour ces cas, Go définit une situation de *panique* (*panic*), inspirée par la fonction éponyme des noyaux UNIX [5].

Cette situation peut être déclenchée par l'appel de la fonction intégrée `panic`, aux caractéristiques suivantes :

1. `panic` accepte un unique argument, de type quelconque, et ne renvoie aucun résultat ;
2. la panique interrompt l'exécution de la fonction en cours, qui reprend dans les fonctions invoquées par **`defer`** pour la fonction en cours ;
3. l'exécution repasse à la fonction appelante ;
4. l'exécution continue ainsi jusqu'à la fonction `main`, à la sortie de laquelle le programme se termine en affichant une trace d'erreur commençant par la valeur de l'argument passé à la fonction `panic` et se poursuivant par la pile d'appel des goroutines actives comme dans le listing 8.7, qui illustre la différence entre une trace d'appel traditionnel, comme aux lignes 4 à 7 du résultat, et une création de goroutine par **`go`**, identifiée par la mention `created by`, comme à la ligne 8 du résultat.

La remontée de la pile d'appels en propagation de panique décrite à l'étape 3 peut être évitée si l'une des fonctions différées exécutées durant la chaîne des retours exécute la fonction intégrée `recover`, comme dans le listing 8.8.

Cette dernière est une fonction sans paramètres, dont la signature spécifie un retour de type **`interface`**`{}`, et qui n'a de sens que dans un bloc **`defer`** :

- ✓ Si elle est invoquée pendant une situation de panique, à l'étape 2 décrite ci-dessus, elle interrompt cette situation, et l'exécution du bloc **`defer`** suit son cours normal, comme aux lignes 14 et 36, où le `recover` de la fonction `f1` récupère la panique déclenchée dans `f2`.
- ✓ Si elle est invoquée hors du contexte d'un bloc **`defer`** comme aux lignes 21 et 35, ou si aucune situation de panique n'est en cours durant un bloc **`defer`**, elle renvoie **`nil`**, comme aux lignes 26 et 38 de l'exemple.

5. https://osinet.fr/go/builtins/wiki-panic

© Dunod – Toute reproduction non autorisée est un délit.

Listing 8.7 – Situation de panique
https://osinet.fr/go/builtins/play-panic

```go
package main

import "time"

func sousPanique() { panic("Panique") }

func panique() { sousPanique() }

func main() {
        go panique()
        time.Sleep(time.Microsecond)
}
```

```
panic: Panique

goroutine 5 [running]:
main.sousPanique(...)
        /tmp/sandbox741896348/prog.go:5
main.panique()
        /tmp/sandbox741896348/prog.go:7 +0x40
created by main.main
        /tmp/sandbox741896348/prog.go:10 +0x40
```

Ceci ressemble à l'exécution réalisée – dans d'autres langages – dans la branche `catch` d'un traitement d'exception, à une différence majeure près : cette récupération ne peut avoir lieu que dans du code terminal, et non en ligne dans la séquence d'exécution principale d'une fonction.

8.9.2 Paniques d'exécution

La fonction `panic` est utilisé par le runtime Go de façon fonctionnellement identique à son usage dans le code utilisateur, pour les situations d'anomalie déjà évoquées.

En pareil cas, il invoque cette fonction en lui passant pour argument une valeur du type intégré `runtime.Error`, un type interface composé :

- ✓ d'une interface **`error`** intégrée ;
- ✓ d'une méthode additionnelle `RuntimeError()` documentée comme sans effet (*no-op*), dont le seul rôle est de permettre la différenciation de ces erreurs par rapport aux autres.

Listing 8.8 – Utilisation de `panic` et `recover`
https://osinet.fr/go/builtins/play-recover

```go
package main

import "fmt"

func f2() (n int) {
        n = 2
        panic("Err:2") // pas de return nécessaire
}

func f1() (n int) {
        n = 1
        defer func() {
                v := recover()
                fmt.Printf("Dans f1: %v %d\n", v, n)
        }()
        n = f2() // Affectation non exécutée.
        return
}

func main() {
        fmt.Println("Hors defer:", recover())
        defer func() {
                if v := recover(); v != nil {
                        fmt.Printf("Dans main: %v\n", v)
                } else {
                        fmt.Println("Main: R.A.S.")
                }
        }()
        n := f1()
        fmt.Printf("Résultat: %d\n", n)

}

// Résultat:
//   Hors defer <nil  : Recover hors defer dans main
//   Dans f1: Err:2 1 : Recover dans f1
//   Résultat: 1      : Exécution normale dans main
//   Main: R.A.S.     : Defer dans main
```

Ce type spécifique permet, dans un code de récupération, d'identifier les erreurs de type panique d'exécution, ou de les simuler, comme dans le listing 8.9.

© Dunod – Toute reproduction non autorisée est un délit.

Listing 8.9 – Paniques d'exécution réelles et simulées
https://osinet.fr/go/builtins/play-runtime-panic

```
package main

import (
        "errors"
        "fmt"
        "runtime"
)
func isRuntimePanic(v error) bool {
        var rt runtime.Error
        return errors.As(v, &rt)
}
func recouvrer() {
        v := recover().(error)
        fmt.Printf("%t %T\n", isRuntimePanic(v), v)
}
func pseudoRuntime() {
        defer recouvrer()
        var p runtime.Error = &runtime.TypeAssertionError{}
        panic(p)
}
func vraieRuntime() {
        defer recouvrer()
        i, j := 1, 0
        fmt.Println(i / j)
}
func ordinaire() {
        defer recouvrer()
        panic(errors.New("Ordinaire"))
}
func main() {
        ordinaire()      // false *errors.errorString
        vraieRuntime()   // true runtime.errorString
        pseudoRuntime() // true *runtime.TypeAssertionError
}
```

8.9.3 Utilisation appropriée

Comme le démontrent les exemples de cette partie, bien qu'il soit possible de créer des paniques à volonté avec `panic(uneValeur)`, à l'image d'un `throw new MonException()` Java ou PHP, toute la logique environnante est conçue pour ne pas utiliser ce modèle, et réserver les paniques aux situations vraiment exceptionnelles.

L'idée générale, souvent exprimée sous diverses formes dans la littérature Go, est que les paniques, à la différence de l'usage habituel des exceptions, ne sont pas des structures de contrôle de la logique d'exécution, mais sont réservées à des situations imprévues :

- ✓ un contenu de requête mal formé n'est pas exceptionnel, c'est plus probablement une attaque ;
- ✓ une connexion réseau qui échoue n'est pas exceptionnelle : aucun réseau n'a une disponibilité de 100 % ;
- ✓ une fonction effectuant une division qui tente une division par zéro en entiers est exceptionnelle : c'est probablement un bogue à corriger et, selon le rôle de la fonction, il peut être légitime de poursuivre l'exécution ;
- ✓ une application qui ne trouve pas sa configuration durant son initialisation est exceptionnelle, et il est légitime de ne pas tenter de poursuivre.

Ceci permet de cadrer l'utilisation appropriée des paniques :

- ✓ un paquet peut rencontrer des situations de paniques, notamment des paniques d'exécution ;
- ✓ il est de sa responsabilité d'intercepter ces paniques, afin d'exposer à ses consommateurs une API idiomatique renvoyant des résultats de types **`error`** structurés, sans paniquer : la limite de paquet est une frontière naturelle pour les paniques ;
- ✓ par extension, la limite entre le paquet `main` et le processus parent est aussi une frontière : ce paquet n'expose pas d'API, et il est légitime de ne pas forcer inutilement une tentative de continuation d'exécution dans certaines situations où des prérequis ne sont pas disponibles.

9 Organisation du code en paquets

9.1 PROGRAMMES ET PAQUETS

Inspiré pour l'organisation du code des langages de la famille Pascal / Modula / Oberon-2, Go organise son code en paquets [1] (*package*), avec un paquet au nom réservé `main` caractérisant les programmes exécutables par opposition aux bibliothèques.

Tout programme, comme illustré par les exemples des chapitres précédents, est construit à partir d'un paquet `main` – détaillé au § 9.5, qui utilisera généralement des données et services fournis par d'autres paquets qu'il importera, et qui en importeront eux-mêmes d'autres. Les cycles d'importation sont interdits et provoquent une erreur de compilation.

Le tableau 9.1 résume des similitudes de concepts dans les langages usuels, sans pour autant impliquer une stricte équivalence.

Tableau 9.1 – Équivalent des paquets Go dans d'autres langages

Langage	Notion	Notion Go comparable
ECMAScript 2015 etc	module (monofichier)	paquet (multifichier)
Java	package	fichier
	module Maven	paquet
PHP	namespace	paquet
	fichier multi-namespace	pas d'équivalent
Python	module (monofichier)	paquet (multifichier)
.NET	assembly	paquet

1. Équivalent des `MODULE` de Modula et, plus encore, d'Oberon-2.

Lors de la construction des programmes, chaque paquet est compilé, en commençant par ses dépendances, et c'est cette version compilée en un seul fichier qui est relue par les paquets qui en dépendent lors de leur propre compilation. Ce mécanisme permet d'éviter le recours à des *headers* comme en C/C++ avec les problèmes de lenteur et de risques de désynchronisation qui leur sont inhérents, l'importation de chaque paquet n'ouvrant que le fichier d'archive – avec l'extension `.a` – déjà créé.

L'organisation du code dans ces paquets est libre, mais quelques bonnes pratiques sont reconnues :

✓ Dans une arborescence de paquets, tout répertoire sauf `cmd`, `internal`, et `vendor`, est supposé contenir du code source Go, au lieu d'être seulement un conteneur de paquets.
  - `cmd` est conventionnellement utilisé comme parent pour les paquets correspondant aux commandes exécutables fournies par un projet. Il ne contient de code que si le projet n'inclut qu'une seule commande, et ne requiert donc pas de sous-répertoires.
  - `internal` est un répertoire réservé au code privé, détaillé au § 9.6.3. Ce n'est pas un paquet par lui-même : les paquets internes sont dans ses sous-répertoires.
  - `vendor` est un répertoire dédié aux paquets tiers copiés dans un projet pour garantir leur contenu lors de la construction, comme détaillé au § 9.9.3.3.

✓ Lors de l'organisation du projet, il est préférable de créer un nombre limité de paquets, en particulier au premier niveau et hors du paquet `internal` : multiplier les paquets amène à définir un grand nombre de types publics, ce qui crée une API étendue et peu profonde, donc contraignante à maintenir.

Go fournit plusieurs outils pour analyser le contenu des paquets compilés au format archive `*.a`, comme des exécutables. Les principaux sont décrits dans le tableau 9.2, et exploitent le format DWARF [2] utilisé par la chaîne de compilation Go.

Tableau 9.2 – Commandes d'inspection et désassemblage de paquets

Commande	Rôle	Traditionnel
`go tool nm`	Affichage des symboles	`nm`
`go tool pack`	Examen des fichiers d'une archive	`ar`
`go tool objdump`	Désassemblage	`objdump`

Pour l'aide intégrée de ces commandes, ajouter le drapeau `-h` à la commande, comme dans : `go tool nm -h`, suivant l'usage établi pour les programes Go.

2. DWARF est un format de représentations des informations de déboguage.
Présentation https://osinet.fr/go/paquets/wiki-dwarf
Site officiel https://osinet.fr/go/paquets/dwarf

9.2 LA CLAUSE package

9.2.1 Définition

Un paquet est constitué des fichiers portant la même clause de paquet, de la forme :

```
package unIdentificateur // chemin d'importation (facultatif)
```

L'identificateur est quelconque, sauf l'identificateur vierge. L'usage veut toutefois qu'il ne commence pas par une majuscule : ceci caractérise les identificateurs exportés, mais la notion d'export n'a pas de sens pour un paquet.

Si un chemin d'importation est précisé dans le commentaire facultatif, il va restreindre les chemins par lesquels le paquet pourra être importé, comme décrit au § 9.9.4.2.

Cette clause n'est pas une déclaration : en effet le nom du paquet n'apparaît dans aucune portée. Il sert uniquement d'étiquette commune pour :

- ✓ regrouper des fichiers dans un paquet ;
- ✓ définir le nom par défaut du paquet dans les déclarations d'importation de paquet

9.2.2 Clause paquets et compilation par fichier

Plusieurs fichiers dans un même répertoire peuvent porter des clauses `package` différentes, que ce soient des programmes exécutables ou des bibliothèques.

Ces fichiers peuvent importer d'autres paquets sans restriction particulière ; en revanche, ils ne peuvent être compilés qu'isolément :

- ✓ Pour tous, une compilation par `go build [options] leFichier.go` est possible, produisant un exécutable ou un greffon si la clause est `package main`, ou une bibliothèque dans les autres cas.
- ✓ Pour les fichiers avec une clause `package main` seulement, une compilation avec exécution directe est possible par `go run leFichier.go`.
  Dans ce cas, le programme est compilé dans un répertoire temporaire, d'où il est exécuté ; puis le programme et le répertoire temporaire sont supprimés en fin d'exécution.

Les répertoires contenant de tels fichiers avec des clauses paquet différentes ne sont pas utilisables comme des paquets, et sont typiques de code jetable et de fragments destinés à la démonstration. En effet, dans ce mode d'utilisation, le paquet construit lors de la compilation ne comporte que le fichier expressément désigné dans la commande de compilation et ses dépendances, et non les éventuels autres fichiers qui porteraient la même clause paquet.

Dans tous les autres cas, il est préférable d'utiliser des répertoires où tous les fichiers portent la même clause paquet, constituant des paquets valides.

© Dunod – Toute reproduction non autorisée est un délit.

9.3 RÉPERTOIRE D'UN PAQUET

Alors que, à une exception près, tous les fichiers d'un même répertoire doivent porter une même clause **`package`** pour constituer un paquet, la spécification Go n'impose pas que les fichiers d'un même paquet soient situés dans le même répertoire.

Cette contrainte existe en pratique :

- ✓ la spécification du langage autorise les implémentations Go à l'exiger ;
- ✓ la FAQ du langage indique que le stockage des fichiers dans un même répertoire est le moyen de créer un paquet multi-fichiers.

L'exception concerne les paquets de test : afin de simplifier à la fois la réalisation de tests en boîte blanche [3] (*white-box testing*) et en boîte noire [4] (*black-box testing*), il est possible de placer dans le répertoire d'un paquet `p` non seulement les fichiers de test du paquet `p` pour le test en boîte blanche, mais aussi ceux du paquet `p_test` pour le test en boîte noire.

Ces derniers n'appartenant pas au paquet testé n'ont accès qu'aux identificateurs exportés du paquet testé, ce qui caractérise le test en boîte noire ; à la différence des tests en boîte blanche situés dans le paquet testé lui-même, qui ont accès à l'ensemble des identificateurs du paquet testé.

Par simplicité, les tests unitaires gagnent à être réalisés en boîte blanche. En revanche les tests de type `Example` [5] gagnent à être réalisés en boîte noire : en effet, ce mode les contraint à qualifier les identificateurs du paquet testé, de sorte que leur contenu, une fois affiché dans l'outil de documentation `godoc`, sera directement utilisable par copier/coller par les programmes utilisant le paquet en question, qui sont nécessairement eux aussi externes à celui-ci.

Enfin, à l'exception du paquet `main`, conventionnellement nommé du nom de la commande à construire, il est d'usage que le nom du paquet corresponde au nom du répertoire le contenant, pour faciliter le traitement des chemins d'import (cf. § 9.9).

Les mécanismes liés aux noms des répertoires de paquets sont détaillés au § 9.9.

9.4 FICHIERS D'UN PAQUET

9.4.1 Répartition du code dans les fichiers

Un paquet peut comporter plusieurs fichiers, et le compilateur tire parti de l'organisation en paquets, compilant les paquets en parallèle et les fonctions et méthodes d'un paquet en parallèle, dans les limites imposées par les dépendances entre paquets.

3. https://osinet.fr/go/paquets/wiki-boite-blanche
4. https://osinet.fr/go/paquets/wiki-boite-noire
5. https://osinet.fr/go/paquets/go-tests-exemples

Du fait de cette compilation parallélisée, modifier l'organisation du code en fichiers multiples n'a pas d'incidence significative sur les temps de compilation.

Cette liberté d'organisation est accompagnée de bonnes pratiques identifiées par l'usage :

- ✓ À l'inverse d'autres standards de codage, notamment les recommandations PSR de PHP, qui encouragent la création d'un fichier par type (classe, interface, trait) dans un même namespace, Go encourage de commencer un paquet par un seul fichier.
  Il est d'usage que ce premier fichier porte le nom du paquet.
- ✓ Lorsque ce fichier grossit au fil de son évolution, l'usage est de le décomposer par préoccupation, en conservant dans un même fichier tous les symboles traitant cet aspect, alors que dans d'autres langages la pratique aurait recommandé de placer le code traitant chaque préoccupation dans un sous-paquet.
  Ainsi, dans `compress/gzip`, un seul fichier `gunzip.go` contient tous les types, variables, et fonctions relatifs à la décompression ; pendant que le fichier `gzip.go` contient l'équivalent relatif à la compression.
- ✓ Si même après un tel découpage, un fichier devient trop gros même pour une seule préoccupation, une règle pour choisir comment le répartir dans plusieurs fichiers sera de déplacer les symboles utilisant les mêmes imports dans des fichiers séparés.
  Si par la suite une décomposition différente en paquets multiples se révèle nécessaire, il est probable qu'elle soit facilitée par cette séparation par imports différents.
- ✓ Inversement, si un paquet en vient à contenir un grand nombre de fichiers de petite taille, par exemple lors d'une répartition en paquets multiples, il peut être souhaitable de regrouper le contenu de ces petits fichiers en un nombre plus limité de fichiers plus longs.
  Pour choisir quels fichiers regrouper, il est pratique de s'appuyer sur les la liste des paquets importés par chaque fichier, en plaçant dans le même fichier fusionné le contenu des fichiers qui importaient les mêmes paquets.

9.4.2 Conventions de nommage

Les noms de paquets sont libres, mais des conventions existent :

- ✓ Les noms de paquets sont au singulier, et intégralement en minuscules, tant dans la clause paquet que dans les identificateurs qui les déclarent dans les déclarations d'importation. Ils ne comportent pas de caractères `_`.
- ✓ Ces noms étant appelés à être largement utilisés, il est souhaitable qu'ils soient à la fois courts et représentatifs, évitant les noms vagues comme `common`, `lib`, `tool`, ou `util`.

© Dunod – Toute reproduction non autorisée est un délit.

✓ Si un paquet doit néanmoins comporter une collection d'utilitaires, compléter le nom générique par la nature des utilitaires, et conserver le nom au singulier, comme `io/ioutil`, `http/httputil`, ou `crypto/internal/randutil`.
Cette répétition est une exception à la règle de nommage Go recommandant d'éviter le « bégaiement » (*stuttering*) dans les identificateurs.

Les noms de fichiers dans un paquet sont libres, mais là aussi des conventions existent :

✓ le nom du premier fichier est le nom du répertoire contenant le paquet ;

✓ pour tout paquet autre que `main`, le nom de ce fichier est celui du paquet.

Lorsqu'un paquet comporte plus d'un fichier, il est recommandé que les noms de fichier soient des noms – au sens large – plutôt que des verbes ou d'autres mots, mais cette règle n'est pas toujours respectée. Ainsi :

✓ Dans `encoding/csv`, on trouve bien `reader.go` et `writer.go` plutôt que `read.go` ou `write.go`.

✓ Le paquet voisin `encoding/xml` comporte bien un fichier `xml.go`, mais aussi deux fichiers nommés par des verbes : `marshal.go` et `read.go`.

Lorsqu'un paquet comporte une documentation rédigée, en plus de la documentation individuelle de chaque identificateur exporté, celle-ci est placée dans un fichier dédié nommé `doc.go`.

Listing 9.1 – Documentation de paquet : extrait de `fmt/doc.go`
https://osinet.fr/go/paquets/go-fmt

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6     Package fmt implements formatted I/O with functions (...)
7  */
8 package fmt
```

Comme dans le listing 9.1, qui résume la structure du fichier `doc.go` du paquet standard `fmt`, un fichier `doc.go` ne comporte, en plus du commentaire de fichier relatif aux informations de propriété intellectuelle qu'une clause paquet en *dernière* ligne, précédée d'un commentaire unique de documentation de paquet.

9.5 PROGRAMMES EXÉCUTABLES ET GREFFONS : LE PAQUET `main`

Parmi tous les identificateurs possibles pour la clause paquet, un seul d'entre eux est réservé : `main`.

Lorsqu'un paquet porte cet identificateur :

- ✓ Il ne peut être importé par un autre paquet Go.
- ✓ Il doit contenir, dans l'un quelconque de ses fichiers, une fonction `main()` dont la signature ne comporte ni paramètres ni résultats.
- ✓ Lors de sa compilation, au lieu de produire un paquet importable au format archive `*.a` (cf. § 9.1), il produit par défaut un programme exécutable.
- ✓ Si la compilation utilise l'option `-buildmode=plugin`, elle produit en revanche un greffon (*plugin*) utilisable par chargement dynamique sur Linux et macOS [6].

Lorsqu'un paquet `main` est utilisé pour construire un exécutable autonome, ce qui est le cas par défaut, ses identificateurs de portée paquet ne pourront pas être importés. Ils n'ont donc pas de raison d'être exportés. L'exportation peut en revanche être nécessaire pour construire un greffon ou être exporté par CGO.

Lors de l'invocation du programme par le système d'exploitation, par exemple depuis la ligne de commande, une fois l'initialisation effectuée (cf. § 9.8), l'exécution est transférée à la fonction `main` dans une unique goroutine.

Au retour de cette fonction, ou à l'invocation de la fonction `os.Exit`, l'exécution se termine, même si d'autres goroutines sont encore actives.

En l'absence de résultats dans la signature de la fonction `main`, il n'est pas possible de définir le code de sortie du processus par une valeur de retour comme c'est le cas en C et C++ :

- ✓ Il vaut 0 par défaut.
- ✓ Il vaut 1 en cas d'anomalie, comme une situation de panique (cf. § 8.9) non récupérée, ou une interruption par un signal POSIX.
- ✓ Il peut être défini manuellement par l'argument passé à `os.Exit` [7], qui doit être compris entre 0 et 125 pour maximiser la portabilité. Cette fonction présente l'inconvénient de mettre fin immédiatement au programme, sans même exécuter les fonctions différées par **`defer`**.

Terminer l'exécution de la goroutine dans laquelle s'exécute la fonction `main` par l'invocation de la fonction `runtime.Goexit()` plutôt que `os.Exit` permet d'exécuter le code différé, mais ne permet pas au postlude du programme d'être exécuté, et aboutit à une panique.

Si la fin d'exécution de `main` doit déclencher du code différé, il est pratique et lisible de combiner deux **`defer`** pour obtenir une séquence de fin d'exécution qui permette à la fois un nettoyage ordonné et un code de sortie choisi, comme dans le listing 9.2.

Dans cet exemple, le code différé est exécuté normalement, contrôlant la fin de la goroutine secondaire représentant un traitement fictif, puis le premier **`defer`** déclaré dans `main` est exécuté en dernier et permet de définir le code de sortie du processus.

6. https://osinet.fr/go/paquets/go-plugin (cf. *Plus en ligne*).
7. https://osinet.fr/go/paquets/go-exit

© Dunod – Toute reproduction non autorisée est un délit.

Listing 9.2 – Fin d'exécution ordonnée avec et `os.Exit`
https://osinet.fr/go/paquets/play-exit

```
package main

import (
        "os"
        "time"
)

func traitementFictif(fin chan<- bool) {
        for i := 1; i <= 100; i++ {
                print(i, " ")
                time.Sleep(time.Nanosecond)
        }
        fin <- true
}

func main() {
        defer func() { // Premier defer, exécuté en dernier.
                println("\nSortie avec 2")
                os.Exit(2)
        }()

        fin := make(chan bool)
        defer func() { // Second defer, exécuté en premier.
                println("\nSéquence de fin")
                <-fin
        }()

        go traitementFictif(fin)
        println("Activité dans main")
        time.Sleep(100*time.Microsecond)
}

// Résultat:
// Activité dans main
// 1 2 3 4 5 ...
// Séquence de fin
// ...996 997 998 999 1000 (vide sur le playground)
// Sortie avec 2
// --> Le code de retour du programme est bien 2.
```

Attention ! Pour exécuter cet exemple, construire le programme par `go build` puis l'exécuter : le code de retour d'une exécution par `go run` n'est pas celui du programme.

9.6 ORGANISATION DES PAQUETS

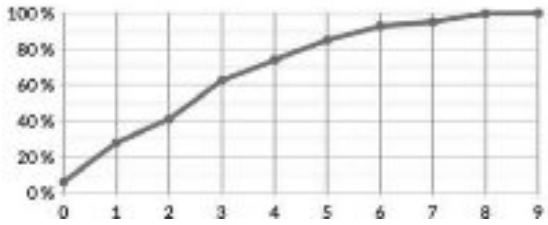
9.6.1 Hiérarchisation des paquets

À la différence des conventions d'autres langages, l'organisation en répertoires et sous-répertoires n'a pas de signification hiérarchique pour les paquets, même si elle doit avoir une signification sémantique pour faciliter la compréhension du code : il n'existe pas de sous-paquets, simplement des paquets situés dans des sous-répertoires.

Ainsi, le paquet `container/heap` n'est pas un sous-paquet de `container` : importer `container` ne donne pas accès aux identificateurs exportés de `container/heap`, fût-ce au prix d'une qualification.

Plus généralement, en raison de l'absence d'héritage dans le système de types et de l'absence de notion de sous-paquet, il est préférable de ne pas répartir le code des projets dans des hiérarchies de code profondes par défaut. À titre d'exemple, dans le projet Go lui-même, avec près de deux millions de lignes source, plus de 80% des paquets ont un chemin d'une profondeur inférieure ou égale à 5, comme illustré par la figure 9.1.

Figure 9.1 – Profondeur des chemins de paquets en Go 1.14



Un symptôme courant d'un découpage trop marqué du code est la présence de répertoires autres que `internal` ne comportant que des sous-répertoires de paquets et pas de code : comme indiqué plus haut, chaque répertoire de paquet doit contenir du code en propre.

Une éventuelle exception peut être une bibliothèque thématique contenant un ensemble d'implémentations indépendantes :

- ✓ Le paquet `container` de la bibliothèque standard est dans cette situation, n'existant que comme conteneur des diverses implémentations : tas (*heap*), liste doublement chaînée (*list*), et liste circulaire (*ring*).
- ✓ Mais même dans ce cas, il existe généralement du code dans le paquet de plus haut niveau, contenant au moins les interfaces communes utilisées par les implémentations définies dans les paquets de niveau inférieur. C'est par exemple le cas du paquet `encoding`, qui définit des interfaces communes à `encoding/gob`, `encoding/json`, et `encoding/xml`.

© Dunod – Toute reproduction non autorisée est un délit.

9.6.2 Organisation des paquets d'un projet

Dès lors qu'un projet se compose de plus d'un paquet, il est préférable de minimiser le contenu du paquet `main`.

Dans cette approche recommandée, le rôle du paquet `main` se limite à :

- ✓ Analyser la ligne de commandes pour identifier les drapeaux, commandes ou sous-commandes, et arguments.
- ✓ Initialiser les services d'usage continu, comme la journalisation et les Connexions aux bases de données, caches, et files d'attente.
- ✓ Charger la configuration depuis l'environnement et/ou des fichiers de configuration.
- ✓ Transférer l'exécution à un objet de haut niveau d'abstraction en lui injectant ces dépendances de services et paramètres, et situé dans un autre paquet.

Ce mécanisme permet de maximiser la testabilité du code :

- ✓ L'objet extérieur recevant ces dépendances injectées est plus facile à tester que si sa logique était liée aux spécificités de la fonction `main.main` et ses interactions avec l'environnement et la séquence d'initialisation des programmes.
- ✓ Le code de `main`, avec ces interactions plus compliquées à tester, se trouve être d'une complexité intrinsèque limitée.

Il permet aussi de réutiliser la même logique pour plusieurs interfaces utilisateur, en ayant des `main` multiples placées dans des répertoires de commandes distincts, placés chacun dans un paquet `<projet>/cmd/<commande>` différent.

9.6.3 Réduction de la surface d'API avec les paquets `internal`

Il est fréquent qu'une donnée utilisée dans l'implémentation d'un paquet provienne d'un paquet présent dans l'un de ses sous-répertoires, et doive être portée par un identificateur exporté pour y être utilisée.

Une conséquence de ces exportations est que les identificateurs concernés deviennent alors globalement accessibles par une importation directe de ces paquets pourtant destinés à l'usage interne, augmentant d'autant la surface effective d'API exposée par l'ajout de ces symboles, bien qu'ils ne soient pas destinés à un usage externe.

Jusqu'en Go 1.4, la seule solution consistait à ne pas utiliser un tel découpage, et placer tous les symboles dans un même paquet, ou à accepter cette expansion de la surface d'API.

Go 1.4 a introduit la notion de paquets *internes* (*internal packages*), caractérisés par le fait d'être placés dans des sous-répertoires d'un répertoire dénommé `internal` [8], qui ne peut pas lui-même comporter de code.

Tout identificateur exporté présent dans un paquet situé dans un chemin situé sous un tel répertoire ne peut être importé que par des paquets situés sous la même racine que le répertoire `internal` concerné, comme illustré par le tableau 9.3.

Le format des déclarations d'import n'est pas affecté, incluant `internal` sans aucune particularité.

Tableau 9.3 – Accès aux symboles exportés des paquets internes

Chemin de paquet	Accès aux exports de `c`
`a/`	Non
`a/b/`	Oui : parent direct d'`internal`
`a/b/internal/`	Non, `internal` n'est pas un paquet
`a/b/internal/c`	Oui : sous le parent direct d'`internal`
`a/b/internal/c/d`	Oui : sous le parent direct d'`internal`
`a/e/`	Non
`a/e/f`	Non
`g/`	Non

Ce mécanisme ne comporte aucun mécanisme comme le `friend` C++, qui permettrait de lister d'autres paquets comme autorisés.

Introduit dans Go 1.4 uniquement pour le code du projet Go sous `GOROOT`, il a été généralisé à l'ensemble des paquets à partir de Go 1.5, et est depuis lors recommandé à tout projet recourant à un tel découpage en paquets situés dans des sous-répertoires, pour éviter les exports non souhaités, et réduire d'autant la surface d'API effective correspondante.

9.7 DÉCLARATION D'IMPORTATION `import`

Une *déclaration* d'importation indique que le paquet en cours dépend d'un autre paquet dit *importé*. Cette déclaration ne peut avoir pour effet d'introduire un cycle qui rendrait un paquet dépendant de lui-même.

© Dunod – Toute reproduction non autorisée est un délit.

8. https://osinet.fr/go/paquets/go-internal

Elle a trois effets sur le programme :

- ✓ optionnel : définir un identificateur local pour le paquet importé ;
- ✓ exécuter le code d'initialisation du paquet importé ;
- ✓ ouvrir l'accès aux identificateurs exportés (cf. § 2.3.2) du paquet importé, directement ou par des expressions de qualification (cf. § 6.3) de ces identificateurs.

Tout identificateur de paquet introduit par une telle déclaration a l'obligation d'être utilisé, sous peine d'une erreur de compilation.

Sa notation pour une déclaration unique est de l'une des trois formes, dont l'une avec deux cas, illustrés par le tableau 9.4 :

- ✓ `import "unpaquet"`
  importe le paquet `unpaquet` et lui associe l'identificateur `unpaquet`, qui devra être utilisé dans au moins une expression de qualification d'identificateur pour accéder à un identificateur exporté du paquet.
- ✓ `import unIdentificateur "unpaquet"`
  importe le paquet `unpaquet` et lui associe l'identificateur `unIdentificateur`.
  - Si l'identificateur défini par cette déclaration n'est pas l'identificateur vierge, il devra être utilisé dans au moins une expression de qualification d'identificateur pour accéder à un identificateur exporté du paquet.
  - En revanche, s'il est l'identificateur vierge, il n'est pas possible de l'utiliser pour qualifier un identificateur, et le paquet n'est importé que pour les effets de bord de son initialisation (cf. § 9.8).
- ✓ `import . "unpaquet"`
  importe le paquet `unpaquet` et ajoute tous ses identificateurs exportés au paquet en cours, sans qualification.
  Le code du paquet en cours devra utiliser au moins l'un de ceux-ci pour éviter une erreur de compilation.

Tableau 9.4 – Exemple des quatre cas de déclarations d'import

Type d'import	Exemple	Accès aux imports
Par défaut	`import "foo/bar"`	`bar.Id`
Aliasé	`import quux "foo/bar"`	`quux.Id`
	`import _ "foo/bar"`	*impossible*
Local	`import . "foo/bar"`	`Id`

Comme les autres types de déclaration, il est possible de les regrouper avec des parenthèses, comme illustré par la plupart des listings de cet ouvrage.

Bien que ce soit hors des normes de codage Go, il est possible dans une déclaration multiple de paquets de regrouper des déclarations sur une seule ligne, en séparant les paquets à importer par des points-virgules, comme dans le listing 9.3.

Listing 9.3 – Importation de paquets multiples sur une seule ligne
https://osinet.fr/go/paquets/play-import-multiple

```
1  package main
2
3  import ( "fmt"; . "math"; _ "github.com/go-sql-driver/mysql" )
4
5
6  func main() {
7          fmt.Println(Cos(Pi))
8  }
```

Cet exemple montre les trois formats groupés sur une seule ligne :

- ✓ le paquet `fmt` est importé par défaut : l'identificateur qui lui est associé est le nom du dernier (et seul) répertoire dans le chemin de paquet indiqué, donc `fmt`, comme le montre l'usage de l'identificateur qualifié `fmt.Println` ;
- ✓ le paquet `math` est importé sans identificateur : ses identificateurs exportés sont disponibles dans le paquet en cours et il n'est ni nécessaire ni possible de les qualifier, comme le montre l'usage de la fonction `math.Cos` et de la constante `math.Pi` sans les qualifier ;
- ✓ le paquet `github.com/go-sql-driver/mysql` est importé avec l'identificateur vierge : il n'est donc pas possible d'utiliser ses identificateurs exportés, et il n'est importé que pour les effets de bord de son initialisation.
  Comme sa déclaration d'importation n'introduit aucun identificateur, le fait qu'il ne soit pas utilisé dans le paquet en cours ne provoque pas d'erreur.

La spécification du chemin d'importation (`unpaquet` dans ces exemples) est dépendante de l'implémentation, et donc théoriquement presque entièrement libre : la spécification Go précise que toutefois que :

- ✓ elle est généralement une sous-chaîne du chemin complet d'accès au paquet compilé ;
- ✓ elle peut être relative à un dépôt de paquets déjà installés ;
- ✓ les diverses implémentations de Go sont autorisées à :
  - limiter les chemins à des chaînes non vides composées de caractères des catégories Unicode L (lettre), M (marque), N (nombre), P (ponctuation) et S (symbole) [9],
  - exclure de cette liste la rune de remplacement Unicode `U+FFFD`, les espaces, et les caractères usuels de l'informatique, spécifiquement :
    `! " # $ % & ' ( ) * , : ; < = > ? [ \ ] ^ ` { | }`

9. Référence : https://osinet.fr/go/paquets/unicode-categorie

© Dunod – Toute reproduction non autorisée est un délit.

Dans la pratique, les implémentations existantes s'alignent sur celle du projet Go lui-même, avec quatre formes de chemin d'import :

✓ Un chemin simple, comme `database/sql`, principalement utilisé pour les paquets fournis par la bibliothèque standard Go.

✓ Un chemin plus complet comme `github.com/go-sql-driver/mysql`, correspondant à des imports de paquets distants.
Ces deux premiers usages sont examinés en détail au § 9.9.

✓ Un chemin relatif comme `./util`.
Ce format permet, lors d'essais rapides sur un seul fichier principal compilé avec `go run unfichier.go`, d'utiliser des paquets situés dans des chemins relatifs au fichier en cours de compilation.
- Il n'est pas autorisé si les fichiers figurent dans un des répertoire de recherche d'import, que ce soit par le système de modules ou par le mécanisme historique basé sur `GOPATH` (cf. § 9.10).
- Il n'est pas utilisable pour la construction de programmes multi-fichiers par `go build`, ni l'installation par `go install`.
- Son retrait fait l'objet d'un débat pour l'évolution Go 2.

Compte tenu de ces restrictions, il est préférable de ne pas y recourir.

✓ Un chemin réservé « `C` », qui importe un pseudo-paquet désignant le paquet en cours comme requérant des dépendances en langage C, et modifie la construction du programme pour utiliser CGO [10] au lieu du compilateur par défaut.
Dans ce cas, les commentaires de la forme `#include "<stdio.h>"` précédant la déclaration `import "C"` auront pour effet d'exposer les symboles définis dans les fichiers d'en-tête C ainsi désignés, en les rendant accessibles comme identificateurs qualifiés par l'identificateur de pseudo-paquet `C`, comme dans le listing 9.4.

L'intégration C avec CGO mentionnée à propos de l'import `C` est un usage avancé de Go, hors du périmètre de cet ouvrage. Pour citer une fois encore un proverbe Go : « *CGO is not Go.* »

9.8 INITIALISATION

Du fait de la possibilité d'existence de variables dans la portée paquet, celles-ci nécessitent une initialisation.

Sans rentrer dans les détails, qui concernent les auteurs de compilateurs, les règles définissant l'ordre de ces initialisation sont les suivantes :

✓ **Entre paquets** :
- L'initialisation des paquets même indépendants est séquentielle, un seul paquet à la fois, dans une seule goroutine.

10. https://osinet.fr/go/paquets/go-cgo

Listing 9.4 – Importation de code en langage C avec CGO
Non utilisable sur https://play.golang.org

```
package main

/*
#include <stdlib.h>
*/
import "C"

import "time"

func main() {
        C.srand(C.uint(time.Now().Nanosecond()))
        println(C.rand())
}
```

- L'initialisation des paquets importés est réalisée avant celle des paquets importateurs.
- Lorsqu'un paquet est importé à plusieurs reprises dans le graphe d'importation d'un programme, son initialisation n'est réalisée qu'une seule fois.

✓ **Dans un même paquet** :

- L'ordre dans lequel les fichiers d'un même paquet sont soumis à la compilation ne fait pas partie de la spécification, mais celle-ci *recommande* que cet ordre soit reproductible, et soit basé sur l'ordre lexical des noms des fichiers.
- Les variables sont d'abord allouées avec leur valeur vierge.
- Le prélude du programme exécute dans une seule goroutine une boucle initialisant toutes les variables non encore initialisées mais prêtes à l'être, en fonction de leurs dépendances, qui peuvent être d'autres variables, ou des fonctions ou méthodes dépendant de celles-ci par utilisation dans leurs instructions, de même que des expressions et valeurs de méthodes.
  Toute boucle de dépendances d'initialisation est interdite, provoquant une erreur de compilation, de sorte qu'à l'issue d'un nombre fini d'itérations toutes les variables sont garanties initialisées.
  À chaque itération, les variables prêtes à être initialisées le sont dans l'ordre de leur apparition dans le source du fichier qui contient leur définition.
- Une fois cette étape terminée, toutes les fonctions `init()` présentes dans les divers fichiers du paquet sont exécutées, dans l'ordre de présentation des fichiers à la compilation et, au sein d'un même fichier, dans leur ordre d'apparition dans ce fichier.
  Bien qu'elles puissent elles-mêmes créer d'autres goroutines, l'exécution des fonctions `init` est séquentielle, et est lancée dans une seule goroutine.

© Dunod – Toute reproduction non autorisée est un délit.

L'identificateur `init` des fonctions `init` n'est pas déclaré. À la différence des fonctions dont l'identificateur est déclaré, y compris `main` :

- ✓ il peut en exister plusieurs dans un même paquet, et même dans un même fichier ;
- ✓ leur identificateur n'étant pas déclaré, elles ne peuvent pas faire l'objet d'un appel de fonction direct : le runtime Go procède à une manipulation de pointeurs pour les invoquer, dans la boucle finale de la fonction `runtime.doInit`.

Leur seule raison d'être est de permettre d'initialiser des variables de portée paquet dont la valeur ne pourrait pas être représentée par un littéral.

Bien qu'il soit possible de les employer à d'autres fins sans restrictions techniques, c'est une pratique déconseillée : ne pouvant être invoquées manuellement, ces fonctions ne peuvent être testées, et affectent la séquence de démarrage globale du programme.

Lorsque l'ordre des dépendances ne peut pas être établi par le compilateur pour une initialisation, typiquement en présence de types interfaces, l'ordre de résolution des dépendances n'est pas spécifié, chaque implémentation étant libre de sa stratégie.

Compte tenu de cette limitation, et du fait que la spécification ne fasse que *recommander* que l'ordre de présentation des fichiers soit basé sur l'ordre lexical de leurs noms, il est préférable d'éviter toute construction de code dépendant de cet ordre d'évaluation.

9.9 CHEMINS D'IMPORTATION

Comme indiqué à propos des déclarations d'import, l'interprétation du chemin d'import dans les spécifications de paquet est du ressort de chaque implémentation, mais les implémentations usuelles traitent ceux-ci comme des chemins dans un système de fichiers.

En faisant abstraction du cas du chemin réservé « `C` » destiné à CGO et des chemins relatifs à l'usage déconseillé, ces chemins sont donc en pratique interprétés comme des chemins relatifs à diverses racines, détaillées ci-après.

9.9.1 Notion de *workspace* et GOPATH

Depuis son origine, Go emploie une notion d'espace de travail ou *workspace* pour organiser le code.

Un tel espace est un répertoire comportant au moins deux sous-répertoires :

- ✓ **`src`** : contient des sources Go ;
- ✓ **`bin`** : contient les exécutables résultant de l'installation des programmes Go compilés sur le système en cours ;

✓ **pkg** : contient les paquets de bibliothèques compilés installés, ainsi que les versions locales des dépendances utilisés par un projet utilisant les modules VGO (cf. § 9.10).
Il faisait partie des workspaces jusqu'à Go 1.10, est devenu optionnel en Go 1.11 à la suite du changement de fonctionnement de la chaîne de compilation standard, et a toujours été considéré comme un artefact des débuts du projet [11], appelé à disparaître dans une version ultérieure.

Des origines de Go jusqu'à la généralisation des modules VGO (cf. § 9.10) avec Go 1.13, puis leur utilisation recommandée en priorité à partir de Go 1.14, l'organisation recommandée [12] a consisté a placer l'ensemble de son code dans un unique espace de travail. Cet espace de travail est situé par défaut dans le sous-répertoire `go` du répertoire personnel de l'utilisateur sur son poste de travail.

Cette technique s'est rapidement révélée difficile à généraliser à tous les types d'environnements de travail, la majorité des autres langages ayant au contraire adopté un modèle utilisant un espace de travail par projet, et couplant celui-ci avec le système de gestion de versions.

Pour soutenir ces deux modes (espace unique ou espace par projet), il est possible de définir autant d'espaces de travail que souhaité, au prix de la reproduction dans chaque projet de la structure d'espace de travail avec ses répertoires `src` et `bin`, voire `pkg`.

Pour résoudre les chemins dans les déclarations d'importation de paquets, Go utilise une variable d'environnement `GOPATH`, qui est composée de la liste [13] des espaces de travail dans lesquels rechercher les paquets.

Son format dépend du système d'exploitation utilisé, en particulier :

✓ **Windows** : chaîne composée de répertoires séparés par des points-virgules, à l'image du `PATH` Windows.

✓ **UNIX, Linux, macOS et autres systèmes POSIX** : chaîne composée de répertoires séparés par des deux-points, à l'image du `PATH` UNIX.

Lors des téléchargements de paquets distants par la commande `go get`, les sources téléchargés sont placés dans des sous-répertoires du sous-répertoire `src` du premier espace de travail du `GOPATH`.

Lors de la résolution des imports, le compilateur recherche dans le sous-répertoire `src` de chacun des composants du `GOPATH` un chemin correspondant au chemin d'importation spécifié, dans l'ordre d'apparitition des espaces de travail dans cette

11. https://osinet.fr/go/paquets/gh-go-4719
12. https://osinet.fr/go/paquets/go-workspaces
13. C'est une erreur courante que de supposer que `GOPATH` ne contient que le chemin d'un unique espace de travail : ceci n'est pas vrai dans le cas général.

© Dunod – Toute reproduction non autorisée est un délit.

variable : ceci permet de ne pas avoir à répliquer le téléchargement des paquets dans chaque composant du `GOPATH`, les téléchargements dans le premier composant étant toujours disponibles.

9.9.2 Bibliothèque standard et `GOROOT`

Le répertoire d'installation de Go est décrit par la variable d'environnement `GOROOT`, dont la valeur par défaut est :

- ✓ **Windows** : `C:\Go`
- ✓ **UNIX, Linux, macOS et autres systèmes POSIX** : `/usr/local/go`

Ce répertoire présente la structure d'espace de travail complète, et est ajouté à la liste des chemins de recherche avant le premier élément de `GOPATH`.

De la sorte, lors de la recherche, les déclarations d'importation simples comme `import "fmt"` ou `import "container/heap"` trouvent un paquet correspondant au chemin dans `$GOROOT/src/fmt` ou `$GOROOT/src/container/heap` et ne poursuivent pas la recherche de paquets : même si un autre paquet homonyme se trouve défini dans un composant du `GOPATH`, la version de la bibliothèque standard est bien celle qui est résolue, évitant toute surcharge de ses paquets.

9.9.3 Paquets distants avec `go get`

9.9.3.1 La commande `go get` *en mode* `GOPATH`

L'immense majorité des projets incorporant du code de tierce partie, Go inclut une commande standard pour télécharger et compiler des paquets stockés dans des dépôts distants : la commande `go get`.

Son comportement dépendant du mode de développement de projet, le comportement listé ci-après est celui applicable en l'absence de modules VGO, dans toutes les versions Go, basé sur `GOPATH`. Son fonctionnement en leur présence est décrit au § 9.10.

```
go get [-d] [-f] [-t] [-u] [-v] [-fix] [-insecure] [flags] [paquets]
```

Les drapeaux propres à cette commande sont :

- ✓ **`-d`** : demande de télécharger les paquets sans les compiler.
- ✓ **`-f`** : applicable uniquement en conjonction avec `-u`, force à ne pas vérifier l'origine du code, permettant l'utilisant de *forks* et chemins d'importation non canoniques.
- ✓ **`-fix`** : demande l'application de `go tool fix` sur le code téléchargé avant d'en résoudre les dépendances ou de le compiler.
- ✓ **`-insecure`** : autorise le téléchargement depuis des dépôts dont l'URL utilise un protocole non sûr, comme HTTP plutôt que HTTPS ou SSH. Son utilisation constitue une vulnérabilité, et doit donc être évitée autant que possible.

- ✓ `-t` : demande d'inclure les dépendances des tests des paquets téléchargés en plus des dépendances des paquets eux-mêmes.
- ✓ `-u` : demande la vérification d'existence d'une mise à jour pour les paquets déjà téléchargés : par défaut, `go get` ne réalise pas de téléchargement pour les paquets déjà présents dans l'un des espaces de travail du `GOPATH`.
- ✓ `-v` : augmente la quantité d'information affichées durant les téléchargements.

Une page d'histoire

« En 2010, nous avons publié un outil appelé `goinstall`, que nous avons décrit comme "une expérience en matière d'installation de paquets".

Il téléchargeait les dépendances, et les enregistrait dans la distribution Go sur votre poste, dans `GOROOT`.

Au fil de l'expérimentation avec `goinstall`, nous avons découvert qu'il fallait séparer la distribution Go et les paquets installés, afin de pouvoir changer de distribution sans perdre ses paquets.

En 2011, nous avons donc introduit `GOPATH`, une variable d'environnement qui spécifiait où rechercher des paquets ailleurs que dans la distribution Go.

L'ajout de `GOPATH` a créé de nouveaux emplacements possibles pour les paquets Go, mais il a globalement simplifié le développement Go, en séparant la distribution Go de vos bibliothèques.

`Goinstall` est devenu `go get` à la sortie de Go 1.»

Russ Cox - *Experiment, Simplify, Ship*
https://osinet.fr/go/paquets/go-experiment

Elle accepte également les drapeaux de la commande `go build`, et les applique durant l'étape de compilation qui suit le téléchargement, en l'absence du drapeau `-d`.

Ces téléchargements incluent les informations de gestion de version, et ne sont pas de simples archives.

Les spécifications de paquet à télécharger ne sont pas au format URL, mais sont de la forme `<forge>/<chemin>/<paquet>`, par exemple `github.com/golang/text`, qui sont interprétés par la commande.

9.9.3.2 Spécification de dépôt

La commande `go get` emploie trois mécanismes distincts pour localiser un paquet à télécharger à partir des spécifications de paquet :

- ✓ **Connaissance interne** : certaines forges logicielles sont connues « en dur » par l'outil. La liste a évolué au fil des versions [14] ; en Go 1.14, elle comporte :
  - **Apache** : `git.apache.org` - dépôts Git ;
  - **Bitbucket** : `bitbucket.org` - dépôts Git et Mercurial ;

14. Pour toute version, elle est visible dans le fichier `cmd/go/internal/get/vcs.go`.

© Dunod – Toute reproduction non autorisée est un délit.

- **Chisel** : `chiselapp.com` - dépôts Fossil ;
- **Github** : `github.com` - dépôts Git ;
- **IBM DevOps Services (JazzHub)** : `hub.jazz.net/git` - dépôts Git ;
- **Launchpad** : `launchpad.net` - dépôts Bazaar ;
- **OpenStack** : `git.openstack.org` - dépôts Git.

✓ **Spécification du gestionnaire** : pour les forges inconnues, si le chemin d'importation du paquet comporte un suffixe désignant un gestionnaire de versions, `go get` emploie les mécanismes propres à celui-ci, essayant si besoin plusieurs protocoles pour un même gestionnaire en cas d'échec.
Les suffixes reconnus sont :

- **`.bzr`** : Bazaar ;
- **`.fossil`** : Fossil ;
- **`.git`** : Git ;
- **`.hg`** : Mercurial ;
- **`.svn`** : Subversion.

✓ **Métadonnées** : si aucun des mécanismes précédents n'a permis d'identifier le gestionnaire approprié pour l'import d'un paquet, `go get` recourt à un mécanisme basé sur HTTP(S) [15] :

1. Il construit un URL HTTPS/HTTP particulier basé sur le chemin spécifié. Ainsi, pour un paquet désigné par `uneforge.tld/org/paquet`, l'URL construit sera `https://uneforge.tld/org/paquet?go-get=1`.
2. En cas de retour en échec, le téléchargement échoue.
3. En cas de succès, il va examiner le `<head>` de la réponse, à la recherche d'un élément de la forme :

   ```
   <meta name="go-import" content="préfixe vcs racine">
   ```

   La commande `go get` n'incluant pas un analyseur HTML complet, il est préférable que cette méta-donnée apparaisse dès que possible dans le résultat, pour éviter toute erreur de décodage, notamment en présence de code CSS ou JavaScript dans la réponse.
4. En l'absence de cette méta-donnée, le téléchargement échoue.
5. En cas de présence, le fragment `vcs` doit correspondre à l'un des six gestionnaires connus : `bzr`, `fossil`, `git`, `hg`, `svn`, ou `mod` lors de l'utilisation des modules VGO ; à défaut le téléchargement échoue.
6. En cas de succès :
   - le préfixe va être utilisé pour une seconde requête vers
     `https://<préfixe>/?go-get=1`

15. HTTP est utilisé dans un second temps si l'accès HTTPS échoue, et si le drapeau `-insecure` est actif.

qui devra répondre avec la même méta-donnée, précisant le même préfixe, sous peine d'abandon du téléchargement ; ou vers la `racine`, interprétée comme un mandataire pour le système de modules, si le gestionnaire de versions est `mod`.

- L'URL de la racine sera alors utilisé conjointement avec la spécification du gestionnaire de versions pour accéder au clonage du dépôt contenant le paquet distant.

Une fois le paquet téléchargé, il est placé dans le premier espace de travail listé dans `GOPATH`, sous le chemin correspondant au paquet spécifié, comme

```
src/uneforge.tld/org/paquet
```

Son code est alors analysé pour identifier ses dépendances à importer, qui font récursivement l'objet du même mécanisme de téléchargement.

Enfin, à moins que le drapeau `-d` n'ait été spécifié, le paquet est compilé, en commençant par toutes ses dépendances.

Méta-donnée de documentation

Le même mécanisme de méta-donnée est utilisé – hors du contexte de la compilation – par le générateur de documentation `godoc`, pour fournir l'accès à la documentation des paquets distants.

La clause utilisée est de la forme :

```
<meta name="go-source" content="préfixe accueil rép fich">
```

- ✓ le champ `préfixe` correspond au chemin d'importation du paquet ;
- ✓ le champ `accueil` est l'URL de la page d'accueil du dépôt contenant le paquet ;
- ✓ le champ `rép` est un modèle pouvant contenir des éléments `{dir}` et/ou `{/dir}` pour désigner le répertoire du paquet dans le dépôt ;
- ✓ le champ `fich` est un modèle pouvant contenir un élément `{file}` correspondant au nom de fichier et un élément `{line}` correspondant à une ligne à désigner dans le fichier.

Hormis le préfixe, chacun des champs peut être remplacé par le caractère `_`, pour utiliser les valeurs par défaut utilisées par le serveur de documentation officiel https://godoc.org

Ce format permet à un outil de construire des URL absolus pour pointer vers toute ligne du source de tout fichier de tout paquet Go présent sur une forge exploitant ce mécanisme.

9.9.3.3 Fixations de versions : vendoring

L'un des soucis associés avec l'usage de paquets distants tient à la continuité de leur existence, et à leurs évolutions :

- ✓ Un paquet peut subitement disparaître sans préavis, comme la débâcle « left-pad » [16] l'a amplement démontré aux développeurs de milliers de projets JavaScript en 2016.

16. https://osinet.fr/go/paquets/npm-left-pad

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ Un mainteneur peut décider de supprimer une version, généralement pour raisons de sécurité, ou sous contrainte juridique.
- ✓ Sur certaines forges seulement, un projet peut décider de modifier le contenu d'une version publiée, soit en la déplaçant vers un autre commit, soit en récrivant l'histoire (par exemple avec `git push --force`), bien que ce soit une pratique vivemen déconseillée.
- ✓ Dans les systèmes non protégés par un mécanisme de signature cryptographique fort, le contenu d'un téléchargement peut être modifié par une tierce partie.
- ✓ Même en l'absence de tout changement de version publiée, un projet peut voir du code malicieux apparaître d'une version à la suivante, comme l'affaire `event-stream` l'a démontré aux développeurs JavaScript en 2018 [17], impliquant par exemple nombre d'extensions de l'IDE Visual Studio Code [18].

L'approche la plus fiable pour protéger ses projets, hormis l'abandon pur et simple de dépendances externes – qui est rarement une solution pragmatique – consiste à utiliser une version spécifique, la certifier en interne, et utiliser dans ses projets la version locale certifiée, typiquement vérifiée par une signature cryptographique.

Cette approche est souvent requise pour des questions de conformité réglementaire (*compliance*), et le projet Go indique qu'elle était utilisée en interne par Google jusqu'à l'introduction des modules VGO (cf. § 9.10).

Pour la simplifier, et au vu des divers outils communautaires qui en avaient proposé diverses implémentations au fil des ans en complément ou en remplacement de `go get`, un mécanisme de *vendoring* a été ajouté au fonctionnement de `go get` en mode `GOPATH` :

- ✓ Lors de la résolution d'un chemin d'import, les paquets placés directement sous le répertoire `vendor` d'un paquet ne sont importables que par les paquets situés sous ce parent direct, à l'image des restrictions appliquées aux paquets internes (cf. § 9.6.3).
- ✓ Les répertoires sous `internal` sont traités comme un composant additionnel situé pour la résolution des chemins entre `GOROOT` et le premier espace de travail de `GOPATH`. Sous ce répertoire, les commentaires de chemin d'importation canonique (cf. § 9.9.4.2) sont ignorés.
- ✓ Du fait de ce placement, les chemins trouvés dans un paquet sous `vendor` sont utilisés de préférence à ceux trouvés ailleurs dans les espaces de travail pour le sous-arbre situé sous le parent direct du répertoire `vendor` : cela permet de figer la version utilisée par un projet donné, tout en continuant à suivre l'évolution du paquet dans le cas général, notamment pour certifier au fil du temps ses versions suivantes.

17. https://osinet.fr/go/paquets/npm-event-stream
18. https://osinet.fr/go/paquets/vs-event-stream

Chaque projet peut donc ajouter les versions qu'il souhaite de chaque projet dont il dépend, y compris des versions modifiées localement, et les enregistrer avec son projet, pour bénéficier de ce mécanisme sur chaque environnement de développement.

Le degré de soutien de ce mécanisme a évolué au fil des versions :

- ✓ en Go 1.5, il était désactivé par défaut, et activé en définissant la variable d'environnement `GO15VENDOREXPERIMENT` sur `true` ;
- ✓ en Go 1.6, il est devenu activé par défaut, et désactivé en définissant la variable d'environnement `GO15VENDOREXPERIMENT` sur `false` ;
- ✓ depuis Go 1.7, il est toujours actif.

En raison des problèmes, notamment de volumétrie, et de la charge de travail représentée par la certification des versions, cette approche est toutefois depuis l'apparition des modules VGO vue comme un mécanisme historique, à remplacer par l'utilisation des modules, qui incluent une version du vendoring adaptée à leur fonctionnement.

9.9.4 Organisation des paquets en mode `GOPATH`

À moins d'utiliser le système de modules VGO (cf. § 9.10) comme recommandé à partir de Go 1.14, le mécanisme de téléchargement par `go get` dans le premier espace de travail de `GOPATH` fournit une organisation par défaut pour les projets, de la forme :

```
<racine de l'espace>/forge.tld/chemin/paquet
```

Cette organisation basée sur `GOPATH`, permet à chaque projet d'avoir un emplacement prédéfini unique, évitant la multiplication des téléchargements, problème récurrent des projets gérés avec Composer, NPM/Yarn, ou Cargo.

Afin d'assurer l'homogénéité de ces espaces, l'organisation recommandée pour les projets consiste à adopter également ce format pour les projets développés, et non seulement pour les dépendances, en associant à tout projet une forge de référence et un chemin, qui sera le plus souvent le compte personnel ou d'organisation utilisé sur la forge pour publier le projet : ceci évite de trouver un même projet cloné sous des emplacements différents par ses divers contributeurs, ce qui poserait problème pour l'importation du fait des différences de chemin dans l'espace de travail.

9.9.4.1 *Gestion des* forks

Cette organisation présente toutefois un problème dans les processus de travail à base de *forks* et *pull requests* / *merge request* sur les projets utilisant Git.

En effet, dans ce processus extrêmement populaire, le développeur souhaitant apporter une modification à un projet va typiquement procéder à un fork sur la forge du projet, comme https://osinet.fr/go/paquets/gh-gogsgogs20, un fork du serveur Git Gogs [19], dont le dépôt est https://osinet.fr/go/paquets/gh-gogs

19. Gogs est un serveur Git écrit en Go : https://osinet.fr/go/paquets/gogs

© Dunod – Toute reproduction non autorisée est un délit.

Pour travailler sur le fork, si le développeur utilise `go get` pour télécharger son fork, le téléchargement va bien cloner le dépôt Git du fork, et le code ainsi téléchargé va contenir les déclarations d'importation du fork, qui feront presque toutes référence au projet d'origine. De ce fait, le projet d'origine sera également téléchargé, ainsi que ses dépendances.

Par la suite, si le développeur du fork souhaite modifier un paquet autre que celui du premier niveau, il va devoir modifier les déclarations d'importation dans les sources de son fork, ce qui causera des conflits lors de l'émission de la requête de fusion avec le projet amont.

La solution consiste à :

1. Ne pas télécharger le fork, mais le projet original :
   `go get -u -v github.com/gogs/gogs`
   ce qui produira un clone Git du dépôt original.
2. Ajouter un *remote* alternatif, pointant non vers le projet d'origine mais vers le fork, sans supprimer le *remote* d'origine, par exemple :
   `git remote add fork https://github.com/gogsgogs20/gogs.git`
3. Travailler normalement sur le fork, en créant des branches locales liées aux branches du fork et non de l'original, comme :
   `git checkout -b 23-un-ticket fork/23-un-ticket`
   En les poussant, elles remonteront sur le dépôt du fork et non sur le dépôt d'origine, cependant que les mises à jour par `go get -u` actualiseront le clone local depuis le dépôt original.
4. Une fois finalisées, ces branches pourront alors donner lieu à des requêtes de fusion sans causer de conflits, les déclarations d'importation ayant été préservées.

9.9.4.2 Chemin d'importation canonique

Un défaut de l'approche consistant à inclure le nom de la forge dans les chemins d'importation d'un projet réside dans le fait que cela introduit un couplage de fait à celle-ci.

Que se passe-t-il si elle vient à disparaître ? Ou simplement si le projet doit changer de forge pour une raison ou une autre ? D'un instant à l'autre, toutes les déclarations d'importation doivent être modifiées, et tous les projets qui en dépendent doivent se mettre à jour sans délai avec les nouveaux chemins d'importation incorporant la nouvelle forge.

L'existence du mécanisme d'autodétection du dépôt par la méta-donnée `go-source` (cf. § 9.9.3.2) fournit une solution : il suffit que le projet utilise comme URL canonique un serveur qui lui soit propre, et qui répondra aux requêtes `go get` en incluant une méta-donnée `go-import` pointant vers la forge utilisée, tout en préservant la possibilité d'en changer sans causer de disruption pour les projets qui en dépendent.

Ceci introduit une autre difficulté : un projet peut se retrouver cloné soit depuis son URL officiel par la commande `go get`, soit depuis sa forge courante, aussi bien par `go get` que par les commandes natives de clonage du système de gestion de versions utilisé. Et, du fait de l'organisation consistant à localiser les projets dans un espace de travail sous un répertoire contenant la forge utilisée, le même projet peut ainsi se retrouver cloné sous au moins deux emplacements, sans compter d'éventuels clonages depuis des forks.

Depuis Go 1.4, les clauses de paquet (cf. § 9.2.1) peuvent inclure après le nom de paquet un commentaire spécifiant le chemin d'importation canonique, comme dans cet exemple du SDK Google Cloud :

```
package cloud // import "cloud.google.com/go"
```

Initialement proposé sous le nom de *custom import* [20], ce mécanisme est plus connu sous le nom de *vanity import*.

Dès lors qu'une clause d'import adopte cette forme, tout téléchargement par `go get` va refuser de s'exécuter si le chemin d'import utilisé ne correspond pas au chemin canonique, nécessitant la combinaison des drapeaux `-f` et `-u` pour que cette sécurité soit ignorée. Les dépendances placées dans un répertoire `vendor` (cf. § 9.9.3.3) ne sont pas affectées.

Ceci permet d'éviter les clones multiples par erreur, sans empêcher les manipulations souhaitant explicitement utiliser un emplacement alternatif.

9.10 MODULES VGO

Avertissement

La version 1.14 est la première à déclarer le système de modules comme prêt pour l'utilisation en production.

La majorité des fonctionnalités décrites sont néanmoins utilisables avec Go 1.13 et, de façon plus limitée, en Go 1.9.7, 1.10.3, et toutes les versions Go 1.11.x et 1.12.x, pour les développeurs contraints d'utiliser provisoirement une version de Go plus ancienne le temps d'une conversion à la version courante.

20. https://osinet.fr/go/paquets/go-vanity-imports

© Dunod – Toute reproduction non autorisée est un délit.

9.10.1 Inconvénients du mécanisme des espaces de travail

Le mécanisme originel d'assemblage de paquets basé sur les espaces de travail et `GOPATH`, tel que décrit depuis le début de ce chapitre, présente plusieurs inconvénients :

- ✓ Le mécanisme de téléchargement des imports distants par `go get` n'offre aucun mécanisme permettant de spécifier une version particulière d'un module, et utilise par défaut toujours la dernière version disponible de la branche par défaut du dépôt contenant le paquet recherché.
- ✓ Il ne dispose intrinsèquement d'aucun moyen de garantir l'intégrité des versions, que ce soit en instantané ou dans la durée.
- ✓ Les développeurs n'utilisent que rarement Go comme premier langage, et s'attendent à pouvoir utiliser une approche où tout le code d'un projet se trouve sur le projet, au lieu d'être rangé dans des espaces de travail perçus comme « fourre-tout » ; et à pouvoir à l'intérieur de ces projets spécifier les règles d'importation et de construction sur la base du *Semantic Versioning*, reflétant la pratique des autres langages courants :
  - Java avec Gradle étendu pour semver ;
  - JavaScript avec NPM ou Yarn ;
  - PHP avec Composer ;
  - Python avec PIP et le standard PEP440 ;
  - Rust avec Cargo.

L'avantage de non-duplication du code mis en avant dans l'organisation du code originelle (cf. § 9.9.4) n'est généralement pas considéré comme pertinent dans ces situations, reflétant en partie l'explosion des espaces disques disponibles.

9.10.2 Solutions antérieures aux modules

Conséquence de ces limitations, la question de la gestion des dépendances est depuis les débuts de Go l'un des principaux points durs de la réalisation de projets, et les solutions n'ont pas manqué, adoptant diverses stratégies :

- ✓ **Vendoring** : la première réponse a été l'utilisation du *vendoring*.
  Initiée par des outils tiers simples comme `govendor` et `vendorize` dès 2013, elle a permis en son temps de résoudre le problème au prix d'une réécriture des chemins d'import pour incorporer un composant `vendor`.
- ✓ **Mandataire** : une approche radicalement différente a été introduite par le site https://gopkg.in/ à partir de mars 2014, utilisant le mécanisme de la méta-donnée `go-import` (cf. 9.9.3.2) pour répondre à des URL incorporant un numéro de version majeure avec des liens vers les URL correspondant à la branche désignée sur Github.

Très populaire, ce mécanisme est toutefois limité en ce qu'il ne fournit pas de garantie de reproductibilité, se limite aux projets hébergés sur Github, ne gère que les branches majeures, et dépend d'un service tiers sans garantie de continuité.

- ✓ **Contraintes et verrou** : dès 2013, d'autres outils ont tenté de résoudre les problèmes des deux premières approches en ajoutant un fichier de spécification d'expressions de version et/ou un fichier de verrou de version une fois les spécifications résolues, à la façon des équivalents PHP ou JavaScript.
  La plupart de ces outils ont eu une durée de vie limitée : `glock` avec son `GLOCKFILE`, `gigo` « Pip for Go », `gobs` avec `goproject.json` et `.goproject`, `gopack` inspiré du Rebar de Haskell, `gopin` avec un fork de `go get` permettant la spécification d'une étiquette de version.
  Deux d'entre eux ont survécu, avec pour approche l'utilisation d'une commande spécifique invoquant `go get` : `godeps`, basé sur des étiquettes de construction [21] (*build tags*) dans les sources, et `gorepoman` avec `gomanifest.json`.
  Les problèmes de ces solutions sont la nécessité d'outillage non standard, la diversité des solutions, et la nécessité d'instrumenter spécifiquement le code (`godeps`).
- ✓ **Vendoring et versions** : en 2014, le logiciel `glide` est devenu la première contribution à combiner *vendoring* et spécification de versions *semantic versioning*, comparable aux outils disponibles dans d'autres langages, devenant pour un temps la référence de fait.
  Il a connu deux principaux dérivés, encore actifs : `gogradle`, un greffon Gradle, et `trash`, qui optimise les opérations de copie.
  D'autres outils ont adopté la même approche sans connaître le même succès : `gopm`, `bunch`, `goop` inspirée du `bundler` de Ruby, et `govend`.
- ✓ **Vendoring Go 1.5** : validant toutes ces approches basées sur le *vendoring*, Go 1.5 en a standardisé une version compatible avec l'outillage standard basé sur `go get` (cf. § 9.9.3.3).
  Les outils contribués actifs s'y sont ralliés : `dep`, `godep`, `godm`, `gom`, `govend`, `govendor`, `gsv`, `gv`, `gvt`, `manul`, `rubigo`, `trash`, `vendetta`, `vexp`, etc.
- ✓ **Vendoring et `GOPATH`** : d'autres expérimentations ont combiné *vendoring* et `GOPATH`, en manipulant ce dernier en cours de compilation, et/ou en s'appuyant sur le *vendoring* standardisé par Go 1.5 : `gb`, `goat`, `wgo`, et surtout `godep`, le prédécesseur de `dep`.
- ✓ **`dep`, l'expérience officielle** : la poursuite de ces expériences, et les retours de la communauté des développeurs continuant à ne pas faire état d'une parfaite satisfaction des besoins, le projet Go a lancé une « expérience officielle », le projet `dep`, pour expérimenter dans cet espace afin d'informer une future solution supposée définitive à toutes difficultés sur ce sujet.

21. https://osinet.fr/go/paquets/go-build-tags

© Dunod – Toute reproduction non autorisée est un délit.

Fort de ce caractère unificateur, `dep` a reçu des efforts de développement conséquents, et a proposé une solution compatible `GOPATH`, utilisation des expressions semantic versioning, et basée sur un résolveur SAT [22], à l'image des outils équivalents de JavaScript, PHP ou Rust.
Les projets utilisant `dep` sont nombreux, et reconnaissables à la présence à la racine de leur dépôt de deux fichiers :

- **`Gopkg.toml`** : description d'un jeu de contraintes à base d'expressions *semantic versioning* à résoudre, spécifié sur https://osinet.fr/go/paquets/gopkg-toml
- **`Gopkg.lock`** : description du résultat de la résolution des contraintes, et signature cryptographique des téléchargements résultants pour en garantir l'intégrité.

9.10.3 Des objectifs du projet Go aux modules VGO

Parallèlement aux besoins identifiés par la communauté des développeurs au fil des ans, l'équipe du projet Go s'était donné certains objectifs additionnels :

✓ Favoriser l'utilisation de versions (*releases*) des dépendances, identifiées par des étiquettes (*tags*), plutôt que d'utiliser systématiquement le dernier commit d'une branche, toujours variable.

✓ Découpler le mécanisme de téléchargement des dépendances du gestionnaire de versions utilisé, facilitant l'utilisation de projets utilisant des gestionnaires de version moins populaires que Git, comme Bazaar, Fossil, ou Mercurial, voire de systèmes historiques basés sur Subversion (SVN).

✓ Découpler la gestion des dépendances de celle de dépôt de gestion de versions, pour permettre en particulier un mode d'utilisation de type *monorepo* aussi facilement qu'avec le mécanisme originel basé sur `GOPATH`.

✓ Permettre – mais ne pas requérir – l'utilisation de serveurs mandataires pour le téléchargement des dépendances, pour garantir la stabilité et la disponibilité des versions de dépendances dans le temps.

✓ Permettre la cohabitation de deux versions majeures différentes d'une même dépendance dans un même projet.

✓ Ne pas dépendre d'un annuaire (*registry*) tiers comparable à https://npmjs.org, https://packagist.org, ou https://crates.io, l'expérience au fil des ans ayant établi que ceux-ci pouvaient devenir un point de contention lors de pics de trafic, et ne disposant pas nécessairement des ressources nécessaires à garantir un bon niveau de qualité de service.

✓ Supprimer le recours au *vendoring* ou, à tout le moins, faire en sorte de proposer des garanties d'intégrité et disponibilité équivalentes en son absence, tout en permettant d'y recourir pour les organisations tenant à le préserver, au moins pour des raisons historiques.

22. https://osinet.fr/go/paquets/wiki-sat

Observant les résultats de l'expérience `dep`, le projet Go a finalement décidé de construire une solution assez largement assez différente pour atteindre ces objectifs : les modules VGO.

Ce mécanisme a été introduit progressivement depuis Go 1.11 :

✓ **Go 1.9** : La compatibilité avec le mécanisme des modules a été introduite rétroactivement en Go 1.9.7, la dernière version de Go 1.9.

✓ **Go 1.10** : La compatibilité avec le mécanisme des modules a été introduite rétroactivement en Go 1.10.3.

✓ **Go 1.11** : Go 1.11 a été la première version à intégrer un support « préliminaire » des modules, marqués à l'époque comme expérimentaux.
Dans cette version, les commandes liées aux modules ne fonctionnenent que dans les répertoires de modules, et les modules ne peuvent être utilisés que dans des répertoires ne figurant pas dans les espaces de travail du `GOPATH`.

✓ **Go 1.12** : La variable d'environnement `GO111MODULE` permet d'autoriser l'utilisation des commandes liées aux modules même dans les espaces de travail, pour la valeur `on` uniquement.

✓ **Go 1.13** : Les commandes liées aux modules sont disponibles partout, à moins que `GO111MODULE` ne force leur interdiction avec la valeur `off`, dès lors que le répertoire courant ou l'un de ses parents comporte un fichier `go.mod`.
Les mécanismes de serveurs mandataires sont étendus pour supporter des mandataires multiples, et un mandataire par défaut est fourni par le projet Go, ainsi qu'une base de sommes de contrôle cryptographiques.

✓ **Go 1.14** :
- La commande `go` accepte le drapeau `-mod=(mod|readonly|vendor)` pour contrôler l'utilisation du vendoring, les répertoires `vendor` étant dorénavant ignorés, sauf en ce qui concerne le répertoire `vendor` à la racine d'un module, qui peut être utilisé avec la commande `go build -mod=vendor`. Ce drapeau doit être passé au moyen de la variable d'environnement `GOFLAGS`, et est rejeté par la sous-commande `go get`.
- Elle exclut les versions étiquetées `+incompatible` lors des mises à jour, sauf en cas de spécification explicite.
- Soutien des dépôts Subversion (`svn`) en mode modules.
- `-modcacherw` évite que le système de modules ne modifie les permissions des modules téléchargés.
- `-modfile=<fichier>` permet de spécifier l'utilisation de fichiers `go.mod` et `go.sum` alternatifs, le nom du second étant dérivé de celui du premier en remplaçant son suffixe `.mod` par `.sum`.

✓ **Au-delà** (prévisions) :
- Dépréciation de `GOPATH`, prévue pour Go 1.13 et finalement reportée.

© Dunod – Toute reproduction non autorisée est un délit.

- Fonctionnement du marquage de versions de modules comme obsolètes décrit dès la proposition initiale VGO [23], mais pas encore implémenté.
- Ajout d'un mécanisme de déclaration de versions à risque *après* leur publication.

9.10.4 Fonctionnement

9.10.4.1 Versionnage sémantique

L'un des objectifs du projet Go lors de la création du système de modules était la capacité à utiliser simultanément deux versions majeures – au sens *semantic versioning* – d'une même dépendance dans un même projet.

Semantic Versioning

Semantic Versioning est une spécification de format de versions destinée à simplifier et standardiser la sélection des dépendances dans les projets logiciels.
La spécification, publiée en 2009 par Tom Preston-Werner, est détaillée sur https://semver.org/, et est rapidement devenue populaire dans de multiples langages.
Sans entrer dans les détails, la principale idée autour de laquelle elle s'organise est l'utilisation de chaînes de versions à trois composants :

1. Le premier nombre est le numéro de version *majeure*. L'incrémentation de ce premier composant marque une évolution incompatible avec les versions antérieures. Pour cette raison, il est parfois également appelé *breaking*.
2. Le deuxième est le numéro de version *mineure*. Son incrémentation marque une évolution du code par ajout de nouvelle fonctionnalité, compatible avec les précédentes versions.
3. Le troisième est le numéro de *correctif*. Son incrémentation marque une simple correction d'anomalie, sans évolution fonctionnelle.

La version majeure 0 est utilisée pour les projets non encore finalisés, et de ce fait supposés plus instables. Dans leur cas, le deuxième nombre reflète les évolutions incompatibles, et le troisième aussi bien les évolutions compatibles que les simples corrections ; et ce jusqu'à la version 1.0.0, première version finalisée.
Ces versions à trois composants peuvent être précisées par des compléments, auxquels la spécification attribue également des rôles.
La description d'un langage d'expressions permettant de spécifier des critères de sélection sur la base de ces versions, ne fait *pas* partie de la spécification Semantic Versioning.

Or le format des déclarations d'importation (cf. § 9.7) n'a pas changé avec ce mécanisme, et n'inclut pas d'extension pour spécifier une version.

La solution utilisée a donc été de modifier la chaîne de spécification d'importation : là où elle correspondait en mode `GOPATH` exactement à un paquet, elle peut avec les modules être complétée d'un segment supplémentaire, pour un format similaire à `forge.tld/org/paquet/vX` où `X` est le numéro de version *majeure* du paquet importé, pour toute version majeure supérieure ou égale à 2. Pour un paquet situé dans un sous-répertoire, le chemin se situe après le segment de version, sous la forme `forge.tld/org/paquet/vX/sous/paquet`.

23. https://osinet.fr/go/paquets/cox-vgo-modules sous *Deprecated versions*.

Ce mécanisme d'extension du chemin d'import est dit *semantic import versioning*.

Le module référencé pour importer le paquet est choisi dans sa version la plus élevée correspondant à la version majeure ainsi spécifiée, avec pour valeur par défaut la version majeure 1 si le segment est absent.

Les versions antérieures à `v1.0.0` n'étant pas destinées à la publication puisqu'elles correspondent à un état incomplet en *semantic versioning*, elles ne peuvent être spécifiées par ce moyen, et sont référencées par des *pseudo-versions* définies à partir de leur référence de commit dans le gestionnaire de versions du projet importé, comme détaillé au § 9.10.4.3.

Par extension, et pour maintenir avec le système historique de https://gopkg.in/ (cf. § 9.10.2), les chemins d'import utilisant ce domaine conservent avec les modules la même interprétation qu'en mode `GOPATH`.

Le mécanisme de sélection des versions, basé sur l'algorithme *Minimal Version Selection* ou MVS, est décrit en détail avec l'ensemble des mécanismes associés, dans la série d'articles consacrés à l'élaboration du système de modules [24] par Russ Cox, son principal créateur, et synthétisé dans la proposition initiale des modules, présentée sur https://osinet.fr/go/paquets/gh-go-24301

9.10.4.2 Système de fichiers

Le principal nouveau concept avec le système de modules est précisément la notion même de module, rapprochée dans le tableau 9.5 des concepts voisins, activé par un fichier `go.mod`.

- ✓ Un module est un sous-arbre du système de fichiers, dont le premier niveau comporte au moins un fichier `go.mod`.
- ✓ Il se compose du contenu de tous les sous-répertoires de ce premier niveau, à l'exception de ceux qui comportent eux-mêmes un fichier `go.mod`, et définissent par là-même un autre module. Il diffère en cela des paquets qui sont confinés à un seul répertoire (cf. § 9.6.1).
- ✓ Il correspond généralement à un dépôt de système de gestion de versions, mais ce n'est pas une nécessité.

Tableau 9.5 – Modules et concepts voisins

	Dépôt	Paquet	Module
Atomique ?	Non	Oui	Non
Versionné ?	Oui	Non	Oui, semver
Contenu	Plusieurs modules Plusieurs paquets	Plusieurs fichiers Un répertoire	Plusieurs paquets
Lié à un VCS ?	Oui	Non	Non

24. https://osinet.fr/go/paquets/cox-vgo

© Dunod – Toute reproduction non autorisée est un délit.

Les modules sont l'unité de distribution de code, alors que les paquets qu'ils contiennent en sont l'unité d'exécution.

Les modules remplacent `GOPATH` pour la résolution des déclarations d'importation, mais pas entièrement pour le stockage : lors d'un téléchargement de module, ceux-ci sont stockés dans le premier espace de travail du `GOPATH`, ce qui permet de limiter l'espace requis par des projets multiples en mutualisant le téléchargement des modules communs à plusieurs projets.

À la différence des téléchargements hors modules par `go get`, leur enregistrement n'a toutefois pas lieu sous le répertoire `src/uneforge.tld/org/paquet`, mais sous `pkg/mod/uneforge.tld/org/paquet`, et les noms des répertoires de paquet peuvent y incorporer ou non une spécification de version. Il est ensuite possible de les copier localement dans le répertoire `vendor` du module (*vendoring*) par `go mod vendor`.

9.10.4.3 Le fichier `go.mod`

Le fichier `go.mod` contient la description d'un module. Celui situé à la racine d'un projet basé sur les modules est son fichier de contrôle principal, qui suffit à décrire intégralement la construction du projet.

Un fichier `go.mod` est un fichier texte orienté ligné, acceptant des commentaires entre `//` et une fin de ligne, et cinq types de directives :

- ✓ **module** : une directive de la forme

  ```
  module forge.tld/org/lemodule
  ```

  définit le *chemin du module*, qui sert de préfixe commun aux chemins d'importation de tous les paquets du module, et se combine avec le chemin d'accès au module sur disque pour définir le chemin d'accès aux paquets. Cette directive ne peut apparaître qu'une seule fois par fichier.

- ✓ **go** : une directive de la forme `go 1.14`, ou tout autre numéro, définit la version de Go à utiliser pour interpréter le code des paquets du module. Elle est facultative, ne peut apparaître qu'une fois, et n'existe qu'à partir de Go 1.12.
  En son absence, la valeur par défaut est celle du compilateur utilisé pour construire la commande de modules en cours.
  Si le compilateur utilisé est d'une version antérieure à celle spécifiée par cette directive, la compilation est tentée malgré tout, et ne signale le problème qu'en cas d'échec de compilation.
  Ce mécanisme est nécessaire pour certains fichiers comportant des étiquettes de construction spécifiant une version de SDK.

- ✓ **require** : cette directive permet de définir des modules à importer, en spécifiant une version précise, sous l'une des deux formes :
  - `v1.2.3`, où les trois chiffres correspondent aux trois composants d'une version *semantic versioning* publiée pour le module importé.

- `v0.0.0-yyyymmddhhmmss-abcdefabcdef` désigne l'importation d'un commit ne correspondant pas à une version publiée du module importé. Dans ce format, la sous-chaîne `yyyymmddhhmmss` correspond à l'horodatage du commit, et la sous-chaîne `abcdefabcdef` à 12 caractères de préfixe de son numéro de commit.
  C'était en Go 1.11 la seule pseudo-version. Depuis Go 1.13, les chiffres `v0.0.0` sont normalement remplacés par la dernière version publiée avant le commit choisi, mais la valeur `v0.0.0` demeure acceptée par souci de compatibilité.
- `vX.Y.Z-pre.0.yyyymmddhhmmss-abcdefabcdef`, avec le même format, désigne un commit spécifique ne correspondant pas à une version publiée, dans le cas où `vX.Y.Z-pre` est la plus récente *pré-version* publiée.
- `vX.Y.(Z+1)-0.yyyymmddhhmmss-abcdefabcde`, avec le même format, désigne un commit spécifique ne correspondant pas à une version publiée, dans le cas où `vX.Y.Z` est la plus récente *version* publiée.

Les trois dernières formes sont dites *pseudo-versions*. Dans ce format, la partie horaire permet entre deux commits de déterminer lequel est le plus récent, les numéros de commit ne formant pas dans la plupart des systèmes de gestion de version une séquence ordonnée.
Ces pseudo-versions sont fabriquées automatiquement par les commandes liées aux modules lors de la résolution des requêtes de modules.

✓ **exclude** : une directive de la forme

```
exclude forge.tld/org/lemodule vX.Y.Z
```

permet d'interdire le téléchargement d'une version spécifique.
Comme les directives `require` spécifient les versions précises à importer, cette clause n'est utile que pour contrôler la sélection des versions lors des opérations de mise à jour.

✓ **replace** : une directive de la forme

```
replace mod1 vX.Y.Z => mod2 vP.Q.R
```

permet de demander le remplacement de la version de module à gauche de la flèche par celle de droite. Les spécifications de version de chaque côté sont facultatives.
La directive `replace` fournit un moyen simple d'utiliser une version localement modifiée d'un paquet distant, en ajoutant une directive de la forme :

```
replace forge.tld/org/module => /chemin/local/module
```

Les directives `require`, `replace` et `exclude` pouvant apparaître plusieurs fois dans le fichier, il est possible de les remplacer par une unique directive identique, encadrant de parenthèses une liste de lignes de modules et versions, à la façon des déclarations Go comme **`const`** ou **`var`**.

© Dunod – Toute reproduction non autorisée est un délit.

Les directives `replace` et `exclude` ne sont appliquées que dans le fichier `go.mod` principal, et ignorées dans les modules importés : ceci permet de garantir que la liste de construction (*build list*) d'un projet à base de modules est intégralement décrite par ce fichier, même lors de la recherche de mises à jour.

Par extension à ce format, les directives `require` peuvent aussi prendre la forme d'une requête de module, comme décrit au § 9.10.5.2. Toutefois, dès qu'une commande Go interviendra sur le fichier, la requête sera résolue et remplacée par son résultat, de sorte que ce n'est pas une pratique recommandée.

9.10.4.4 Le fichier `go.sum`

Comme le montre la description de son format, le fichier `go.mod` n'est pas l'équivalent d'un `composer.json`, `package.json`, ou `Cargo.toml`.

En effet, là où ces fichiers spécifient des expressions *semantic versioning* – des requêtes de modules en terminologie Go – le fichier `go.mod` précise exactement la version à télécharger.

Il est en ce sens plus proche du rôle d'un fichier verrou comme `composer.lock`, `yarn.lock`, ou `Cargo.lock`, d'autant que, comme eux, il n'est pas prévu pour être édité manuellement, mais créé et modifié par les commandes liées aux modules.

Mais, à la différence de ces derniers, il n'inclut pas les sommes de contrôle cryptographiques des modules téléchargés. Ce rôle est dévolu au fichier `go.sum`, dont le rôle unique est de garantir l'intégrité des versions.

Lors de chaque opération amenant un projet à prendre connaissance d'une version de dépendance utilisée, sa signature y est enregistrée.

Cela permet, lors de chaque téléchargement, de s'assurer que la version téléchargée n'a pas changé de contenu par rapport à la version utilisée pour la construction sur un autre poste.

Par la suite, même si le projet vient à requérir une version antérieure ou ultérieure, les signatures demeurent, même inutilisées ; cela permet lors d'une mise à jour future de détecter un changement de version, même si elle n'a pas été utilisée entre temps, accroissant ainsi l'étendue de la plage de versions dont l'intégrité est protégée à toutes celles utilisées par le projet à tout stade de son développement.

Comme les autres fichiers verrous, son contenu n'est pas destiné à une exploitation manuelle, bien qu'il s'agisse là aussi d'un fichier texte.

Puisque ce fichier est uniquement un mécanisme de contrôle d'intégrité, il a vocation à être committé dans tout projet, que ce soit une application finale ou un module destiné à l'importation.

9.10.5 Utilisation au quotidien

9.10.5.1 Créer un nouveau module

Un module n'est rien d'autre qu'un paquet complété par un fichier `go.mod`.

La conversion d'un simple paquet en module se résume à la commande `go mod init`, complétée dans la plupart des cas par le chemin d'importation canonique du paquet, de la forme :

```
go mod init forge.tld/org/module
```

Si le répertoire du paquet se situe dans un espace de travail `GOPATH`, le chemin du paquet n'est pas nécessaire : il est automatiquement identifié par le chemin dans l'espace de travail.

Cette commande a pour effet de créer un fichier `go.mod` minimal :

Listing 9.5 – Fichier `go.mod` minimal

```
1 module forge.tld/org/module
2
3 go 1.14
```

9.10.5.2 Modifier les dépendances

À ce stade, même si le code dépend de certaines dépendances, celles-ci ne sont pas encore importées, et sont inconnues du `go.mod`.

Toute commande nécessitant une construction, comme `go build`, `go run`, ou `go test` va automatiquement identifier les dépendances à partir du code source, les ajouter au fichier `go.mod` et créer le fichier `go.sum` pour y enregistrer les sommes de contrôle des paquets importés.

Les dépendances peuvent dans cette approche utiliser le format *semantic import versioning* (cf. § 9.10.4.1) pour spécifier les dépendances sur des versions majeures supérieures à 1, en ajoutant un segment de version au chemin présent dans la déclaration d'importation.

Ainsi, pour utiliser la dernière version 4.x du fork du module de journalisation `glog` de Google par Airbrake, un module pourra spécifier dans son code :

```
import "github.com/airbrake/glog/v4"
```

…ce qui aura pour effet, lors de la prochaine requête de module, d'insérer une ligne de la forme (la version peut évidemment varier avec le temps) :

```
require github.com/airbrake/glog/v4 v4.0.1
```

Il est également possible de forcer une actualisation du fichier `go.mod`, incluant à la fois l'ajout des nouvelles dépendances et la suppression des anciennes dépendances avec la commande `go mod tidy`.

© Dunod – Toute reproduction non autorisée est un délit.

Dans cette approche, les dépendances sont détectées et téléchargées par Go, mais il est aussi possible de spécifier des dépendances avant de les utiliser, avec le commande `go get`, comme en mode `GOPATH`.

Par défaut, la commande acquiert la version la plus élevée de la branche majeure 1 du module mais, à la différence du mode `GOPATH`, il est possible de spécifier une requête de module (*module query*) pour demander à Go de rechercher une version à partir d'une contrainte, à la façon d'un `composer require` ou d'un `yarn add`. Les formats disponibles sont les suivants :

- ✓ **`v1.3.2`** : correspond à une version sémantique totalement spécifiée, et requiert exactement cette version.
- ✓ **`v1` ou `v1.3`** : correspond à un préfixe de version sémantique, requérant la plus récente version disponible comportant ce préfixe.
- ✓ **`<v1.3.2` ou `>=v1.4.3`** : est une comparaison de version sémantique, requérant la plus récente version disponible validant cette comparaison.
- ✓ **`latest`** : requiert la plus récente version si une version existe, ou à défaut le commit non étiqueté le plus récent de la branche par défaut du gestionnaire de versions du module requis. Si le module est déjà requis à un commit plus récent, cela aura pour effet de revenir à une version plus ancienne.
- ✓ **`upgrade`** : est similaire à `latest` mais, si le module est déjà requis à un commit plus récent, il y demeurera.
- ✓ **`patch`** : est similaire à `upgrade` mais, si un module est déjà requis – directement ou par l'un de ses paquets – à une version donnée, la mise à jour se limitera à la plus récente version corrective de la version mineure actuellement utilisée.
- ✓ **Référence du gestionnaire de versions** : toute autre chaîne pouvant être interprétée par le gestionnaire de versions est utilisable, comme un numéro de commit, ou le nom d'une branche, interprété comme le dernier commit de la branche.

Une fois la version appropriée identifiée, la spécification de l'exacte version utilisée – et non la requête de module – et la somme de contrôle du module téléchargé sont stockées dans les fichiers `go.mod` et `go.sum`. En particulier, si la requête de module spécifie un commit ou une branche et que le commit ainsi spécifié correspond à une version, c'est la référence de version qui est préservée.

Lors d'un développement itératif, par opposition à la migration d'un projet existant, cette approche est souvent la plus pratique, car elle permet de bénéficier de l'ensemble des symboles des paquets du module importé dans son éditeur ou IDE.

Enfin, il est possible, comme avec `go get` en mode `GOPATH`, de demander la mise à jour d'un, de plusieurs ou de tous les modules de la liste de construction avec la commande `go get -u <modules>`

9.10.5.3 Publier une version

Go privilégiant l'immuabilité des versions, il est préférable que chaque version ne soit publiée qu'à l'issue d'étapes de préparation :

1. Nettoyer la liste de dépendances avec `go mod tidy`
2. Exécuter la suite de tests complète, avec le détecteur de situations de compétition, avec `go test -race ./...`
3. Vérifier que l'état de l'arborescence de travail est propre pour le gestionnaire de versions, par exemple avec `git status` ; à défaut réaliser un commit pour obtenir un état propre.
4. Étiqueter le commit avec une étiquette de la forme `vX.Y.Z`, où les nombres X, Y, et Z forment une version conforme *semantic versioning*.
5. Publier la branche et l'étiquette sur le dépôt correspondant à son chemin d'import.

À ce stade, le module devient importable :

- ✓ Si la version est encore une `0.Y.Z`, une requête de module spécifiant l'étiquette, comme `go get forge.tld/org/module@v0.Y.Z` sera nécessaire.
- ✓ Si la version est une `1.Y.Z`, son chemin d'import sera un chemin simple similaire à un chemin en mode `GOPATH`.
- ✓ Si la version est une `X.Y.Z` avec `X ⩾ 2`, un chemin d'import de la forme `forge.tld/org/paquet/vX` sera utilisable.

9.10.5.4 Autres opérations sur les modules

À tout moment, il est possible de vérifier l'état du module et de ses dépendances avec les variantes de la commande `go list -m`, le drapeau `-m` étant nécessaire pour qu'elle opère sur les modules plutôt que sur les paquets. Ses options sont nombreuses et décrites par la commande `go help list`.

Les commandes utilisables en mode `GOPATH` sont également utilisables en mode modules. Une seule commande est spécifique : `go mod`. Elle centralise les opérations spécifiques aux modules, sans équivalent en mode `GOPATH` :

- ✓ **`go mod download`** : télécharge les modules déjà spécifiés dans le cache local présent dans le premier espace de travail du `GOPATH`.
- ✓ **`go mod edit`** : permet de manipuler le fichier `go.mod`, ce qui est plus pratique pour les scripts d'automatisation que de devoir utiliser de simples manipulations de texte, et garantit la conformité du format de fichier à sa grammaire.
- ✓ **`go mod graph`** : produit une liste d'arêtes reliant chaque module du graphe de dépendances à l'une de ses dépendances. Divers outils Open Source exploitent ce format pour en dériver des graphiques avec GraphViz.
- ✓ **`go mod init`** : convertit un simple paquet en module.

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ **`go mod tidy`** : ajoute les nouvelles dépendances découvertes dans les sources du module, et supprime celles qui ne sont plus utilisées. Recommandé avant tout enregistrement d'une nouvelle version.
- ✓ **`go mod vendor`** : copie dans le répertoire `vendor` du projet les modules importés, permettant la réalisation de dépôts complets à construire sans téléchargements. Cette copie est réalisée depuis le cache de modules dans le premier espace de travail du `GOPATH` et est donc très rapide, ne nécessitant pas de téléchargement.
- ✓ **`go mod verify`** : vérifie que le code source des modules téléchargés dans le cache de modules du premier espace de travail du `GOPATH` n'a pas été modifié localement.
- ✓ **`go mod why`** : indique la raison de la présence d'un module dans le fichier `go.mod`. Il est généralement nécessaire de l'utiliser sous la forme `go mod why -m <module>`, pour que la recherche examine les dépendances des paquets dans les sous-répertoires des modules : à défaut, elle se limite à rechercher les dépendances directes sur le paquet principal du module, ce qui est rarement le but recherché.

Outre sa fonction de création initiale de module, `go mod init` est également capable, dans nombre de cas, de réimporter les configurations de ses prédécesseurs, avec des taux de succès variables selon la configuration du projet.

Elle reconnaît en particulier :

- ✓ `Gopkg.lock` de `dep`,
- ✓ `GLOCKFILE` de `glock`,
- ✓ `Godeps/Godeps.json` de l'ancien outil `godep`, remplacé par `dep`,
- ✓ `dependencies.tsv` de `godeps`,
- ✓ `glide.lock` de `glide`,
- ✓ `vendor.conf` de `trash`,
- ✓ `vendor.yml` de `govend`,
- ✓ `vendor/manifest` de `gvt`,
- ✓ `vendor/vendor.json` de `govendor`.

En cas d'échec, ou en présence d'autres systèmes, il est souvent efficace de supprimer la configuration de l'ancien gestionnaire, et effectuer un `go mod init` suivi d'un `go mod tidy` pour régénérer les dépendances à partir des sources.

9.10.5.5 Télécharger des modules distants

La commande `go get` utilisée avec les modules est capable d'utiliser des serveurs mandataires (*proxies*) plutôt que de télécharger directement depuis les serveurs origine.

La configuration de l'accès à ces serveurs est détaillée au § 9.10.5.6.

Ceci présente de nombreux avantages, pour peu que le serveur en question comporte un stockage persistant pour les modules téléchargés :

- ✓ **Protection contre les pannes amont** : du fait de la présence locale des modules clonés après une première utilisation, ceux-ci demeurent disponibles même pendant les pannes.
- ✓ **Protection contre problèmes d'évolution** : du fait de la présence de ces copies locales, en cas de modification d'une version déjà publiée, le mandataire peut signaler l'anomalie, et continuer à fournir la version validée antérieurement. De même, si le projet vient à être supprimé comme dans l'affaire `left-pad` (cf. § 9.9.3.3).
- ✓ **Performance** : lors du téléchargement, le client Go peut utiliser des mécanismes de téléchargement par copie, comme `git archive`, qui peuvent être considérablement plus rapides qu'un clonage direct depuis le gestionnaire de versions.
- ✓ **Performance** : une fois les paquets disponibles dans le cache du mandataire, le téléchargement ne nécessite plus d'accès distant, mais opère au niveau de performance du réseau local de l'entreprise.
- ✓ **Traçabilité, politique** : si tous les téléchargements sont centralisés par le mandataire, celui-ci est en mesure de limiter les sources de modules autorisées selon la politique d'utilisation de projets extérieurs de l'entreprise, et fournir des rapports d'utilisation.

Les serveurs mandataires ne sont pas de simples mandataires HTTP(S), mais implémentent un protocole de téléchargement spécifique et très simple [25] en surcouche de HTTP(S).

Divers projets libres et/ou commerciaux implémentent ce protocole :

- ✓ Le projet Go fournit un serveur de démonstration, sur https://osinet.fr/go/paquets/go-mandataire-exemple
- ✓ Le projet Athens, initié par Microsoft, est un projet Open Source d'installation très simple, notamment par Docker : https://docs.gomods.io
- ✓ Google héberge des services partagés, utilisés par défaut à partir de Go 1.13, dans le cadre d'une approche visant à sécuriser les téléchargement de modules [26] par défaut :
  - un serveur mandataire https://proxy.golang.org/
  - un serveur de sommes de contrôle https://sum.golang.org/
  - un index des nouvelles versions sur https://index.golang.org/

25. https://osinet.fr/go/paquets/go-protocole-telechargement
26. https://osinet.fr/go/paquets/go-securite-modules

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ L'éditeur JFrog héberge https://gocenter.io, et intègre les fonctions associées dans son offre commerciale Artifactory [27].
- ✓ L'éditeur Sonatype héberge https://gonexus.dev/ et intègre les fonctions associées dans son offre commerciale Nexus [28].

Enfin, le projet Go a introduit fin 2019 un nouveau serveur de découverte https://go.dev/ destiné à fournir une meilleure expérience de recherche, documentation et évaluation des modules publiés, susceptible de remplacer le serveur historique https://godoc.org/ bien que son positionnement soit plus complémentaire que concurrent, comme décrit sur https://osinet.fr/go/paquets/go-blog-discovery

9.10.5.6 Configuration

Le système de modules dans son ensemble est contrôlé par la variable d'environnement `GO111MODULE`, dont l'impact a évolué au fil des versions comme décrit au § 9.10.3.

Ses valeurs sont les suivantes :

- ✓ **`off`** : les modules sont désactivés, seul le mode `GOPATH` est utilisable.
- ✓ **`on`** : les modules sont nécessaires, et `GOPATH` n'est utilisé que pour la destination de stockage des modules téléchargés, comme décrit au § 9.10.4.2, et pour l'installation des commandes exécutables par `go install`. Il est ignoré pour toute autre opération.
- ✓ **Vide ou `auto`** : la reconnaissance des modules est automatique, ne s'activant que dans les répertoires contenant un fichier `go.mod` ou dont l'un des ancêtres en contient un.

Le système de serveurs mandataires pour les téléchargements est configurable par un jeu de variables d'environnement spécifiques :

- ✓ **`GOPROXY`** : définit comment les commandes de modules doivent se connecter aux serveurs, en listant les serveurs ou en spécifiant `direct` pour demander un téléchargement sans mandataire.
  En Go 1.13 les valeurs possibles sont une concaténation de chaînes d'URL de serveurs mandataires (*proxies*), et de la chaîne `direct` en dernière position, séparées par des virgules. Dans les versions antérieures, un seul serveur pouvait être désigné au lieu d'une liste à virgules. Les URL peuvent être de la forme `file:///chemin`.
  À chaque requête, Go tente le téléchargement en séquence depuis chacun des serveurs. En cas d'erreur de téléchargement HTTP 404 ou 410 depuis un serveur mandataire, Go passe au serveur suivant dans la liste.

27. https://osinet.fr/go/paquets/intro-artifactory
28. https://osinet.fr/go/paquets/intro-gonexus

La variable peut aussi porter la valeur réservée `off` pour interdire tout téléchargement.
Sa valeur par défaut, vide avant Go 1.13, est depuis lors :

https://proxy.golang.org/

- ✓ **GOPRIVATE** : une liste de préfixes de modules pour lesquels le téléchargement ne doit utiliser ni serveur mandataire, ni serveur de sommes de contrôle.
- ✓ **GONOPROXY** : une liste de préfixes de modules ne devant pas être téléchargés depuis les serveurs mandataires. Cette valeur est vide par défaut, et son contenu surcharge `GOPRIVATE`.
- ✓ **GOSUMDB** : le nom réservé `off` ou le nom d'une base de données de sommes de contrôle, pour les vérifications d'intégrité, depuis Go 1.13, éventuellement complété de sa clef publique et de son URL. Sa valeur par défaut est :

  `sum.golang.org`

  En République populaire de Chine, le nom `sum.golang.google.cn` désigne la même base. Dans tous les cas, utiliser `off` ou ajouter le drapeau `-insecure` aux commandes de modules désactive la vérification des sommes de contrôle, renonçant ce faisant aux garanties d'intégrité qu'elles apportent.
- ✓ **GONOSUMDB** : une liste de préfixes de modules ne devant pas faire l'objet d'une vérification depuis le serveur de sommes de contrôle. Cette valeur est vide par défaut, et son contenu surcharge `GOPRIVATE`.
- ✓ **GOINSECURE** : introduite en Go 1.14, cette ariable permet d'autoriser la commande `go` à ne pas exiger l'utilisation de HTTPS, et à ne pas vérifier les certificats présentés par les ser eurs HTTPS. À l'instar de `GOPRIVATE`, son format est une liste de motifs simples de modules pour lesquels cette exemption s'applique, séparés par des virgules.

Les valeurs autres que la valeur par défaut pour les paramètres autres que `GOPROXY` sont principalement utiles dans les projets d'entreprises, où un serveur mandataire privé comme Artifactory, Athens ou Nexus permet la sélection, le contrôle d'intégrité, et la traçabilité des modules utilisés. Ceci est particulièrement utile dans les démarches de conformité réglementaire (*compliance*).

Outre cette utilisation, la variable `GOPROXY` est aussi particulièrement utile pour les développeurs nomades, qui peuvent installer un serveur mandataire à stockage local comme Athens sur leur poste de travail pour accéder au téléchargement de modules même hors ligne, et pour toute structure souhaitant accélérer ses téléchargements par l'accès à un cache de modules local.

Programmation concurrente

10.1 CONCURRENCE ET PARALLÉLISME

L'un des objectifs lors de la création de Go était de tirer parti des nouvelles architectures multi-processeurs et multi-cœurs (cf. § 1.2), afin d'optimiser l'efficacité énergétique des machines modernes face à la fin de la progression des vitesses d'exécution scalaires en fil d'exécution unique.

La technique choisie pour atteindre cet objectif en Go a été la mise en œuvre d'un modèle de programmation concurrente basée sur les notions introduites par CSP (cf. § 1.1.2.4).

Cette notion de concurrence est souvent mal comprise et confondue avec le parallélisme :

- ✓ **Concurrence** : un mode d'écriture des applications dans lequel la fonctionnalité est découpée en tâches unitaires pouvant s'exécuter indépendamment de l'état des autres tâches.
  Ceci correspond le plus souvent, en Go, à des fonctions ne dépendant pas d'un état global mutable, et communiquant leurs résultats sur des canaux.
- ✓ **Parallélisme** : un mode d'exécution des applications dans lequel les tâches à exécuter le sont simultanément sur des processeurs ou cœurs différents.

Comme le montre cette description, la concurrence est un mode de conception, alors que le parallélisme est un mode d'exécution, que facilite la programmation concurrente, en démultipliant les tâches disponibles indépendantes.

L'exécution d'un programme concurrent n'a aucun besoin d'être parallèle.

Cette distinction permet de concevoir les programmes Go sans avoir à se préoccuper outre mesure de leur exécution parallèle, dès lors qu'ils sont conçus sur un

modèle concurrent. Elle permet ainsi le développement sur des postes de travail sans capacités particulières d'applications destinées à une exécution parallèle, y compris à grande échelle.

Le runtime Go est depuis ses origines capable de répartir ses unités d'exécution – les goroutines – sur des cœurs multiples pour une exécution parallèle.

La limite du nombre de cœurs à utiliser pour cette exécution est déterminée par la variable d'environnement `GOMAXPROCS`. Définie par défaut à 1 jusqu'en Go 1.4.x, cette variable est depuis Go 1.5 initialisée par défaut au nombre de cœurs disponibles lors de l'exécution.

Elle peut être dynamiquement contrôlée par les programmes, au moyen de fonctions du paquet `runtime`, décrites ci-après.

Il est utile de prendre en compte le fait que l'augmentation du nombre de cœurs affectés à l'exécution parallèle de tâches n'est pas nécessairement synonyme de réduction du temps total d'exécution du fait à la fois de la limite du nombre de tâches effectivement parallélisable, de la section séquentielle (cf. loi d'Amdahl, § 1.3.2 page 17), et du coût de fonctionnement de l'ordonnanceur.

10.2 GOROUTINES

10.2.1 Définition

Fonctionnellement, les goroutines sont :

✓ des fils d'exécution (*threads*) indépendants, exécutés concurremment,

✓ tous situés dans un même espace d'adressage.

La spécification de Go n'en dit pas plus, laissant toute latitude aux auteurs de compilateurs pour choisir des implémentations variables.

Dans l'implémentation officielle du projet Go, les goroutines sont multiplexées sur des fils d'exécution (*threads*) du système d'exploitation sous-jacent, utilisant au maximum un thread actif par cœur de processeur, dans la limite du nombre de cœurs autorisés définie par la variable d'environnement `GOMAXPROCS` décrite ci-dessus, et par son éventuelle surcharge au moyen de la fonction `runtime.GOMAXPROCS`.

Le nombre de threads pouvant être bloqués dans l'attente de retours d'appels système, ou affectés à des appels CGO, n'est en revanche pas limité.

À la différence des threads, elles ne sont connues que du runtime Go, et utilisent une pile de taille très limitée : 4 ko en Go 1.1, 8 ko en Go 1.2, et 2 ko depuis Go 1.3 [1]. La taille de la pile est augmentée selon les besoins par réallocation et copie depuis Go 1.3, après avoir utilisé dans les versions antérieures un mécanisme de pile segmentée, et constaté, comme le projet Rust, certains comportements pathologiques

1. La valeur en est définie par la constante `runtime._StackMin`, dans `src/runtime.stack.go`.

de cette approche [2]. De plus, leur instantiation et leur libération sont réalisées dans l'espace mémoire du processus, sans requérir de nouvelles ressources du système d'exploitation.

Enfin, la commutation n'est pas préemptive par interruptions, avec des tranches de temps prédéfinies, mais coopérative, réalisée par l'ordonnanceur du runtime Go dans diverses situations comme les opérations sur les canaux, l'instruction **go**, les appels système bloquants, ou la fin d'une passe de ramasse-miettes (*garbage collector*). Une préemption par l'ajout d'instructions de commutation dans les boucles (cf. § 7.3.8.4) est possible en utilisant une construction personnalisée du SDK Go avec l'option `GOEXPERIMENT=preemptibleloops`. Bien que cette possibilité existe depuis Go 1.8, elle n'a pas été généralisée, notamment du fait de son coût d'exécution sur les boucles serrées. Voir les *Plus en ligne* pour plus de détails sur l'ordonnanceur et le ramasse-miettes.

Ceci rend l'utilisation de goroutines extrêmement efficace en comparaison avec des threads, ou à plus forte raison avec des processus pour les systèmes où ils diffèrent des threads [3].

10.2.2 Utilisation

Les fonctions `init` et la fonction `main.main`, point de démarrage des programmes, sont exécutées dans une goroutine unique créée dans le prologue d'exécution du runtime Go.

Toutes les autres goroutines sont créées dans le code utilisateur par l'instruction **go**, détaillée au § 7.4.2, et de la forme :

```
go uneFonction(desArguments)
```

La fonction passée à l'instruction **go** peut indifféremment être une fonction définie, une méthode, ou une fonction anonyme. Dans ce dernier cas, une erreur courante est d'oublier d'invoquer la fonction à la fin de sa définition, provoquant une erreur de compilation.

Les éventuels résultats renvoyés par une fonction lancée dans une nouvelle goroutine de cette façon sont toujours ignorés. Pour cette raison, il est nécessaire que la plupart des goroutines disposent d'un moyen de communiquer leur résultat, autre que l'instruction **return**.

Deux grandes approches sont utilisées à cette fin :

- ✓ Passer un ou plusieurs paramètres de types canaux, qui permettent à la goroutine de recevoir des données et d'émettre des résultats en cours d'exécution. C'est la technique la plus générale.
- ✓ Dans le cas des fonctions anonymes seulement, utiliser des variables définies dans la portée englobante. C'est la technique la plus simple.

2. https://osinet.fr/go/concurrence/go-piles-contigues
3. Les deux sont équivalents dans le noyau Linux, représentés par des instances de `struct task_struct`, dans https://osinet.fr/go/concurrence/gh-torvalds-linux-task

© Dunod – Toute reproduction non autorisée est un délit.

Afin d'éviter les problèmes d'état mutable partagé qui peuvent survenir dans cette technique lorsque plusieurs goroutines, y compris la fonction créatrice, doivent coordonner leurs accès à ces variables, cette technique est généralement limitée au cas d'une unique goroutine sous le contrôle de sa fonction créatrice.
Par extension, des fonctions définies peuvent aussi recourir à des variables globales au paquet, voire d'autres paquets, mais cela ne fait qu'amplifier le problème, de sorte que cette approche n'est pas recommandable.

L'utilisation de fonctions anonymes est parfois préconisée pour utiliser la seconde technique et éviter les canaux, avec l'idée que la goroutine ainsi créée ne pourra pas avoir d'occurrences multiples, n'étant pas visible hors de sa fonction créatrice.

Toutefois, rien n'empêche que la fonction créatrice elle-même soit invoquée plusieurs fois, et vienne à créer d'autres occurrences de cette fonction, ne faisant que reporter le problème.

En outre, dans un test unitaire, invoquer des fonctions anonymes internes étant plus compliqué que d'invoquer des fonctions définies, cette pratique n'est pas optimale dans une perspective de qualité.

10.2.3 API

L'API disponible pour interroger et manipuler les goroutines est volontairement orientée vers les développeurs d'outillage, les développeurs de projets Go étant encouragés à traiter les goroutines comme une abstraction essentiellement illimitée dans la plupart des situations, et à ne pas interagir avec l'ordonnanceur (*scheduler*), dont l'implémentation change au fil des versions.

Ceci est en particulier mis en évidence par le fait qu'il n'existe aucune fonction exportée permettant d'accéder à un identificateur ou descripteur de goroutine, un point détaillé dans la FAQ Go : https://osinet.fr/go/concurrence/go-id-goroutine

Les mécanismes disponibles sont les suivants :

✓ **GODEBUG** : cette variable d'environnement peut comporter des paires de valeurs séparées par des virgules, deux de ces valeurs concernant les goroutines, `schedtrace` et `scheddetail` :

- `GODEBUG=schedtrace=MS`
  où `MS` est un nombre de millisecondes, provoque l'émission sur la sortie d'erreur d'informations sur l'état de l'ordonnanceur, comme dans le listing 10.1.
- `GODEBUG=schedtrace=MS,scheddetail=1`
  ajoute à l'affichage précédent des informations plus détaillées dont la liste des goroutines et leur état, comme dans le listing 10.2.
  Dans cette version détaillée, les lignes `Px` désignent les processeurs logiques (cœurs), les lignes `Mx` les threads du système d'exploitation, et les lignes `Gx` les goroutines.

Listing 10.1 – Informations de l'ordonnanceur avec `schedtrace`

```
SCHED 0ms: gomaxprocs=8 idleprocs=6 threads=3 spinningthreads=1
        idlethreads=0 runqueue=0 [0 0 0 0 0 0 0 0]
SCHED 8ms: gomaxprocs=8 idleprocs=8 threads=5 spinningthreads=0
        idlethreads=3 runqueue=0 [0 0 0 0 0 0 0 0]
SCHED 14ms: gomaxprocs=8 idleprocs=8 threads=5 spinningthreads=0
        idlethreads=3 runqueue=0 [0 0 0 0 0 0 0 0]
SCHED 24ms: gomaxprocs=8 idleprocs=8 threads=5 spinningthreads=0
        idlethreads=3 runqueue=0 [0 0 0 0 0 0 0 0]
```

Listing 10.2 – Informations sur les goroutines avec `schedtrace+scheddetail`

```
SCHED 0ms: gomaxprocs=8 idleprocs=5 threads=4 spinningthreads=1 idlethreads=0
       runqueue=0 gcwaiting=0 nmidlelocked=0 stopwait=0 sysmonwait=0
  P0: status=1 schedtick=0 syscalltick=0 m=3 runqsize=1 gfreecnt=0
  P1: status=1 schedtick=0 syscalltick=0 m=2 runqsize=0 gfreecnt=0
  P2: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
  P3: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
  P4: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
  P5: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
  P6: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
  P7: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
  M3: p=0 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0
        spinning=false blocked=false lockedg=-1
  M2: p=1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=2 dying=0
        spinning=false blocked=false lockedg=-1
  M1: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0
        spinning=false blocked=false lockedg=-1
  M0: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0
        spinning=false blocked=false lockedg=1
  G1: status=1(chan receive) m=-1 lockedm=0
  G2: status=1() m=-1 lockedm=-1
  G3: status=1() m=-1 lockedm=-1
  G4: status=4(GC scavenge wait) m=-1 lockedm=-1
```

- ✓ **`GOMAXPROCS`** : cette variable d'environnement définit le nombre maximal de threads actifs (non bloqués), qui ne peut dépasser le nombre de cœurs présents sur le système.
- ✓ **`GORACE`** : configure le fonctionnement du détecteur de situations de compétition (cf. § 10.7), le principal moyen d'identifier les problèmes liés au fonctionnement concurrent des goroutines.
- ✓ **`runtime.Caller()`**, **`runtime.Callers()`**, **`runtime.CallerFrames()`** : renvoient des informations détaillées sur l'état de la pile de la goroutine où elles sont invoquées.
- ✓ **`runtime.GC()`** : déclenche le ramasse-miettes de façon bloquante pour la goroutine qui l'invoque, et potentiellement pour l'ensemble du programme en cours.

© Dunod – Toute reproduction non autorisée est un délit.

- ✓ **`runtime.GOMAXPROCS()`** : permet de redéfinir le nombre maximal de threads actifs, initialisé par la variable d'environnement du même nom, et en renvoie la valeur précédente. Elle ne modifie pas cette valeur si son argument est négatif ou nul, et son exécution provoque une passe bloquante (*stop the world*) du ramasse-miettes.
  Lors de la compilation vers l'architecture Web Assembly (`GOARCH=wasm`), introduite en Go 1.11, ce nombre est automatiquement ramené à 1, dans l'attente d'une version multi-fils de cette architecture.
  L'instruction `runtime.GOMAXPROCS(runtime.NumCPU())`, très populaire aux débuts de Go, n'est plus utile depuis Go 1.5, qui exécute l'équivalent dans le prologue des programmes.
  Plus généralement, le – rare – intérêt de cette fonction est, au contraire, de ne pas utiliser la totalité des cœurs disponibles pour réserver des ressources entre processus coopérants, ou pour identifier durant un test de type `Benchmark` la limite de parallélisation.
  Elle est décrite comme destinée à disparaître dans une version future de Go, comme conséquence des améliorations de l'ordonnanceur (*scheduler*).
- ✓ **`runtime.Goexit()`** : met fin à la goroutine en cours, après avoir exécuté le code différé qu'elle pouvait contenir. Elle ne crée pas une situation de panique, et ne renvoie pas de résultat, comme détaillé au § 9.5 à propos du listing 9.2, page 302.
- ✓ **`runtime.Gosched()`** : cède le contrôle de l'exécution, autorisant l'ordonnanceur à exécuter une autre goroutine, puis reprend l'exécution lorsque l'ordonnanceur réactive la routine concernée.
  Ce comportement est comparable au `Thread.yield` de Java ou .NET, à la différence que ceux-ci permettent la commutation de threads et non de goroutines.
  L'ordonnanceur déclenchant des commutations dans de nombreuses situations, comme les appels de fonctions, ou les instructions `select`, son principal cas d'utilisation concerne les fonctions sans appels à d'autres fonctions, et ne tirant pas parti d'un découpage concurrent, comme ce peut être le cas en calcul numérique.
  Elle peut aussi être utile pour réaliser une boucle d'attente infinie qui ne pénalise pas l'ordonnanceur, de la forme :

```
for { runtime.Gosched() }
```

- ✓ **`runtime.LockOSThread()`**, **`runtime.UnlockOSThread()`** : demandent respectivement à l'ordonnanceur de ne pas déplacer la goroutine appelante vers un autre thread, et de mettre fin à cette restriction.
  Ce verrouillage est destiné aux goroutines qui invoquent des fonctions de code non-Go dépendant de l'état mémoire de leur thread.

- ✓ **`runtime.NumCPU()`** : renvoie le nombre de cœurs disponibles lors de l'initialisation du programme. Ce nombre n'est pas mis à jour par la suite même si le système d'exploitation modifie dynamiquement ce nombre.
- ✓ **`runtime.NumGoroutine()`** : renvoie le nombre de goroutines existantes, quel que soit leur état.
- ✓ **`runtime.Stack()`** : formate une représentation lisible de la pile d'appels (*stack trace*) de la goroutine en cours dans une tranche d'octets *préallouée*, et renvoie la longueur du résultat, comme illustré dans le listing 10.3.
  C'est le format utilisé par le runtime Go dans l'affichage final lors des situations de panique.

D'autres mécanismes existent dans le paquet `runtime`, mais sont encore moins utiles hormis pour les développeurs du projet Go lui-même et les développeurs d'outils de développement.

Aucune de ces fonctions n'est destinée à un usage courant : comme le montrent leurs descriptions, elles sont principalement utiles en situation de déboguage, ou pour des besoins très spécifiques.

Listing 10.3 – Affichage de la pile d'une goroutine
https://osinet.fr/go/concurrence/play-pile-goroutine

```
1  package main
2
3  import "runtime"
4
5  func f2(n1, n2 int) {
6          tr := make([]byte, 1<<20)          // Allouer le tampon
7          count := runtime.Stack(tr, false) // Goroutine courante
8          stack := string(tr[:count])        // Limité au résultat
9          println(stack)
10 }
11
12 func f1(n int) { f2(n, 7) }
13 func main() { f1(5) }
14
15 // Résultat
16 // goroutine 1 [running]:
17 // main.f2(0x5, 0x7)
18 //          /tmp/sandbox958066379/prog.go:7 +0x80
19 // main.f1(...)
20 //          /tmp/sandbox958066379/prog.go:12
21 // main.main()
22 //          /tmp/sandbox958066379/prog.go:13 +0x40
```

© Dunod – Toute reproduction non autorisée est un délit.

10.2.4 Bonnes pratiques

Diverses bonnes pratiques d'utilisation des goroutines ont émergé à l'usage :

✓ Vérifier les limites de mémoire disponible et de nombre de descripteurs de fichiers disponibles dans les situations où un nombre important de goroutines sont susceptibles d'être créées.
La définition d'un nombre important n'est pas explicite, mais pour fixer les idées : quelques dizaines n'est pas un nombre important, alors qu'un million l'est.

✓ Dans le cas de multiples goroutines à exécution potentiellement longue, et sans point de déclenchement de l'ordonnanceur – ce qui devrait être rare – appliquer au moins l'une des techniques suivantes :
- invoquer `runtime.Gosched` pour permettre une commutation ;
- limiter le nombre de goroutines dans cette situation à un nombre strictement inférieur au nombre de threads autorisés, pour permettre à la goroutine parente de disposer d'un thread d'exécution et éviter le blocage du processus.
Ce nombre est disponible via `runtime.GOMAXPROCS(0)`, plutôt que par `runtime.NumCPU()`, qui n'en est que la limite supérieure lors du lancement du programme.

✓ Ne pas créer une goroutine à l'intérieur d'une fonction pour renvoyer ses résultats dans un canal : ceci contraint l'appelant à lire l'ensemble des résultats, la goroutine créée n'ayant pas de moyen de savoir si l'appelant a déjà obtenu les résultats souhaités, et devant donc produire l'ensemble de ses résultats sans pouvoir terminer par anticipation.
Un mécanisme comme le motif Visiteur [4], à l'image de `(sync.Map).Range` (cf. § 10.4.5), permet en revanche une telle interruption par le consommateur.
Plus généralement, il est préférable de laisser au consommateur d'une fonctionnalité le choix de l'exécuter en mode concurrent ou séquentiel [5].

✓ Puisque l'exécution d'une goroutine est autonome, prévoir un moyen de notifier sa fin d'exécution à son créateur ou, inversement, fournir à son créateur un moyen de demander à la goroutine de se terminer.
En l'absence d'API permettant de manipuler les goroutines directement, la technique éprouvée consiste à passer aux fonctions utilisées comme goroutines des canaux de pure signalisation, à contenu de longueur 0, par exemple de type `chan struct{}`.

Cette dernière technique a été formalisée à partir de Go 1.7 par l'inclusion dans le runtime Go du paquet `context`, précédemment apparu à l'époque de Go 1.3 dans le paquet des bibliothèques supplémentaires `golang.org/x/net/context`.

4. https://osinet.fr/go/concurrence/wiki-visiteur
5. Voir notamment la présentation de Dave Cheney à QCon 2019 : https://osinet.fr/go/concurrence/cheney concurrence

10.3 CONTEXTES

10.3.1 Définition

Comme évoqué dans la rubrique précédente, il est généralement nécessaire de pouvoir contrôler l'exécution d'une goroutine, que ce soit pour en limiter la durée ou l'interrompre à volonté, et ce, bien qu'elle soit, après son lancement, totalement indépendante de la goroutine qui l'a créée après son lancement.

Un cas d'utilisation particulièrement notable est celui des contrôleurs dans les projets de services web (implémentations de l'interface `http.Handler`), qui sont lancés sous forme de goroutines indépendantes par les mécanismes de serveurs HTTP [6].

Pour simplifier ceux-ci et maximiser leur testabilité, ils sont généralement décorés par des médiateurs (*intergiciels* en français du Québec, *middlewares* en anglais), utilisés pour fournir :

- ✓ Des services initialisés au lancement, comme un journaliseur (*logger*) ou une connexion de base de données, passés aux handlers par la portée englobante de leur définition lors de la mise en place du routage.
- ✓ Des données spécifiques à la requête servie par une goroutine de *handler*, comme un jeton d'authentification à usage unique *nonce*, qui ne peuvent être produites que lors de la requête elle-même.

Or la signature des handlers ne comporte qu'une destination d'écriture et l'objet requête, et n'accepte aucun paramètre additionnel, du fait de la rigidité des types fonction quant à la signature.

De même, il est fréquent qu'un service web préfère garantir un temps de réponse limité, au prix de résultats en échec, plutôt que de maintenir une connexion avec son client sur une longue durée, et l'indépendance des goroutines ne fournit pas de moyen de les terminer pour revenir par anticipation.

Les contextes d'exécution du paquet `context` apportent une réponse à ces besoins. Par extension, il sont aussi utilisables à tout traitement susceptible d'avoir une durée suffisante pour nécessiter une annulation, comme la consommation d'API web et les opérations de base de données ou de file d'attente.

Ils sont définis par le type interface `Context`, qui spécifie quatre méthodes :

- ✓ **`Deadline() (deadline time.Time, ok bool)`** : renvoie l'instant où le traitement utilisant le contexte devra se terminer, et un booléen indiquant si cette limite existe, ou si le traitement n'a pas de durée limite.
  Ceci permet à un traitement à durée variable, comme un calcul temps réel qui améliore progressivement la précision de son résultat, de savoir quand il lui est préférable de renvoyer un résultat plutôt que de continuer à affiner son calcul.

6. Voir la méthode `Serve()` du type `net/http.Server`.

© Dunod – Toute reproduction non autorisée est un délit.

✓ **`Done() <-chan struct`** : renvoie un canal en réception, sur lequel rien n'est jamais émis.
Ce canal peut être **`nil`** si le contexte n'est pas annulable. Sinon, il sera fermé par le créateur du contexte lorsque les goroutines utilisant le contexte devront s'interrompre.
Les canaux fermés n'étant pas bloquants, leur utilisation dans un **`select`** par les goroutines concernées fournit une signalisation partageable entre de multiples goroutines pour leur indiquer que leur créateur souhaite les voir se terminer [7].

✓ **`Err() error`** : renvoie la raison pour laquelle le canal de `Done()` a été fermé :
- **`nil`** s'il n'est pas encore fermé ;
- `context.Canceled` si l'annulation du contexte a été demandée ;
- `context.DeadlineExceeded` si la durée d'exécution allouée au contexte est dépassée.

✓ **`Value(key interface) interface`** : fournit un accès de type clef/valeur, renvoyant **`nil`** pour les clefs inexistantes. Ce format implique en particulier qu'il n'est pas possible de différencier l'absence de résultat pour une clef donnée de la présence pour cette clef d'une valeur **`nil`**, à la différence de l'utilisation d'un type carte.
Le type des clefs est libre pourvu que ses valeurs soient comparables pour l'égalité ; il n'est pas nécessaire qu'elles le soient pour l'ordre.

10.3.2 Utilisation

Les règles d'utilisation des contextes ne se limitent pas à l'API, mais aussi à leur mode d'utilisation :

✓ Toute requête reçue du réseau doit établir un contexte, du type interface `context.Context` aussi tôt que possible, en utilisant la valeur renvoyée par `context.Background()`.
Ceci ne concerne pas seulement les requêtes HTTP, mais toute autre requête sur un protocole quelconque, comme SMTP [8], SSH [9], ou tout service personnalisé réalisé sur TCP/IP ou UDP/IP avec le paquet `net` de la bibliothèque standard.

✓ Toute requête émise vers un service réseau doit comporter un paramètre de type `context.Context`.
Si la requête sortante est créée dans le cadre d'une requête entrante, le contexte passé doit être créé à partir du contexte reçu par celle-ci.

✓ Pour permettre le bon fonctionnement des outils d'analyse statique, les contextes doivent être utilisés comme paramètres indépendants, plutôt que d'être utilisés comme des champs d'un type structure.

7. L'article https://osinet.fr/go/concurrence/blog-pipelines présente d'autres façons d'utiliser ce type de signalisation.
8. https://osinet.fr/go/concurrence/gh-guerrilla
9. https://osinet.fr/go/concurrence/gopheracademy-ssh

✓ Par convention, toute fonction acceptant un argument `context.Context` doit l'accepter comme premier paramètre, avec l'identificateur `ctx` [10].

✓ Bien que **`nil`** soit une valeur valide pour le type `context.Context` puisqu'il s'agit d'un type interface, il n'est pas souhaitable de passer **`nil`** comme argument pour un paramètre de ce type.
Si le contenu exact du contexte n'a pas encore été identifié au cours d'un développement non finalisé, utiliser le contexte standard `context.TODO` prévu à cet effet.

✓ Les contextes ne sont pas destinés à être modifiés : pour ajouter des propriétés à un contexte reçu, il faut en créer un nouveau avec l'une des fonctions spécialisées :

- **`WithCancel(Context) (Context, CancelFunc)`** : renvoie un nouveau contexte basé sur son premier argument, mais avec un canal `Done` initialisé.
  La fonction renvoyée comme second résultat permet de fermer ce canal, signalant à tous les contextes dérivés qu'ils doivent se terminer.
- **`WithDeadline(ctx Context, d time.Time)(Context,CancelFunc)`** renvoie un nouveau contexte basé sur son premier argument, mais avec un horodatage limite.
  Si le canal reçu comportait déjà un horodatage limite, et que le paramètre reçu est ultérieur à l'existant, celui-ci n'est pas modifié. En revanche, s'il est antérieur à l'existant, il se substitue à l'existant dans le canal renvoyé.
  La fonction renvoyée comme second résultat permet de fermer ce canal, signalant à tous les contextes dérivés qu'ils doivent se terminer. La fermeture peut également être déclenchée par la fermeture du canal `Done` du contexte parent `ctx`, ou par le fait que l'instant désigné par le paramètre d'horodatage soit atteint.
- **`WithTimeout(Context, time.Duration) (Context, CancelFunc)`** : est équivalente à `WithDeadline`, mais avec une durée exprimée à partir de l'instant où elle est invoquée au lieu d'un horodatage fixe.
  Son utilisation est identique.
- **`WithValue(ctx Context, key, val interface{}) Context`** : est différente des trois précédentes, en ce qu'elle n'ajoute pas de signaux d'annulation mais permet d'ajouter des valeurs au contexte.
  Ces valeurs sont supposées être uniquement celles qui sont spécifiques à la requête en cours, et non des données persistantes définies lors de l'initialisation des services. Pour ces dernières, utiliser de préférence d'autres mécanismes, le plus souvent des médiateurs, comme décrit au § 10.3.1.

10. Dès leur introduction dans Go 1.7 en 2014, Google requérait que les programmeurs Go passent un `context.Context` comme premier argument de toute fonction située sur le chemin d'exécution entre entrée et sortie de requête, pour faciliter l'interopérabilité de telles fonctions, et garantir un mécanisme commun d'annulation, de limite de durée et de transmission des données de sécurité.
Source : https://osinet.fr/go/concurrence/go-context

© Dunod – Toute reproduction non autorisée est un délit.

Dès lors qu'un contexte comporte un canal `Done` d'expiration, il est nécessaire d'invoquer la fonction reçue comme second résultat des fonctions suivantes :

- ✓ `context.WithCancel`,
- ✓ `context.WithDeadline`,
- ✓ `context.WithTimeout`.

En effet, leur utilisation crée des goroutines qui vont maintenir des références en mémoire durant toute leur existence, et empêcheront la récupération mémoire par le ramasse-miettes si elle ne sont pas terminées par l'invocation de cette fonction d'annulation ou l'arrivée à l'horodatage limite s'il est défini, chacun de ces événements invoquant récursivement l'annulation de tous les contextes dérivés.

Enfin, divers paquets autres que `context` proposent des fonctions permettant de produire des contextes dérivés ou d'utiliser des contextes, parmi lequels certains peuvent servir d'inspiration :

- ✓ **`WithLabels(context.Context, LabelSet) context.Context`** : cette fonction du paquet `pprof` [11] dérive un nouveau contexte avec des étiquettes de profilage.
- ✓ **`WithClientTrace(context.Context, *ClientTrace) context.Context`** cette fonction du paquet `httptrace` [12] dérive un nouveau contexte complétant une trace de requête de serveur à client, pour faciliter la production de traces en environnement de services répartis.
- ✓ **`http.TimeoutHandler(Handler, time.Duration, string) Handler`** cette fonction est un *médiateur* du paquet `net/http` [13] qui décore un `http.Handler` préexistant par une limitation de durée d'exécution.
  La rubrique suivante fournit un exemple de son utilisation, et illustre ce faisant l'API des contextes.

10.3.3 Exemple : annulation de handler HTTP

L'exemple du listing 10.4 illustre trois approches basées sur les contextes pour réaliser un *handler* HTTP réagissant à un contexte d'exécution limitée dans le temps, produit à la ligne 46 à partir d'un handler quelconque en lui ajoutant un délai maximum d'exécution et un message en cas de dépassement de délai. Cette modification est réalisée au moyen de la fonction `http.TimeoutHandler`.

Ce médiateur intercepte le flux de réponse du *handler* d'origine, et exécute celui-ci dans une goroutine.

11. https://osinet.fr/go/concurrence/go-pprof
12. https://osinet.fr/go/concurrence/go-httptrace
13. https://osinet.fr/go/concurrence/go-http

Listing 10.4 – Trois approches pour un handler HTTP à durée limitée
https://osinet.fr/go/concurrence/play-handler-temporise
(code non exécutable sur le playground Go)

```
package main

import ( "fmt"; "io"; "net/http"; "strconv"; "time" )

func affiche(w io.Writer, msg string) {
        fmt.Println(msg)      // Console serveur
        fmt.Fprintln(w, msg) // Réponse HTTP
}

func travailSimulé(w io.Writer, i int) {
        if i%20 == 0 {
                time.Sleep(time.Microsecond)
                w.Write([]byte(strconv.Itoa(i) + " "))
        }
}

func rootHandler(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context() // Non-nil ici.
        done := ctx.Done()
        deadLine, _ := ctx.Deadline() // Deadline existe.
        // Marge de sécurité avant la limite effective.
        deadLine = deadLine.Add(- time.Millisecond)
        for i := 1; i <= 1E5; i++ {
                // Approche proactive: vérifier l'échéance.
                if time.Now().After(deadLine) {
                        affiche(w, "Délai atteint")
                        return
                }

                // Approche réactive: détecter l'expiration.
                select {       // Contrôle non bloquant
                case <-done:  // du canal done...
                        affiche(w, "Délai dépassé")
                        return
                default:       // ...grâce au "default".
                }
                travailSimulé(w, i)
        }
        affiche(w, "Calcul terminé")
}
```

© Dunod – Toute reproduction non autorisée est un délit.

```
42 func main() {
43         rh := http.HandlerFunc(rootHandler)
44         const délai = 5 * time.Millisecond
45         message := fmt.Sprintf("Délai de %v expiré\n", délai)
46         http.Handle("/", http.TimeoutHandler(rh, délai, message))
47         http.ListenAndServe(":8000", nil)
48 }
```

Attention ! Compte tenu des restrictions du playground Go, l'exemple ne peut y être exécuté : le lien n'est fourni que pour faciliter le copier/coller.

S'il se termine dans les délais, son résultat est alors copié sur le flux de réponse et le médiateur revient avec succès. À défaut, il se termine à l'expiration du délai, pendant que le *handler* d'origine poursuit son exécution et émet ses résultats vers le *handler* du médiateur, qui les ignorera.

✓ La version telle que proposée par le listing 10.4 est proactive, et a connaissance de l'existence du délai.
Elle calcule (ligne 20) l'échéance de fin de calcul et prend une marge de sécurité pour terminer avant cette échéance (lignes 21, 22), puis à chaque itération de son travail propre, elle vérifie si ce délai est atteint, et termine d'elle-même son exécution avant la limite de délai.
Dans ce mode, le client HTTP recevra une réponse `200` et les résultats calculés dans l'intervalle de temps alloué.
Cette méthode est bien appropriée aux services non répliqués avec une limite de temps de réponse, comme les rendus graphiques interactifs, ou la négociation en temps réel.

✓ Dans une approche inverse, en supprimant l'ensemble de l'interprétation du contexte (lignes 18-22, 24-36), on retombe sur un *handler* naïf ignorant totalement l'existence de la limitation de durée d'exécution.
C'est la situation dans laquelle l'application utilise sans le modifier un *handler* existant, tout en garantissant une durée maximale d'exécution.
Dans ce cas, la logique de la fonction `TimeoutHandler` joue à plein : le *handler* d'origine s'exécute jusqu'à sa fin normale, mais la réponse est fournie par le médiateur en temps garanti.
Si le résultat est atteint avant la limite de délai, le client HTTP reçoit une réponse `200` et les résultats complets. Sinon, ses résultats sont abandonnés, et le client HTTP reçoit une réponse `503` avec le message `Délai de 5ms expiré` fourni par le médiateur.

✓ En ne commentant que les lignes 20-22 et 24-28, le handler adopte une approche réactive : il détecte la fermeture du canal `Done` du contexte, qui lui indique que son temps de travail est expiré, et se termine à ce stade.
L'instruction `select` exécute la clause `default` tant que le délai n'est pas expiré, et abandonne dès qu'il est atteint.
Du point de vue du client HTTP, le comportement est le même que dans le cas du handler naïf. En revanche, du point de vue du serveur, cette approche est préférable, la charge CPU du handler se terminant dès l'expiration du délai.

La dernière méthode peut servir dans le cas de services répliqués, dans lesquels un client va instancier de multiples demandes à des serveurs secondaires, et renvoyer le premier résultat obtenu, utilisant le résultat du serveur le plus rapide car le moins chargé, tandis que les serveurs plus chargés et donc plus lents abandonneront du fait de la lenteur.

Ce n'est toutefois *pas* la meilleure façon de traiter ce cas, mais uniquement la plus simple pour démontrer la détection d'annulation du canal.

10.4 PRIMITIVES DE SYNCHRONISATION : LE PAQUET `sync`

La triade (canaux, goroutines, `select`) est conçue pour être l'outil de travail quotidien des développeurs d'applications concurrentes.

C'est le mécanisme par défaut, suffisant pour le plus grand nombre de cas ; non pour sa performance absolue, mais parce qu'il est le plus facile d'emploi pour produire des programmes concurrents corrects avec le plus faible risque d'erreurs.

Son utilisation dans certains algorithmes concurrents amène fréquemment à devoir traiter deux cas particuliers : le besoin d'exécution une fois au plus, et l'attente de fin d'exécution d'un groupe de goroutines, à la façon de `Promise.all()` pour une promesse JavaScript. Pour ces deux cas, le paquet `sync` comporte les types `sync.Once` et `sync.WaitGroup`.

Pour des utilisations un peu plus avancées, notamment lorsque des questions de performance sous la microseconde entrent en considération – ce qui n'est pas le cas de la vaste majorité des microservices, et encore moins des sites web – le recours à des mécanismes plus traditionnels à base de verrouillage sont disponibles sous la forme des types `sync.Mutex` et `sync.RWMutex`.

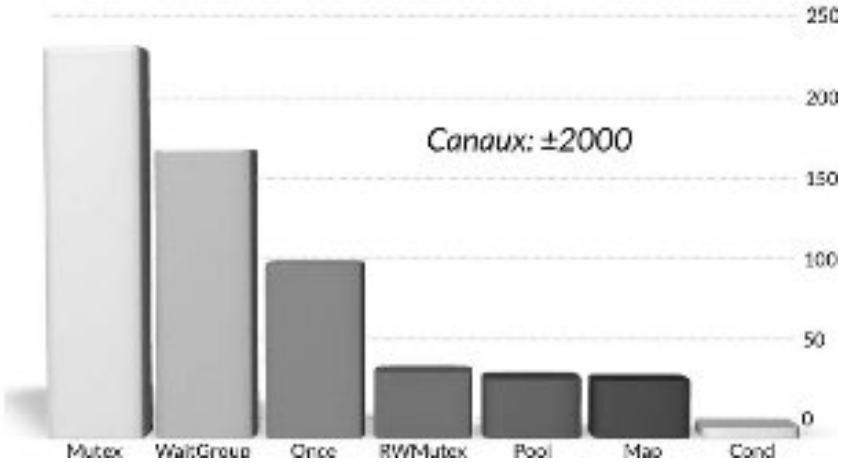
Enfin, pour un nombre limité de situations, d'autres mécanismes plus spécialisés sont inclus dans ce paquet : `sync.Pool` pour limiter la pression sur le ramasse-miettes, `sync.Map` comme une alternative aux types cartes natifs gérant correctement les accès concurrents, et, d'utilisation encore plus rare, les variables de conditions `sync.Cond` utilisées pour l'implémentation du mécanisme de moniteurs [14].

14. https://osinet.fr/go/concurrence/wiki-moniteur. La page équivalente dans la Wikipedia française est beaucoup moins précise.

© Dunod – Toute reproduction non autorisée est un délit.

La figure 10.1 représente le nombre d'occurrences de chacun de ces types dans le SDK Go 1.13, qui est représentatif d'une base de code Go mature et optimisée : une très vaste utilisation des canaux, près de dix fois plus importante que celle des `Mutex`, un bon nombre de `WaitGroup` et `Once`, puis des utilisations progressivement plus rares des autres types.

Figure 10.1 – Utilisation des types du paquet `sync` dans le SDK Go 1.13



Ces types ont en commun d'avoir des valeurs vierges directement prêtes à l'emploi – à l'exception de `Pool` et `Cond` – et de ne pas pouvoir être copiés après un début d'utilisation, ce qui interdit aux types structures qui les embarquent comme champs d'avoir des méthodes à récepteur valeur, dont l'appel a pour effet de réaliser une copie du récepteur.

10.4.1 Groupe des tâches : `sync.WaitGroup`

Le type `sync.WaitGroup` est une structure opaque, permettant d'attendre la fin d'un ensemble de goroutines sans utiliser manuellement la combinaison canal et **`select`**.

Le fonctionnement est très intuitif, comme démontré par le listing 10.5 :

1. La fonction englobante déclare (ligne 31) une variable de type `sync.WaitGroup`, dont la valeur vierge est directement utilisable, comme recommandé au § 5.4.3.
2. Elle lance de multiples goroutines et, pour chacune d'entre elles, incrémente le nombre de goroutines du groupe d'attente, en invoquant la méthode `Add()` du groupe, comme à la ligne 33.
3. Chaque goroutine indique le fait qu'elle se termine au groupe d'attente, en invoquant la méthode `Done` du groupe.

Pour s'assurer d'invoquer la méthode dans tous les cas, il est d'usage d'invoquer la méthode dans un **`defer`**, comme à la ligne 17.

4. L'appelant, lorsqu'il est prêt à attendre la fin de l'ensemble des goroutines, invoque la méthode `Wait()` du groupe, comme à la ligne 36.

Listing 10.5 – Utilisation de `sync.WaitGroup`
https://osinet.fr/go/concurrence/play-waitgroup

```
10 type routine func(*sync.WaitGroup)
11
12 func créeTâches(max int) []routine {
13         tasks := make([]routine, max)
14         for i := 0; i < max; i++ {
15                 i := i // Explication dans le texte.
16                 tasks[i] = func(wg *sync.WaitGroup) {
17                         msec := time.Duration(rand.Intn(max))
18                         time.Sleep(msec * time.Millisecond)
19                         fmt.Printf("%c ", 'a'+i)
20                         wg.Done()
21                 }
22         }
23         return tasks
24 }
25
26 func main() {
27         const n = 26
28         tâches := créeTâches(n)
29         t0 := time.Now()
30
31         var wg sync.WaitGroup
32         for i := 0; i < n; i++ {
33                 wg.Add(1)
34                 go tâches[i](&wg)
35         }
36         wg.Wait()
37
38         délai := time.Now().Sub(t0)
39         fmt.Printf("\nDélai total: %v\n", délai)
40 }
```

Comme le montre une exécution répétée de cet exemple, sa durée tourne autour de 26 millisecondes, soit la durée de la plus longue des goroutines : avec une écriture très lisible, le programme a attendu l'exécution des 26 tâches concurrentes avant de poursuivre son exécution.

© Dunod – Toute reproduction non autorisée est un délit.

Pourquoi `i := i` ?

L'instruction potentiellement la plus surprenante du listing 10.5 n'est pas liée aux `WaitGroup` eux-mêmes, mais est le `i := i` à la ligne 15 :

Dans une boucle comme celle de la fonction `créerTâches`, en l'absence d'une telle instruction, la valeur de `i` à la ligne 19 est celle de la variable dans la portée englobante qui, à la fin de la boucle, est égale à la valeur maximale… dans chacune des goroutines, ce qui n'est pas l'effet recherché.

Pour éviter ce phénomène, l'instruction ajoutée utilise une déclaration courte pour créer une nouvelle variable dans la portée du **`for`**, qui masquera la variable de la portée englobante une fois initialisée.

Le fait de la nommer de la même façon tire parti de l'ordre d'évaluation (cf. § 7.2.5.3), évaluant d'abord chaque identificateur avant de procéder à l'affectation, obtenant ainsi la valeur extérieure pour l'affecter à la valeur de la portée locale.

10.4.2 Exécution unique : `Once`

Un autre besoin courant pour ce qui est des exécutions concurrentes est le fait de n'exécuter un traitement qu'une seule fois.

Le paquet `sync` inclut à cet effet le type `sync.Once`, utilisé comme dans le listing 10.6 :

1. Placer le traitement à protéger dans une fonction de signature **`func`**`()`.
2. Allouer une variable de type `sync.Once`, dont la valeur vierge est directement utilisable, comme recommandé au § 5.4.3.
3. Invoquer le traitement autant de fois que souhaité par la méthode `Do` de la variable de type `sync.Once`, comme à la ligne 11 du listing : l'action ne sera exécutée qu'une seule fois.

Listing 10.6 – Utilisation de `sync.Once`
https://osinet.fr/go/concurrence/play-once

```
10          for i := 0; i < max; i++ {
11                  go func() {
12                          unique.Do(action)
13                          fini <- struct{}{}
14                  }()
15          }
16          // Vérifier l'exécution des multiples passes
17          for i := 0; i < max; i++ {
18                  println("Passe", i)
19                  <-fini
20          }
21  }
```

Cette situation ne nécessiterait pas un type spécialisé dans le cas de programmation séquentielle, et en ne prenant pas en compte les erreurs.

Mais les handlers de services réseau, en particulier, sont des goroutines indépendantes, de sorte qu'il peut être nécessaire de coordonner leur exécution de certains traitements à usage unique comme la compilation de modèles (*templates*) à la volée lorsqu'ils ne sont pas compilés durant l'initialisation.

De même, `Once` gère correctement le cas où le traitement provoque une situation de panique : si la panique est récupérée, évitant la fin du programme, l'action est bien considérée comme ayant été exécutée, et ne sera donc pas rejouée.

10.4.3 Exclusion mutuelle : `Mutex / RWMutex`

« *Channels orchestrate ; mutexes serialize.* »

Rob Pike - *Go Proverbs*
https://osinet.fr/go/concurrence/yt-pike2015-canaux

Un degré d'abstraction plus bas que les types vus précédemment, le paquet `sync` comporte une interface et deux types de verrous d'exclusion mutuelle, de facture classique, décrits par le tableau 10.1 :

- ✓ **`sync.Mutex`** : un verrou simple traditionnel, à un seul mode d'accès.
- ✓ **`sync.RWMutex`** : un verrou séparant l'accès en lecture de l'accès en écriture afin de permettre la combinaison d'un accès concurrent en lecture avec un accès synchronisé en écriture.
- ✓ **`sync.Locker`** : un type interface limité aux méthodes `Lock()` et `Unlock()`.

Tableau 10.1 – Méthodes des mutexes

	Mutex	RWMutex	
`Lock()`	✓	✓	acquiert un verrou exclusif
`RLock()`		✓	acquiert un verrou en lecture
`Unlock()`	✓	✓	libère un verrou exclusif acquis
`RUnlock()`		✓	libère un verrou en lecture acquis
`RLocker()`		✓	traite un verrou en lecture comme un verrou exclusif

La méthode `RWMutex.RLocker` renvoie son récepteur `RWMutex` converti en un type interface `Locker` implémenté par un type dynamique anonyme dont la méthode `Lock` effectue un `RLock` sur le `RWMutex` d'origine, et de même pour `Unlock`.

© Dunod – Toute reproduction non autorisée est un délit.

Les méthodes `Lock` et `RLock` sont bloquantes sans limitation de durée, jusqu'à réussite. En cas de tentatives de verrouillage concurrentes introduisant un interblocage (étreinte mortelle, *deadlock*) [15], le runtime le détecte et place le code dans une situation de panique.

Leur utilisation se déroule à la base comme suit :

1. Déclarer une variable `Mutex` ou `RWMutex` dans la fonction devant effectuer un accès concurrent à une donnée.
   Sa valeur vierge est un mutex non verrouillé, directement utilisable, comme recommandé au § 5.4.3.
2. Acquérir un verrou exclusif ou en lecture selon le besoin, avec `Lock` ou `RLock`.
3. Effectuer l'accès protégé.
4. Libérer le verrou par `RLock` ou `RUnlock`.

Comme tous les types du paquet `sync`, les mutexes ne doivent jamais être copiés pour être réutilisés : ils doivent être créés localement dans la fonction qui les utilise, comme indiqué ci-dessus.

Les verrous acquis par les mutexes ne sont pas associés à la goroutine qui les a créés : dans certains algorithmes concurrents, une goroutine peut relâcher un verrou acquis par une autre.

C'est précisément là, au niveau algorithmique, que se situe la difficulté d'une bonne utilisation des mutexes : dès lors qu'entrent en jeu des tâches plus complexes que la protection de l'accès à un champ d'une variable partagée, la complexité intrinsèque de la programmation concurrente, masquée par les mécanismes plus simples abordés jusqu'ici, réapparaît.

C'est la raison pour laquelle il est préférable de ne pas utiliser les mutexes en première intention dans la plupart des cas, mais d'organiser le code autant que possible avec canaux, goroutines, et `select`, complétés si besoin par des `WaitGroup` et `Once`. Le proverbe cité au début de cette rubrique fournit une idée générale des situations dans lesquelles utiliser les mutexes plutôt que des mécanismes de plus haut niveau.

Deux outils sont particulièrement utiles pour travailler sur le code intégrant des mutexes :

✓ Le détecteur de situations de compétition (cf. § 10.7), pour identifier les erreurs d'accès non protégé, indiquant par là même où l'utilisation de mutexes peut se révéler nécessaire.

✓ Le profil prédéfini `mutex` du profileur Go `pprof`, pour analyser les problèmes de contentions sur les verrous acquis par les mutexes. L'utilisation de ce profil est décrite sur : https://osinet.fr/go/concurrence/go-profilage

15. https://osinet.fr/go/concurrence/wiki-interblocage

10.4.4 Allocations réutilisées : Pool

Le type sync.Pool est destiné à l'optimisation de la gestion mémoire, que ce soit lors d'accès concurrents ou pas.

En effet, il arrive fréquemment qu'un algorithme ait besoin de traiter à répétition des instances d'un même type pour une durée courte, et ne les libère que pour en instancier d'autres : qu'il suffise de penser aux algorithmes de manipulation de graphes et autres structures de conteneurs, ou aux données de formatage et analyse élémentaires dans les fonctions des groupes fmt.Print et fmt.Scan.

Ceci accroît inutilement la charge de l'allocateur et du ramasse-miettes, puisqu'ils doivent en permanence allouer et recycler des zones de mémoire de taille constante, qu'il serait plus efficace de réutiliser, y compris entre goroutines multiples.

C'est précisément ce service que fournit le type sync.Pool, introduit en Go 1.3 et amélioré côté ramasse-miettes en Go 1.13.

À la différence de la plupart des autres types du paquet sync, celui-ci nécessite une initialisation. En effet, puisque son rôle est de fournir des objets initialisés, il a besoin de disposer d'une fonction initialisant ces éléments.

Pool est un type structure avec un seul champ New, de type **func** () **interface**{} : il s'agit bien là d'un champ fonction et non d'une méthode (cf. § 3.4.4).

Le type de retour de New ne doit pas faire illusion : bien que le type des objects qu'il alloue soit libre, il doit toujours renvoyer des objets du même type pour un Pool donné, puisque ceux-ci seront recyclés ou réutilisés selon la stratégie – non spécifiée – du Pool.

L'utilisation se déroule comme suit :

1. Initialiser un pool p en lui fournissant une fonction d'allocation dans le champ New. Puisque ces fonctions sont des allocateurs pour un type d'élément prédéfini, elles sont couramment de la forme :

   ```
   func () interface{} { return new(typeDesÉléments) }
   ```

2. Pour obtenir un élément, invoquer p.Get(). Cette fonction va retirer un élément du lot disponible et le renvoyer à l'appelant. Elle peut décider, selon la situation, d'en allouer un nouveau en invoquant p.New().
3. Pour recycler un élément, invoquer p.Put(). La fonction peut le remettre dans le lot disponible, ou décider de le libérer en fonction de la pression sur la mémoire et de l'utilisation du ramasse-miettes.

Pour utiliser les Pool efficacement, il faut que le lot de données soit utilisé par un objet à longue durée de vie, en particulier s'il est consommé par des goroutines indépendantes. Son coût de fonctionnement n'est en revanche pas utilisé efficacement pour gérer la liste d'allocations d'un objet à courte durée de vie et/ou utilisé par une seule goroutine.

© Dunod – Toute reproduction non autorisée est un délit.

10.4.5 Tableaux associatifs : `Map`

Les types cartes natifs du langage ne supportent pas l'accès concurrent, et sont pourtant massivement utilisés.

Pour simplifier leur utilisation dans deux cas très répandus, un nouveau type `sync.Map` a été introduit dans Go 1.9, à partir d'une implémentation précédemment trouvée dans le paquet des bibliothèques supplémentaires `golang.org/x/sync`.

L'API est comparable à celle des cartes, à ceci près qu'elle est à base de méthodes, le langage ne pouvant être modifié.

Comme la plupart des autres types du paquet `sync`, la valeur vierge est prête à l'emploi, et une carte ne doit pas être copiée après l'appel d'une première méthode, interdisant aux types qui les contiennent de comporter des méthodes à récepteur valeur.

- ✓ **`Delete(key interface{})`** : supprime la valeur associée à une clef, sans erreur en l'absence de valeur.
- ✓ **`Load(key interface{}) (interface{}, bool)`** : renvoie la valeur associée à une clef si elle existe, ou **`nil`** sinon, et un booléen indiquant si la valeur était présente.
  Comme cette méthode n'est pas un élément du langage, elle ne peut renvoyer qu'un type **`interface{}`** pour permettre de manipuler tout type de données, de sorte qu'il est généralement nécessaire de procéder ensuite à une assertion de type sur le type d'élément souhaité.
- ✓ **`LoadOrStore(key, value interface{}) (interface{}, bool)`** : renvoie une paire de valeurs associée à une clef, comme `Load`.
  En revanche, si aucune valeur n'était précédemment associée à la clef, l'argument valeur passé à la méthode est stocké dans la carte, et utilisé comme valeur résultat. Le second résultat, booléen, vaut `true` si la valeur préexistait, `false` si elle a été stockée à partir des arguments.
- ✓ **`Range(f func(key, value interface{}) bool)`** : implémente un patron de conception Visiteur [16], invoquant sur chaque paire clef/valeur de la carte la fonction passée comme argument, et en interrompant son parcours si la fonction renvoie `false`.
  Comme ces cartes sont destinées à un fonctionnement concurrent, la méthode est compatible avec des modifications de la carte en cours de parcours, de sorte que :
  - elle ne garantit pas que le parcours effectivement réalisé corresponde à un quelconque état instantané de la carte ;

16. https://osinet.fr/go/concurrence/wiki-visiteur

- si une valeur est modifiée ou supprimée durant le parcours, elle ne garantit pas que cette valeur sera passée à la fonction, et, si elle l'est, ne garantit pas que la valeur soit spécifiquement celle présente avant ou après la modification.

Elle garantit en revanche que la fonction sera invoquée au maximum une fois pour chaque clef, indépendamment des changements de valeurs.

Sa complexité d'exécution peut être en $\mathcal{O}(n)$ du nombre d'éléments, même si son parcours est interrompu après un nombre d'appels fixe et inférieur au nombre d'éléments.

✓ **`Store(key, value interface{})`** : stocke une valeur en l'associant à une clef.

Aussi pratique que soit ce type, il est important de prendre en compte ses limitations et de ne pas l'utiliser systématiquement à la place d'une carte native, en raison de son coût plus élevé. Il est spécifiquement optimisé pour deux situations :

✓ lorsque la valeur associée à une clef donnée est beaucoup plus souvent lue qu'elle n'est modifiée : c'est la situation typique des antémémoires (*cache*) ;

✓ lorsque les jeux de clefs utilisés par les diverses goroutines accédant à la carte sont disjoints.

Dans ces deux cas, les situations de contention sur le verrou de la carte sous-jacente sont rares, et le surcoût de résolution de la contention par rapport à une approche simple est amorti par l'efficacité de l'accès en lecture (pour les caches) ou l'absence de conflits d'accès (pour les jeux de clefs disjoints).

Dans les autres cas, il est préférable d'utiliser :

✓ soit une implémentation personnalisée bâtie en fonction du type d'accès autour d'un verrou par `RWMutex` ;

✓ soit, si le type d'accès s'y prête, une version personnalisée des autres implémentations de carte concurrente pré-optimisées pour d'autres situations, fournies par le paquet des bibliothèques supplémentaires `golang.org/x/sync` (cf. § 10.6.4).

10.4.6 Moniteurs : `Cond`

Le dernier type fourni par le paquet est `sync.Cond`, qui fournit un mécanisme de variables de condition [17], sous-ensemble du mécanisme de moniteurs, défini par Brich Hansen [18] et C.A.R. Hoare [19].

En substance, une variable de condition est un type de données par lequel un ensemble de tâches indépendantes – des goroutines dans le cas de Go – peut attendre qu'une condition devienne vérifiée pour reprendre son exécution.

17. https://osinet.fr/go/concurrence/wiki-variable-de-condition
18. Shared Classes - https://osinet.fr/go/concurrence/hansen1973-moniteurs.pdf
19. « Monitors : an operating system structuring concept », *Communications of the ACM*, Vol. 17, n° 10, p. 549-557, https://osinet.fr/go/concurrence/hoare1974-moniteurs

© Dunod – Toute reproduction non autorisée est un délit.

Dans les cas non répétitifs, ceci est réalisé en Go en passant un canal ouvert sur lequel les goroutines se placent en réception, et restent bloquées. La fermeture du canal leur permet de reprendre leur exécution, comme évoqué à propos de la manipulation des types canaux (cf. § 4.5.4) et de l'instruction d'émission (cf. § 7.2.3).

La limitation de cette approche tient au fait qu'un canal une fois fermé ne peut être rouvert, de sorte que, une fois les goroutines débloquées par la fermeture du canal, elles ne peuvent pas se replacer en attente d'un rendez-vous suivant : des mécanismes plus complexes sont nécessaires.

Le type `sync.Cond` répond à ce besoin, avec les mécanismes caractéristiques des variables de conditions :

- ✓ **Création** : une variable de condition nécessitant un mécanisme d'exclusion mutuelle, les variables `sync.Cond` sont créées par une fonction fabrique dans une goroutine créatrice.

  ```
  vc := func NewCond(l Locker) *Cond
  ```

  Le paramètre doit recevoir un argument du type interface `Locker`, implémenté concrètement par les types `sync.Mutex`, `sync.RWMutex`, et par le type anonyme renvoyé par `RWMutex.Locker()`.

- ✓ **Mise en attente** : une goroutine en attente d'un rendez-vous sur une variable de condition `vc` commence par acquérir son verrou par `vc.Lock()`, puis se place en attente en invoquant `vc.Wait()`.
  Durant l'attente, la méthode `vc.Wait()` libère le verrou obtenu jusqu'à réception d'une notification, puis l'acquiert de nouveau avant de se terminer, rendant le contrôle avec le verrou actif comme s'il n'avait jamais cessé de l'être, tout en l'ayant en pratique libéré durant l'attente.

- ✓ **Notification des goroutines** : la goroutine créatrice notifie son arrivée au point de rendez-vous en utilisant l'une des deux méthodes `vc.Signal()` et `vc.Broadcast()`.
  Les deux fournissent le même service, la différence étant que `vc.Broadcast()` va diffuser la notification à toutes les goroutines en attente, tandis que `vc.Signal()` ne va transmettre la notification qu'à une seule d'entre elles, s'il en existe au moins une. Il n'existe pas de moyen de sélectionner quelle goroutine en attente sera choisie pour recevoir la notification.
  La goroutine créatrice qui invoque ces méthodes peut indifféremment détenir ou non le verrou de la variable de condition lorsqu'elle les invoque.

Dans la plupart des cas, la condition qu'attendent les goroutines n'est pas nécessairement vérifiée immédiatement au retour de `vc.Wait()`, de sorte qu'il est préférable de placer son invocation dans une boucle dans l'attente de la condition, comme dans la méthode `http2pipe.Read` du paquet `net/http`.

Bien que ce type soit présent depuis la version weekly 2011-02-24, antérieure à Go 1.0, il est utilisé moins de dix fois dans le SDK Go 1.13 (hors tests), et encore moins avant l'ajout du protocole HTTP/2, confirmant son caractère d'usage exceptionnel.

10.5 BAS NIVEAU : LE PAQUET sync/atomic

Au plus près du système d'exploitation, voire du matériel, le paquet sync/atomic fournit des primitives d'accès mémoire concurrent. Il n'est *pas* destiné à l'usage des développeurs d'applications.

Comme le précise sa documentation, les fonctions de ce niveau nécessitent une utilisation particulièrement prudente, de sorte que, à l'exception d'applications très particulières devant opérer à bas niveau, il est préférable d'utiliser les mécanismes fournis par le langage – canaux, goroutines et select – ou par le paquet sync, et par extension ceux du paquet x/sync, à l'écriture desquels ces fonctions sont destinées.

Les fonctions de ce paquet sont listées dans le tableau 10.2.

Tableau 10.2 – Fonctions atomiques

Fonction	[u]int(32|64)	uintptr	unsafe.Pointer
Add<type>	✓	✓	
CompareAndSwap<type>	✓	✓	✓
Load<type>	✓	✓	✓
Store<type>	✓	✓	✓
Swap<type>	✓	✓	✓

Toutes ces fonctions exécutent l'opération indiquée de façon atomique, en tirant parti au mieux des mécanismes sous-jacents disponibles sur l'architecture cible. Le paquet définit également un type atomic.Value, implémentant des méthodes Load() et Store() équivalentes à ces fonctions :

Pour un type T, ces fonctions sont les suivantes :

- ✓ **AddT(addr *T, delta T) T** : ajoute deux valeurs du type choisi, la première passée par adresse et la seconde par valeur, et renvoie leur somme.
- ✓ **CompareAndSwapT(addr *T, old, new T) bool** : compare la valeur pointée par le premier argument à celle du second. Si ces deux valeurs sont égales, elle remplace cette valeur par la troisième et renvoie true ; sinon, elle renvoie false.
- ✓ **LoadT(addr *T) T** : déréférence l'adresse passée en argument et renvoie la valeur pointée.
- ✓ **StoreT(addr *T, val T)** : enregistre la valeur du second argument à l'adresse pointée par le premier.

© Dunod – Toute reproduction non autorisée est un délit.

✓ **`SwapT(addr *T, new T) T`** : enregistre la valeur du second argument à l'adresse pointée par le premier, et renvoie la valeur qui s'y trouvait précédemment.

Conséquence de leur proximité du matériel, ces fonctions font l'objet de restrictions variables selon les versions. Ainsi, pour Go 1.14 :

✓ Sur l'architecture x86-32, les fonctions opérant sur des types 64 bits nécessitent des instructions du jeu MMX [20].

✓ Sur les architectures ARM non Linux, ces fonctions nécessitent au minimum un cœur ARM v6K.

✓ Sur les architectures ARM, x86-32 et MIPS RISC 32 bits, l'alignement mémoire des mots 64 bits est du ressort de l'appelant, le runtime Go garantissant uniquement l'alignement du premier mot des variables, structures, tableaux et tranches sur un alignement 64 bits.

10.6 MÉCANISMES SPÉCIALISÉS : LE PAQUET `golang.org/x/sync`

Le module `golang.org/x/sync` contient des primitives de synchronisation similaires à celles du paquet standard `sync`, mais d'un usage plus spécialisé :

✓ à bas niveau, des sémaphores avec poids dans le paquet `semaphore` ;

✓ un mécanisme de regroupement des erreurs survenant dans des goroutines réalisant des parties d'un même traitement, dans le paquet `errgroup` ;

✓ un mécanisme de sérialisation des exécutions concurrentes de fonction, dans le paquet `singleflight` ;

✓ trois implémentations de cartes à accès concurrent, similaires à `sync.Map`, dans le paquet `syncmap`.

10.6.1 Sémaphores : le paquet `semaphore`

Les sémaphores fournis dans le paquet `golang.org/x/sync/semaphore` sont une extension des sémaphores simples de la littérature, disposant de la notion de poids et tirant parti du mécanisme des contextes (cf. § 10.3) pour l'annulation.

Leur API est la suivante :

✓ **Création** : `s := semaphore.NewWeighted(int64) *semaphore.Weighted`
L'argument passé à cette fonction fabrique est le poids du sémaphore.

✓ **Acquisition** : une goroutine peut tenter d'acquérir un sémaphore aussi bien en bloquant qu'en ne bloquant pas :

- **`s.Acquire(ctx context.Context, n int64) error`** : tente une acquisition bloquante du sémaphore `s` pour le poids passé en argument.

20. https://osinet.fr/go/concurrence/wiki-mmx

Elle reste bloquée sur l'acquisition jusqu'au succès, hormis en cas de fermeture du contexte reçu, auquel cas elle renvoie le résultat de `ctx.Err()`, l'erreur ayant entraîné la fermeture du contexte.

- **`s.TryAcquire(TryAcquire(n int64) bool`** : tente une acquisition non bloquante du sémaphore `s` pour le poids passé en argument.
  Le résultat reflète le succès de l'acquisition. L'opération n'ayant pas de durée n'utilise pas de contexte.

✓ **Libération** : `s.Release(n int64)` libère le sémaphore.
À l'encontre des pratiques de traitement d'erreurs recommandées, cette fonction déclenche une panique au lieu de renvoyer une erreur si le code utilisateur demande une libération pour un poids supérieur au poids courant du sémaphore.

Les sémaphores sont des structures `Weighted` incluant un champ de type `sync.Mutex`, donc les méthodes ci-dessus sont toutes à récepteur pointeur.

10.6.2 Regroupement d'erreurs et annulation : le paquet `errgroup`

Les groupes d'erreurs du paquet `golang.org/x/sync/errgroup` permettent la synchronisation, la propagation d'erreurs et l'annulation de contexte pour des groupes de goroutines exécutées comme sous-tâches d'une même tâche parente.

Leur implémentation, dans le type `errgroup.Group`, est conçue autour des mécanismes de `sync.WaitGroup` et `sync.Once`.

Leur API est la suivante :

✓ **`eg := WithContext(context.Context) (*Group, context.Context)`**
crée un groupe d'erreurs pourvu d'un contexte et renvoie ce groupe et son contexte comme résultats.
Le contexte renvoyé est annulé dès que l'une de deux conditions suivantes vient à être vérifiée :
- un résultat non-**`nil`** de l'une des goroutines ;
- la fin de l'ensemble des tâches avec succès, de sorte que l'appel à la méthode `eg.Wait()` est revenu avec un résultat **`nil`**.

✓ **Valeur vierge** : il est aussi possible d'utiliser la valeur vierge du type pour obtenir un groupe avec la même capacité de résultat groupé, mais sans contexte associé.

✓ **`eg.Go(f func() error)`** : exécute dans une goroutine la tâche représentée par la fonction passée comme unique argument.
Cette fonction doit être niladique et renvoyer une valeur de type **`error`**, avec pour valeur **`nil`** en cas de succès et une autre valeur en cas d'erreur.
En cas d'erreur, si le groupe d'erreurs comporte un contexte, sa fonction d'annulation est invoquée.

© Dunod – Toute reproduction non autorisée est un délit.

✓ **`eg.Wait()`** : attend la fin d'exécution de l'ensemble des goroutines créées par les invocations de `eg.Do()`, puis invoque la fonction d'annulation du contexte qu'elle porte si elle en porte un.
Un retour d'erreur non-**`nil`** de l'une des tâches ne provoque pas de fin d'exécution anticipée du groupe.
Dans tous les cas, `Wait()` renvoie la première erreur non-**`nil`** rencontrée, ou **`nil`** si aucune tâche n'est revenue en erreur.

Un point à prendre en compte est le fait que même lorsqu'un groupe d'erreurs est créé par `errgroup.WithContext(ctx)`, le contexte n'est pas transmis par défaut aux tâches : s'il est souhaité qu'elles en dépendent, il faudra le leur transmettre, typiquement en créant les tâches comme des fonctions anonymes dans la portée de la fonction créant le groupe d'erreurs, de sorte que le contexte leur soit accessible par le jeu des portées, à la façon du test `TestWeightedDoesntBlockIfTooBig` du paquet `golang.org/x/sync/semaphore`.

10.6.3 Dédoublonnage d'exécution : le paquet `singleflight`

Un problème relativement courant dans les architectures à base de services est la tendance à surconsommer les services amont en réponse à des requêtes aval. Si les requêtes sont espacées, il est possible d'utiliser un cache de résultats, mais cela ne résoud pas le cas des requêtes entrantes requérant la même donnée amont avant que la précédente ne soit terminée (*bounce*) : la résolution de ce problème (*debouncing*) passe alors souvent par un mécanisme de verrou.

Ces appels concurrents étaient l'un des problèmes de la résolution DNS jusqu'en Go 1.1, et le mécanisme de `singleflight` de la bibliothèque a été importé du projet Camlistore [21] pour lui donner une solution plus simple d'emploi pour obtenir une résolution efficace.

Ce paquet étant devenu `internal` depuis Go 1.4, il n'est toutefois pas utilisable par les développeurs d'applications. Le paquet `golang.org/x/sync/singleflight` en est la version publique destinée à l'usage général.

Son utilisation est la suivante :

✓ **Création** : la valeur vierge du type `singleflight.Group` est prête à l'emploi :

```
var sf singleflight.Group
```

✓ **Soumission de tâches avec réception des résultats** : `sf.Do(string, func() (interface{}, error)) (interface{}, error, bool)` soumet la tâche passée en second paramètre sous l'identité passée en premier.
Si aucune tâche n'est en cours avec la même clef, son exécution est lancée dans un `WaitGroup`, et le résultat de la fonction est renvoyé par cette méthode, d'où il pourra être extrait par une assertion de type.

21. Perkeep, https://perkeep.org/, dénommé Camlistore à l'époque, est une bibliothèque de stockage créée par Brad Fitzpatrick, le créateur de LiveJournal, Memcached, OpenID, et PubSubHubbub.

Le résultat de type **error** est le deuxième résultat de la fonction exécutée, et le troisième indique si plus d'une instance de la fonction a été soumise lors de l'obtention du résultat.
Si une tâche est en cours d'exécution sous la même clef, les demandes suivantes sont mises en attente, et reçoivent le même résultat au moment où l'exécution initiale se termine. Le troisième résultat vaut alors toujours `true`.

- ✓ **Soumission de tâches avec canal résultats** :
`sf.DoChan(string, func() (interface{}, error)) <-chan Result`
fonctionne de la même façon que `Do()` mais, au lieu de renvoyer les résultats de façon synchrone, elle renvoie immédiatement un canal d'éléments de type `singleflight.Result`, sur lequel elle émet les résultats lors de leur obtention, chaque invocation recevant un canal différent.
- ✓ **Oubli d'une tâche** : `sf.Forget(string)` indique au groupe d'oublier l'identité de la tâche passée comme argument.
Cela n'affecte pas les exécutions en cours, mais la prochaine invocation de `s.Do()` ou `s.DoChan()` déclenchera une nouvelle exécution de la fonction au lieu de venir se greffer sur le groupe d'exécution en attente.

L'exemple de référence de l'utilisation de `DoChan()` est la méthode `lookupIPAddr` du type `net.Resolver`, visible dans le fichier `src/net/lookup.go` du SDK Go.

10.6.4 Cartes concurrentes : le paquet `syncmap`

En raison de la diversité des situations de concurrence, les cartes standard de Go sont optimisées pour l'accès non concurrent, déléguant à des couches de niveau supérieur leur utilisation en mode concurrent.

Outre les implémentations sur mesure, optimisées pour le profil d'accès spécifique à une application, le paquet `golang.org/x/sync/syncmap` comporte trois implémentations correspondant à des situations rencontrées plus ou moins couramment :

- ✓ Pour les situations de caches et de groupe d'accès disjoints, le type `syncmap.Map` implémente un accès en temps constant amorti.
Depuis Go 1.9, il a été promu dans la bibliothèque standard en tant que `sync.Map` (cf. § 10.4.5), et n'est plus qu'un alias vers celui-ci.
- ✓ Le type `syncmap_test.RWMutexMap` est optimisé pour l'accès en lecture en utilisant un `sync.RWMutex`.
- ✓ Le type `syncmap_test.DeepCopyMap` est une autre implémentation utilisant un `sync.Mutex` et une `atomic.Value`.

Aucun de ces types n'est destiné à une utilisation directe : depuis Go 1.9, l'implémentation de référence n'est qu'un alias de `sync.Map`, et les deux autres implémentations n'existent que dans le paquet de test en boîte blanche (cf. § 9.3). Elles servent

© Dunod – Toute reproduction non autorisée est un délit.

à valider le comportement et la performance de l'implémentation de référence, pour s'assurer que celle-ci demeure au moins aussi performante que les autres dans les cas pour lesquels elle est conçue.

Du point de vue des développeurs d'application, leur intérêt est de fournir des exemples auxquels comparer leurs implémentations spécifiques.

10.7 SITUATIONS DE COMPÉTITION : LE *RACE DETECTOR*

Pour faciliter l'identification des situations de compétition (*race condition*), et faciliter ainsi la correction des algorithmes concurrents, Go inclut depuis sa version 1.1 un détecteur de situations de compétition (*race detector*).

Son utilisation est des plus simples : il suffit de rajouter l'option `-race` aux commandes de test ou de construction : `go test`, `go build`, `go run`, et `go install`.

Avec cette option, le code source est modifié à la volée pour insérer une instrumentation autour de chaque accès mémoire, instrumentation qui sera utilisée par le runtime lors de l'exécution du programme, pour valider la cohérence des accès.

En cas de détection d'une situation de compétition, un rapport est émis et le programme se termine après son émission. Pour chaque accès mémoire participant à la situation de compétition, le rapport indique l'emplacement source de l'opération et la pile d'appels de la goroutine dans laquelle il se situe.

Cette réécriture ralentit *considérablement* l'exécution du code : un facteur de ralentissement de 10 n'est pas exceptionnel. L'utilisation des versions construites avec ce détecteur est donc à réserver aux environnements de développement, voire dans des cas extrêmes de situations problématiques n'apparaissant qu'en production, à une activation par *feature flag* sur une faible portion sélectionnée du trafic.

Son usage est documenté et illustré d'exemples dans la documentation système :

- ✓ Introduction : https://osinet.fr/go/concurrence/go-intro-detecteur
- ✓ Référence : https://osinet.fr/go/concurrence/go-detecteur

Limitations de disponibilité

Introduit initialement pour les seules architectures x86-64 `amd64`, et uniquement pour Linux, macOS, et Windows, son support a progressivement été étendu aux architectures `arm64` (en pratique, uniquement avec des adresses virtuelles de 48 bits) et PowerPC 64 bits `ppc64le` pour Linux ; et aux systèmes FreeBSD et NetBSD pour `amd64`.

Son option *memory sanitizer* ne fonctionne en Go 1.14 que sur Linux, et uniquement sur les architectures `amd64` et `arm64`.

Pour vérifier dans une version donnée son support pour une architecture, se reporter au fichier `src/cmd/internal/sys/supported.go` du SDK Go.

Index

La présence éventuelle de « *Plus en ligne* » après les numéros de pages signifie que l'entrée correspondante se trouve aussi traitée dans les *Plus en ligne*.

0V, 162, 163

A

abortion clause, 268
adresse, 35, 56
affectabilité, 102
affectation, 231
  opération, 233
ALGOL W, 170
alias de type, 100, 160
Amdahl
  Gene, 17
  loi, 17, 338
AMPhp, 11
antémémoire, 359
AOT, 6
appel, 71, 206
array, 109
ASCII, 9, 105, 106, 260
assertion de type, 95, 204
assignment
  instruction, 231
  operation, 233
attente active, 255
autoforward, 12, 126

B

backquote, 54
backtick, 54
base type, 57
BASIC, 8, 265, 267
Bazaar, 314, 322
BCPL, 38
bégaiement, 300
black-box testing, 298
bloc, 237
  de clause, 22
  de fichier, 22
  de paquet, 22
  instruction, 22, 241
  univers, 22
block, 241
bootstrapping, 276
bounce, 364
break, 263
build
  list, 328
  tags, 321
builtins, 50, 275
byte, 44
bytecode, 5

C

C, 3, 6, 8, 265
C++, 11, 62
cache, 359
call, 206
canaux, 11, 137
caractères, 44
Cargo, 317, 320
carte, 122
cast, 212
casting, 211
CGO, 301, 308, 310, 338
chaînes de caractères, 51

champs, 126
  intégrés, 129
  étiquettes, 131
channel, 11
  types, 137
Cheney, Dave, 99, 127, 285, 344
classes, 126
clause
  d'abandon, 268
  d'étendue, 259
  de cas, 245, 250, 252
  de garde, 243, 284
  for, 257
  paquet, 238
  range, 259
closure, 63, 66
Cobol, 8, 127, 265, 267
commutateur
  d'expression, 188, 244
  de type, 96, 188, 248
comparaison, 32
compilation croisée, 7
compliance, 316, 335
Composer, 317, 320
CompoundString, 9
condition, variable de…, 359
conformité, 316, 335
const, 156
constantes, 152
container, 303
contextes, 345
continue, 263
conversion, 211
copie creuse, 67
Cox, Russ, 86, 115, 272, 288, 313, 325
CSP, 11, 137, 337
custom import, 318, 319

D

deadlock, 356
déballage d'une erreur, 287
debouncing, 364
déclaration
  courte de variable, 149
  de premier niveau, 238
  des constantes, 156
  des variables, 147, 148
  instruction de…, 237
découpage, 198
délégation automatique, 12, 126
dépassement de capacité, 31, 45
déréférencement, 35, 60
désassemblage, 296
désérialisation, 125
destructeur, 170
Dijkstra, E.W., 267
directionnalité, 138
DLL Hell, 7
Donovan, Alan, 162
DWARF, 296
dynamic dispatch, 82

E

EBCDIC, 9
ECMAScript, 19
édition de liens, 7
ellipse, 111
emballage d'une erreur, 288
embedded fields, 129
énergie, 13, 255
ensembles, 128
entiers, 44
enveloppe thermique, 13
Erlang, 138
erreurs
  déballage, 286
  emballage, 286
  traitement, 283
error
  interface…, 283
  unwrapping, 287
  wrapping, 287
espace de travail, 310
étendue d'une indexation, 197
étiquette
  d'instruction, 240
  de champ, 128, 131
  lecture, 174
  de construction, 321
étreinte mortelle, 356
expression
  commutateur d'…, 188
  constante, 221
  d'appel de fonction/méthode, 206
  d'assertion de type, 204
  d'index, 196
  de conversion, 211
  de méthode, 193
  de sélection, 189
  de tranchage (découpage), 198
  ordre d'évaluation, 222
  primaire, 188
  valeur de méthode, 194

extern, 25
exécutables, 300

F

fallthrough, 247
feature flag, 366
fermeture, 63, 66
field, 126
FIFO, 137
FILLER, 127
finaliseur, 171
Fitzpatrick, Brad, 364
FMA, 51
fonction, 63
  anonyme, 65
  d'ordre supérieur, 63
  médiateur, 63
  variadique, 67
for, 257
forever, 257
fork, 312, 317, 319
Fortran, 8, 19, 265, 267
Fossil, 314, 322
friend, 305

G

garbage collector, 171, 339, *Plus en ligne*
génériques, 277, 280
Go 2, 62, 287, 308
go-import, 314
go-source, 315
go.mod, 326
go.sum, 328
GOPATH, 311
  inconvénients, 320
GOROOT, 312
goroutine, 11, 270, 338
goto, 265
Gradle, 320
greffon, 301, *Plus en ligne*
Griesemer, Robert, 10
guard clause, 243, 284

H

Hansen, Brich, 359
headers, 296
heap, 303
higher order function, 63
hissage, 41, 240, 267
Hoare, C.A.R., 11, 137, 170, 359
hoisting, 41, 240, 267

I

Icicle, 11
identificateur, 23
  exporté, 25
  qualifié, 187
identité de types, 100
IEEE-754, 107
if/else, 241
immuable, 52
import, 187, 305
  chemin canonique, 318
  custom…, 318
  vanity…, 318
in the large, 1
indirection, 35
init statement, 261
injection de dépendances, 304
instruction
  affectation, 231
  déclaration, 237
  déclaration courte, 234
  émission, 230
  expression, 228
  incrémentation, 230
  simple, 150, 225, 227
  terminale, 225
  vide, 228
interblocage, 356
intercession, 174, 179
interfaces, 85
intergiciel, 345
internal package, 305
Internet of Things, 7
introspection, 174, 179
invocation, 71, 206
  dynamique, 82
IoT, 7
iota, 158
IPC, 10
ISO 10646, 9
ISO 8859-1, 9, 106
ISO 8859-15, 9

J

J2SE, 12
Java, 6, 11
JavaScript, 8
JIT, 6
JVM, 6

K

keywords, 40

L

label, 240
lambda, 65
Latin 1, 9
Latin 9, 9
LIGNORE, 42
Lisp, 267
list, 303
liste
  circulaire, 303
  de construction, 328
  doublement chaînée, 303
littéraux
  cartes, 122
  chaînes, 54
  complexes, 48
  entiers, 46
  flottants, 48
  fonctions, 65
  pointeurs, 57
  runes, 46
  structures, 132
  tableaux, 111
  tranches, 116
logger, 345
logique, 34
longjmp, 266

M

main, 300
map, 122
masquage, 25, 235
médiateur, 63, 345, 348
Mercurial, 314, 322
merge request, 317
métaprogrammation, 174
méthode, 85
  statique, 74, 126, 208
middleware, 63, 345
Minimal Version Selection, 325
Modula, 295
module
  alternatives, 320
  définition, 325
  query, 330
moniteur, 359
monkey patching, 126
monorepo, 322
mots-clefs, 40
  secrets, 42
MVS, 325

N

n-uplet, 73, 231
namespace, 42
nil, 165
nombres
  complexes, 47
  flottants, 47
nonce, 345
NPM, 317, 320

O

Oberon-2, 74, 295
Objective-C, 11
Occam, 137
opérateur, 26
  additif, 31
  adresse, 35
  de comparaison, 32
  logique, 34
  multiplicatif, 30
  réception, 36
opération d'affectation, 233
ordonnanceur, 340, 342, *Plus en ligne*
ORM, 131, 174
overflow, 31, 45

P

package, 295
  clause, 238, 297
  import, 305
  internal…, 305
panic, 289
  run-time…, 290
panique, 289
  d'exécution, 290
paquet, 295
  clause…, 238, 297
  déclaration, 305
  fichiers d'un…, 298
  import, 305
  interne, 305
  main, 300
  plugin, *Plus en ligne*
  unsafe, *Plus en ligne*
Pascal, 295
PERL, 5
PHP, 8
  6.x, 9
Pike, Rob, XV, 9, 91, 98, 164, 174, 181, 283, 355
pile d'appels, 343

PIP, 320
playground, 14
plugin, 301, *Plus en ligne*
pointeurs, 56
ponctuation, 37
POO, 11
portée, 237
POSIX, 10, 301
post statement, 261
pré-version, 327
précision des constantes, 155
Preston-Werner, Tom, 324
priorité des opérateurs, 28
profondeur, 189
programmes, 300
promesse
  de compatibilité Go 1, 4, 276
  JavaScript, 351
promotion d'un champ ou méthode, 130
protocole de téléchargement, 333
proxy, 332, 334
  protocol, 333
pseudo-version, 327
pthread, 11
pull request, 317
Python, 5

Q

qualification, 187
quasi-variables, 150, 235, 269

R

race
  condition, 142, 366
  detector, 366
ramasse-miettes, 170–173, 339, 342, 357, *Plus en ligne*
range
  for, 259
  index, 197
ReactPHP, 11
receiver, 73
récepteur, 73
receveur, 73
reflect, 174
réflexion, 131, 174
  fonctions génériques, 177
  intercession, 179
  introspection, 179
représentabilité, 105
requête de module, 330
ring, 303
Ruby, 5
run-time panic, 290
rune, 10, 44
runtime, 181
Rust, 338

S

SAT, 322
SBCS, 9
scheduler, 340, 342, *Plus en ligne*
secrets, 42
select, 251
sélecteurs, 189
sélection, 189
semantic
  import versioning, 325, 329
  versioning, 324
set, 128
setjmp, 266
shadowing, 25, 235
short variable declaration, 149
signal, 301
simple instruction, 227
Simula 67, 11
Single Unix Specification, 10
situation de compétition, 142, 366
slice, 114
slicing, 198
Smalltalk, 11
SOLID, 8, 148
soupassement, 107, 217
stack trace, 288, 343
stop the world, 342
stratégie, 84
struct, 126
stuttering, 300
Subversion (SVN), 314, 322
sucre syntaxique, 76, 167
surcharge, 65
switch
  d'expression, 244
  de type, 248
syntaxic sugar, 76

T

tableau, 109
tag
  build, 321
  field, 128, 131

tas, 303
TCO, 2
TDP, 13
temporisation, 255
ternaire, 241
tests
  boîte blanche, 298
  boîte noire, 298
Thompson, Ken, 9, 38
thread, 338
timeout, 255
top-level declaration, 238
tranchage, 198
tranche, 114
trois-points, 64, 67, 111, 185, 209
tuple, 73, 231
type
  aliasing, 100, 160
  switch, 96
type
  alias de..., 100, 160
  assertion de..., 204
  commutateur de..., 188, 248
  de base d'un pointeur, 57
  défini, 159
  des constantes, 154
  dynamique, 8, 28, 82, 92, 95, 97, 104, 122, 147, 166, 179, 205, 211
  sous-jacent, 159
  statique, 8, 147, 149, 150, 205, 211
types, 159
  canaux, 137
  cartes, 122
  chaînes, 51
  composites, 109
  ensembles, 128
  flottants, 47
  fonctions, 63
  interfaces, 85
  pointeurs, 56
  scalaires, 43
  simples, 43
  structures, 126
  tableaux, 109
  tranches, 114

U

UCS, 9
uintptr, 44
underlying type, 159
Unicode
  norme..., 9
  point de code..., 217
  rune de remplacement..., 217, 260, 307
UNIX, 251, 289
unmarshalling, 125
unwrapping, error..., 287
UTF-8, 9, 260

V

valeur vierge, 162
vanity import, 318
variables, 147
  anonymes, 151
  de condition, 359
  déclaration, 147, 148
variant, 8
vendoring, 315

W

WGL-4, 106
white-box testing, 298
Wirth, Niklaus, 14, 267
workspace, 310
wrapping, error..., 287

X

XmString, 9

Y

Yarn, 317, 320
yield, 342

Z

zéro négatif, 107, 153
zero value, 162
zerobase, 60