Python : modules et packages

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Go

Packages

- En **Go**, le code est organisé dans des **packages**
- Un package permet de regrouper des fonctions, variables, structures...
- Chaque fichier **Go** commence par une déclaration `package`
- Un package facilite la réutilisation et l'organisation du code

Go

Packages

- En **Go**, le code est organisé dans des **packages**
- Un package permet de regrouper des fonctions, variables, structures...
- Chaque fichier **Go** commence par une déclaration `package`
- Un package facilite la réutilisation et l'organisation du code

Un package = un dossier contenant des fichiers `.go` qui partagent le même `package nom`.

Go

Pourquoi utiliser des packages ?

- Réutiliser du code
- Organiser les projets
- Réduire le couplage
- Faciliter la maintenance
- Favoriser le travail collaboratif

Go

Avant de commencer

- Créez un nouveau projet `go-packages`
- Créez deux fichiers `main.go` et `fonctions.go`

Go

Étant donné le fichier `fonctions.go` (à créer à la racine du projet)

```go
package main

func Somme(a int, b int) int {
    return a + b
}

func Produit(a int, b int) int {
    return a * b
}
```

Go

Pour utiliser ces fonctions dans `main.go`

```
package main

import "fmt"

func main() {
    fmt.Println(Somme(2, 3))
    fmt.Println(Produit(2, 3))
}
```

Go

Pour utiliser ces fonctions dans `main.go`

```
package main

import "fmt"

func main() {
    fmt.Println(Somme(2, 3))
    fmt.Println(Produit(2, 3))
}
```


En relançant la commande suivante ⇒ erreur car `fonctions.go` n'a pas été compilé

```
go run main.go
```

Go

Pour compiler les deux

```
go run main.go fonctions.go
```

Go

Pour compiler les deux

```
go run main.go fonctions.go
```

Ou

```
go run .
```

Go

Pour compiler les deux

```
go run main.go fonctions.go
```

Ou

```
go run .
```


Résultat

```
5
6
```

Go

Avant de commencer

- Créez un nouveau package `utils`
- Déplacez `fonctions.go` dans `utils`
- Initialisez le module

Go

Avant de commencer

- Créez un nouveau package `utils`
- Déplacez `fonctions.go` dans `utils`
- Initialisez le module

Ensuite, exécutez

```
go mod init go-packages
```

Go

Metttons à jour le package dans `fonctions.go`

```
package utils

func Somme(a int, b int) int {
    return a + b
}

func Produit(a int, b int) int {
    return a * b
}
```

Go

Pour utiliser le package `utils` dans `main.go`, il faut l'importer

```
package main

import "fmt"
import "go-packages/utils"

func main() {
        fmt.Println(utils.Somme(2, 3))
        fmt.Println(utils.Produit(2, 3))
}
```

Go

Pour utiliser le package `utils` dans `main.go`, il faut l'importer

```
package main

import "fmt"
import "go-packages/utils"

func main() {
        fmt.Println(utils.Somme(2, 3))
        fmt.Println(utils.Produit(2, 3))
}
```


Résultat

```
5
6
```

Go

Utilisation d'un bloc d'import

```
package main

import (
        "fmt"
        "go-packages/utils"
)

func main() {
        fmt.Println(utils.Somme(2, 3))
        fmt.Println(utils.Produit(2, 3))
}
```

Go

Utilisation d'un bloc d'import

```
package main

import (
        "fmt"
        "go-packages/utils"
)

func main() {
        fmt.Println(utils.Somme(2, 3))
        fmt.Println(utils.Produit(2, 3))
}
```

Résultat

```
5
6
```

Go

On peut également utiliser un alias

```
package main

import (
        "fmt"
        u "go-packages/utils"
)

func main() {
        fmt.Println(u.Somme(2, 3))
        fmt.Println(u.Produit(2, 3))
}
```

Go

L'import avec . permet d'éviter le préfixe du package

```
package main

import (
        "fmt"
        . "go-packages/utils"
)

func main() {
        fmt.Println(Somme(2, 3))
        fmt.Println(Produit(2, 3))
}
```

Go

L'import avec . permet d'éviter le préfixe du package

```
package main

import (
        "fmt"
        . "go-packages/utils"
)

func main() {
        fmt.Println(Somme(2, 3))
        fmt.Println(Produit(2, 3))
}
```

Remarque

Utilisation déconseillée dans les projets professionnels.

Go

Import uniquement pour exécuter le code d'initialisation

```
import (
    _ "github.com/go-sql-driver/mysql"
)
```

Go

Import uniquement pour exécuter le code d'initialisation

```
import (
    _ "github.com/go-sql-driver/mysql"
)
```

Utilisation

- Chargement automatique d'un driver
- Exécution des fonctions `init()`
- Aucun accès direct aux fonctions du package

Go

Go : compilé ou interprété ?

- Go est un langage compilé
- Le code source est transformé en binaire natif
- Aucune machine virtuelle n'est nécessaire
- Le binaire peut être distribué directement
- Les performances sont généralement proches de celles du C

Go

Compilation

```
go build
```

Go

Compilation

```
go build
```

Exécution

```
./monprogramme
```

Go

Modules Go

- Introduits officiellement avec Go 1.11
- Permettent la gestion des dépendances
- Remplacent GOPATH
- Reposent sur les fichiers `go.mod` et `go.sum`

Go

Création d'un module

```
go mod init go-packages
```

Go

Création d'un module

```
go mod init go-packages
```

Génère automatiquement

`go.mod`

Go

Contenu d'un fichier `go.mod`

```
module mon-packages

go 1.25
```

Go

Ajout d'une dépendance

```
go get github.com/gin-gonic/gin
```

Go

Ajout d'une dépendance

```
go get github.com/gin-gonic/gin
```

Puis

```
import "github.com/gin-gonic/gin"
```

Go

go.sum

- Généré automatiquement
- Contient les empreintes cryptographiques
- Garantit l'intégrité des dépendances téléchargées
- Ne doit généralement pas être modifié manuellement

Go

Règle de visibilité

- Majuscule = exporté
- Minuscule = privé au package

Go

Règle de visibilité

- Majuscule = exporté
- Minuscule = privé au package


```
func Somme() {}
func produit() {}
```

Go

Exemple

```
package calculs

func Somme(a int, b int) int {
    return a+b
}

func produit(a int, b int) int {
    return a*b
}
```

Go

Fonction `init()`

```
package utils

import "fmt"

func init() {
    fmt.Println("Initialisation")
}
```

Go

Caractéristiques

- Exécutée automatiquement
- Avant `main()`
- Peut exister dans plusieurs fichiers
- Aucun appel explicite nécessaire

Go

Package internal

- Introduit pour limiter l'accès à certains packages
- Réservé au projet courant
- Renforce l'encapsulation
- Évite les dépendances non souhaitées

Go

Organisation possible

```
go-packages/
|
+-- cmd/
|
+-- internal/
|   |
|   +-- utils/
|
+-- pkg/
|
+-- go.mod
```

Go

Workspace

- Introduit avec Go 1.18
- Permet de travailler sur plusieurs modules
- Facilite le développement de projets complexes

Go

Création d'un workspace

```
go work init
```

Go

Création d'un workspace

```
go work init
```

Génère

```
go.work
```

Go

Exemple

```
workspace/
|
+-- api/
|
+-- common/
|
+-- frontend/
|
+-- go.work
```

Go

Bonnes pratiques

- Un package = une responsabilité
- Utiliser `gofmt`
- Respecter les conventions de nommage
- Limiter l'utilisation de `init()`
- Éviter les imports avec `.`
- Utiliser `internal` pour le code métier
- Initialiser systématiquement un module Go
- Versionner `go.mod` et `go.sum`