2

Démarrage du runtime et initialisation

Le démarrage du runtime de Go, appelé *bootstrap*, constitue une séquence complexe d'étapes qui transforment un exécutable binaire en un programme Go fonctionnel[1]. Ce processus comprend des phases distinctes : l'initialisation du runtime lui-même, puis l'initialisation des packages utilisateur.

L'ensemble du bootstrap s'exécute dans un seul thread système, de manière strictement séquentielle. Cette approche garantit un état cohérent pendant l'initialisation : le scheduler n'est pas encore opérationnel, les structures de concurrence ne sont pas configurées et les goroutines ne peuvent pas encore être créées. Cette exécution monothread élimine tout risque de race condition pendant la phase délicate de mise en place des fondations du runtime.

2.1. Signification et convention de nommage *rt0*

Le préfixe `rt0` dans les fonctions de bootstrap Go fait référence à `Runtime 0`, où le `0` signifie *le tout début* [the very beginning]. Cette convention provient du monde Unix et du langage C, où `crt0` (C Runtime 0) désigne l'ensemble des routines de démarrage qui s'exécutent avant la fonction `main()`.

Go adopte une convention de nommage structurée pour ses fonctions de bootstrap : `runtime/rt0_<GOARCH>_<GOOS>.s`[2], où GOARCH représente l'architecture cible (amd64, arm64, 38, etc.) et GOOS le système d'exploitation (linux, windows, darwin, etc.). Cette approche permet d'avoir des points d'entrée spécialisés pour chaque combinaison de plateformes.

Go organise son code de bootstrap dans des fichiers assembleur spécifiques à chaque couple OS/architecture :

[1] Les détails techniques du bootstrap présentés dans ce chapitre s'appuient sur la documentation officielle du runtime. La compréhension de cette séquence est fondamentale pour maîtriser les mécanismes internes de Go.
[2] Tous les fichiers de code source mentionnés dans ce chapitre sont relatifs à https://go.dev/src/, sauf mention contraire.

```
runtime/rt0_darwin_amd64.s
runtime/rt0_darwin_arm64.s
runtime/rt0_linux_amd64.s
runtime/rt0_linux_arm64.s
runtime/rt0_windows_amd64.s
runtime/rt0_windows_arm64.s
[...]
```

Le compilateur Go sélectionne automatiquement le fichier approprié selon les valeurs des variables d'environnement GOOS et GOARCH lors de la compilation. Cette architecture modulaire permet d'adapter finement le bootstrap aux spécificités de chaque plateforme (conventions d'appel, formats exécutables, API système) tout en partageant le maximum de code commun dans les fichiers `runtime/asm_*.s`.

Chaque fichier `runtime/rt0_*.s` est généralement très compact, ne contenant souvent qu'un simple saut vers la fonction commune `_rt0_<GOARCH>`. Cette indirection permet une organisation claire : le code spécifique à la plateforme reste isolé, tandis que la logique de bootstrap principale est mutualisée.

Bien que minimal, ce code spécifique à chaque OS/CPU a pour mission de résoudre les différences fondamentales entre plateformes et de converger vers un état unifié. Pour l'architecture (CPU), les différences sont évidentes : le code assembleur AMD64 diffère de ARM64. Pour l'OS, chaque système a ses propres spécificités que les fichiers `rt0` normalisent :

Conventions de passage d'arguments

La majorité des systèmes Unix placent `argc` et `argv` sur la stack, mais FreeBSD et DragonFly les passent dans des registres spécifiques (DI). Android nécessite une synthèse complète d'arguments factices (`gojni`) pour les bibliothèques partagées.

Initialisations système spécifiques

Plan 9 requiert l'initialisation manuelle du TLS [3] (`_tos`, `_privates`), tandis qu'AIX doit interagir avec le loader système avant de pouvoir démarrer le runtime Go. Windows délègue le parsing des arguments au runtime, qui s'en charge ultérieurement.

[3] TLS (Thread Local Storage) est un mécanisme permettant à chaque thread d'avoir sa propre copie de variables globales, isolée des autres threads. Ce concept est traité en détail à la Section 2.4, Gestion CGO et configuration TLS.

Formats exécutables et conventions d'appel

Chaque OS a ses propres conventions pour les appels système, les formats exécutables (ELF vs PE vs Mach-O) et les mécanismes de démarrage de processus.

Malgré cette diversité, tous les fichiers `rt0` convergent vers un objectif unique : normaliser l'environnement pour appeler `runtime·rt0_go(SB)`[4] avec `argc` et `argv` dans un format standard, permettant à Go de présenter une interface uniforme (`os.Args`) au développeur.

Astuce *› Vous pouvez voir quelles combinaisons GOOS/GOARCH sont supportées en exécutant* `go tool dist list`*. Chaque combinaison listée correspond potentiellement à un fichier* `rt0` *spécifique.*

2.2. Configuration de la stack et structures fondamentales

La fonction `runtime·rt0_go` constitue le cœur du processus de bootstrap. Elle commence par initialiser les structures de données fondamentales du runtime.

L'étape suivante consiste à initialiser la stack dynamique de la première goroutine g0. Contrairement aux futures goroutines qui auront des stacks allouées dans le heap Go, g0 constitue un cas particulier : elle utilise la stack système fournie par le système d'exploitation (typiquement plusieurs mégaoctets), que le runtime adapte pour la faire ressembler à une stack de goroutine normale. Le runtime configure les mécanismes de détection de débordement en définissant les stack guards[5] à environ 64 ko en dessous du sommet de la stack système, créant ainsi une zone de protection. Cette approche permet une transition cohérente entre le monde système et le modèle de stacks redimensionnables de Go.

Après la configuration de la stack, le runtime initialise la structure `g` représentant la goroutine racine g0. Cette structure, définie dans `runtime/runtime2.go`, contient les métadonnées essentielles de chaque goroutine : pointeurs de stack, état d'exécution, et liens vers les autres structures du runtime. La goroutine g0 a un statut particulier car elle sert de base pour l'exécution du bootstrap et du scheduler.

[4] Le caractère · (U+00B7, middle dot) est une convention héritée de Plan 9 utilisée dans l'assembleur Go pour séparer le nom du package du nom de la fonction. Cette notation distingue les symboles Go (`runtime·rt0_go`) des symboles C classiques et évite les collisions de noms entre packages.

[5] Les stack guards sont des adresses de contrôle placées dans la stack pour détecter les débordements. Quand le pointeur de stack (SP) descend en dessous de ces adresses, le runtime déclenche automatiquement la croissance de stack ou signale une erreur de débordement (détaillé Section 4.2, Architecture et gestion de la stack).

Il est important de noter que g0 n'est pas une goroutine au sens habituel du terme. Bien qu'elle utilise la structure de données `g` par commodité d'implémentation, g0 constitue en réalité le contexte d'exécution système du runtime. Contrairement aux goroutines utilisateur, g0 ne peut pas être préemptée, ne possède pas de stack dynamique redimensionnable et n'est jamais soumise au scheduler normal. Son rôle consiste à exécuter le code du runtime lui-même : orchestration du scheduler, exécution du garbage collector et transitions entre goroutines. De manière importante, g0 sert également de contexte d'exécution pour toutes les opérations qui nécessitent de quitter temporairement l'espace utilisateur, notamment les appels système bloquants et les interactions avec le kernel.

Dans le contexte du bootstrap analysé ici, le chapitre se concentre sur la g0 du thread principal (m0). Cependant, il faut noter que ce modèle s'appliquera plus tard à l'ensemble du runtime : chaque thread M que Go créera au fil de l'exécution possédera sa propre g0, constituant ainsi un contexte d'exécution privilégié pour les opérations critiques du runtime. Quand une goroutine utilisateur déclenche le garbage collector, effectue un appel système bloquant ou nécessite toute interaction avec le kernel, le runtime bascule temporairement vers la g0 de son thread porteur pour gérer ces opérations dans l'espace système, puis revient à la goroutine utilisateur. Cette séparation stricte entre contexte système (g0) et contexte utilisateur (goroutines normales) garantit l'isolation et la robustesse du runtime, tout en permettant au scheduler de maintenir un contrôle précis sur les transitions entre espace utilisateur et espace kernel.

Le runtime prépare également la structure `m` (machine) représentant le thread du système d'exploitation principal (m0). Cette structure, également définie dans `runtime/runtime2.go`, maintient la liaison entre les threads OS et les goroutines Go. Le thread m0 sera responsable de l'exécution initiale et du démarrage du scheduler.

Ces initialisations fondamentales, orchestrées principalement dans les fonctions de `runtime/proc.go` et `runtime/stack.go`, établissent les bases du modèle de concurrence de Go avant même que le scheduler ne devienne opérationnel.