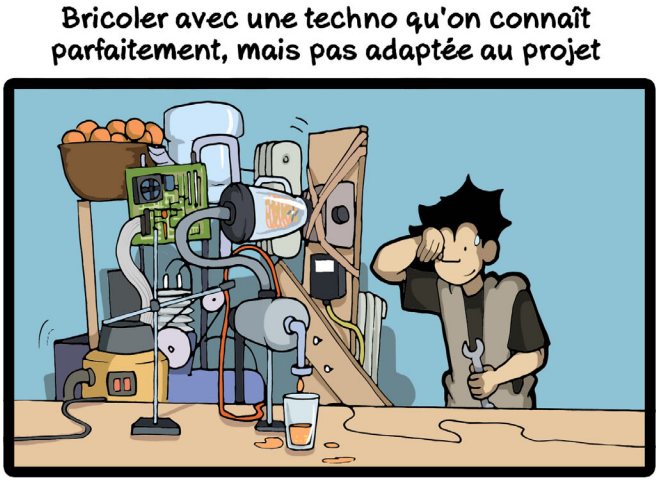
2

Pourquoi Go ?



Un nouveau langage de programmation est typiquement créé dans l'intention de résoudre la frustration de compromis exigés par les langages existants, dans le cadre de certains cas d'utilisation. Tous les langages de programmation sont conçus pour mettre la priorité sur certaines valeurs au détriment d'autres, et Go ne fait pas exception. En conséquence, il présente des points forts exceptionnels pour certains cas d'utilisation, mais exige aussi, comme les autres, certains compromis qu'il faut comprendre.

Nous allons ainsi passer en revue une série de problèmes posés par certains langages et voir les solutions apportées par d'autres, et enfin l'approche de Go sur chaque sujet. Cela vous aidera à cerner les cas d'utilisation pour lesquels Go est particulièrement pertinent.

Note *› Nous ne prendrons en compte pour nos comparaisons que quelques langages typiques de l'industrie, notre objectif étant de mieux comprendre les intentions sous-jacentes à la conception de Go. Cela ne signifie pas pour autant qu'il n'existe pas d'autres solutions avec des langages spécifiques ou que les solutions de Go ne sont pas implémentées ailleurs.*

2.1. Compromis performance vs. portabilité

Le problème

D'un côté du spectre, on a des langages comme C, qui misent énormément sur la performance et très peu sur la portabilité. Du fait de la variété des architectures physiques d'ordinateurs, le même code, une fois compilé, peut avoir des comportements très différents d'un ordinateur à l'autre. Cela pose des problèmes évidents pour une grande quantité de cas d'utilisation dans l'industrie.

De l'autre côté du spectre, on a des langages comme Python ou Ruby, dont le code se comportera toujours de la même manière sur toutes les architectures (à condition de vous assurer d'utiliser la même version du langage, bien sûr). Ces langages arrivent à ce résultat notamment en interprétant le code à la volée, plutôt qu'en le compilant en code machine puis en l'exécutant. Comme vous pouvez le deviner, cette approche est coûteuse en performance, aucun travail n'est prémâché par la compilation, vu qu'il n'y a pas de compilation.

Certaines solutions

Java a une approche très particulière pour résoudre ce problème et obtenir de la performance sans pour autant interpréter : il ne compile pas le code pour l'architecture physique de l'ordinateur mais pour la JVM (*Java Virtual Machine*), sur laquelle s'appuient aussi d'autres langages, tels que Scala et Groovy. Le concept : le code est effectivement compilé, mais au lieu d'obtenir du code machine, vous le compilez en *Java bytecode*. Cet autre langage non lisible par l'humain est une version prémâchée des opérations qui devront être compilées une seconde fois plus tard le plus rapidement possible. Le langage ne contient aucune spécificité de plateforme, ce qui garantit au développeur que ce bytecode sera compris strictement de la même manière par toutes les JVM, quels que soient les systèmes d'exploitation sur lesquels elles sont installées. Lorsque la JVM récupère ce bytecode, elle le compile donc en code machine très rapidement juste avant de l'exécuter ; cela s'appelle la *JIT compilation* (*Just In Time*, dont la traduction littérale est "juste à temps", mais peut être désignée par "compilation à la volée").

L'approche de Go

Go, qui est très regardant sur la performance, est un langage entièrement compilé, directement en langage machine. L'utilisateur n'a donc n'a pas du tout besoin d'installer quoi que ce soit (pas même Go) pour exécuter le binaire initialement écrit en Go, quel que soit son système d'exploitation.

Mais comment alors Go fait-il pour garantir la portabilité du code ? Simplement, le compilateur Go permet de compiler le code sur votre ordinateur directement pour un autre OS, et une autre architecture.

Par exemple, si vous utilisez macOS, vous pouvez passer au compilateur les variables `GOOS` et `GOARCH` de sorte que la compilation cible Windows, puis compiler votre code comme vous le faites d'habitude. Vous obtiendrez un fichier `.exe` directement exécutable sur n'importe quel ordinateur sous Windows !

L'une des raisons pour lesquelles tout ceci marche aussi bien vient de ce que la bibliothèque standard de Go a été conçue pour être facilement portable et permettre le code non portable si besoin. La portabilité effective de Go est un effort à tous les niveaux de la plateforme.

Go, un langage écrit… en Go !

La plupart des langages ont le cœur de leur compilateur ou interpréteur écrit avec un autre langage plus bas niveau (souvent C). Go est lui-même un langage très bas niveau et, depuis Go 1.5, tout le compilateur Go est écrit avec une version précédente de Go lui-même !

Cela lui permet notamment de réaliser sans tricher les fonctionnalités qui ne sont pas reproductibles en C (par exemple, les tableaux qui n'ont pas besoin d'être des pointeurs en Go ou les retours multiples de fonction).

Go va même jusqu'à déborder de sa propre utilisation personnelle, puisqu'il existe maintenant GoRuby, une implémentation de Ruby dont le cœur est entièrement en Go. Cela dit, son utilisation n'est pas commune à l'heure où ces lignes sont écrites, et on lui préfère le plus souvent les implémentations MRI (en C) ou JRuby (en Java).