

Practical Go: conseils pratiques pour la rédaction de programmes Go maintenables

Table des matières

1. Principes directeurs	3
1.1. Simplicité	3
1.2. Lisibilité	4
1.3. Productivité	5
2. Identifiants	5
2.1. Choisir des identifiants par souci de clarté et non de brièveté	6
2.2. Longueur de l'identifiant	7
2.2.1. Le contexte est la clé	8
2.3. Ne nommez pas vos variables par leur type	9
2.4. Utiliser un style de nommage cohérent	11
2.5. Utiliser un style de déclaration cohérent	12
2.6. Être un joueur dans une équipe	15
3. Les commentaires	16
3.1. Les commentaires sur les variables et les constantes devraient décrire leur contenu et non leur objectif.	17
3.2. Toujours documenter les symboles publics	18
4. Conception d'un package	20
4.1. Un bon package commence par son nom.	20
4.1.1. Les bons noms de packages doivent être uniques.	20
4.2. Évitez les noms de packages tels que base, common ou util	21
4.3. Retourner le resultat au plus tôt, plutôt que dans de profondes imbrications de la fonction.	22
4.4. Rendre la valeur zéro utile	23
4.5. Évitez les déclarations globales dans les packages	25
5. Structure du projet	26
5.1. Envisagez des packages moins nombreux et plus grands	26
5.1.1. Classer le code dans des fichiers par instructions d'importation	27
5.1.2. Préférez les tests internes aux tests externes	28
5.1.3. Utilisez des packages internes pour réduire la surface de votre API publique	28
5.2. Garder le package principal le plus petit possible.	29
6. Conception d'API	29
6.1. Concevoir des API qui sont difficiles à utiliser à mauvais escient.	29

6.1.1. Méfiez-vous des fonctions qui prennent plusieurs paramètres du même type	30
6.2. Concevoir des API pour leur cas d'utilisation par défaut	31
6.2.1. Décourager l'utilisation de nil comme paramètre.	31
6.2.2. Préférez les paramètres var args plutôt que []T.	33
6.3. Let functions define the behaviour they requires	34
7. Traitement des erreurs	36
7.1. Éliminer la gestion des erreurs en éliminant les erreurs	36
7.1.1. Compteur de lignes.	36
7.1.2. WriteResponse.	38
7.2. Ne traiter une erreur qu'une seule fois	40
7.2.1. Ajout de contexte aux erreurs	42
7.2.2. Erreurs de wrapper avec github.com/pkg/errors	43
8. Concurrency	45
8.1. Tenez-vous occupé ou faites le travail vous-même	46
8.2. Laisser la concomitance à l'appelant	48
8.3. Ne jamais démarrer une goroutine sans savoir quand elle s'arrêtera.....	50

Introduction

Cet article était à l'origine un atelier organisé pour la QCon Shanghai de 2018. Une mise à jour de cet atelier sera donnée à la GopherCon de Singapore en mai 2019.

Bonjour

Mon objectif au cours des deux prochaines sessions est de vous conseiller sur les meilleures pratiques en matière d'écriture de code en Go.

Ceci est une présentation de type atelier, je vais me passer des diapositives habituelles et nous allons travailler directement à partir du document que vous pouvez emporter avec vous aujourd'hui.

Vous pouvez trouver la dernière version de cette présentation à l'adresse <https://dave.cheney.net/practical-go/presentations/qcon-china.html>

- La traduction en français à l'adresse <https://gauthier.frama.io/practical-go-fr/>
- Le document au format PDF <https://gauthier.frama.io/practical-go-fr/practical-go-fr.pdf>
- Le document au format epub <https://gauthier.frama.io/practical-go-fr/practical-go-fr.epub>



Ce travail est placé sous licence [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

1. Principes directeurs

Si je veux parler des meilleures pratiques dans n'importe quel langage de programmation, j'ai besoin de définir ce que j'entends par *meilleur*. Si vous êtes venu à mon discours d'hier, vous auriez vu cette citation du chef de l'équipe Go, Russ Cox:

Le génie logiciel est ce qui arrive à la programmation lorsque vous ajoutez du temps et d'autres programmeurs.

— Russ Cox

Russ fait la distinction entre la *programmation* logicielle et l'*ingénierie* logicielle: le premier est un programme que vous écrivez vous-même, le second est un produit sur lequel de nombreuses personnes travailleront au fil du temps. Les ingénieurs vont et viennent, les équipes vont grandir et se réduire, les besoins vont changer, les fonctionnalités seront ajoutées et les bugs corrigés: c'est la nature du génie logiciel.

Je suis peut-être l'un des premiers utilisateurs de Go dans cette salle, mais prétendre que mon ancienneté donne plus de poids à mon point de vue est faux. Au lieu de cela, les conseils que je vais vous donner aujourd'hui s'inspirent de ce que je crois être les principes directeurs qui sous-tendent Go lui-même. Ce sont :

1. La simplicité
2. La lisibilité
3. La productivité



Vous remarquerez que je n'ai pas parlé de *performance* ou de *concurrence* : certains langages de programmation sont un peu plus rapides que Go, mais ne sont certainement pas aussi simples que Go. Il existe des langages qui font de la concurrence le meilleur objectif possible, mais ils ne sont pas aussi lisibles, ni aussi productifs.

Les performances et la concurrence sont des attributs importants, mais pas aussi importants que la *simplicité*, la *lisibilité* et la *productivité*.

1.1. Simplicité

La simplicité est une condition préalable à la fiabilité.

— Edsger W. Dijkstra

Pourquoi devons-nous rechercher la simplicité? Pourquoi est-il important que les programmes écrits en Go soient simples?

Nous avons tous été dans une situation où nous nous sommes dit "Je ne comprends pas ce code", n'est-ce pas ? Nous avons tous travaillé sur des programmes dans lesquels nous avons peur de faire un changement parce que nous craignons que cela ne casse une autre partie du programme, une partie que nous ne comprenons pas et que nous ne savons pas réparer: c'est de la complexité.

Il existe deux manières de concevoir un logiciel: la première est de le faire le plus simple de sorte qu'il n'y ait manifestement aucune anomalie, et l'autre est de le rendre si compliquée de sorte qu'il n'y ait pas d'anomalie évidente. La première méthode est beaucoup plus difficile.

— C. A. R. Hoare

La complexité transforme un logiciel fiable en logiciel peu fiable. C'est la complexité qui tue les projets logiciels. Voici pourquoi la simplicité est le but le plus élevé du langage Go. Quels que soient les programmes que nous écrivons, nous devrions être en mesure de convenir qu'ils sont simples.

1.2. Lisibilité

La lisibilité est essentielle à la maintenabilité.

— Mark Reinhold, JVM language summit 2018

Pourquoi est-il important que le code Go soit lisible ? Pourquoi devrions-nous nous efforcer d'écrire du code lisible ?

Les programmes doivent être écrits pour que les gens puissent les lire et, seulement accessoirement, pour que des machines puissent les exécuter.

— Hal Abelson and Gerald Sussman, Structure and Interpretation of Computer Programs

La lisibilité est importante car tous les logiciels, et pas seulement les programmes Go, sont écrits par des humains pour être lus par d'autres humains. Le fait que les logiciels soient exécutés également par des machines est secondaire.

Le code est lu beaucoup plus de fois qu'il n'a été écrit: un seul morceau de code sera lu des centaines, voire des milliers de fois au cours de sa durée de vie.

La compétence la plus importante pour un programmeur est sa capacité à communiquer efficacement ses idées.

— Gastón Jorquera

La lisibilité est essentielle pour comprendre ce que fait le programme. Si vous ne pouvez pas comprendre ce que fait un programme, comment pouvez-vous espérer le maintenir? Si un logiciel ne peut pas être maintenu, il sera réécrit; Et ça sera la dernière fois que votre entreprise investira

dans Go.

Si vous écrivez un programme pour vous-même, il ne devra peut-être fonctionner qu'une seule fois, ou que vous êtes la seule personne qui le verra jamais, dans ce cas faites ce qui vous convient le mieux. Mais s'il s'agit d'un logiciel auquel plus d'une personne contribuera ou qui sera utilisée par des personnes suffisamment longtemps pour que les exigences, les fonctionnalités ou l'environnement dans lequel il sera exécuté évolue, votre objectif doit être que votre programme *puisse être maintenu*.

La première étape vers l'écriture d'un code maintenable consiste à s'assurer que le code est lisible.

1.3. Productivité

Le design est l'art d'arranger le code pour qu'il fonctionne aujourd'hui et puisse être modifié pour toujours.

— Sandi Metz

Le dernier principe sous-jacent que je tiens à souligner est celui de la *productivité* : la productivité des développeurs est un sujet épineux, mais il se résume à ceci: combien de temps consacrez-vous à des travaux utiles dans l'attente de vos outils ou désespérément perdus dans une base de code étrangère? Les développeurs Go devrait sentir qu'ils peuvent faire beaucoup avec Go.

Une blague dit que Go a été conçu en attendant la compilation d'un programme C++. La compilation rapide est l'un des éléments clés de Go, c'est un point essentiel pour attirer de nouveaux développeurs. Bien que la vitesse de compilation reste un champ de bataille constant, il est juste de dire que les compilations prennent quelques minutes dans d'autres langages et quelques secondes dans Go. Cela aide les développeurs Go à se sentir aussi productif que leurs homologues travaillant dans des langages interprétés sans les problèmes de fiabilité inhérents à ces langages.

Plus fondamental à la question de productivité des développeurs, le développeur Go réalise que le code est écrit pour être lu et place ainsi l'acte de lire du code au-dessus de celui de l'écrire. Go va jusqu'à imposer, via l'outillage et la paramétrage, à ce que tout code soit formaté dans un style spécifique. Cela élimine la friction de l'apprentissage d'un dialecte spécifique à un projet et aide à repérer les erreurs parce qu'elles ont simplement l'*air* incorrectes.

Les développeurs Go ne passent pas des jours à déboguer des erreurs de compilation impénétrables. Ils ne perdent pas du temps avec des scripts de compilation compliqués ou à déployer du code en production. Et surtout, ils ne passent pas leur temps à essayer de comprendre ce que leurs collègues ont écrit.

C'est à la productivité que pensent les équipes de développeurs Go lorsqu'ils disent que le langage doit passer à l'échelle.

2. Identifiants

Le premier sujet que nous allons aborder est celui des *identifiants* : un identifiant est un mot fantaisie pour un *nom* : le nom d'une variable, le nom d'une fonction, le nom d'une méthode, le

nom d'un type, le nom d'un package, etc.

Une mauvaise désignation est symptomatique d'une mauvaise conception.

— Dave Cheney

Compte tenu de la syntaxe limitée de Go, les identifiants que nous choisissons pour les éléments de nos programmes ont un impact démesuré sur la lisibilité de nos programmes. La lisibilité est la qualité qui définit un bon code, le choix de bons identifiants est donc crucial pour la lisibilité du code Go.

2.1. Choisir des identifiants par souci de clarté et non de brièveté

L'évidence du code est important. Ce que vous faites sur une ligne doit être fait sur trois lignes.

— Ukiah Smith

Go n'est pas un langage optimisé pour écrire des programmes intelligibles sur une seule ligne. Go n'est pas optimisé pour écrire un programme dans le moins de lignes possibles. Nous ne l'avons pas optimisé en fonction de la taille du code source sur le disque, ni du temps qu'il faut pour taper le code du programme dans un éditeur.

Bien nommer, c'est comme raconter une bonne blague. Si vous devez l'expliquer, ce n'est pas drôle.

— Dave Cheney

La clé de cette clarté réside dans les identifiants que nous choisissons dans nos programmes Go. Parlons des qualités d'un bon identifiant:

- **Un bon nom est concis:** Pas trop court, mais aussi court que possible, il ne doit pas oublier ce qui n'est pas superflu. Il doit avoir un bon rapport signal bruit.
- **Un bon nom est descriptif:** Un bon nom doit décrire l'utilité d'une variable ou d'une constante et *non* leur contenu. Un bon nom doit décrire le résultat d'une fonction ou le comportement d'une méthode, et *non* leur implémentation. Le nom d'un package, *pas* son contenu: plus un nom décrit de manière précise l'élément identifié, meilleur est le nom.
- **Un bon nom doit être prévisible:** Vous devriez être en mesure de déduire la manière dont un symbole sera utilisé à partir de son seul nom. Il est nécessaire de choisir des noms descriptifs mais aussi qui suivent la tradition. C'est de cela que parlent les développeurs Go lorsqu'ils utilisent le mot idiomatique.

Parlons de chacune de ces propriétés plus en détail.

2.2. Longueur de l'identifiant

Parfois, les gens critiquent le style Go qui recommande des noms de variables courts. Comme l'a dit Rob Pike: "Les programmeurs Go veulent des identifiants de bonne longueur".^[1]

Andrew Gerrand suggère d'utiliser des identifiants plus longs pour indiquer au lecteur des choses plus importantes.

Plus la distance entre la déclaration d'un nom et ses utilisation est grande, plus le nom doit être long.^[3]

— Andrew Gerrand

Nous pouvons en tirer quelques lignes directrices:

- Les noms de variables courts fonctionnent bien lorsque la distance entre leur déclaration et leur dernière utilisation est courte.
- Les noms de variable longs doivent se justifier, plus ils sont longs, plus ils ont de valeur à fournir: les noms bureaucratiques longs portent peu de signal par rapport à leur poids sur la page.
- N'incluez pas le nom de votre type dans le nom de votre variable.
- Les constantes devraient décrire la valeur qu'elles détiennent, et non pas comment cette valeur est utilisée.
- Préférez les variables à une lettre pour les boucles et les branches, les mots simples pour les paramètres et les valeurs de retour, les mots multiples pour les fonctions et les déclarations au niveau du package.
- Préférez les mots simples pour les méthodes, les interfaces et les packages.
- Rappelez-vous que le nom d'un package fait partie du nom que l'appelant utilise pour s'y référer, utilisez-le donc.

Regardons cet exemple

```

type Person struct {
    Name string
    Age  int
}

// AverageAge returns the average age of people.
func AverageAge(people []Person) int {
    if len(people) == 0 {
        return 0
    }

    var count, sum int
    for _, p := range people {
        sum += p.Age
        count += 1
    }

    return sum / count
}

```

Dans cet exemple, la variable de plage `p` est déclarée à la ligne 13 et est utilisée une seule fois, à la ligne suivante: `p` vit très peu de temps à la fois sur la page et pendant l'exécution de la fonction. Un lecteur intéressé par la valeur de `p` n'a besoin de lire que deux lignes.

En comparaison `people` est déclaré dans les paramètres de la fonction et vit pendant sept lignes. La même chose est vraie pour `sum` et `count`, cela justifie donc l'utilisation de noms plus longs. Le lecteur doit parcourir un plus grand nombre de lignes pour les localiser, il faut donc qu'elles soient plus lisibles.

J'aurais pu choisir `s` pour `sum` et `c` (ou éventuellement `n`) pour `count` mais cela aurait réduit toutes les variables du programme au même niveau d'importance. J'aurais pu choisir `p` au lieu de `people` mais cela aurait posé problème pour le choix du nom de la variable dans la boucle `for ... range`. Le choix de `person` à la place de `p` dans la boucle aurait l'air bizarre, car la variable éphémère d'itération porte un nom plus long que le groupe de valeurs dont elle est dérivée.



Utilisez des lignes vides pour diviser les étapes d'une fonction de la même manière que vous utilisez des paragraphes pour diviser la partie d'un document. Dans `AverageAge` trois opérations se déroulent en séquence: la première est la condition préalable, qui consiste à vérifier que nous ne divisons pas par zéro s'il n'y a pas de personne, la seconde est l'accumulation de la somme des âges et le comptage des personnes, et la dernière est le calcul de la moyenne.

2.2.1. Le contexte est la clé

Il est important de reconnaître que la plupart des conseils de nommage sont contextuels, j'aime bien dire que c'est un principe et non une règle.

Quelle est la différence entre deux identifiants `i` et `index`? Nous ne pouvons conclure que l'un est

meilleur que l'autre.

Pouvons nous dire que l'exemple ci-dessous

```
for index := 0; index < len(s); index++ {  
    //  
}
```

Est fondamentalement plus lisible que

```
for i := 0; i < len(s); i++ {  
    //  
}
```

Je soutiens que non, car il est probable que la portée de `i`, et `index`, est limitée à la boucle `for` et que la verbosité supplémentaire de cette dernière ajoute peu à la *compréhension* du programme.

Cependant, laquelle de ces fonctions est la plus lisible ?

```
func (s *SNMP) Fetch(oid []int, index int) (int, error)
```

Ou

```
func (s *SNMP) Fetch(o []int, i int) (int, error)
```

Dans cet exemple, `oid` est une abréviation de SNMP Object ID, si vous le raccourcissez par `o`, les développeurs doivent traduire la notation courante qu'ils lisent dans la documentation SNMP en notation plus courte dans votre code. De la même manière, remplacer `index` par `i` ne permet plus de comprendre que nous parlons de messages SNMP, dans lesquels une sous-valeur de chaque OID est appelée un index.



Ne mélangez pas et ne faites pas correspondre des paramètres longs et courts dans une même déclaration.

2.3. Ne nommez pas vos variables par leur type

Vous ne devriez pas nommer vos variables d'après leurs types pour la même raison que vous ne nommez pas vos animaux domestiques "chien" et "chat". Vous ne devriez pas non plus inclure le nom de votre *type* dans le nom de votre variable pour les mêmes raisons.

Le nom de la variable doit décrire son contenu, pas le *type* de contenu. Prenons l'exemple ci-dessous:

```
var usersMap map[string]*User
```

Quels sont les avantages de cette déclaration? Nous pouvons voir que c'est une *map* et que cela à un rapport avec le *type* **User*, c'est probablement une bonne chose. Mais *usersMap* est une *map*, or Go est un langage typé de façon statique, il ne nous permettra pas d'utiliser accidentellement pour autre chose, le suffixe *Map* est donc redondant.

Maintenant considérons que nous ayons à déclarer d'autres variables telles que:

```
var (  
    companiesMap map[string]*Company  
    productsMap  map[string]*Products  
)
```

Nous avons maintenant trois variables de type *map*, *usersMap*, *companiesMap* et *productsMap*, chacun étant un *map* de *strings* sur différents types. Nous savons que ce sont des *maps* et que leurs déclarations nous empêchent d'utiliser l'une à la place de l'autre. Le compilateur déclanchera une erreur si nous essayons d'utiliser *companiesMap* dans le code alors qu'il s'attend à trouver une *map[string]*User*. Dans cette situation, il est clair que le suffixe *Map* n'améliore pas la clarté du code, tout en contraignant à le saisir partout.

Ma suggestion est d'éviter tout suffixe qui ressemble au *type* de la variable.



Si le nom *users* n'est pas suffisamment descriptif, alors *usersMap* ne le sera pas non plus.

Ce conseil s'applique également aux paramètres d'une fonction, par exemple:

```
type Config struct {  
    //  
}  
  
func WriteConfig(w io.Writer, config *Config)
```

Nommer *config* le paramètre de type **Config* est redondant. Nous savons qu'il est de type **Config* puisque c'est écrit.

Dans ce cas, considérer l'utilisation de *conf*, voir même juste *c* si la durée d'utilisation de la variable est suffisamment courte.

S'il y a plus d'une variable de type **Config*, l'usage de *conf1* et *conf2* est moins descriptif que *original* et *updated* car ces dernières risquent moins d'être confondues.

Ne laissez pas les noms des packages voler les bons noms de variables.

Le nom d'un identifiant importé inclut son nom de package. Par exemple le type `Context` dans le package `context` s'appellera `context.Context`. Cela rendra impossible l'usage de `context` comme nom de variable ou de type dans votre package.



```
func WriteLog(context context.Context, message string)
```

Ne compilera pas. C'est pourquoi la déclaration locale pour les types `context.Context` est traditionnellement `ctx`.

```
func WriteLog(ctx context.Context, message string)
```

2.4. Utiliser un style de nommage cohérent

Une autre propriété d'un bon nom est qu'il doit être prévisible. Le lecteur devrait être capable de comprendre l'utilisation d'un nom lorsqu'il le rencontre pour la première fois. Lorsqu'ils rencontrent un nom commun, ils devraient pouvoir supposer qu'il n'a pas changé de signification depuis la dernière fois qu'il l'a vu.

Par exemple, si votre code transmet un *handle* de base de données, assurez-vous qu'il porte le même nom chaque fois qu'il apparaît: plutôt que de combiner `d *sql.DB`, `dbase *sql.DB`, `DB *sql.DB` et `database *sql.DB`, au lieu de cela harmoniser les noms sur quelque chose comme:

```
db *sql.DB
```

Cela favorise la reconnaissance: si vous voyez une `db`, vous savez que c'est un `*sql.DB` et que celle-ci a été identifiée localement ou fournie par l'appelant.

Des conseils similaires s'appliquent aux récepteurs de méthode; utilisez le même nom de récepteur pour chaque méthode de ce type. Cela permet au lecteur de s'approprier plus facilement l'utilisation du récepteur entre les méthodes de ce type.



La convention concernant les noms de destinataires courts dans Go va à l'encontre des conseils donnés jusqu'à présent. C'est l'un des choix faits au début qui est devenu le style préféré, tout comme l'utilisation de `CamelCase` plutôt que celle de `snake_case`.



Le style Go indique que les destinataires ont un nom composé d'une seule lettre ou d'acronymes dérivés de leur type. Vous constaterez peut-être que le nom de votre récepteur est parfois en conflit avec le nom d'un paramètre dans une méthode. Dans ce cas, envisagez d'allonger légèrement le nom du paramètre et n'oubliez pas d'utiliser ce nouveau nom de paramètre de manière cohérente.

Enfin, certaines variables d'une seule lettre sont traditionnellement associées aux boucles et au comptage. Par exemple, `i`, `j` et `k` sont généralement des variable d'incrément pour les boucles `for`. `n` est généralement associé à un compteur ou à un accumulateur. `v` est un raccourci courant pour une valeur dans une fonction de codage générique, `k` est couramment utilisé pour la clé d'une `map` et `s` est souvent utilisé en tant que raccourci pour les paramètres de type chaîne de caractère.

Comme avec l'exemple de base de données ci-dessus, les programmeurs s'attendent à ce que `i` soit une variable d'incrément d'une boucle. Si vous vous assurez que `i` est toujours une variable de boucle, non utilisée dans d'autres contextes en dehors d'une boucle `for`, lorsque les lecteurs rencontrent une variable appelée `i` ou `j`, ils savent qu'une boucle est proche.



Si vous vous retrouvez avec tellement de boucles imbriquées que vous avez épuisé votre réserve de variables `i`, `j` et `k`, il est probablement temps de diviser votre fonction en unités plus petites.

2.5. Utiliser un style de déclaration cohérent

Go propose six façons de déclarer des variables

- `var x int = 1`
- `var x = 1`
- `var x int; x = 1`
- `var x = int(1)`
- `x := 1`

Je suis sûr qu'il y en a d'autres auxquelles je n'ai pas pensé. C'est quelque chose que les concepteurs de Go reconnaisse comme probablement une erreur, mais il est trop tard pour changer les choses maintenant. Avec toutes ces différentes façons de déclarer une variable, comment éviter que chaque développeur Go choisit son propre style ?

Je souhaite présenter des suggestions sur la façon de déclarer des variables dans mes programmes, c'est le style que j'essaie d'utiliser dans la mesure du possible.

- **Lorsque vous déclarez une variable sans l'initialiser, utilisez `var`.** Lorsque vous déclarez une variable qui sera explicitement initialisée ultérieurement dans la fonction, utilisez le mot-clé `var`.

```
var players int    // 0

var things []Thing // an empty slice of Things

var thing Thing    // empty Thing struct
json.Unmarshal(reader, &thing)
```

Le préfixe `var` sert à indiquer que cette variable a été délibérément déclarée comme étant la valeur zéro du type indiqué. Cela correspond également à la nécessité de déclarer des variables au niveau du package à l'aide de `var`, par opposition à la syntaxe de déclaration courte - bien que je ferai

valoir plus tard que vous devriez absolument éviter d'utiliser des variables au niveau du package.

- **Lors de la déclaration et de l'initialisation, utilisez `:=`.** Lorsque vous déclarez et initialisez la variable en même temps, c'est-à-dire que nous ne laissons pas la variable être implicitement initialisée à sa valeur zéro, je vous recommande d'utiliser le formulaire de déclaration de variable courte. Il est clair pour le lecteur que la variable à gauche du `:=` est volontairement initialisée.

Pour expliquer pourquoi, regardons l'exemple précédent, mais cette fois en initialisant délibérément chaque variable:

```
var players int = 0

var things []Thing = nil

var thing *Thing = new(Thing)
json.Unmarshal(reader, thing)
```

Dans les premier et troisième exemples, dans Go, il n'y a pas de conversions automatiques d'un type à un autre; le type situé à gauche de l'opérateur d'affectation doit être identique à celui situé à droite. Le compilateur peut en déduire le type. De la variable déclarée à partir du type du côté droit, l'exemple peut être écrit de manière plus concise, comme ceci:

```
var players = 0

var things []Thing = nil

var thing = new(Thing)
json.Unmarshal(reader, thing)
```

Cela nous laisse avec une initialisation explicite des `players` sur 0 ce qui est redondant car 0 est la valeur zéro des joueurs. Il est donc préférable de préciser que nous allons utiliser la valeur zéro en écrivant à la place

```
var players int
```

Qu'en est-il de la deuxième déclaration? Nous ne pouvons pas écrire

```
var things = nil
```

Parce que `nil` n'a pas de type.^[4] Au lieu de cela, nous avons le choix. Voulons-nous la valeur zéro pour une *slice*?

```
var things []Thing
```

ou voulons-nous créer une *slice* avec zéro éléments ?

```
var things = make([]Thing, 0)
```

Si nous voulions pour ce dernier, que ce ne soit pas la valeur zéro pour la *slice*, alors nous devrions indiquer clairement au lecteur que nous faisons ce choix en utilisant la formulation de déclaration abrégé :

```
things := make([]Thing, 0)
```

Ce qui indique au lecteur que nous avons choisi d'initialiser les *things* explicitement.

Cela nous amène à la troisième déclaration,

```
var thing = new([]Thing)
```

Ce qui à la fois initialise explicitement une variable et introduit l'utilisation inhabituelle du mot clé *new* que certains développeurs Go n'aiment pas. Si nous appliquons notre recommandation de syntaxe de déclaration courte, l'instruction devient

```
thing := new([]Thing)
```

Ce qui indique clairement que *thing* est explicitement initialisée au résultat de *new(Thing)* - un pointeur vers un *Thing* - mais nous laisse toujours l'utilisation inhabituelle de *new*. Nous pourrions résoudre ce point en utilisant la forme compacte littérale initialisation de *struct*,

```
thing := &Thing{}
```

Ce qui fait la même chose que *new(Thing)*, d'où la raison pour laquelle certains développeur Go sont contrariés par cette duplication de fonctionnalité. Cependant, cela signifie que nous initialisons clairement *thing* avec un pointeur sur un *Thing{}*, qui est la valeur zéro pour un *Thing*.

Au lieu de cela, nous devrions reconnaître que *thing* est déclarée comme étant sa valeur zéro et utiliser l'adresse de l'opérateur pour transmettre l'adresse de *thing* à *json.Unmarshal*

```
var thing Thing  
json.Unmarshal(reader, &thing)
```

Bien sûr, comme avec n'importe quelle règle empirique, il y a des exceptions. Par exemple, parfois deux variables sont étroitement liées



```
var min int
max := 1000
```

Paraîtrait étrange. La déclaration serait plus lisible comme ceci

```
min, max := 0, 1000
```

En résumé:

- Lors de la déclaration d'une variable sans initialisation, utilisez la syntaxe `var`.
- Lors de la déclaration et de l'initialisation explicite d'une variable, utilisez `:=`.

Faites des déclarations compliquées évidentes.

Quand quelque chose est compliqué, cela devrait avoir l'air compliqué.

```
var length uint32 = 0x80
```



Ici, `length` peut être utilisé avec une bibliothèque qui requiert un type numérique spécifique et il est plus claire avec cette syntaxe que `length` choisie comme type `uint32` que dans la formulation de déclaration abrégé:

```
length := uint32i(0x80)
```

Dans le premier exemple, j'ai délibérément enfreint la règle qui consiste à utiliser la formulation de déclaration `var` avec un initialiseur explicite. Cette décision de changer la façon de faire habituel est un indice pour le lecteur que quelque chose d'inhabituel se produit.

2.6. Être un joueur dans une équipe

J'ai parlé d'un objectif de l'ingénierie logicielle visant à produire du code lisible et maintenable, et vous allez pouvoir consacrer l'essentiel de votre carrière à des projets dont vous n'êtes pas le seul auteur. Mon conseil dans cette situation est de suivre le style de l'équipe.

Changer de style au milieu d'un fichier est discordant. L'uniformité, même si ce n'est pas votre approche préférée, a plus de valeur pour la maintenabilité que vos préférences personnelles. Ma règle de base est la suivante: si le code passe dans `gofmt` alors cela ne vaut généralement pas la peine de faire une revue de code pour ça.



Si vous voulez renommer une base de code, ne le mélangez pas avec une autre modification: car si quelqu'un utilise `git bisect` pour retrouver un bug, il ne veut pas parcourir des milliers de lignes de renommage pour trouver également le code que vous avez modifié.

3. Les commentaires

Avant de passer à des points plus importants, je voudrais passer quelques minutes à parler de commentaires.

Un bon code a beaucoup de commentaires, un mauvais code nécessite beaucoup de commentaires.

— Dave Thomas and Andrew Hunt, *The Pragmatic Programmer*

Les commentaires sont très importants pour la lisibilité d'un programme Go. Chaque commentaire doit comporter une et une seule des trois informations suivantes:

1. Le commentaire devrait expliquer *quoi*, ce que fait l'objet.
2. Le commentaire devrait expliquer *comment* cela est fait.
3. Le commentaire devrait expliquer *pourquoi*, pour quelle raison cela est fait.

La première forme est idéale pour commenter un symbole public:

```
// Open opens the named file for reading.  
// If successful, methods on the returned file can be used for reading.
```

La seconde forme est idéale pour commenter une méthode:

```
// queue all dependant actions  
var results []chan error  
for _, dep := range a.Deps {  
    results = append(results, execute(seen, dep))  
}
```

La troisième forme, le *pourquoi*, est unique dans la mesure où elle ne déplace pas les deux premières, mais elle ne remplace pas non plus le *quoi* ou le *comment* : le style de commentaire *pourquoi* existe pour expliquer les facteurs externes qui dirige le code que vous lisez sur la page. Souvent, ces facteurs ont rarement un sens pris hors du contexte, le commentaire est là pour fournir ce contexte.


```
return &v2.Cluster_CommonLbConfig{
    // Disable HealthyPanicThreshold
    HealthyPanicThreshold: &envoy_type.Percent{
        Value: 0,
    },
}
```

Dans cet exemple, les effets de la mise à zéro de la valeur de `HealthyPanicThreshold` ne sont peut-être pas clairs immédiatement. Le commentaire est nécessaire pour préciser que la valeur 0 désactive le comportement du seuil de panique.

3.1. Les commentaires sur les variables et les constantes devraient décrire leur contenu et non leur objectif.

Lorsque vous ajoutez un commentaire à une variable ou à une constante, ce commentaire doit décrire le *contenu* de la variable et non son *objectif*.

```
const randomNumber = 6 // determined from an unbiased die
```

Dans cet exemple, le commentaire explique pourquoi la valeur six est attribuée à `randomNumber`. Le commentaire ne décrit pas l'utilisation où `randomNumber` sera utilisé. Voici d'autres exemples:

```
const (
    StatusContinue           = 100 // RFC 7231, 6.2.1
    StatusSwitchingProtocols = 101 // RFC 7231, 6.2.2
    StatusProcessing         = 102 // RFC 2518, 10.1

    StatusOK                 = 200 // RFC 7231, 6.3.1
```

Dans le contexte de *HTTP*, le code 100 est appelé `StatusContinue`, tel que défini dans la RFC 7231, section 6.2.1.

Pour les variables sans valeur initiale, le commentaire doit décrire qui est responsable de l'initialisation de cette variable.

```
// sizeCalculationDisabled indicates whether it is safe
// to calculate Types' widths and alignments. See dowidth.
var sizeCalculationDisabled bool
```

Ici le commentaire indique au lecteur que la fonction `dotwidth` est responsable du maintien de l'état de la variable `sizeCalculationDisabled`.

Se cacher à la vue de tous

Voici une astuce de Kate Gregory.^[6] Parfois, vous trouverez un meilleur nom pour une variable caché dans le commentaire.



```
// registry of SQL drivers
var registry = make(map[string]*sql.Driver)
```

Le commentaire a été ajouté par l'auteur car `registry` ne convient pas suffisamment à définir l'objectif - c'est un registre, mais un registre de quoi? En renommant la variable `sqlDrivers` il est maintenant clair que le but de cette variable est de contenir les drivers SQL.

```
var sqlDrivers = make(map[string]*sql.Driver)
```

Maintenant, le commentaire est redondant et peut être supprimé.

3.2. Toujours documenter les symboles publics

Puisque godoc est la documentation de votre package, vous devriez toujours ajouter un commentaire pour chaque symbole public - variable, constante, fonction et méthode - déclaré dans votre package.

Voici deux règles du guide de style Google

- Toute fonction publique qui n'est pas à la fois évidente et brève doit être commentée.
- Toute fonction d'une bibliothèque doit être commentée quelle que soit sa longueur ou sa complexité.

```
package ioutil
```

```
// ReadAll reads from r until an error or EOF and returns the data it read.  
// A successful call returns err == nil, not err == EOF. Because ReadAll is  
// defined to read from src until EOF, it does not treat an EOF from Read  
// as an error to be reported.  
func ReadAll(r io.Reader) ([]byte, error)
```

Il y a une exception à cette règle : vous n'avez pas besoin de documenter les méthodes qui implémentent une interface. En particulier, ne faites pas ça :

```
// Read implements the io.Reader interface  
func (r *FileReader) Read(buf []byte) (int, error)
```

Ce commentaire ne dit rien. Il ne vous dit pas ce que fait la méthode, en fait c'est pire, il vous dit d'aller chercher la documentation ailleurs. Dans cette situation, je suggère de supprimer complètement ce commentaire.

Voici un exemple du package io

```
// LimitReader returns a Reader that reads from r  
// but stops with EOF after n bytes.  
// The underlying implementation is a *LimitedReader.  
func LimitReader(r Reader, n int64) Reader { return &LimitedReader{r, n} }
```



```
// A LimitedReader reads from R but limits the amount of  
// data returned to just N bytes. Each call to Read  
// updates N to reflect the new amount remaining.  
// Read returns EOF when N <= 0 or when the underlying R returns EOF.  
type LimitedReader struct {  
    R Reader // underlying reader  
    N int64   // max bytes remaining  
}  
  
func (l *LimitedReader) Read(p []byte) (n int, err error) {  
    if l.N <= 0 {  
        return 0, EOF  
    }  
    if int64(len(p)) > l.N {  
        p = p[0:l.N]  
    }  
    n, err = l.R.Read(p)  
    l.N -= int64(n)  
    return  
}
```

Notez que la déclaration de LimitedReader est directement précédée de la fonction qui l'utilise, et la

déclaration de `LimitedReader` suit la déclaration de `LimitedReader` elle-même. Même si `LimitedReader.Read` n'a pas de documentation en soi, il est clair qu'il s'agit d'une implémentation de `io.Reader`.



Avant d'écrire la fonction, écrivez le commentaire décrivant la fonction. Si vous trouvez difficile d'écrire le commentaire, alors c'est un signe que le code que vous allez écrire va être difficile à comprendre.

4. Conception d'un package

Écrire du code timide - des modules qui ne révèlent rien d'inutile aux autres modules et qui ne s'appuient pas sur les implémentations d'autres modules.

— Dave Thomas

Chaque package Go est en fait son propre petit programme Go. Tout comme l'implémentation d'une fonction ou d'une méthode n'est pas importante pour l'appelant, l'implémentation des fonctions, méthodes et types qui composent l'API publique de votre package - son comportement - est sans importance pour l'appelant.

Un bon package Go doit s'efforcer d'avoir un faible degré de couplage de niveau de source de sorte que, au fur et à mesure que le projet se développe, les changements apportés à un package ne se répercutent pas sur l'ensemble du code. Ces refactorings stop-the-world imposent une limite stricte au taux de changement d'une base de code et donc à la productivité des membres travaillant dans cette base de code.

Dans cette section, nous parlerons de la conception d'un package - y compris les types de noms de noms de package, et des conseils pour écrire des méthodes et des fonctions.

4.1. Un bon package commence par son nom

Écrire un bon package Go commence par le choix d'un bon nom du package. Pensez au nom de votre package comme une accroche-éclair pour décrire ce qu'il fait en un seul mot.

Tout comme j'ai parlé des noms de variables dans la section précédente, le nom d'un package est très important. La règle de base que je suis n'est pas "quels types dois-je mettre dans ce package ?". Normalement, la réponse à cette question n'est pas "ce package fournit le type X", mais "ce package vous permet de parler HTTP".



Nommez votre package pour ce qu'il *fournit*, pas pour ce qu'il contient.

4.1.1. Les bons noms de packages doivent être uniques.

Dans votre projet, chaque nom de package doit être unique. Cela devrait être assez facile si vous avez suivi les conseils selon lesquels le nom d'un package devrait dériver de son but. Si vous trouvez que vous avez deux packages qui ont besoin du même nom, il est probable que ce soit l'une ou l'autre des raisons ; * Le nom du package est trop générique. * Le package chevauche un autre

package d'un nom similaire. Dans ce cas, vous devriez soit revoir votre design, soit envisager de fusionner les packages.

4.2. Évitez les noms de packages tels que `base`, `common` ou `util`

Une cause fréquente de mauvais noms de packages est ce que l'on appelle les *packages utilitaires*. Ce sont des packages où les aides et le code utilitaire communs se figent avec le temps. Comme ces packages contiennent un assortiment de fonctions sans rapport, leur utilité est difficile à décrire en termes de ce que le package fournit. Cela conduit souvent à ce que le nom du package soit dérivé de ce qu'il contient - *les utilitaires*.

Les noms de packages comme `utils` ou `helpers` se retrouvent couramment dans les grands projets qui ont développé des hiérarchies de packages profondes et qui veulent partager les fonctions d'aide sans rencontrer de boucles d'importation. En extrayant les fonctions utilitaires vers un nouveau package, la boucle d'importation est rompue, mais parce que le package provient d'un problème de conception dans le projet, son nom ne reflète pas son but, seulement sa fonction de rupture du cycle d'importation.

Ma recommandation pour améliorer le nom des packages d' `utils` ou d' `aides` est d'analyser où ils sont appelés et si possible de déplacer les fonctions pertinentes dans le package de l'appelant. Même si cela implique de dupliquer du code d'aide, c'est mieux que d'introduire une dépendance d'importation entre deux packages.

Une petite duplication est beaucoup moins chère qu'une mauvaise abstraction.

— Sandy Metz

Dans le cas où les fonctions utilitaires sont utilisées à de nombreux endroits, préférez plusieurs packages, chacun centré sur un seul aspect, à un seul package monolithique.



Utilisez le pluriel pour nommer les packages utilitaires. Par exemple `strings` pour les utilitaires de gestion des chaînes de caractères.

Les packages avec des noms tels que `base` ou `commun` sont souvent trouvés lorsque des fonctionnalités communes à deux ou plusieurs implémentations, ou des types communs pour un client et un serveur, ont été refaits dans un package séparé. Je crois que la solution est de réduire le nombre de packages, de combiner le client, le serveur et le code commun en un seul package nommé d'après la fonction du package.

Par exemple, le package `net/http` n'a pas de sous-packageage `client` et `server`, mais un fichier `client.go` et `server.go`, contenant chacun leurs types respectifs, et un fichier `transport.go` pour le code de transport du message commun.

Le nom d'un identificateur comprend le nom de son emballage.

Il est important de se rappeler que le nom d'un identificateur inclut le nom de son package.



- La fonction `Get` du package `net/http` devient `http.get` lorsqu'elle est référencée par un autre package.
- Le type de lecteur du package `Strings` devient `Strings.Reader` lorsqu'il est importé dans d'autres packages.
- L'interface `Error` du package `net` est clairement liée aux erreurs réseau.

4.3. Retourner le resultat au plus tôt, plutôt que dans de profondes imbrications de la fonction.

Comme Go n'utilise pas d'exceptions pour le flux de contrôle, il n'est pas nécessaire d'indenter profondément votre code juste pour fournir une structure de niveau supérieur pour les blocs `try` et `catch`. Plutôt que d'imbriquer le chemin qui réussit de plus en plus profondément vers la droite, le code Go est écrit dans un style où le chemin du succès continue sur l'écran au fur et à mesure que la fonction progresse. Mon ami Mat Ryer appelle cette pratique "le codage en ligne de mire". ^[7]

Ceci est réalisé en utilisant des clauses de garde, des blocs conditionnels avec des conditions préalables à l'entrée d'une fonction. Voici un exemple tiré du package `bytes`,

```
func (b *Buffer) UnreadRune() error {
    if b.lastRead <= opInvalid {
        return errors.New("bytes.Buffer: UnreadRune: previous operation was not a
successful ReadRune")
    }
    if b.off >= int(b.lastRead) {
        b.off -= int(b.lastRead)
    }
    b.lastRead = opInvalid
    return nil
}
```

En entrant dans `UnreadRune`, l'état de `b.lastRead` est vérifié et si l'opération précédente n'était pas `ReadRune`, une erreur est immédiatement renvoyée. De là, le reste de la fonction procède avec l'affirmation que `b.lastRead` est supérieur à `opInvalid`.

Comparez ceci à la même fonction écrite sans clause de garde,

```
func (b *Buffer) UnreadRune() error {
    if b.lastRead > opInvalid {
        if b.off >= int(b.lastRead) {
            b.off -= int(b.lastRead)
        }
        b.lastRead = opInvalid
        return nil
    }
    return errors.New("bytes.Buffer: UnreadRune: previous operation was not a
    successful ReadRune")
}
```

Le bloc de la condition qui réussit, cas le plus courant, est imbriqué à l'intérieur de la première condition `if` et la condition qui réussie retourne `nil`. La découverte de cela nécessite un contrôle minutieux des accolades de fermeture. La dernière ligne de la fonction renvoie maintenant une erreur, et l'appelé doit tracer l'exécution de la fonction jusqu'à l'accolade d'ouverture correspondante pour savoir quand le contrôle va atteindre ce point.

C'est plus sujet aux erreurs pour le lecteur et le développeur qui maintient le code, d'où la raison pour laquelle Go préfère utiliser des clauses de garde et retourner l'erreur au plus tôt.

4.4. Rendre la valeur zéro utile

Chaque déclaration de variable, en supposant qu'aucun initialisateur explicite n'est fourni, sera automatiquement initialisée à une valeur qui correspond au contenu de la mémoire mise à zéro. Il s'agit de la valeur zéro. Le type de la valeur détermine la *valeur zéro* de la valeur ; pour les types numériques, elle est égale à zéro, pour les types de pointeurs, elle est égale à `nil`, de même pour les *slices*, les *maps* et les *channels*.

Cette propriété de toujours mettre une valeur à une valeur par défaut connue est importante pour la sécurité et l'exactitude de votre programme et peut rendre vos programmes Go plus simples et plus compacts. C'est ce dont parlent les programmeurs de Go lorsqu'ils disent "donnez à vos structures une valeur zéro utile".

Considérons le type `sync.Mutex`. `sync.Mutex` contient deux champs entiers non exportés, représentant l'état interne du mutex. Grâce à la valeur zéro, ces champs seront mis à 0 à chaque fois qu'un `sync.Mutex` est déclaré. `sync.Mutex` a été délibérément codé pour profiter de cette propriété, rendant le type utilisable sans initialisation explicite.

```

type MyInt struct {
    mu sync.Mutex
    val int
}

func main() {
    var i MyInt

    // i.mu is usable without explicit initialisation.
    i.mu.Lock()
    i.val++
    i.mu.Unlock()
}

```

Un autre exemple d'un type avec une valeur zéro utile est `bytes.Buffer`. Vous pouvez déclarer un `bytes.Buffer` et commencer à y écrire sans initialisation explicite.

```

func main() {
    var b bytes.Buffer
    b.WriteString("Hello, world!\n")
    io.Copy(os.Stdout, &b)
}

```

Une propriété utile des `slices` est que leur valeur zéro est `nil`. Cela a du sens si l'on considère la définition d'un en-tête de la `slice` dans le runtime.

```

type slice struct {
    array *[]T // pointer to the underlying array
    len   int
    cap   int
}

```

La valeur zéro de cette structure impliquerait que `len` et `cap` ont la valeur `0`, et le `array`, le pointeur vers la mémoire contenant le contenu du tableau de support de la `slice`, serait `nil`. Cela signifie que vous n'avez pas besoin de faire explicitement une `slice`, vous pouvez simplement le déclarer.

```

func main() {
    // s := make([]string, 0)
    // s := []string{}
    var s []string

    s = append(s, "Hello")
    s = append(s, "world")
    fmt.Println(strings.Join(s, " "))
}

```




`var s []string` est similaire aux deux lignes commentées au-dessus, mais pas identique. Il est possible de détecter la différence entre une *slice* de valeur nulle et la valeur d'une *slice* de longueur nulle. Le code suivant affichera `false`.

```
func main() {  
    var s1 = []string{}  
    var s2 []string  
    fmt.Println(reflect.DeepEqual(s1, s2))  
}
```

Une propriété utile, quoique surprenante, des pointeurs non initialisés - aucun pointeur - est que vous pouvez appeler des méthodes sur des types qui ont une valeur nulle. Ceci peut être utilisé pour fournir des valeurs par défaut simplement.

```
type Config struct {  
    path string  
}  
  
func (c *Config) Path() string {  
    if c == nil {  
        return "/usr/home"  
    }  
    return c.path  
}  
  
func main() {  
    var c1 *Config  
    var c2 = &Config{  
        path: "/export",  
    }  
    fmt.Println(c1.Path(), c2.Path())  
}
```

4.5. Évitez les déclarations globales dans les packages

La clé pour écrire des programmes maintenables est qu'ils doivent être couplés de manière lâche - un changement dans un package devrait avoir une faible probabilité d'affecter un autre package qui ne dépend pas directement du premier.

Il y a deux excellentes façons d'obtenir un couplage lâche en Go

1. Utilisez des interfaces pour décrire le comportement de vos fonctions ou méthodes.
2. Éviter l'utilisation de l'état global.

Dans Go, nous pouvons déclarer des variables à la portée de la fonction ou de la méthode, ainsi qu'à la portée du package. Lorsque la variable est publique, avec un identificateur commençant par une majuscule, sa portée est effectivement globale à l'ensemble du programme - tout package peut

observer le type et le contenu de cette variable à tout moment.

L'état global mutable introduit un couplage étroit entre les parties indépendantes de votre programme car les variables globales deviennent un paramètre invisible pour chaque fonction de votre programme ! Toute fonction qui repose sur une variable globale peut être cassée si le type de cette variable change. Toute fonction qui repose sur l'état d'une variable globale peut être cassée si une autre partie du programme change cette variable.

Si vous voulez réduire le couplage, une variable globale est créée,

1. Déplacez les variables pertinentes en tant que champs sur les structures qui en ont besoin.
2. Utiliser des interfaces pour réduire le couplage entre le comportement et la mise en œuvre de ce comportement.

5. Structure du projet

Parlons de la combinaison des packages dans un projet. Généralement, il s'agira d'un dépôt git unique. A l'avenir, les développeurs de Go utiliseront les termes *module* et *projet* de manière interchangeable.

Tout comme un ensemble, chaque projet doit avoir un objectif clair. Si votre projet est une bibliothèque, il devrait fournir une chose, comme l'analyse XML ou la journalisation. Vous devriez éviter de combiner plusieurs buts en un seul projet, cela vous aidera à éviter la redoutable bibliothèque `commun`.



D'après mon expérience, le repo `commun` finit par être étroitement lié à son plus gros consommateur, ce qui rend difficile la mise à jour des correctifs de back-port sans mise à niveau à la fois du commun et du consommateur, ce qui entraîne de nombreux changements non liés et une rupture de l'API en cours de route.

Si votre projet est une application, comme votre application web, le contrôleur Kubernetes, et ainsi de suite, alors vous pourriez avoir un ou plusieurs packages principaux dans votre projet. Par exemple, le contrôleur Kubernetes sur lequel je travaille possède un seul package `cmd/contour` qui sert à la fois de serveur déployé sur un cluster Kubernetes et de client pour le débogage.

5.1. Envisagez des packages moins nombreux et plus grands

Une des choses que j'ai tendance à retenir dans la révision de code pour les programmeurs qui passent d'autres langages à Go est qu'ils ont tendance à trop utiliser les packages.

Go ne fournit pas de moyens élaborés d'établir la visibilité. Go manque de modificateurs d'accès de Java, `public`, `protected`, `private` et de façon implicite `default`. Il n'y a pas d'équivalent à la notion de classes `friend` du C++.

En Go nous n'avons que deux modificateurs d'accès, `public` et `privé`, indiqués par la majuscule de la première lettre de l'identifiant. Si un identifiant est `public`, son nom commence par une majuscule,

cet identifiant peut être référencé par tout autre package Go.



Il se peut que vous entendiez des gens dire "*exporté*" et "*non exporté*" comme synonymes pour public et privé.

Étant donné le choix limité pour contrôler l'accès aux symboles d'un package, quelles pratiques les programmeurs Go doivent-ils suivre pour éviter de créer des hiérarchies de packages trop compliquées ?



Chaque package, à l'exception de `cmd/` et `internal/`, doit contenir du code source.

Le conseil que je me surprends à répéter est de préférer des packages moins nombreux et plus grands. Votre position par défaut devrait être de ne pas créer un nouveau package. Cela conduira à ce que trop de types soient rendus publics, créant ainsi une surface API large et peu profonde pour votre package...

Les sections qui suivent explorent cette suggestion plus en détail.



Vous venez de Java ?

Si vous venez de Java ou C#, considérez cette règle empirique. - Un package Java est l'équivalent d'un seul fichier source `.go`. - Un package Go est l'équivalent d'un module Maven complet ou d'un assemblage `.NET`.

5.1.1. Classer le code dans des fichiers par instructions d'importation

Si vous organisez vos packages en fonction de ce qu'ils fournissent aux appelants, devriez-vous faire de même pour les fichiers d'un package Go ? Comment savez-vous quand vous devez diviser un fichier `.go` en plusieurs fichiers ? Comment savoir si vous êtes allé trop loin et si vous devriez envisager de consolider vos fichiers `.go` ?

Voici les lignes directrices que j'utilise :

- Commencez chaque package avec un fichier `.go`. Donnez à ce fichier le même nom que le nom du dossier. Par exemple, le package `http` doit être placé dans un fichier appelé `http.go` dans un répertoire nommé `http`.
- Par exemple, `messages.go` contient les types `Request` et `Response`, `client.go` contient le type `Client`, `server.go` contient le type `Server`.
- Si vous trouvez que vos fichiers ont des déclarations d'`import` similaires, envisagez de les combiner. Sinon, identifiez les différences entre les ensembles d'importation et déplacez-les.
- Différents fichiers devraient être responsables de différentes parties du package. `messages.go` peut être responsable de la répartition des requêtes HTTP et des réponses connecté et déconnecté, `http.go` peut contenir la logique de gestion de réseau de bas niveau, `client.go` et `server.go` implémentent la logique métier HTTP de construction ou de routage des requêtes, etc.



Préférez des noms (plutôt que des verbes) pour nommer vos fichiers sources.



Le compilateur Go compile chaque package en parallèle. Dans un package, le compilateur compile chaque fonction en parallèle (les méthodes sont juste des fonctions fantaisistes dans Go). Changer la disposition de votre code à l'intérieur d'un package ne devrait pas affecter le temps de compilation.

5.1.2. Préférez les tests internes aux tests externes

L'outil `go test` permet d'écrire vos tests de packages à deux endroits. En supposant que votre package s'appelle `http2`, vous pouvez écrire un fichier `http2_test.go` et utiliser la déclaration `http2` du package. Cela compilera le code dans `http2_test.go` comme s'il faisait partie du package `http2`. C'est ce qu'on appelle familièrement un *test interne*.

L'outil `go test` prend également en charge une déclaration de package spéciale, se terminant par `test`, dans notre exemple le package s'appellera `http_test`. Cela permet à vos fichiers de test de cohabiter avec votre code dans le même package, mais lorsque ces tests sont compilés, ils ne font pas partie du code de votre package, ils vivent dans leur propre package. Cela vous permet d'écrire vos tests comme si vous étiez un autre package appelant dans votre code. C'est ce qu'on appelle un *test externe*.

Je vous recommande d'utiliser des tests internes lorsque vous écrivez des tests unitaires pour votre *package*. Cela vous permet de tester chaque fonction ou méthode directement, en évitant la bureaucratie des tests externes.

Toutefois, vous *devez placer* vos exemples de fonctions de test dans un fichier de test externe. Ceci permet de s'assurer que lorsqu'ils sont visualisés dans godoc, les exemples ont le préfixe de package approprié et peuvent être facilement copiés et collés.



Éviter les hiérarchies de packages complexes, résister au désir d'appliquer la taxonomie

À une exception près, dont nous parlerons plus loin, la hiérarchie des packages Go n'a aucune signification pour l'outil `go`. Par exemple, le package `net/http` n'est pas un enfant ou un sous-packageage du package `net`.

Si vous trouvez que vous avez créé des répertoires intermédiaires dans votre projet qui ne contiennent pas de `fichiers.go`, vous n'avez peut-être pas suivi ce conseil.

5.1.3. Utilisez des packages internes pour réduire la surface de votre API publique

Si votre projet contient plusieurs packages, vous pouvez trouver des fonctions exportées qui sont destinées à être utilisées par d'autres packages dans votre projet, mais qui ne sont pas destinées à faire partie de l'API publique de votre projet. Si vous vous trouvez dans cette situation, l'outil `go` reconnaît un nom de dossier spécial - et non un nom de package -, `internal/` qui peut être utilisé pour placer du code qui est public pour votre projet, mais privé pour les autres projets.

Pour créer un tel package, placez-le dans un répertoire nommé `internal/` ou dans un sous-

répertoire d'un répertoire nommé `internal/`. Lorsque la commande `go` voit l'importation d'un package dont le chemin d'accès est `internal`, elle vérifie que le package qui effectue l'importation se trouve dans l'arborescence enracinée dans le `parent` du répertoire `internal`.

Par exemple, un package `.../a/b/c/internal/d/e/f` ne peut être importé que par le code présent dans l'arborescence de répertoires en racine à `.../a/b/c`. Il ne peut pas être importé par le code présent dans `.../a/b/g` ou dans tout autre référentiel.^[8]

5.2. Garder le package principal le plus petit possible.

Votre fonction `main` et votre package `main` devraient en faire le moins possible. C'est parce que `main.main` agit comme un singleton ; il ne peut y avoir qu'une seule fonction principale dans un programme, y compris les tests.

Parce que `main.main` est un singleton, il y a beaucoup d'hypothèses intégrées dans les choses que `main.main` appellera qu'elles ne seront appelées que pendant `main.main` ou `main.init`, et seulement une fois. Il est donc difficile d'écrire des tests pour du code écrit dans `main.main`, donc vous devriez vous efforcer de déplacer le plus possible votre logique métier hors de votre fonction principale et idéalement hors de votre package `main`.



`func main()` devrait analyser les `flags` ouvrir les connexions aux bases de données, les loggers, et ainsi de suite, puis transférer l'exécution à un objet de haut niveau.

6. Conception d'API

Le dernier conseil de conception que je vais donner aujourd'hui est le plus important.

Toutes les suggestions que j'ai faites jusqu'ici ne sont que des suggestions. C'est comme ça que j'essaie d'écrire mes programmes Go, mais je ne vais pas les pousser à fond dans la révision du code.

Cependant, lorsqu'il s'agit d'examiner les API lors de l'examen du code, je suis moins indulgent. C'est parce que tout ce dont j'ai parlé jusqu'à présent peut être corrigé sans rompre la rétrocompatibilité ; ce sont, pour la plupart, des détails de mise en œuvre.

Lorsqu'il s'agit de l'API publique d'un package, il est payant de réfléchir longuement à la conception initiale, car la modification ultérieure de cette conception va perturber les personnes qui utilisent déjà votre API.

6.1. Concevoir des API qui sont difficiles à utiliser à mauvais escient.

Les API devraient être faciles à utiliser et difficiles à mal utiliser.^[10]

— Josh Bloch

Si vous deviez retirer quelque chose de cette présentation, ce devrait être ce conseil de Josh Bloch.

Si une API est difficile à utiliser pour des choses simples, alors chaque appel de l'API aura l'air compliquée. Lorsque l'appel réelle de l'API est compliquée, elle sera moins évidente et plus susceptible d'être négligée.

6.1.1. Méfiez-vous des fonctions qui prennent plusieurs paramètres du même type

Un bon exemple d'API simple, mais difficile à utiliser correctement est celle qui prend deux ou plusieurs paramètres du même type. Comparons deux fonctions :

```
func Max(a, b int) int
func CopyFile(to, from string) error
```

Quelle est la différence entre ces deux fonctions ? Évidemment, l'une renvoie le maximum de deux chiffres, l'autre copie un fichier, mais ce n'est pas la chose la plus importante.

```
Max(8, 10) // 10
Max(10, 8) // 10
```

Max est *commutatif* ; l'ordre de ses paramètres n'a pas d'importance. Le maximum de huit et dix est dix, peu importe si je compare huit et dix ou dix et huit. Cependant, cette propriété ne s'applique pas à *CopyFile*.

```
CopyFile("/tmp/backup", "presentation.md")
CopyFile("presentation.md", "/tmp/backup")
```

Lequel de ces appels a fait une copie de sauvegarde de votre présentation et lequel a remplacé votre présentation par la version de la semaine dernière ? On ne peut pas le dire sans consulter la documentation. La personne qui va réviser le code ne peut pas savoir si vous avez passé la bonne commande sans consulter la documentation.

Une solution possible est d'introduire un type d'aide qui sera responsable de l'appel correct de *CopyFile*.

```
type Source string

func (src Source) CopyTo(dest string) error {
    return CopyFile(dest, string(src))
}

func main() {
    var from Source = "presentation.md"
    from.CopyTo("/tmp/backup")
}
```

De cette façon, `CopyFile` est toujours appelé correctement - ce qui peut être affirmé avec un test unitaire - et peut éventuellement être rendu privé, ce qui réduit davantage les risques d'utilisation abusive.



Les APIs avec plusieurs paramètres du même type sont difficiles à utiliser correctement.

6.2. Concevoir des API pour leur cas d'utilisation par défaut

Il y a quelques années, j'ai donné une conférence ^[11] sur l'utilisation des options fonctionnelles ^[12] pour rendre les API plus faciles à utiliser pour leur cas par défaut. L'essentiel de cet exposé était que vous devriez concevoir vos APIs pour le cas d'utilisation commun. Autrement dit, votre API ne devrait pas exiger de l'appelant qu'il fournisse des paramètres dont il ne se soucie pas.

6.2.1. Décourager l'utilisation de `nil` comme paramètre.

J'ai ouvert ce chapitre avec la suggestion que vous ne devriez pas forcer l'appelant de votre API à vous fournir des paramètres quand ils ne se soucient pas vraiment de ce que ces paramètres signifient. C'est ce que je veux dire quand je dis *concevoir des APIs pour leur cas d'utilisation par défaut*.

Voici un exemple du paquet `net/http`

```
package http

// ListenAndServe listens on the TCP network address addr and then calls
// Serve with handler to handle requests on incoming connections.
// Accepted connections are configured to enable TCP keep-alives.
//
// The handler is typically nil, in which case the DefaultServeMux is used.
//
// ListenAndServe always returns a non-nil error.
func ListenAndServe(addr string, handler Handler) error {
```

`ListenAndServe` prend deux paramètres, une adresse TCP pour écouter les connexions entrantes et `http.Handler` pour gérer la requête HTTP entrante. `Serve` permet au second paramètre d'être nul, et note que l'appelant passera généralement `nil` indiquant qu'il veut utiliser `http.DefaultServeMux` comme paramètre implicite.

Maintenant, l'appelant de `Serve` a deux façons de faire la même chose.

```
http.ListenAndServe("0.0.0.0:8080", nil)
http.ListenAndServe("0.0.0.0:8080", http.DefaultServeMux)
```

Les deux font exactement la même chose.

Ce comportement `nil` est viral. Le paquet `http` dispose également d'un assistant `http.Serve`, que vous pouvez raisonnablement imaginer sur lequel `ListenAndServe` s'appuie comme ceci

```
func ListenAndServe(addr string, handler Handler) error {
    l, err := net.Listen("tcp", addr)
    if err != nil {
        return err
    }
    defer l.Close()
    return Serve(l, handler)
}
```

Comme `ListenAndServe` permet à l'appelant de passer `nil` pour le second paramètre, `http.Serve` supporte également ce comportement. En fait, `http.Serve` est celui qui implémente la logique "si `handler` est `nil`, utilise `DefaultServeMux`". Accepter `nil` pour un paramètre peut amener l'appelant à penser qu'il peut passer `nil` pour les deux paramètres. Quoi qu'il en soit, appeler `Serve` comme ça,

`http.Serve(nil, nil)`

provoque une affreuse panique.



Ne mélangez pas les paramètres `nil` et non `nil-able` dans la même déclaration de fonction.

L'auteur de `http.ListenAndServe` essayait de rendre la vie de l'utilisateur de l'API plus facile dans le cas commun, mais a peut-être rendu le paquet plus difficile à utiliser en toute sécurité.

Il n'y a pas de différence dans le nombre de lignes entre l'utilisation de `DefaultServeMux` explicitement, ou implicitement via `nil`.

```
const root = http.Dir("/htdocs")
http.Handle("/", http.FileServer(root))
http.ListenAndServe("0.0.0.0:8080", nil)
```

par rapport à

```
const root = http.Dir("/htdocs")
http.Handle("/", http.FileServer(root))
http.ListenAndServe("0.0.0.0.0.0:8080", http.DefaultServeMux)
```

et cette confusion valait-elle vraiment la peine de sauver une ligne ?

```
const root = http.Dir("/htdocs")
mux := http.NewServeMux()
mux.Handle("/", http.FileServer(root))
http.ListenAndServe("0.0.0.0.0.0:8080", mux)
```




Réfléchissez sérieusement au temps que les fonctions d'aide vont permettre d'économiser au programmeur. Clair, c'est mieux que concis.



Éviter les API publiques avec des paramètres de test uniquement

Évitez d'exposer des API dont les valeurs ne diffèrent que par la portée du test. Au lieu de cela, utilisez des wrappers publics pour masquer ces paramètres, utilisez les aides de portée de test pour définir la propriété dans la portée de test.

6.2.2. Préférez les paramètres `var args` plutôt que `[]T`

Il est très courant d'écrire une fonction ou une méthode qui prend en paramètre une *slice*.

```
func ShutdownVMs(ids []string) error
```

C'est juste un exemple que j'ai inventé, mais c'est commun à beaucoup de code sur lequel j'ai travaillé. Le problème avec de telles déclaration de fonction, c'est qu'elles supposent qu'elles seront appelées avec plus d'une entrée. Cependant, ce que j'ai trouvé, c'est que plusieurs fois ce type de fonctions sont appelées avec un seul argument, qui doit être "mis en boîte" à l'intérieur d'une slice juste pour répondre aux exigences de la signature des fonctions.

De plus, comme le paramètre `ids` est une *slice*, vous pouvez passer une slice vide ou `nil` à la fonction et le compilateur sera content. Cela ajoute une charge de test supplémentaire parce que vous devriez couvrir ces cas dans vos tests.

Pour donner un exemple de cette classe d'API, j'ai récemment refactorisé un élément de logique qui m'obligeait à définir des champs supplémentaires si au moins un des paramètres d'un ensemble était non nul. La logique ressemblait à ceci :

```
if svc.MaxConnections > 0 || svc.MaxPendingRequests > 0 || svc.MaxRequests > 0 || svc
.MaxRetries > 0 {
    // apply the non zero parameters
}
```

Comme l'instruction `if` devenait très longue, j'ai voulu tirer la logique du check out dans sa propre fonction. Voilà ce que j'ai trouvé :

```
// anyPositive indicates if any value is greater than zero.
func anyPositive(values ...int) bool {
    for _, v := range values {
        if v > 0 {
            return true
        }
    }
    return false
}
```

Cela m'a permis de rendre clair pour le lecteur la condition où le bloc intérieur sera exécuté :

```
if anyPositive(svc.MaxConnections, svc.MaxPendingRequests, svc.MaxRequests, svc
    .MaxRetries) {
    // apply the non zero parameters
}
```

Cependant il y a un problème avec `anyPositif`, quelqu'un pourrait accidentellement l'invoquer comme ceci:

```
if anyPositive() { ... }
```

Dans ce cas, `anyPositive` retournerait `false` car il exécuterait zéro itération et retournerait immédiatement `false`. Ce n'est pas la pire chose au monde - ça le serait si `anyPositif` retournait `true` quand aucun argument n'est passé. Néanmoins, il vaudrait mieux que nous puissions changer la déclaration de `anyPositif` pour imposer que l'appelant passe au moins un argument. Nous pouvons le faire en combinant des paramètres normaux et `var arg` comme ceci :

```
// anyPositive indicates if any value is greater than zero.
func anyPositive(first int, rest ...int) bool {
    if first > 0 {
        return true
    }
    for _, v := range rest {
        if v > 0 {
            return true
        }
    }
    return false
}
```

Maintenant `anyPositif` ne peut pas être appelé avec moins d'un argument.

6.3. Let functions define the behaviour they requires

Supposons que l'on m'ait confié la tâche d'écrire une fonction qui permet de rendre persistante sur disque une structue `Documents`.

```
// Save writes the contents of doc to the file f.
func Save(f *os.File, doc *Document) error
```

Je pourrais écrire cette fonction, `Save`, qui prend `*os.File` comme destination pour écrire le Document. Mais cela pose quelques problèmes

La déclaration de `Save` empêche la possibilité d'écrire les données dans un emplacement réseau. En

supposant que le stockage en réseau est susceptible de devenir nécessaire plus tard, la définition de cette fonction devrait changer, ce qui aurait un impact sur tous ses appelants. De plus `Save` est pénible à tester, car il fonctionne directement avec des fichiers sur disque. Ainsi, pour vérifier son fonctionnement, le test devrait lire le contenu du fichier après avoir été écrit. Et je devrais également m'assurer que `f` est écrit dans un endroit temporaire et toujours supprimé par la suite.

`*os.File` définit également un grand nombre de méthodes qui ne sont pas pertinentes pour l'enregistrement, comme la lecture des répertoires et la vérification pour voir si un chemin est un lien symbolique. Il serait utile que la définition de la fonction `Save` ne puisse décrire que les parties de `*os.File` qui sont pertinentes.

Que pouvons-nous faire ?

```
// Save writes the contents of doc to the supplied
// ReadWriterCloser.
func Save(rwc io.ReadWriterCloser, doc *Document) error
```

En utilisant `io.ReadWriterCloser` nous pouvons appliquer le principe de ségrégation d'interface pour redéfinir `Save` pour prendre une interface qui décrit des choses plus générales en forme de fichier.

Avec ce changement, n'importe quel type qui implémente l'interface `io.ReadWriterCloser` peut être substitué au précédent `*os.File`.

Ceci rend l'enregistrement à la fois plus large dans son application, et clarifie à l'appelant de l'enregistrement quelles méthodes du type `*os.File` sont pertinentes pour son opération.

Et en tant qu'auteur de `Save`, je n'ai plus la possibilité d'appeler ces méthodes non liées sur `*os.File` car elles sont cachées derrière l'interface `io.ReadWriterCloser`.

Mais nous pouvons aller un peu plus loin avec le principe de la ségrégation des interfaces.

Premièrement, il est peu probable que si la fonction `Save` suit le principe de la responsabilité unique, elle lira le fichier qu'elle vient d'écrire pour en vérifier le contenu, ce qui devrait être la responsabilité d'un autre code.

```
// Save writes the contents of doc to the supplied
// WriterCloser.
func Save(wc io.WriterCloser, doc *Document) error
```

Ainsi, nous pouvons réduire la spécification de l'interface que nous passons à `Save` à l'écriture et à la fermeture.

Deuxièmement, en dotant `Save` d'un mécanisme de fermeture de son flux, dont nous avons hérité dans ce désir de le faire ressembler encore à un fichier, la question se pose de savoir dans quelles circonstances le `wc` sera fermé.

Éventuellement, `Save` appellera `Close` sans condition, ou peut-être `Close` sera appelé en cas de succès. Cela pose un problème pour l'appelant de `Save` car il peut vouloir écrire des données

supplémentaires dans le flux après que le document soit écrit.

```
// Save writes the contents of doc to the supplied
// Writer.
func Save(w io.Writer, doc *Document) error
```

Une meilleure solution serait de redéfinir `Save` pour ne prendre qu'un `io.Writer`, le décharger complètement de la responsabilité de faire autre chose que d'écrire des données dans un flux.

En appliquant le principe de ségrégation d'interface à notre fonction `Save`, les résultats ont été simultanément une fonction qui est la plus spécifique par rapport à ses exigences - elle n'a besoin que d'une chose inscriptible - et la plus générale par rapport à sa fonction, nous pouvons maintenant utiliser `Save` pour sauvegarder nos données dans tout fichier qui implémente `io.Writer`.

7. Traitement des erreurs

J'ai donné plusieurs présentations sur la gestion des erreurs footnote:[1] et j'ai beaucoup écrit sur la gestion des erreurs sur mon blog. J'ai aussi beaucoup parlé de la gestion des erreurs lors de la séance d'hier, alors je ne répéterai pas ce que j'ai dit.

- <https://dave.cheney.net/2014/12/24/inspecting-errors>
- <https://dave.cheney.net/2016/04/07/constant-errors>

Au lieu de cela, je veux couvrir deux autres domaines liés à la gestion des erreurs.

7.1. Éliminer la gestion des erreurs en éliminant les erreurs

Si vous étiez dans ma présentation hier, j'ai parlé des ébauches de propositions visant à améliorer le traitement des erreurs. Mais savez-vous ce qui est mieux qu'une syntaxe améliorée pour la gestion des erreurs ? Pas besoin de gérer les erreurs du tout.



Je ne dis pas "supprimer la gestion des erreurs". Ce que je suggère, c'est de changer votre code pour que vous n'ayez pas d'erreurs à gérer.

Cette section s'inspire du livre récent de John Ousterhout, *A philosophy of Software Design* footnote:[2]. L'un des chapitres de ce livre s'intitule "Définir les erreurs hors de l'existence". Nous allons essayer d'appliquer ce conseil à Go.

7.1.1. Compteur de lignes

Écrivons une fonction pour compter le nombre de lignes dans un fichier.

```
func CountLines(r io.Reader) (int, error) {
    var (
        br    = bufio.NewReader(r)
        lines int
        err    error
    )

    for {
        _, err = br.ReadString('\n')
        lines++
        if err != nil {
            break
        }
    }

    if err != io.EOF {
        return 0, err
    }
    return lines, nil
}
```

Parce que nous suivons nos conseils des sections précédentes, `CountLines` prend un `io.Reader`, pas un `*os.File` ; c'est le travail de l'appelant de fournir le `io.Reader` qui est le contenu que nous voulons compter.

Nous construisons un `bufio.Reader`, puis nous nous asseyons dans une boucle en appelant la méthode `ReadString`, en incrémentant un compteur jusqu'à atteindre la fin du fichier, puis nous retournons le nombre de lignes lues. Au moins c'est le code que nous voulons écrire, mais au lieu de cela cette fonction est rendue plus compliquée par la gestion des erreurs. Par exemple, il y a cette construction étrange,

```
_, err = br.ReadString('\n')
lines++
if err != nil {
    break
}
```

Nous incrémentons le nombre de lignes avant de vérifier l'erreur, qui semble étrange.

La raison pour laquelle nous devons l'écrire de cette façon est que `ReadString` retournera une erreur s'il rencontre un caractère de fin de fichier avant de frapper une nouvelle ligne. Cela peut se produire s'il n'y a pas de nouvelle ligne finale dans le fichier.

Pour essayer de résoudre ce problème, nous réarrangeons la logique pour incrémenter le nombre de lignes, puis voyons si nous avons besoin de sortir de la boucle.



cette logique n'est toujours pas parfaite, pouvez-vous repérer le bug ?

Mais nous n'avons pas encore fini de vérifier les erreurs. `ReadString` renvoie `io.EOF` lorsqu'il atteint la fin du fichier. Cela est attendu, `ReadString` a besoin d'un moyen de dire stop, il n'y a plus rien à lire. Donc avant de renvoyer l'erreur à l'appelant de `CountLine`, nous devons vérifier si l'erreur n'était pas `io.EOF`, et dans ce cas la propager, sinon nous retournons zéro pour dire que tout fonctionne bien.

Je pense que c'est un bon exemple de l'observation de Russ Cox selon laquelle la gestion des erreurs peut obscurcir le fonctionnement de la fonction. Regardons une version améliorée.

```
func CountLines(r io.Reader) (int, error) {
    sc := bufio.NewScanner(r)
    lines := 0

    for sc.Scan() {
        lines++
    }
    return lines, sc.Err()
}
```

Cette version améliorée passe de `bufio.Reader` à `bufio.Scanner`.

Sous le capot `bufio.Scanner` utilise `bufio.Reader`, mais il ajoute une belle couche d'abstraction qui aide à supprimer la gestion des erreurs avec le fonctionnement obscurci de `CountLines`.



`bufio.Scanner` peut rechercher n'importe quel motif, mais par défaut, il recherche les nouvelles lignes.

La méthode `sc.Scan()` retourne `true` si l'analyseur a trouvé une ligne de texte et n'a pas rencontré d'erreur. Ainsi, le corps de notre boucle `for` sera appelé seulement quand il y a une ligne de texte dans le tampon du scanner. Cela signifie que notre `CountLines` révisé gère correctement le cas où il n'y a pas de nouvelle ligne de suivi, et traite également le cas où le fichier était vide.

Deuxièmement, comme `sc.Scan` renvoie `false` lorsqu'une erreur est rencontrée, notre boucle `for` se ferme lorsque la fin du fichier est atteinte ou qu'une erreur est rencontrée. Le type `bufio.Scanner` mémorise la première erreur qu'il a rencontrée et nous pouvons récupérer cette erreur une fois que nous avons quitté la boucle en utilisant la méthode `sc.Err()`.

Enfin, `sc.Err()` se charge de gérer `io.EOF` et le convertira en `nil` si la fin du fichier est atteinte sans rencontrer une autre erreur.



Lorsque vous vous trouvez confronté à la gestion des erreurs, essayez d'extraire certaines opérations dans un type d'aide.

7.1.2. WriteResponse

Mon deuxième exemple est inspiré du post du blog [Errors are values](#).^{footnote:[]}

Plus tôt dans cette présentation, nous avons vu des exemples portant sur l'ouverture, la rédaction et

la fermeture de dossiers. La gestion des erreurs est présente, mais pas écrasante car les opérations peuvent être encapsulées dans des aides comme `ioutil.ReadFile` et `ioutil.WriteFile`. Cependant, lorsqu'il s'agit de protocoles réseau de bas niveau, il devient nécessaire de construire la réponse directement en utilisant des primitives d'E/S, le traitement des erreurs peut devenir répétitif. Considérez ce fragment d'un serveur HTTP qui construit la réponse HTTP.

```
type Header struct {
    Key, Value string
}

type Status struct {
    Code    int
    Reason  string
}

func WriteResponse(w io.Writer, st Status, headers []Header, body io.Reader) error {
    _, err := fmt.Fprintf(w, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)
    if err != nil {
        return err
    }

    for _, h := range headers {
        _, err := fmt.Fprintf(w, "%s: %s\r\n", h.Key, h.Value)
        if err != nil {
            return err
        }
    }

    if _, err := fmt.Fprint(w, "\r\n"); err != nil {
        return err
    }

    _, err = io.Copy(w, body)
    return err
}
```

Tout d'abord, nous construisons la ligne d'état en utilisant `fmt.Fprintf`, et vérifions l'erreur. Ensuite, pour chaque en-tête, nous écrivons la clé et la valeur de l'en-tête, en vérifiant l'erreur à chaque fois. Enfin, nous terminons la section d'en-tête avec un `\r\n` supplémentaire, vérifions l'erreur, et copions le corps de la réponse au client. Enfin, bien que nous n'ayons pas besoin de vérifier l'erreur de `io.Copy`, nous devons la traduire des deux formulaires de valeur de retour que `io.Copy` renvoie dans la valeur de retour unique que `WriteResponse` renvoie.

C'est beaucoup de travail répétitif. Mais nous pouvons nous faciliter la tâche en introduisant un petit *wrapper*, `errWriter`.

`errWriter` remplit le contrat `io.Writer` de sorte qu'il peut être utilisé pour envelopper un `io.Writer` existant. `errWriter` transmet les écrits à son auteur sous-jacent jusqu'à ce qu'une erreur soit détectée. À partir de ce moment, il supprime toute écriture et renvoie l'erreur précédente.

```

type errWriter struct {
    io.Writer
    err error
}

func (e *errWriter) Write(buf []byte) (int, error) {
    if e.err != nil {
        return 0, e.err
    }
    var n int
    n, e.err = e.Writer.Write(buf)
    return n, nil
}

func WriteResponse(w io.Writer, st Status, headers []Header, body io.Reader) error {
    ew := &errWriter{Writer: w}
    fmt.Fprintf(ew, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)

    for _, h := range headers {
        fmt.Fprintf(ew, "%s: %s\r\n", h.Key, h.Value)
    }

    fmt.Fprint(ew, "\r\n")
    io.Copy(ew, body)
    return ew.err
}

```

Appliquer `errWriter` à `WriteResponse` améliore considérablement la clarté du code. Chacune des opérations n'a plus besoin de se mettre entre crochets avec un contrôle d'erreur. Signaler l'erreur est déplacé à la fin de la fonction en inspectant le champ `ew.err`, évitant la traduction ennuyeuse des valeurs de retour de `io.Copy`.

7.2. Ne traiter une erreur qu'une seule fois

Enfin, je tiens à mentionner que vous ne devez traiter les erreurs qu'une seule fois. Gérer une erreur signifie inspecter la valeur de l'erreur et prendre une seule décision.

```

// WriteAll writes the contents of buf to the supplied writer.
func WriteAll(w io.Writer, buf []byte) {
    w.Write(buf)
}

```

Si vous prenez moins d'une décision, vous ignorez l'erreur. Comme nous le voyons ici, l'erreur de `w.WriteAll` est éliminée.

Mais prendre plus d'une décision en réponse à une seule erreur est également problématique. Ce qui suit est un code que je rencontre fréquemment.


```
func WriteAll(w io.Writer, buf []byte) error {
    _, err := w.Write(buf)
    if err != nil {
        log.Println("unable to write:", err) // annotated error goes to log file
        return err                          // unannotated error returned to caller
    }
    return nil
}
```

Dans cet exemple, si une erreur se produit pendant `w.Write`, une ligne sera écrite dans un fichier journal, notant le fichier et la ligne où l'erreur s'est produite, et l'erreur est également renvoyée à l'appelant, qui éventuellement va le journaliser, et renvoyer jusqu'en haut du programme.

L'appelant fait probablement la même chose

```
func WriteConfig(w io.Writer, conf *Config) error {
    buf, err := json.Marshal(conf)
    if err != nil {
        log.Printf("could not marshal config: %v", err)
        return err
    }
    if err := WriteAll(w, buf); err != nil {
        log.Println("could not write config: %v", err)
        return err
    }
    return nil
}
```

Vous obtenez ainsi une pile de lignes en double dans vos logs,

```
unable to write: io.EOF
could not write config: io.EOF
```

mais en haut du programme, vous obtenez l'erreur originale sans aucun contexte.

```
err := WriteConfig(f, &conf)
fmt.Println(err) // io.EOF
```

J'aimerais aller un peu plus loin parce que je ne considère pas que les problèmes liés à l'enregistrement et au retour ne sont qu'une question de préférence personnelle.

```
func WriteConfig(w io.Writer, conf *Config) error {
    buf, err := json.Marshal(conf)
    if err != nil {
        log.Printf("could not marshal config: %v", err)
        // oops, forgot to return
    }
    if err := WriteAll(w, buf); err != nil {
        log.Println("could not write config: %v", err)
        return err
    }
    return nil
}
```

Le problème que je vois beaucoup est que les programmeurs oublient de revenir d'une erreur. Comme nous l'avons déjà dit, le style Go consiste à utiliser des clauses de garde, à vérifier les conditions préalables au fur et à mesure que la fonction progresse et à revenir tôt.

Dans cet exemple, l'auteur a vérifié l'erreur, l'a enregistrée, mais a oublié de revenir. Cela a causé un bug subtil.

Le contrat de traitement des erreurs dans Go stipule que vous ne pouvez pas faire d'hypothèses sur le contenu d'autres valeurs de retour en présence d'une erreur. Comme la formation JSON a échoué, le contenu de `buf` est inconnu, peut-être qu'il ne contient rien, mais pire il pourrait contenir un fragment JSON à moitié écrit.

Parce que le programmeur a oublié de revenir après avoir vérifié et journalisé l'erreur, le tampon corrompu sera passé à `WriteAll`, qui réussira probablement et donc le fichier de configuration sera écrit incorrectement. Cependant, la fonction retournera très bien, et la seule indication qu'un problème s'est produit sera une simple ligne de log se plaignant de la formation de JSON, et non un échec à écrire la configuration.

7.2.1. Ajout de contexte aux erreurs

Le bogue s'est produit parce que l'auteur essayait d'ajouter un contexte au message d'erreur. Ils essayaient de se laisser une miette de pain pour remonter à la source de l'erreur.

Regardons une autre façon de faire la même chose avec `fmt.Errorf`.

```
func WriteConfig(w io.Writer, conf *Config) error {
    buf, err := json.Marshal(conf)
    if err != nil {
        return fmt.Errorf("could not marshal config: %v", err)
    }
    if err := WriteAll(w, buf); err != nil {
        return fmt.Errorf("could not write config: %v", err)
    }
    return nil
}

func WriteAll(w io.Writer, buf []byte) error {
    _, err := w.Write(buf)
    if err != nil {
        return fmt.Errorf("write failed: %v", err)
    }
    return nil
}
```

En combinant l'annotation de l'erreur avec le retour sur une ligne, il est plus difficile d'oublier de retourner une erreur et d'éviter de continuer accidentellement.

Si une erreur d'E/S se produit lors de l'écriture du fichier, la méthode `Error()` de l'erreur signalera quelque chose comme ceci ;

`could not write config: write failed: input/output error`

7.2.2. Erreurs de wrapper avec github.com/pkg/errors

Le modèle `fmt.Errorf` fonctionne bien pour annoter le message d'erreur, mais il le fait au prix d'obscurcir le type de l'erreur originale. J'ai fait valoir que le traitement des erreurs comme des valeurs opaques est important pour produire un logiciel qui est lâchement couplé, de sorte que le visage que le type de l'erreur originale ne devrait pas importer si la seule chose que vous faites avec une valeur d'erreur est

- Vérifiez qu'il n'est pas nul.
- Imprimez-le ou enregistrez-le.

Cependant, il y a certains cas, je crois qu'ils sont peu fréquents, où vous avez besoin de récupérer l'erreur originale. Dans ce cas, vous pouvez utiliser quelque chose comme mon paquet d'erreurs pour annoter des erreurs comme ceci

```

func ReadFile(path string) ([]byte, error) {
    f, err := os.Open(path)
    if err != nil {
        return nil, errors.Wrap(err, "open failed")
    }
    defer f.Close()

    buf, err := ioutil.ReadAll(f)
    if err != nil {
        return nil, errors.Wrap(err, "read failed")
    }
    return buf, nil
}

func ReadConfig() ([]byte, error) {
    home := os.Getenv("HOME")
    config, err := ReadFile(filepath.Join(home, ".settings.xml"))
    return config, errors.WithMessage(err, "could not read config")
}

func main() {
    _, err := ReadConfig()
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}

```

Maintenant l'erreur rapportée sera la belle erreur de K&D

could not read config: open failed: open /Users/dfc/.settings.xml: no such file or directory

et la valeur de l'erreur conserve une référence à la cause initiale.

```

func main() {
    _, err := ReadConfig()
    if err != nil {
        fmt.Printf("original error: %T %v\n", errors.Cause(err), errors.Cause(err))
        fmt.Printf("stack trace:\n%+v\n", err)
        os.Exit(1)
    }
}

```

Ainsi, vous pouvez récupérer l'erreur d'origine et imprimer une trace de pile ;

```
original error: *os.PathError open /Users/dfc/.settings.xml: no such file or directory
stack trace:
open /Users/dfc/.settings.xml: no such file or directory
open failed
main.ReadFile
    /Users/dfc/devel/practical-go/src/errors/readfile2.go:16
main.ReadConfig
    /Users/dfc/devel/practical-go/src/errors/readfile2.go:29
main.main
    /Users/dfc/devel/practical-go/src/errors/readfile2.go:35
runtime.main
    /Users/dfc/go/src/runtime/proc.go:201
runtime.goexit
    /Users/dfc/go/src/runtime/asm_amd64.s:1333
could not read config
```

L'utilisation du paquet d'erreurs vous donne la possibilité d'ajouter un contexte aux valeurs d'erreur, d'une manière qui est inspectable à la fois par un humain et une machine. Si vous êtes venu à ma présentation hier, vous saurez que le wrapper est en train d'être intégré à la bibliothèque standard dans une prochaine version de Go.

8. Concurrency

Souvent, Go est choisi pour un projet en raison de ses caractéristiques concurrentes. L'équipe de Go s'est donné beaucoup de mal pour rendre la concomitance dans Go bon marché (en termes de ressources matérielles) et performante, cependant il est possible d'utiliser les fonctionnalités de concomitance de Go pour écrire du code qui n'est ni performant ni fiable. Avec le temps qu'il me reste, j'aimerais vous donner quelques conseils pour éviter certains des pièges qui viennent avec les fonctions de simultanéité de Go.

Go offre une prise en charge de première classe pour la simultanéité avec les canaux et les instructions `select` et `go`. Si vous avez appris Go à partir d'un livre ou d'un cours de formation, vous avez peut-être remarqué que la section sur la concurrence est toujours l'une des dernières que vous allez couvrir. Cet atelier n'est pas différent, j'ai choisi de couvrir la concurrence en dernier, comme s'il s'agissait en quelque sorte d'un complément aux compétences habituelles qu'un programmeur Go doit maîtriser.

Il y a là une dichotomie ; la caractéristique phare de Go est notre modèle de concurrence simple et léger. En tant que produit, notre langage se vend presque toute seule sur cette seule fonctionnalité. D'un autre côté, il y a un récit selon lequel la concurrence n'est pas si facile à utiliser, sinon les auteurs n'en feraient pas le dernier chapitre de leur livre et nous ne regarderions pas en arrière avec regret nos efforts de formation.

Cette section discute de quelques pièges de l'utilisation naïve des fonctions de simultanéité de Go.

8.1. Tenez-vous occupé ou faites le travail vous-même

Quel est le problème avec ce programme ?

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    for {
    }
}
```

Le programme fait ce que nous avons prévu, il sert un simple serveur web. Mais il fait aussi quelque chose d'autre en même temps, il gaspille le CPU dans une boucle infinie. C'est parce que le `for{}` sur la dernière ligne de `main` va bloquer la goroutine principal parce qu'il ne fait aucune E/S, n'attend pas sur un verrou, n'envoie ou ne reçoit pas sur un canal, ou ne communique pas avec le *scheduler*.

Comme le runtime Go est généralement planifié en coopération, ce programme va tourner en vain sur un seul processeur, et peut finir par être bloqué en temps réel.

Comment pourrions-nous régler ce problème ? Voici une suggestion.

```

import (
    "fmt"
    "log"
    "net/http"
    "runtime"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    for {
        runtime.Gosched()
    }
}

```

Cela peut paraître stupide, mais c'est une solution commune que je vois couramment dans la nature. C'est symptomatique de ne pas comprendre le problème sous-jacent.

Maintenant, si vous êtes un peu plus expérimenté en go, vous pouvez écrire quelque chose comme ceci.

```

package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    select {}
}

```

Une instruction `select` vide sera bloquée indéfiniment. C'est une propriété utile parce que maintenant nous ne faisons pas tourner un CPU entier juste pour appeler `runtime.Gosched()`. Cependant, nous ne traitons que le symptôme, pas la cause.

Je veux vous présenter une autre solution qui, je l'espère, vous est déjà venue à l'esprit. Plutôt que d'exécuter `http.ListenAndServe` dans un goroutine, nous laissons avec le problème de ce qu'il faut faire avec le goroutine principal, exécutez simplement `http.ListenAndServe` sur le goroutine principal lui-même.



Si la fonction principale `main` d'un programme Go revient, alors le programme Go quittera inconditionnellement, peu importe ce que font les autres goroutines lancées par le programme au fil du temps.

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

Voici donc mon premier conseil : si votre goroutine ne peut pas progresser tant qu'elle n'a pas obtenu le résultat d'une autre, il est souvent plus simple de faire le travail vous-même plutôt que de le déléguer. Ceci élimine souvent beaucoup de suivi d'état et de manipulation de canal nécessaire pour sonder un résultat depuis un goroutine jusqu'à son initiateur.



Beaucoup de programmeurs de Go surutilisent les goroutines, surtout lorsqu'ils démarrent. Comme pour toute chose dans la vie, la modération est la clé du succès.

8.2. Laisser la concomitance à l'appelant

Quelle est la différence entre ces deux API ?


```
// ListDirectory returns the contents of dir.
func ListDirectory(dir string) ([]string, error)

// ListDirectory returns a channel over which
// directory entries will be published. When the list
// of entries is exhausted, the channel will be closed.
func ListDirectory(dir string) chan string
```

Tout d'abord, les différences évidentes ; le premier exemple lit un répertoire dans une *slice* puis retourne la *slice* entière, ou une erreur si quelque chose a mal tourné. Cela se produit de manière synchrone, l'appelant de `ListDirectory` bloque jusqu'à ce que toutes les entrées du répertoire aient été lues. En fonction de la taille du répertoire, cela peut prendre beaucoup de temps, et pourrait potentiellement allouer beaucoup de mémoire à la construction de la diapositive des noms des entrées du répertoire.

Examinons le deuxième exemple. C'est un peu plus Go comme, `ListDirectory` renvoie un canal sur lequel les entrées de répertoire seront passées. Quand le canal est fermé, c'est votre indication qu'il n'y a plus d'entrées de répertoire. Comme la population du canal se produit après le retour de `ListDirectory`, `ListDirectory` est probablement en train de démarrer un goroutine pour peupler le canal.



Il n'est pas nécessaire pour la seconde version d'utiliser une routine Go ; elle peut allouer un canal suffisant pour contenir toutes les entrées du répertoire sans bloquer, remplir le canal, le fermer, puis le renvoyer à l'appelant. Mais c'est peu probable, car cela poserait les mêmes problèmes avec la consommation d'une grande quantité de mémoire pour mettre en mémoire tampon tous les résultats dans un canal.

La version canal de `ListDirectory` a deux autres problèmes :

- En utilisant un canal fermé comme signal qu'il n'y a plus d'éléments à traiter, `ListDirectory` ne peut pas dire à l'appelant que l'ensemble des éléments retournés sur le canal est incomplet car une erreur est survenue en cours de route. Il n'y a aucun moyen pour l'appelant de faire la différence entre un répertoire vide et une erreur à lire entièrement dans le répertoire. Dans les deux cas, un canal est renvoyé depuis `ListDirectory`, qui semble être fermé immédiatement.
- L'appelant doit continuer à lire à partir du canal jusqu'à ce qu'il soit fermé parce que c'est le seul moyen pour lui de savoir que le goroutine qui a commencé à remplir le canal s'est arrêté. C'est une limitation sérieuse de l'utilisation de `ListDirectory`, l'appelant doit passer du temps à lire sur le canal même s'il a reçu la réponse qu'il voulait. Il est probablement plus efficace en termes d'utilisation de mémoire pour les annuaires de taille moyenne à grande, mais cette méthode n'est pas plus rapide que la méthode originale basée sur les slices.

La solution aux problèmes des deux implémentations est d'utiliser un `callback`, une fonction qui est appelée dans le contexte de chaque entrée de répertoire lors de son exécution.

```
func ListDirectory(dir string, fn func(string))
```

Il n'est pas surprenant que c'est ainsi que fonctionne la fonction `filepath.WalkDir`.



Si votre fonction lance un goroutine, vous devez fournir à l'appelant un moyen d'arrêter explicitement ce goroutine. Il est souvent plus facile de laisser la décision d'exécuter une fonction de manière asynchrone à l'appelant de cette fonction.

8.3. Ne jamais démarrer une goroutine sans savoir quand elle s'arrêtera.

L'exemple précédent montrait l'utilisation d'une goroutine quand elle n'était pas vraiment nécessaire. Mais l'une des raisons principales de l'utilisation de Go est la première classe de fonctionnalités de concomitance des offres du langage Go. En effet, il existe de nombreux cas où vous souhaitez exploiter le parallélisme disponible dans votre matériel. Pour ce faire, vous devez utiliser des goroutines.

Cette application simple sert le trafic http sur deux ports différents, le port `8080` pour le trafic applicatif et le port `8001` pour l'accès au terminal `/debug/pprof`.

```
package main

import (
    "fmt"
    "net/http"
    _ "net/http/pprof"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    go http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux) // debug
    http.ListenAndServe("0.0.0.0:8080", mux)                       // app traffic
}
```

Bien que ce programme ne soit pas très compliqué, il représente la base d'une application réelle.

Il y a quelques problèmes avec l'application telle qu'elle est qui se révéleront au fur et à mesure que l'application grandit, alors adressons-en quelques-unes maintenant.

```
func serveApp() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    http.ListenAndServe("0.0.0.0:8080", mux)
}

func serveDebug() {
    http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux)
}

func main() {
    go serveDebug()
    serveApp()
}
```

En divisant les gestionnaires `serveApp` et `serveDebug` en leurs propres fonctions, nous les avons découplés de `main.main`. Nous avons également suivi les conseils d'en haut et nous nous assurons que `serveApp` et `serveDebug` laissent leur concours à l'appelant.

Mais il y a quelques problèmes d'opérabilité avec ce programme. Si `serveApp` revient, `main.main` reviendra, provoquant l'arrêt du programme et son redémarrage par le gestionnaire de processus que vous utilisez.



Tout comme les fonctions dans Go laissent la concomitance à l'appelant, les applications devraient quitter le travail de surveillance de leur statut et de redémarrage s'ils échouent au programme qui les a invoquées. Ne rendez pas vos applications responsables de leur redémarrage, c'est une procédure qu'il vaut mieux gérer de l'extérieur de l'application.

Cependant, `serveDebug` est exécuté dans un goroutine séparé et s'il retourne juste ce goroutine sortira pendant que le reste du programme continue. Votre personnel d'exploitation ne sera pas heureux de constater qu'il ne peut pas obtenir les statistiques de votre application quand il le souhaite parce que le gestionnaire de bogues `/debug` a cessé de fonctionner depuis longtemps.

Ce que nous voulons nous assurer, c'est que si l'un des goroutines responsables de servir cette application s'arrête, nous fermons l'application.

```

func serveApp() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    if err := http.ListenAndServe("0.0.0.0:8080", mux); err != nil {
        log.Fatal(err)
    }
}

func serveDebug() {
    if err := http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux); err != nil {
        log.Fatal(err)
    }
}

func main() {
    go serveDebug()
    go serveApp()
    select {}
}

```

Maintenant `serveApp` et `serveDebug` vérifient l'erreur renvoyée par `ListenAndServe` et appellent `log.Fatal` si nécessaire. Parce que les deux maîtres-chiens courent dans des goroutines, nous stationnons la goroutine principale dans une `select{}`.

Cette approche pose un certain nombre de problèmes :

- Si `ListenAndServe` revient avec une erreur nulle, `log.Fatal` ne sera pas appelé et le service HTTP sur ce port s'arrêtera sans arrêter l'application.
- `log`.appels fatals `os.Exit` qui quittera inconditionnellement le programme ; les reports ne seront pas appelés, les autres goroutines ne seront pas notifiées pour s'arrêter, le programme s'arrêtera. Il est donc difficile d'écrire des tests pour ces fonctions.



N'utilisez que `log.Fatal` des fonctions `main.main` ou `init`.

Ce que nous aimerions vraiment, c'est transmettre toute erreur qui se produit à l'initiateur du goroutine pour qu'il sache pourquoi le goroutine s'est arrêté, qu'il puisse arrêter le processus proprement.

```

func serveApp() error {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    return http.ListenAndServe("0.0.0.0:8080", mux)
}

func serveDebug() error {
    return http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux)
}

func main() {
    done := make(chan error, 2)
    go func() {
        done <- serveDebug()
    }()
    go func() {
        done <- serveApp()
    }()

    for i := 0; i < cap(done); i++ {
        if err := <-done; err != nil {
            fmt.Println("error: %v", err)
        }
    }
}

```

Nous pouvons utiliser un canal pour collecter l'état de retour du goroutine. La taille du canal est égale au nombre de goroutines que nous voulons gérer pour que l'envoi sur le canal fait ne bloque pas, car cela bloque l'arrêt du goroutine, ce qui provoque sa fuite.

Comme il n'y a aucun moyen de fermer en toute sécurité le canal `done`, nous ne pouvons pas utiliser l'idiome `for range` pour boucler le canal jusqu'à ce que tous les goroutines aient fait leur rapport, au lieu de cela nous bouclons pour autant de goroutines que nous avons commencé, qui est égale à la capacité du canal.

Maintenant nous avons un moyen d'attendre que chaque goroutine sorte proprement et enregistre toute erreur qu'elle rencontre. Tout ce qu'il faut, c'est un moyen d'acheminer le signal d'arrêt du premier goroutine qui sort vers les autres.

Il s'avère que demander à un `http.Server` de s'éteindre est un peu compliqué, donc j'ai transformé cette logique en fonction d'aide. L'assistant de service prend une adresse et `http.Handler`, similaire à `http.ListenAndServe`, et aussi un canal d'arrêt que nous utilisons pour déclencher la méthode `Shutdown`.

```

func serve(addr string, handler http.Handler, stop <-chan struct{}) error {
    s := http.Server{
        Addr:    addr,
        Handler: handler,
    }

    go func() {
        <-stop // wait for stop signal
        s.Shutdown(context.Background())
    }()

    return s.ListenAndServe()
}

func serveApp(stop <-chan struct{}) error {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    return serve("0.0.0.0:8080", mux, stop)
}

func serveDebug(stop <-chan struct{}) error {
    return serve("127.0.0.1:8001", http.DefaultServeMux, stop)
}

func main() {
    done := make(chan error, 2)
    stop := make(chan struct{})
    go func() {
        done <- serveDebug(stop)
    }()
    go func() {
        done <- serveApp(stop)
    }()

    var stopped bool
    for i := 0; i < cap(done); i++ {
        if err := <-done; err != nil {
            fmt.Println("error: %v", err)
        }
        if !stopped {
            stopped = true
            close(stop)
        }
    }
}

```

Maintenant, chaque fois que nous recevons une valeur sur le canal terminé, nous fermons le canal d'arrêt qui fait que tous les goroutines qui attendent sur ce canal arrêtent leur serveur http. Ceci

entraînera à son tour le retour de tous les goroutines `ListenAndServe` restantes. Une fois que toutes les goroutines que nous avons commencé se sont arrêtées, `main.main` retourne et le processus s'arrête proprement.

[1] <https://www.lysator.liu.se/c/pikestyle.html>

[2] <https://talks.golang.org/2014/names.slide#4>

[3] <https://talks.golang.org/2014/names.slide#4>

[4] <https://speakerdeck.com/campoy/understanding-nil>

[5] <https://www.youtube.com/watch?v=Ic2y6w8lMPA>

[6] <https://www.youtube.com/watch?v=Ic2y6w8lMPA>

[7] <https://medium.com/@matryer/line-of-sight-in-code-186dd7cdea88>

[8] <https://golang.org/doc/go1.4#internalpackages>

[9] <https://www.infoq.com/articles/API-Design-Joshua-Bloch>

[10] <https://www.infoq.com/articles/API-Design-Joshua-Bloch>

[11] <https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis>

[12] <https://commandcenter.blogspot.com/2014/01/self-referential-functions-and-design.html>