



Go 泛型

泛型限制及其对中间件研发的影响

大明

Go 泛型

1 泛型简介

Introduction

2 泛型限制

Limitation

3 泛型影响

Impact

4 总结

Summary

1

泛型简介

定义和约束

基本语法

使用[T constraint]的语法来声明使用泛型：

- 可以声明在接口上
- 可以声明在结构体上
- 可以声明在方法上

```
- type Set[T any] interface {  
    Put(key T) error  
    Exist(key T) error  
}
```

```
type HashSet[T any] struct {  
}
```

```
- func Print[T any](t T) {  
    fmt.Printf(format: "%v", t)  
}
```

约束

约束是泛型里面新引入的语法元素。约束简单来说就是，类型参数所要满足的条件。约束具体来说可以分成：

- **基本类型和内置类型**：基本类型和内置虽然也能作为约束，但是实际中可能并不常用。

```
func PrintBool[T bool](t T) {  
    fmt.Printf(format: "%v", t)  
}  
  
func PrintSlice[T []int](t T) {  
    fmt.Printf(format: "%v", t)  
}  
  
func PrintArray[T [3]int](t T) {  
    fmt.Printf(format: "%v", t)  
}  
  
func PrintMap[T map[string]string](t T) {  
    fmt.Printf(format: "%v", t)  
}  
  
func PrintChan[T chan int](ch T) {  
    val := <-ch  
    fmt.Printf(format: "%v", val)  
}
```

约束

约束是泛型里面新引入的语法元素。约束简单来说就是，类型参数所要满足的条件。约束具体来说可以分成：

- **基本类型和内置类型**：基本类型和内置虽然也能作为约束，但是实际中可能并不常用。channel 这边类型推断看上去还是不太智能的样子，还得自己手动转换

```
func PrintReadOnlyChan[T <-chan int](ch T) {  
    val := <-ch  
    fmt.Printf(format: "%v", val)  
}
```

```
func TestPrintReadOnlyChan(t *testing.T) {  
    ch := make(chan int, 2)  
    ch <- 123  
    ch <- 234  
    PrintChan(ch)  
    // 这样会编译错误  
    // PrintReadOnlyChan(ch)  
    var ch1 <-chan int  
    ch1 = ch  
    PrintReadOnlyChan(ch1)  
}
```

约束

约束是泛型里面新引入的语法元素。约束简单来说就是，类型参数所要满足的条件。约束具体来说可以分成：

- 基本类型和内置类型
- **内置约束**：any 和 comparable。前者代表任意的类型，后者代表的是可比较类型，也就是 Go 在没有泛型时候就有的可比较的概念。例如在和 map 结合使用的时候，Key 必须满足 comparable 的约束。严格来说，any 和 comparable 也只不过是内置类型

```
1 type Set[T any] interface {  
2     Put(key T) error  
3     Exist(key T) error  
4 }  
5  
6 type HashSet[T comparable] struct {  
7     m map[T]struct{}  
8 }
```

约束

约束是泛型里面新引入的语法元素。约束简单来说就是，类型参数所要满足的条件。约束具体来说可以分成：

- 基本类型和内置类型
- 内置约束
- 普通接口：普通接口都可以被用作约束

```
type Hashable interface {  
    HashCode() int64  
}
```

```
type HashMap[T Hashable] struct {  
}
```



约束

约束是泛型里面新引入的语法元素。约束简单来说就是，类型参数所要满足的条件。

约束具体来说可以分成：

- 基本类型和内置类型
- 内置约束
- 普通接口
- `type X Y` 定义的类型
- **普通的结构体**：普通的结构体用作泛型约束，将无法调用任何方法，任何字段。也就是说，当成整体来用是可以的，但是不能访问字段或者方法

```
type User struct {
    Name string
}

func (u User) GetName() string {
    return u.Name
}

func PrintUser[T User](t T) {
    // 这个 OK
    fmt.Printf(format: "%v", t)

    // 编译错误
    // type bound for T has no method Name
    fmt.Printf(format: "%v", t.Name)
    // type bound for T has no method GetName
    fmt.Printf(format: "%v", t.GetName())
}
```

约束

约束是泛型里面新引入的语法元素。约束简单来说就是，类型参数所要满足的条件。

约束具体来说可以分成：

- 基本类型和内置类型
- 内置约束
- 普通接口
- 普通的结构体
- `type X Y` 定义的类型：和 Y 的类型直接相关

```
type Buyer User

func (b Buyer) GetName() string {
    return b.Name
}

func PrintBuyer[T Buyer](t T) {
    fmt.Printf(format: "%v", t.GetName())
}
```

约束

约束是泛型里面新引入的语法元素。约束简单来说就是，类型参数所要满足的条件。

约束具体来说可以分成：

- 基本类型和内置类型
- 内置约束
- 普通接口
- 普通的结构体
- `type X Y` 定义的类型：和 `Y` 的类型直接相关
- **约束接口**：用符号 `|` 来组合类型，用符号 `~` 来表达 `type X Y` 这种形式的衍生类型

```
type Number interface {  
    int32 | int64  
}  
  
func PrintNumber[T Number](t T) {  
    fmt.Printf(format: "%v", t)  
}
```

```
func TestPrintStringMap(t *testing.T) {  
    var h Headers = map[string]string{"Method": "POST"}  
    PrintStringMap(h)  
}
```



2

泛型限制

Limitation



无法限制必须组合某个结构体

结构体可以作为泛型参数，但是无法访问任何字段和方法。

由此带来的就是我们在 Go 内无法做到类似于别的语言用泛型表达“类型必须继承某个抽象类”的效果

换言之，我们无法限定类型必须要组合某个类型。

```
type User struct {  
    Name string  
}  
  
func (u User) GetName() string {  
    return u.Name  
}  
  
func PrintUser[T User](t T) {  
    // 这个 OK  
    fmt.Printf(format: "%v", t)  
    // 编译错误  
    // type bound for T has no method Name  
    fmt.Printf(format: "%v", t.Name)  
    // type bound for T has no method GetName  
    fmt.Printf(format: "%v", t.GetName())  
}
```



I 泛型简介——无法限制必须组合某个结构体

业务开发受限更多，尤其是希望在公司推行一些规范的时候，无法利用泛型来加强检测。例如要求所有的数据库实体都必须组合一个 BaseEntity，BaseEntity 里面有公司在数据库表创建方面的各种强制字段。

```
func Insert[T BaseEntity](t T) {  
    fmt.Printf(format: "insert %v", t)  
}
```

```
func TestInsert(t *testing.T) {  
    Insert(UserEntity{})  
}
```

```
./orm_test.go:108:8: UserEntity does not implement BaseEntity
```

```
Compilation finished with exit code 2
```



约束类型只能用于泛型

约束类型无法被用作类型声明，只能用于泛型。

这导致我们无法表达：我只接收特定几种类型作为输入的语义。

假如说我现在想要实现一个求和的函数，能够将 `int` 类型和 `float` 类型进行相加

```
type Number interface {
    ~int8 | ~int16 | ~int32 | ~int64 | ~int | ~float32 | ~float64
}

func Sum[T Number](ts ...T) T {
    var res T
    for _, t := range ts {
        res += t
    }
    return res
}
```

```
func TestSum(t *testing.T) {
    res := Sum[int](1, 2, 3, 4, 5)
    fmt.Printf(format: "%v", res)
}
```


泛型限制——约束类型只能用于泛型

Number 是一个泛型约束类型，所以无法被用作普通的类型，它只能出现在泛型里面。

所以右边的写法是错误的。

同样的，也无法声明一个 Number 变量

```
func SumV1(ns ...Number) Number {  
    }  
  
func TestSumV1(t *testing.T) {  
    res := SumV1(ns...: 1.1, 2, 3, 4, 5)  
    fmt.Printf(format: "%v", res)  
}
```


泛型限制——约束类型只能用于泛型

我们日常开发，或者说中间件开发的过程中，经常会碰到某个接口只接收特定几种类型的情况，目前的做法都是将参数声明成 `interface {}` 并且结合 `switch-case` 来处理，在最后肯定是在 `default` 里面进行错误输入处理。

这种样板代码将会长期存在

```
func SumV2(ns ...interface{}) float64 {  
    var res float64 = 0  
    for _, n := range ns {  
        switch val := n.(type) {  
        case float64:  
            res += val  
        case int:  
            res += float64(val)  
        default:  
            panic(v: "invalid types")  
        }  
    }  
    return res  
}
```

结构体无法声明泛型方法

接口或者结构体都可以是泛型的，但是它们不能声明泛型方法。

我个人认为这是最强的限制，没有之一。

它几乎断绝了所有的客户端类型的中间件利用泛型的道路

```
type Stream[T any] struct {  
    values []T  
}  
  
// 注意，这个方法在 Go 里面不被认为是泛型方法，因为它没有泛型参数  
// 虽然它的接收器是一个泛型  
func (s *Stream[T]) Filter(func(t T) bool) *Stream[T] {  
    return s  
}  
  
// 编译错误  
func (s *Stream[T]) Map[E any](func(t T) E) *Stream[T] {  
    return s  
}
```

泛型限制——结构体无法声明泛型方法

右边的写法全部无法通过编译。

```
type Cache interface {  
    Get[T any](key string) (T, error)  
}  
  
type Orm interface {  
    Create[T any](t T) error  
}  
  
type Config interface {  
    Get[T any](key string) (T, error)  
}  
  
type HttpClient interface {  
    Post[T any](endpoint string) (T, error)  
}
```

泛型限制——结构体无法声明泛型方法

如果硬要使用泛型，就需要将泛型声明在类型定义上，而后再在每次使用的时候都需要用具体类型来创建一个实例。

这种做法严重违背了单例设计原则。

客户端类型的中间件和我们日常开发最贴近，但是因为泛型的这一个限制，不能太期望这一类的客户端中间件会带来大的变更。

```
type CacheV1[T any] interface {  
    Get(key string) (T, error)  
}
```

```
var intCache CacheV1[int]
```

```
type OrmV1[T any] interface {  
    BatchInsert(ts ...T) error  
}
```

```
var userOrm OrmV1[User]
```

泛型限制——结构体无法声明泛型方法

对比 Java 的写法。

可以看到，Java 的泛型，同样是 `Stream<T>`，但是 Java 的泛型是可以进一步引入 `E` 的。

而 `Cache` 本身不是泛型的，但是可以额外声明泛型方法

```
class Stream<T> {  
    public <E> Stream<E> map() {  
        return new Stream<>();  
    }  
}  
  
class Cache {  
    public <T> T get(String key) {  
        throw new RuntimeException();  
    }  
}
```

```
String a = cache.get("aa");  
int b = cache.get("bbb");  
}  
  
public static final Cache cache = new Cache();
```

switch 无法操作类型参数

虽然在大多数场景下，使用了泛型参数，内部还要 switch 是一个很奇怪的用法。但是偶尔还是可能需要这么一个语法特性。

目前来说，Go 泛型支持不是很好。
switch 类型参数这个特性还处于 proposal 戒断

```
func Get[T any](key string) (T, error) {  
    var t T  
    switch t.(type) {  
    case int:  
        // 即便我们 switch 了类型，但是此时我们依旧不能对 t 进行赋值  
        t = 10  
        return t, nil  
    }  
}
```

```
func GetV1[T any](key string) (T, error) {  
    var t T  
    switch t.(type) {  
    case int:  
        // 即便我们 switch 了类型，但是此时我们依旧不能对 t 进行赋值  
        return 10, nil  
    }  
}
```

```
func GetV1[T any](key string) (T, error) {  
    // 无法将类型作为 switch 的对象  
    switch T {  
    case int:  
        return 10, nil  
    }  
}
```

泛型限制——switch 无法操作类型参数

注：直播的时候漏了这两种写法

```
func GetV4[T any](key string) (T, error) {
    var t T
    // 编译错误
    // cannot use type switch on type parameter value t (variable of
    switch val := t.(type) {
    case int:
        val = 10
        fmt.Printf(format: "t is %v and val is %d", t, val)
    }
    return t, nil
}

func GetV5[T any](key string) (T, error) {
    var t T
    var a interface{} = t
    switch val := a.(type) {
    case int:
        val = 10
        fmt.Printf(format: "t is %v and val is %d", t, val)
        // t is 0 and val is 10
        // 基础类型赋值是没有效果的
    }
    return t, nil
}
```


泛型限制——switch 无法操作类型参数

类似的需求还是只能通过指针来达成目标, 并且指针要赋值给一个 `interface {}` 类型才能进一步进行 `switch`

```
func GetV3[T any](key string) (T, error) {  
    var t T  
    var tp interface{} = &t  
    switch val := tp.(type) {  
    case *int:  
        *val = 10  
    }  
    return t, nil  
}
```


泛型限制——switch 无法操作类型参数

这一类的准备由非泛型转泛型的 API，就难以避免要写出这种恶心的代码。

同时，另外一个疑问是，如果已有的代码已经有了 GetInt, GetString, GetFloat32 等方法，那么还有必要引入泛型的版本？

```
func GetV3[T any](key string) (T, error) {  
    var t T  
    var tp interface{} = &t  
    switch val := tp.(type) {  
    case *int:  
        *val = GetInt()  
    case *string:  
        *val = GetString()  
        // other cases  
    }  
    return t, nil  
}
```



3

影响


Impact

数据结构与算法的类库

前述的这些限制对数据结构与算法的类库几乎没有影响。所以它们会迎来比较大的发展。

数据结构：例如 Map，Set 等。目前来看默认的 map 的核心缺陷在于 key 必须是 comparable 的，而在一些使用复杂结构体作为 key 的场景下，难以使用。以及 map 的变种，例如有序 map，最求高效率的小 map。


又如树形结构

 [paliimx/Data-Structures-and-Algorithms](#)
Data Structures and Algorithms implementation in Go

go golang algorithm algorithms datastructures data-structures algo

algorithms-and-data-structures

☆ 2.4k ● Go MIT license Updated on Oct 23, 2021

 [arnauddri/algorithms](#)
Algorithms & Data Structures in Go

☆ 1.8k ● Go Updated on Feb 5, 2021

 [0xAX/go-algorithms](#)
Algorithms and data structures for golang

data-structures algorithm go golang sort tree-structure hacktoberfe

☆ 1.6k ● Go Updated on Feb 20

 [timtadh/data-structures](#)



池

池一类的也可以迎来一定的改进。

比如典型的 `sync.Pool` 可以考虑使用泛型进行封装。

也可以设计通用的资源池。这一类的资源池可以满足：

- 资源一定时间不被使用就会被释放
- 控制住空闲资源的数量

连接池、对象池可以看做是这种通用资源池的特例

```
type Pool[T any] struct {  
    pool sync.Pool  
}  
  
func NewPool[T any](factory func() T) *Pool[T] {  
    return &Pool[T]{pool: sync.Pool{  
        New: func() interface{} {  
            return factory()  
        },  
    }}  
  
func (p *Pool[T]) Get() T {  
    res := p.pool.Get()  
    return res.(T)  
}
```

```
func TestPool_Get(t *testing.T) {  
    i := 0  
    p := NewPool[int](func() int {  
        i++  
        return i  
    })  
    fmt.Println(p.Get() + p.Get())  
}
```

缓存模式会有显著改进

缓存模式可以说将迎来显著地，用户体验上的改进。

核心在于早期我们设计缓存模式接口，如 write-through, read-through 的时候，要么直接使用 `interface {}`，用户则会陷入类型断言中。

要么使用具体类型，或者复制粘贴代码，或者使用代码生成策略。

但是因为 `T any` 不能被看成是 `interface {}`，所以虽然代码看起来是装饰器，但是 `ReadThroughCache` 在 Go 里面并不被认为实现了 `Cache` 接口。

```
type Cache interface {
    Get(key string) (interface{}, error)
    Set(key string, val interface{}) error
}

type ReadThroughCache[T any] struct {
    c      Cache
    readFunc func() (T, error)
}

func (c *ReadThroughCache[T]) Get(key string) (T, error) {
    res, err := c.c.Get(key)
    if err == nil {
        return res.(T), nil
    }
    val, err := c.readFunc()
    if err != nil {
        return val, err
    }
    _ = c.c.Set(key, val)
    return val, err
}
```

影响——缓存模式

但是因为 `T any` 不能被看成是 `interface {}`，所以虽然代码看起来是装饰器，但是 `ReadThroughCache` 在 Go 里面并不被认为实现了 `Cache` 接口。

即便是用 `interface {}` 来作为具体类型来实例化，还是会编译错误

```
type Cache interface {
    Get(key string) (interface{}, error)
    Set(key string, val interface{}) error
}

var a Cache = &ReadThroughCache[interface{}]()

type ReadThroughCache[T any] struct {
    c      Cache
    readFunc func() (T, error)
}

func (c *ReadThroughCache[T]) Get(key string) (T, error) {...}

func (c *ReadThroughCache[T]) Set(key string, val T) error {...}
```


基于 Builder 模式的库

Builder 模式在很大程度上能够避开之前限制。

例如我们可以利用 Builder 模式来设计更为友好的 http 客户端接口。

```
type HttpGetter[T any] struct {
    req *http.Request
}

func (get *HttpGetter[T]) Host(host string) *HttpGetter[T] {
    get.req.Host = host
    return get
}

func (get *HttpGetter[T]) Path(path string) *HttpGetter[T] {
    get.req.RequestURI = path
    return get
}

func (get *HttpGetter[T]) Get() (T, error) {
    var t T
    client := http.Client{}
    resp, err := client.Do(get.req)
    if err != nil {
        return t, err
    }
    bytes, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return t, err
    }
    err = json.Unmarshal(bytes, &t)
    return t, err
}
```

基于 Builder 模式的库

ORM 可以考虑走利用 Builder 模式的路子。

通过 QueryBuilder 来构建 SQL，最终发起查询，并且得到结果。

这方面，SELECT 语句比较合适，因为 SELECT 语句更加复杂。

```
func (s *Selector[R]) Select(columns ...Selectable) *Selector[R] {
    s.columns = columns
    return s
}

// From specifies the table which must be pointer of structure
func (s *Selector[R]) From(table interface{}) *Selector[R] {
    s.table = table
    return s
}

// Where accepts predicates
func (s *Selector[R]) Where(predicates ...Predicate) *Selector[R] {
    s.where = predicates
    return s
}

func (s *Selector[R]) Get() (*R, error) {
    query, err := s.s.Build()
    if err != nil {
        return nil, err
    }
    var r R
    rp := &r
    // 这是基于 sqlx 的实现
    err = s.db.sqlxDB.Get(rp, query.SQL, query.Args...)
    return rp, err
}
```

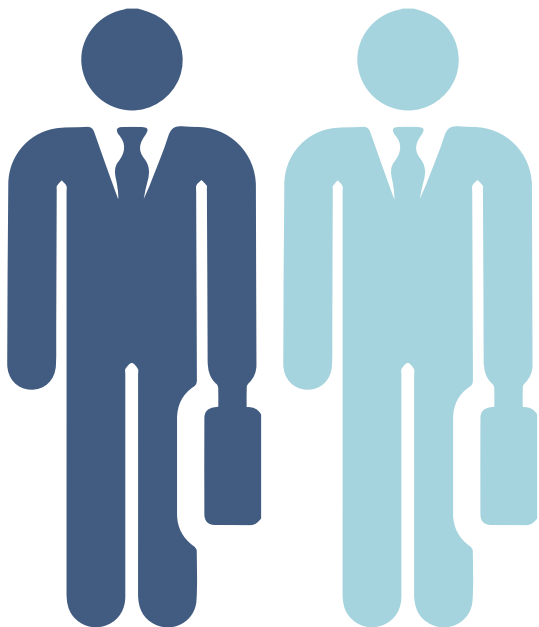



4

总结

Summary





generic, not general



适合函数式风格和过程式编程

也就是不适合面向对象风格编程，至少效果差很多



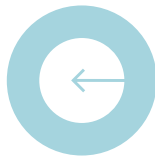
对中间件影响有限

对 API 设计影响较大，但是对实现影响很有限



对业务开发影响不大

业务开发普遍是针对业务对象，因而天然不适用泛型



不必刻意追求泛型

类库还有待发展，限制还过多





感谢 & Q&A

大明