

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.

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage. Le Code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements	d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée. Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).
	

© Dunod, 2020
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-080410-8

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2^e et 3^e 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 langage	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	Chainage 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
	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
	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 <code>import</code>	305
9.8	Initialisation	308
9.9	Chemins d'importation	310
9.9.1	Notion de <i>workspace</i> et <code>GOPATH</code>	310
9.9.2	Bibliothèque standard et <code>GOROOT</code>	312
9.9.3	Paquets distants avec <code>go get</code>	312
9.9.4	Organisation des paquets en mode <code>GOPATH</code>	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 <code>sync</code>	351
10.4.1	Groupe des tâches : <code>sync.WaitGroup</code>	352
10.4.2	Exécution unique : <code>Once</code>	354
10.4.3	Exclusion mutuelle : <code>Mutex</code> / <code>RWMutex</code>	355
10.4.4	Allocations réutilisées : <code>Pool</code>	357
10.4.5	Tableaux associatifs : <code>Map</code>	358
10.4.6	Moniteurs : <code>Cond</code>	359
10.5	Bas niveau : le paquet <code>sync/atomic</code>	361
10.6	Mécanismes spécialisés : le paquet <code>golang.org/x/sync</code>	362
10.6.1	Sémaphores : le paquet <code>semaphore</code>	362
10.6.2	Regroupement d'erreurs et annulation : le paquet <code>errgroup</code>	363
10.6.3	Dédoublonnage d'exécution : le paquet <code>singleflight</code>	364
10.6.4	Cartes concurrentes : le paquet <code>syncmap</code>	365
10.7	Situations de compétition : le <i>race detector</i>	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 <code>append</code>	118
4.7	Fonction <code>copy</code> 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 <code>len</code> et <code>cap</code>	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	<code>nil</code> n'est pas toujours égal à lui-même	167
5.15	Appel de méthodes sur <code>nil</code> concret	168
5.16	Appel de méthodes sur <code>nil</code> interface	169
5.17	Finaliseur et <code>runtime.KeepAlive</code>	172
5.18	Étiquettes structurées dans l'ORM GORM	175
5.19	Lecture d'étiquettes de structure	176
5.20	Comparaison par <code>reflect.DeepEqual</code>	178
5.21	Permutation dans une tranche avec <code>reflect.Swapper</code>	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 <code>nil</code>	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 <code>nil</code>	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 « <code>_</code> » et « <code>init</code> »	239
7.10	Instructions <code>if</code> , <code>else</code> et ternaires	242
7.11	L'instruction <code>switch</code> d'expression	244
7.12	Instruction <code>fallthrough</code>	248
7.13	Utilisation pratique de <code>fallthrough</code>	249
7.14	L'instruction <code>switch</code> de type	250
7.15	L'instruction <code>select</code>	253
7.16	Temporisation avec <code>time.After</code>	256
7.17	Quatre <code>for</code> de 0 à 9	258
7.18	Instructions simples de clause <code>for</code>	259
7.19	Modification d'une expression de boucle <code>for</code>	262
7.20	Interruption avec <code>break</code>	264
7.21	Itération anticipée avec <code>continue</code>	265
7.22	Émulation d'une boucle <code>for</code> avec <code>goto</code>	266
7.23	Les quatre formes de <code>return</code>	269
7.24	Exécution concurrente avec <code>go</code>	271
7.25	Code terminal avec <code>defer</code>	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 <code>new</code> et <code>make</code>	279
8.4	Longueur et capacité des tranches	281
8.5	Équivalent de <code>delete</code> pour tranches et tableaux	282
8.6	Erreurs constantes	286
8.7	Situation de panique	290
8.8	Utilisation de <code>panic</code> et <code>recover</code>	291
8.9	Paniques d'exécution réelles et simulées	292
9.1	Documentation de paquet : extrait de <code>fmt/doc.go</code>	300
9.2	Fin d'exécution ordonnée avec <code>et os.Exit</code>	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 <code>go.mod</code> minimal	329
10.1	Informations de l'ordonnanceur avec <code>schedtrace</code>	341
10.2	Informations sur les goroutines avec <code>schedtrace+scheddetail</code>	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 <code>sync.WaitGroup</code>	353
10.6	Utilisation de <code>sync.Once</code>	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.org/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 versions 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.

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.



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 JavaScript 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, Docker³ 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 :

$\lim_{t \rightarrow \infty} Pv(t) = 1$ reflétant l'adéquation du produit aux besoins de sa cible ;
 $\lim_{t \rightarrow \infty} Cm(t) = \infty$ 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 :

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

Donc, puisque C_0 et V_0 sont constants :

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

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.

- 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 débogage 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 langage

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 laquelle il existe trois grandes stratégies :

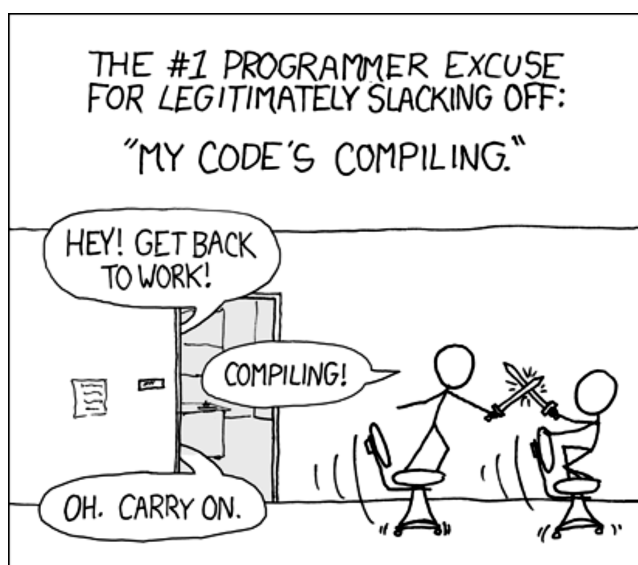
- ✓ **Interprétation** : l'interpréteur analyse le source à la volée, empilant les jetons analysés au fil de l'analyse syntaxique, et les dépilant dès qu'il identifie un fragment correspondant à une instruction exécutable complète. C'est le modèle de la plupart des langages de scripts jusqu'à l'apparition 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

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 système, 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>

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 Liskov¹¹ dans le cadre d'une programmation SOLID¹².

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 EBCDIC¹³ d'IBMTM.¹⁴ ou la norme états-unienne ASCII¹⁵ 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/Motif¹⁷, 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 Unicode¹⁹ 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 Pike²⁰, 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 finalisées²¹, 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>

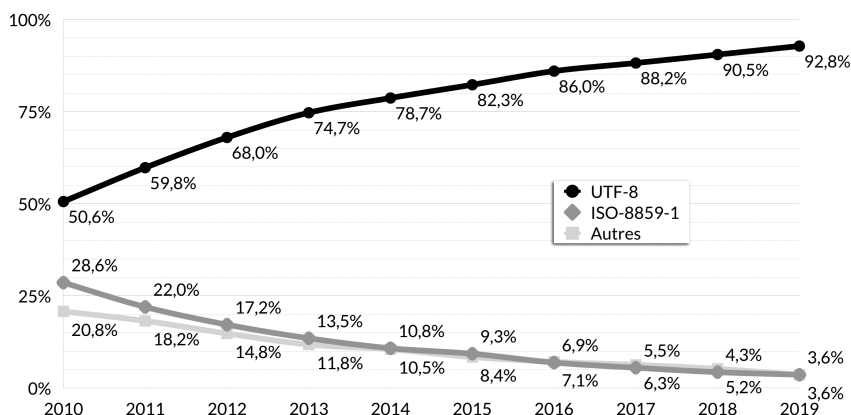
18. <https://osinet.fr/go/intro/iec-iso-10646>

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>

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 Griesemer²² avec une première version publique en 2009, 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.