Engineer

Go

langage compilé à typage fort avec ramasse-miettes

Ce que c’est

Go est un langage de programmation généraliste né chez Google, conçu pour la programmation système, avec des mécanismes de programmation concurrente intégrés au langage plutôt qu’ajoutés par une bibliothèque. On l’appelle souvent golang, d’après la première adresse web du projet, golang.org ; sa FAQ précise fermement que le langage s’appelle simplement Go, écrit Go et jamais GO.

Robert Griesemer, Rob Pike et Ken Thompson ont esquissé ses objectifs sur un tableau blanc le 21 septembre 2007. Le 10 novembre 2009, le langage a été rendu public comme projet open source. Go 1 a suivi le 28 mars 2012, avec la promesse que les programmes écrits pour cette version continueraient à se compiler et à fonctionner sans changement avec toutes les versions ultérieures de Go 1. La promesse tient toujours : la spécification décrit désormais la version go1.27 du langage, et Go 1.27.0 est sorti le 19 août 2026. Les génériques sont arrivés avec Go 1.18 ; les concepteurs les ont qualifiés de plus grand changement depuis la première publication open source.

Selon la FAQ, le langage est né d’un choix que ses auteurs ne voulaient plus faire : entre compilation efficace, exécution efficace et facilité de programmation. Go vise les trois. Un programme se compose de paquets, et une seule commande, `go`, donne accès aux outils de compilation, de test, de formatage et de récupération des dépendances. Le mot-clé `go` devant un appel de fonction démarre une goroutine, une fonction exécutée aux côtés du reste du programme. Leur coût est si faible que, selon la FAQ, des centaines de milliers peuvent partager un espace d’adressage. Des canaux transmettent les valeurs entre elles, et Effective Go résume le style voulu dans une devise : « Ne communiquez pas en partageant la mémoire ; partagez plutôt la mémoire en communiquant. »

Deux habitudes distinguent le code Go partout où il apparaît. Le programme `gofmt` présente chaque fichier source selon une disposition standard, ce qui écarte les débats de formatage lors des revues. Le compilateur refuse aussi un programme contenant une variable ou un import inutilisé : la première peut cacher un bug, le second ralentit toutes les compilations suivantes.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que Go ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. La partie invisible pour les utilisateurs : où résident les données, quelles règles s'appliquent et comment les autres systèmes peuvent y accéder.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Pourquoi c’est important

On envisage généralement Go lorsqu’il faut choisir dans quel langage écrire un service réseau, un outil en ligne de commande ou une partie de l’infrastructure. Les alternatives se situent de part et d’autre : un langage de script d’un côté, C, C++ ou Rust de l’autre.

Face à un langage de script, Go offre un déploiement simple et une détection précoce des erreurs. L’éditeur de liens produit par défaut des binaires à liaison statique : le résultat est un fichier qui embarque son environnement d’exécution et fonctionne sur un serveur sans rien d’autre à installer. La FAQ en chiffre le coût à environ deux mégaoctets pour un programme affichant « hello, world », contre environ 750 kB pour le même programme en C. Les erreurs de type apparaissent à la compilation plutôt que lors de l’exécution en production d’une branche rarement utilisée. Un des objectifs initiaux était que même un gros exécutable puisse être compilé en quelques secondes sur un seul ordinateur.

Face à C, C++ ou Rust, Go offre simplicité et sûreté mémoire au prix d’un moindre contrôle. Un ramasse-miettes récupère la mémoire, ce qui épargne au code ordinaire toute une catégorie d’erreurs d’utilisation après libération. En contrepartie, le programme ne décide pas exactement quand la mémoire est libérée, et chaque binaire embarque l’environnement qui s’en charge. Pour la plupart des serveurs, c’est le bon compromis. Pour un noyau, un petit appareil embarqué ou une boucle critique ne tolérant pas de ramasse-miettes, c’est le mauvais.

L'argument moins visible est le temps. En 2023, Russ Cox a écrit qu'un Go 2 cassant les anciens programmes « n'arrivera jamais » et que le projet allait plutôt « renforcer encore son engagement en faveur de la compatibilité ». La promesse se paie en mises à niveau plutôt qu'en réécritures. Une nouvelle version majeure paraît tous les six mois, en février et en août, et reste prise en charge jusqu'à la publication de deux versions plus récentes. Une équipe qui veut les correctifs de sécurité passe donc à une version plus récente environ une fois par an.

La simplicité du langage demande aussi d'écrire certaines choses plus longuement. Les erreurs sont des valeurs ordinaires, renvoyées et vérifiées là où elles surviennent : c'est à peu près aussi explicite que répétitif.

À quoi cela ressemble en pratique

Une goroutine est facile à lancer et tout aussi facile à perdre. La page sur le transfert du travail lent vers une file de tâches River le dit clairement : le travail lent lancé dans une goroutine disparaît dès que le processus redémarre, en plein déploiement, sans trace de ce qui aurait dû se faire. L'entreprise l'a transféré à River, qui conserve sa file dans PostgreSQL : une tâche et les lignes qu'elle touche sont ainsi validées ou annulées ensemble. Derrière se trouvent 15 modules d'exécution des tâches et 8 tâches planifiées, avec 20 tâches pg_cron en dessous. Une goroutine permet l'exécution concurrente ; la durabilité doit venir d'ailleurs.

Les tests sont livrés avec les outils, et la suite de tests en couches en montre le résultat. Son API Go contient 981 fonctions de test dans 310 fichiers : une des quatre couches testées chacune là où réside son comportement, à côté des spécifications de test du frontend et des propres tests SQL de la base.

Le programme continu de sécurité s'appuie aussi sur les outils du langage. Les dépendances Go sont vérifiées avec govulncheck : une vulnérabilité reconnue par les contrôles configurés peut faire échouer la compilation. La page cite le volume de signalements comme le coût réel des analyses. govulncheck vise précisément ce coût : il ne signale une vulnérabilité que lorsque les fonctions du code appellent la fonction affectée, directement ou par l'intermédiaire d'autres fonctions.

Où les problèmes surviennent

Une fonction signale une erreur alors que rien n'a échoué. Une valeur d'interface Go contient un type et une valeur. Elle n'est égale à nil que si aucun des deux n'est défini. Une fonction renvoyant un pointeur nil d'un type d'erreur concret par l'interface `error` donne à son appelant une interface contenant un type, qui n'est donc pas nil :

```
func save() error {
    var err *ValidationError // a nil pointer
    if invalid() {
        err = &ValidationError{}
    }
    return err // never equal to nil: the interface holds a type
}
```

Renvoyez un `nil` littéral en cas de réussite et déclarez chaque résultat d'erreur comme `error` plutôt que comme un type concret.

Un service ayant réussi tous les tests plante avec le trafic réel. Les opérations sur les maps sont volontairement non synchronisées : verrouiller chaque accès ralentirait la plupart des programmes. La FAQ avertit qu'« un accès non contrôlé à une map peut faire planter le programme ». La cause est une goroutine écrivant dans une map tandis qu'une autre la lit ou y écrit ; des lectures seules sont sûres. Protégez la map avec un mutex, confiez-la à une seule goroutine que les autres joignent par un canal, ou utilisez `sync.Map` dans les cas pour lesquels il est conçu. Exécutez ensuite les tests avec `go test -race` . Pour un programme typique, le détecteur peut multiplier la mémoire utilisée par cinq à dix et le temps d'exécution par deux à vingt. Il ne trouve que les conditions de concurrence sur les chemins réellement exécutés : un binaire compilé avec `-race` et soumis à une charge réaliste en trouve donc davantage.

Les goroutines lancées dans une boucle travaillent toutes sur le dernier élément. Avant Go 1.22, une boucle `for` partageait une variable entre toutes les itérations. Une fermeture qui la capturait voyait donc sa valeur au moment où la goroutine finissait par s'exécuter. Go 1.22 donne une variable à chaque itération, mais seulement dans les modules dont go.mod déclare `go 1.22` ou plus récent. Le vieux code garde son sens même lorsqu'il est compilé par des outils plus récents. Relevez la version de la ligne `go` , puis exécutez `go vet` et les tests. Un test qui commence à échouer après ce changement vérifiait souvent la mauvaise valeur depuis le départ.

Une compilation sur une nouvelle machine s'arrête pour télécharger un autre Go. Depuis Go 1.21, la ligne `go` indique la version minimale requise. Lorsque la chaîne d'outils installée est plus ancienne que celle demandée par le module, la commande `go` télécharge cette version plus récente sous forme de module et l'exécute depuis le cache des modules. C'est pratique sur un portable, mais surprenant dans une compilation isolée du réseau ou une image de conteneur figée. Installez la chaîne d'outils indiquée par go.mod et définissez la variable d'environnement `GOTOOLCHAIN` lorsqu'une compilation doit utiliser une version exacte.

Pour aller plus loin

- Spécification du langage Go — Le manuel de référence de toutes les constructions du langage, versionné avec chaque publication.
- Questions fréquentes — Les raisons de la création du langage, ses choix de conception et les problèmes d'erreur nil et de maps décrits plus haut.
- Go 1 et l'avenir des programmes Go — La promesse de compatibilité et les exceptions qu'elle autorise.
- Historique des versions — Toutes les versions depuis Go 1, avec la politique de prise en charge en tête.
- Détecteur de conditions de concurrence — Son exécution, ce qu'il détecte et son coût.
- Corriger les boucles for dans Go 1.22 — Le changement des variables de boucle, son lien avec go.mod et la façon de tester l'ancien code avec ce nouveau comportement.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que Go ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/go/