GoのGenerics関連Proposal最新状況まとめと簡単な解説 (2021年8月版)

syumai

Go 1.17 リリースパーティー

自己紹介

syumai

`Go Language Specification 輪読会` と言う勉強会を主催しています

普段はGoとTypeScriptを書きつつ生活しています

Twitter: @__syumai

本日のテーマ

GoのGenerics関連Proposal

そもそも、Proposalってどこに出ているの?

Proposalの探し方

- 基本は、golang/goのGitHub Issue上で `Proposal` タグが付いているものを探せばOK
  - ジェネリクス関係は `generics` タグも付いている
- golang/proposal宛ての変更としてレビューを先に行ってからIssueがOpenされるものもある (表の `go/ast` の提案がこれに該当)
  - Gerrit上で `repo: proposal` のものを見ると探しやすい

見付けたProposal一覧を表にしました

Proposal	Status	Author	GitHub Issue	Proposal Document / Gerrit
type parameters	**accepted (2021/2/11)**	ianlancetaylor	#43651	Proposal
type sets	**accepted (2021/7/22)**	ianlancetaylor	#45346	Gerrit
constraints package	**accepted (2021/8/19)**	ianlancetaylor	#45458	
slices package	**accepted (2021/8/12)**	ianlancetaylor	#45955	
maps package	議論中 (2021/8/20現在)	rsc	#47649	
sync, sync/atomic: add PoolOf, MapOf, ValueOf	議論中 (2021/8/20現在)	ianlancetaylor	#47657	
go/ast changes for generics	議論中 (2021/8/20現在)	findleyr	#47781	Proposal
go/types changes for generics	議論中 (2021/8/20現在)	findleyr	-	Gerrit
go/parser: add a mode flag to disallow the new syntax	議論中 (2021/8/20現在)	findleyr	#47783	
disallow type parameters as RHS of type declarations	議論中 (2021/8/20現在)	findleyr	#45639	
Generic parameterization of array sizes	議論中 (2021/8/20現在)	ajwerner	#44253	Proposal
container/heap package	議論中 (2021/8/20現在)	cespare	#47632	

Genericsに出ているProposalのざっくり分類

1. 言語仕様についてのProposal

- type parameters
- type sets
- Generic parameterization of array sizes

2. package追加 / 既存のpackageの変更のProposal

- constraints package
- slices package
- maps package
- sync, sync/atomic package

3. 静的解析関連のpackageの変更のProposal

- go/ast
- go/types

今日は `package追加 / 既存のpackageの変更のProposal` を中心に紹介します

各Proposalの紹介

言語仕様についてのProposal

(他に優れた資料があるので、簡単な紹介に留めます)

type parameters

Status: accepted

- Goでジェネリックなプログラミングを行えるようにするために、型や関数が *type parameter* を受け付けることを出来るようにする提案
- 型パラメータが受け付ける型に対しての *constraints* の導入や、型推論のルールについてもこの提案に含まれている
- 先ほどの表にあるProposalは全てこれをベースに提案が行われている

概要については tenntennさんの資料 を参照いただくのをおすすめします

Proposal内のコード例

- Stringerと言う名前で、String()メソッドを持つconstraintを定義している
- Stringify関数は、type parameterとして `T` を宣言し、constraintにStringerを指定している
  - slice sの要素型 `T` は Stringer を満たしているので、 `String()` メソッドを呼ぶことが出来る

```
type Stringer interface {
        String() string
}

func Stringify[T Stringer](s []T) (ret []string) {
        for _, v := range s {
                ret = append(ret, v.String())
        }
        return ret
}
```

https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md

type sets

Status: accepted

- type parameters proposalがacceptされた時点で含まれていた、constraintsにおける *type list* を置き換える提案
- type listのわかりにくさを解消し、より一般的な解決法を提案したもの

コード例

```
type PredeclaredSignedInteger interface {
        int | int8 | int16 | int32 | int64
}

type SignedInteger interface {
        ~int | ~int8 | ~int16 | ~int32 | ~int64
}
```

type sets

- 2021年8月現在、Type Parameter Proposalのドキュメントがtype sets版への書き換えが行われている
- また、言語仕様の変更も既にtype sets版で作業が行われている

type setsの詳細については、Nobishiiさんの記事を参照いただくことをおすすめします

- Go の "Type Sets" proposal を読む - Zenn
- Type Sets Proposalを読む(2) - Zenn

package追加 / 既存のpackageの変更のProposal

constraints package

Status: accepted

- type parameterのconstraintsに頻繁に使われるであろう定義をまとめたpackage
- 例えば、 `constraints.Integer` は全ての整数型にマッチする制約
    - 下記のような整数型の値のみを受け付ける関数を宣言する時に使える

```
import "constraints"

func DoubleInteger[T constraints.Integer](i T) T {
  return i + i
}

func main() {
  DoubleInteger(int(1))   // => int(2)
  DoubleInteger(uint8(2)) // => uint8(4)
  DoubleInteger(int64(3)) // => int64(6)
}
```

constraints packageに定義されている型の一覧

```
package constraints

/* 数値型系 */
// 符号付き整数型の制約
type Signed interface { ... }
// 符号なし整数型の制約
type Unsigned interface { ... }
// 整数型の制約
type Integer interface { ... }
// 浮動小数点数型の制約
type Float interface { ... }
// 複素数型の制約
type Complex interface { ... }

/* 演算子系 */
// 順序付けが可能な型の制約（次の演算子をサポートする型 < <= >= >）
type Ordered interface { ... }

/* 複合型系 */
// スライス型の制約
type Slice[Elem any] interface { ~[]Elem }
// マップ型の制約
type Map[Key comparable, Val any] interface { ~map[Key]Val }
// チャネル型の制約
type Chan[Elem any] interface { ~chan Elem }
```

constraints packageを使ったコード例

任意の型のsliceのソート

- `Slice` と `Ordered` を使った例
- 並び替え可能な要素型を持つ任意の型のsliceを受け取り、ソートして返す

```
import (
  "fmt"
  "sort"
)

func SortSlice[S constraints.Slice[T], T constraints.Ordered](s S) {
  sort.Slice(s, func(i, j int) bool { return s[i] < s[j] })
}

func main() {
  ints := []int{3,1,4,2,5}
  SortSlice(ints)
  fmt.Println(ints) // [1 2 3 4 5]

  strs := []string{"c", "b", "a"}
  SortSlice(strs)
  fmt.Println(strs) // [a b c]
}
```

任意の型のチャネルのバッファ内の値を全てSliceに出力する

- FlushSliceは、constraints.Chanで制約された任意の要素型のチャネルを受け付ける
- 同じ要素型のsliceにバッファの内容を読み出して返す

```
import (
  "fmt"
)

func FlushSlice[C constraints.Chan[T], T any](ch C) []T {
  result := make([]T, 0, cap(ch)))
Loop:
  for {
    select {
      case v, ok := <-ch:
        if !ok {
          break Loop
        }
        result = append(result, v)
      default:
        break Loop
    }
  }
  return result
}

func main() {
  intCh := make(chan int, 3)
  intCh <- 1; intCh <- 2; intCh <- 3

  strCh := make(chan string, 3)
  strCh <- "a"; strCh <- "b"; strCh <- "c"

  fmt.Println(FlushSlice(intCh)) // [1 2 3]
  fmt.Println(FlushSlice(strCh)) // [a b c]
}
```

補足

`Slice`, `Map`, `Chan` constraintの使いどころについて

- SortSliceの例は、実は `func SortSlice[T constraints.Ordered](s []T) {}` と書けるので、 `constraints.Slice` は使わなくても良い
- これらのconstraintが必要になるのは、引数として受け取ったsliceの型の値をそのまま返したい場合
  - 例えば、スライスを操作した結果を返す関数に `type Ints []int` を渡した場合、戻り値は `Ints` 型であって欲しい

```
func F[S constraints.Slice[T], T any](s S) S {} // 戻り値の型は S (Ints)
func F[T any](s []T) []T {} // 戻り値の型は []T ([]int)
```

補足

`Chan` の補足

- `Chan` は `~chan Elem` を制約とするので、実は `<-chan Elem` はこの制約を満たさない
  - `chan Elem` は `<- chan Elem` のunderlying typeではないため
  - どんな channel に対しても使える制約ではない点に注意が必要
- 次のようなコードはcompile error

```
func main() {
  intCh := make(chan int, 3)
  intCh <- 1; intCh <- 2; intCh <- 3

  var intRecvCh <-chan int = intCh // intChを <-chan int 型に変換
  fmt.Println(FlushSlice(intRecvCh)) // error: <-chan int は constraints.Chan[T]を満たさない
}
```

補足

その他のよく使われそうなconstraintsについて

- Go本体に組み込まれる `any` と `comparable` と言うconstraintもある
- anyは、全ての型を受け付けるconstraintで、comparableは `比較可能` な型 (`==, !=, <, <=, >, >=` をサポートする型) を受け付けるconstraint
- これらと、constraints packageを使い分けながらコードを書いていくことになります

slices package

Status: accepted

- ジェネリックなslice操作を行うためのpackageを導入する提案
- 次のような操作が簡単に出来るようになる
  - slice同士の比較
  - sliceの一部を取り除いたり、sliceの途中に要素を挿入したりする操作
    - (これまでSliceTricksを駆使する必要があった)

slices packageで宣言されている関数の一覧

```
package slices

import "constraints"

/* 比較系 */
// 2つのsliceの長さが同じで、含まれる要素とその順番が等しいかどうかを返す
func Equal[T comparable](s1, s2 []T) bool
func EqualFunc[T1, T2 any](s1 []T1, s2 []T2, eq func(T1, T2) bool) bool
// 2つのsliceを比較する
// 結果は `0 if s1==s2, -1 if s1 < s2, and +1 if s1 > s2` の数値で得られる
func Compare[T constraints.Ordered](s1, s2 []T) int
func CompareFunc[T any](s1, s2 []T, cmp func(T, T) int) int

/* 検索系 */
// vのs内でのindexを返す
func Index[T comparable](s []T, v T) int
func IndexFunc[T any](s []T, f func(T) bool) int
// vがsに含まれているかどうかを返す
func Contains[T comparable](s []T, v T) bool

/* 要素操作系 */
// vをsのi番目に挿入し、変更されたsliceを返す
func Insert[S constraints.Slice[T], T any](s S, i int, v ...T) S
// s[i:j]をsから除去して、変更されたsliceを返す
func Delete[S constraints.Slice[T], T any](s S, i, j int) S

/* 複製系 */
// sを複製したsliceを返す
func Clone[S constraints.Slice[T], T any](s S) S
// 等しい要素を取り除いたsliceを返す。(Unixのuniq commandのようなイメージ)
func Compact[S constraints.Slice[T], T comparable](s S) S
func CompactFunc[S constraints.Slice[T], T any](s S, cmp func(T, T) bool) S

/* 容量操作系 */
// 容量をn増やしたsliceを返す
func Grow[S constraints.Slice[T], T any](s S, n int) S
// sliceの使われていない容量を取り除いたsliceを返す
func Clip[S constraints.Slice[T], T any](s S) S
```

これまでの書き方との比較

sliceの比較

```
// slicesなし
func EqualInts(a, b []int) bool {
  if len(a) != len(b) {
    return false
  }
  for i := 0; i < len(a); i++ {
    if a[i] != b[i] {
      return false
    }
  }
  return true
}

func EqualStrs(a, b []string) bool {
  ... // string用の全く同じ実装
}

func main() {
  is1 := []int{1, 2, 3}
  is2 := []int{1, 2, 4} // not equal

  ss1 := []string{"a", "b", "c"}
  ss2 := []string{"a", "b", "c"} // equal

  fmt.Println(EqualInts(is1, is2)) // false
  fmt.Println(EqualStrs(ss1, ss2)) // true
}

// slicesあり
func main() {
  is1 := []int{1, 2, 3}
  is2 := []int{1, 2, 4}

  ss1 := []string{"a", "b", "c"}
  ss2 := []string{"a", "b", "c"}

  fmt.Println(slices.Equal(is1, is2)) // false
  fmt.Println(slices.Equal(ss1, ss2)) // true
}
```

sliceへのInsert / Delete

- Insert / Deleteは SliceTricks から持ってきている。これまでは行いたい操作に対して実装が複雑すぎたが、シンプルに書けるようになった

```
// slicesなし
func InsertInt(a []int, x, i int) []int {
  return append(a[:i], append([]T{x}, a[i:]...)...)
}

func DeleteInt(a []int, i int) []int {
  copy(a[i:], a[i+1:])
  a[len(a)-1] = 0
  return a[:len(a)-1]
}

func main() {
  a := []int{1, 2, 3, 4, 5}
  a = InsertInt(a, 2, 100) // index: 2に100を挿入
  a = DeleteInt(a, 3) // index: 3の要素を削除
  fmt.Println(a) // [1, 2, 100, 4, 5]
}

// slicesあり
func main() {
  a := []int{1, 2, 3, 4, 5}
  a = slices.Insert(a, 2, 100) // index: 2に100を挿入
  a = slices.Delete(a, 3, 4) // index: 3:4 の要素を削除
  fmt.Println(a) // [1, 2, 100, 4, 5]
}
```

補足

- Map, Filter, Reduceはここには含まれていない
  - どこか、より包括的な *streams API* の一部になるとよいだろう、とrscがコメントしている

(2021/8/20時点でacceptedなのはここまで)

maps package

Status: 議論中

- ジェネリックなmap操作を行うためのpackageを導入する提案

maps packageで宣言されている関数の一覧

注) 下記の内容は今後変わる可能性が非常に高いです

```
package maps

/* キー、値の抽出 */
// map `m` のキーのsliceを返す。順序は不定
func Keys[K comparable, V any](m map[K]V) []K
// `m` の値のsliceを返す。順序は不定
func Values[K comparable, V any](m map[K]V) []V

/* 比較系 */
// 2つのmapが同じキーと値のペアを保持しているかを返す
func Equal[K, V comparable](m1, m2 map[K]V) bool
func EqualFunc[K comparable, V1, V2 any](m1 map[K]V1, m2 map[K]V2, cmp func(V1, V2) bool) bool

/* 複製系 */
// `m` のコピーを返す。浅いクローン（新しい map のキーと値への単純な代入）となる
func Clone[K comparable, V any](m map[K]V) map[K]V

/* 要素操作系 */
// `m` の要素を全て削除する
func Clear[K comparable, V any](m map[K]V)
// map `src` のキーと値のペアを全て map `dst` に追加する
// 重複したキーの値は上書きされる
func Add[K comparable, V any](dst, src map[K]V)
// `m` 対して、関数 `keep` が false を返すキーと値のペアを全て削除する
func Filter[K comparable, V any](m map[K]V, keep func(K, V) bool)
```

これまでの書き方との比較

mapのキーのsliceの取得

```
// mapsなし
func IntMapKeys(m map[int]bool) []int {
  s := make([]int, 0, len(m))
  for k := range m {
    s = append(s, k)
  }
  return s
}

func main() {
  m := map[int]bool{
    1: true,
    2: false,
    3: true
  }
  fmt.Println(IntMapKeys(m)) // 例）[2, 1, 3]（順序は不定）
}

// mapsあり
func main() {
  m := map[int]bool{
    1: true,
    2: false,
    3: true
  }
  fmt.Println(maps.Keys(m)) // 例）[3, 2, 1]（順序は不定）
}
```

その他の使用例

奇数と偶数のmapを分ける

```
func SeparateEvenOddMaps[M constraints.Map[K, V], K comparable, V constraints.Integer](m M) (even, odd M) {
        even, odd = maps.Clone[M, K, V](m), maps.Clone[M, K, V](m) // Note: 手元の実装で [M, K, V] は `m` から推論出来なかった
        maps.Filter(even, func (k K, v V) bool { return v % 2 == 0 })
        maps.Filter(odd, func (k K, v V) bool { return v % 2 == 1 })
        return
}

func main() {
        strIntMap := map[string]int{
                "A": 1,
                "B": 2,
                "C": 3,
                "D": 4,
        }
        even, odd := SeparateEvenOddMaps(strIntMap)
        fmt.Println(even) // 例) map[B:2 D:4] (順序は不定)
        fmt.Println(odd) // 例) map[A:1 C:3] (順序は不定)
}
```

sync, sync/atomic: add PoolOf, MapOf, ValueOf

Status: 議論中 (2021/8/20現在)

- sync.Pool / sync.Map / atomic.Valueをジェネリックにする提案
- これまで、これらは `interface{}` 型の値を受け付けるのみだったが、コンパイル時に型を決定して安全に扱えるようにする

sync packageの抜粋

```
package sync

// 従来のPool
type Pool struct {
        New func() interface{}
}
// (*Pool) Get / Put => interface{}

// T型の値のPool
type PoolOf[T any] struct {
    ...
    New func() T
}
// (*PoolOf[T]) Get / Put => T

// ---

// 従来のMap
type Map struct { ... }
// (*Map) Load(key interface{}) => interface{}

// K, Vをキーと値の型に持つMap
type MapOf[K comparable, V any] struct { ... }
// (*MapOf[K, V]) Load(key K) => V
```

atomic packageの抜粋

```
package atomic

// 従来のValue
type Value struct { ... }
// (*Value) Load() => interface{}

// T型のValue
type ValueOf[T any] struct { ... }
// (*ValueOf[T]) Load() => T
```

atomic.ValueOfの利用イメージ

```
import "atomic"

type Config struct {
  A int
  B string
}

var config atomic.ValueOf[Config]

func Load() Config {
  return config.Load()
}

func Store(c Config) {
  config.Store(c)
}
```

型アサーションが不要となり、安全に扱えるようになっています

Generic parameterization of array sizes

Status: 議論中 (2021/8/20現在)

- 配列は長さによって型が異なるので、constraintを簡単に書くことが出来ない
- これを例外的に許容するための独自の文法を追加する提案

注) 本Proposalの内容は `type list` のままで書かれているので、独自にtype setsで解釈して紹介します。

配列を受け付けるconstraintの例

```
type IntArray interface {
  ~[1]int | ~[2]int | ~[3]int | ~[4]int | ... | ~[100]int // 必要な分を全部unionで定義する必要がある
}

func PrintInts(ints IntArray) {
  fmt.Println(ints)
}

func main() {
  i1 := [100]int{1,2, ..., 100}
  PrintInts(i) // ok
  i2 := [101]int{1,2, ..., 101}
  PrintInts(i) // ng（100までしか定義に含んでいないため）
}
```

提案されている内容 (をtype setで解釈したもの)

```
type IntArray interface {
  ~[...]int // どんな長さの配列も許容する
}

func PrintInts(ints IntArray) {
  fmt.Println(ints)
}

func main() {
  i1 := [100]int{1,2, ..., 100}
  PrintInts(i) // ok
  i2 := [101]int{1,2, ..., 101}
  PrintInts(i) // ok
}
```

- 多重配列 (matrix) のサポートについても提案に含まれている
- これについては、下記の `len(D)` のようにtype parameterから長さの情報を取得する想定 (個人的には、やや難しい気がする)

```
type Dim interface {
  ~[...]struct{}
}

type Matrix2D[D Dim, T any] [len(D)][len(D)]T

func main() {
  var m Matrix2D[[3]struct{}, int] // [3][3]int
}
```

その他のProposalについて

- 言語仕様が変わるので、静的解析に使われる go/ast や go/types などに対する変更もProposalが出されており、現在議論中のようです

引き続きProposalの状況を追っていきます！