

G.I. Go

Générateurs & Itérateurs en Go



Image de ChatGPT

Benoît Masson

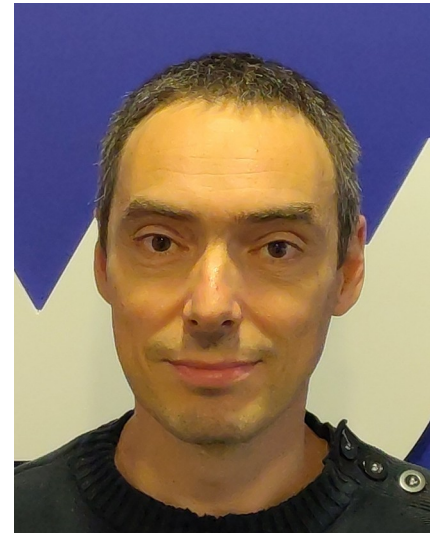


7 février 2025

Générique

- Développeur Go depuis 2015
- Software Craftsman

@OVHcloud Rennes
Noms de Domaines depuis 2020



benoitmasson

Mais pourquoi ?

- Uniformisation des pratiques observées
 - `archive/tar.Reader.Next`
 - `bufio.Reader.ReadByte`
 - `bufio.Scanner.Scan`
 - `container/ring.Ring.Do`
 - `database/sql.Rows`
 - `expvar.Do`
 - `flag.Visit`
 - `go/token.FileSet.Iterate`
 - `path/filepath.Walk`
 - `runtime.Frames.Next`
 - `sync.Map.Range`
 - ...

Mais pourquoi ?

- Uniformisation des pratiques observées

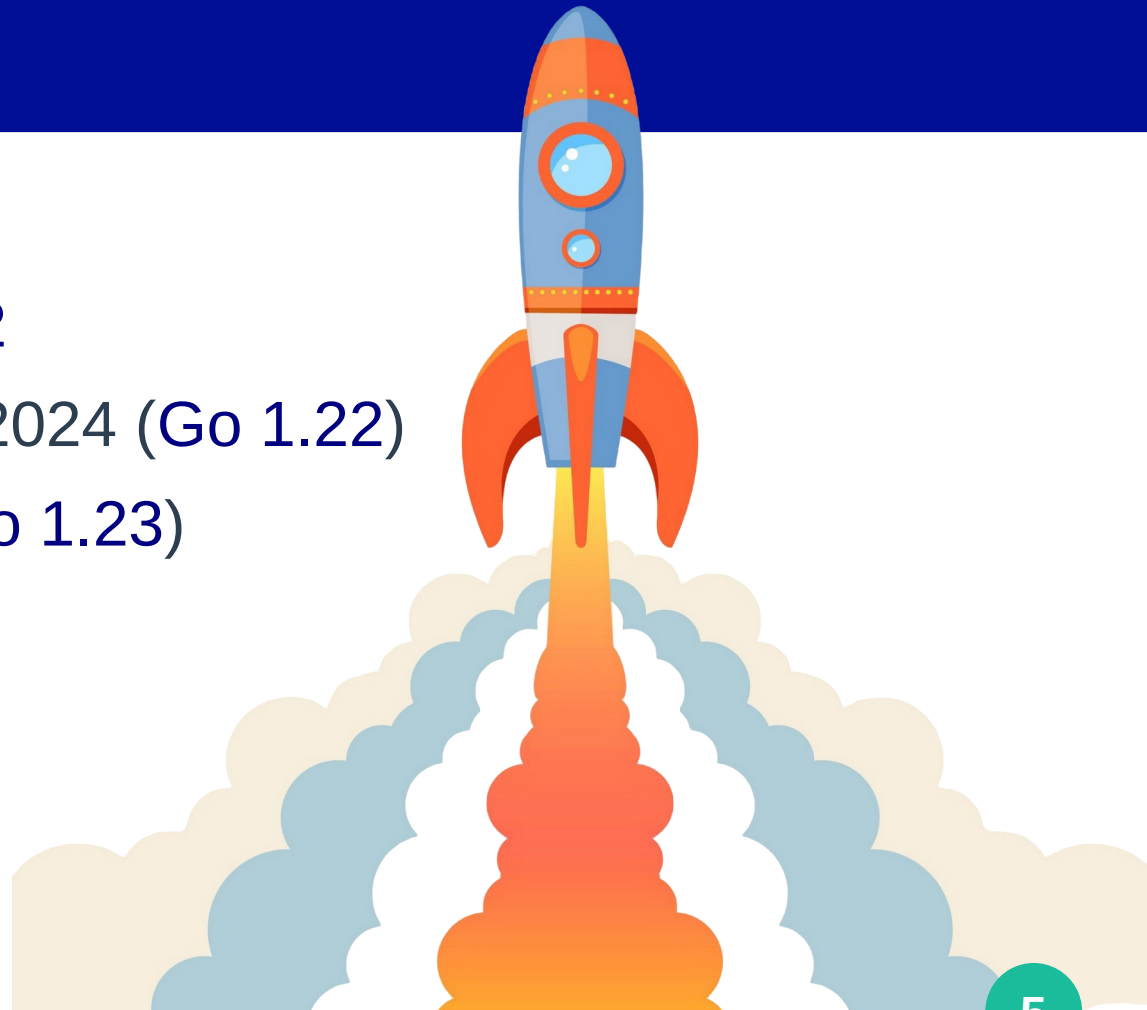
- `archive/tar.Reader.Read`
- `bufio.Reader.Read`
- `bufio.Scanner.Scan`
- `container/ring.Ring`
- `database/sql.Row`
- `expvar.Do`
- `flag.Visit`
- `go/token.FileSet.Iterate`
- `path/filepath.Walk`
- `runtime.Frames.Next`
- `sync.Map.Range`



- S'appuie sur les génériques, disponibles depuis 2022 (Go 1.18)

Historique

- Discuté depuis octobre 2022
- Expérimenté depuis février 2024 (Go 1.22)
- Généralisé en août 2024 (Go 1.23)



Comment ça marche



Comment ça marche



- itérateur « push » = **générateur**
 - 1 ou 2 valeurs
- `yield`, `break`
- **itérateur** « pull »
- test unitaire

Bibliothèque standard

slices

- `All([]E) iter.Seq2[int, E]`
- `Values([]E) iter.Seq[E]`
- `Collect(iter.Seq[E]) []E`
- `AppendSeq([]E, iter.Seq[E]) []E`
- `Backward([]E) iter.Seq2[int, E]`
- `Sorted(iter.Seq[E]) []E`
- `SortedFunc(iter.Seq[E], func(E, E) int) []E`
- `SortedStableFunc(iter.Seq[E], func(E, E) int) []E`
- `Repeat([]E, int) []E`
- `Chunk([]E, int) iter.Seq([]E)`

maps

- `All(map[K]V) iter.Seq2[K, V]`
- `Keys(map[K]V) iter.Seq[K]`
- `Values(map[K]V) iter.Seq[V]`
- `Collect(iter.Seq2[K, V]) map[K, V]`
- `Insert(map[K, V], iter.Seq2[K, V])`

Cas d'usage : Requêtes paginées

```
func AllItems(client api.Client) iter.Seq2[Item, error] {
    return func(yield func(string, error) bool) {
        var cursor string
        var items []string
        for {
            items, cursor, err := client.getPage(cursor)
            if err != nil {
                if errors.Is(err, io.EOF) {
                    return
                }
                if !yield(Item{}, fmt.Errorf("failed to get page: %w", err)) {
                    return
                }
            }
            for _, item := range items {
                if !yield(item, nil) {
                    return
                }
            }
        }
    }
}
```

Usage

```
for item, err := range AllItems(client) {
    if err != nil {
        // ...
        break
    }
    // ...
}
```

Cas d'usage : Scan SQL

Source : github.com/achille-roussel/sqlrange

```
for p, err := range sqlrange.Query[Point](db, `select x, y from points`) {  
    if err != nil {  
        ...  
    }  
    ...  
}
```

Critiques & Conclusion

Lire par exemple : `go-evolves-in-the-wrong-direction`

- Deux façons d'itérer sur les anciennes bibliothèques
- Signature de l'itérateur restrictif, pas adapté à toutes les situations
- Augmentation de la complexité implicite du *range*
- Vérification manuelle des erreurs à chaque itération

Critiques & Conclusion

Lire par exemple : `go-evolves-in-the-wrong-direction`

- Deux façons d'itérer sur les anciennes bibliothèques
- Signature de l'itérateur restrictif, pas adapté à toutes les situations
- Augmentation de la complexité implicite du *range*
- Vérification manuelle des erreurs à chaque itération

Mon avis : pas un vrai problème, la complexité étant déportée dans une fonction isolée, l'usage est **uniformisé, simple et clair**

Merci !

Feedbacks



Code & Slides

