Go泛型前瞻

它带来了什么？

内容

- 为什么需要泛型
- 泛型简介
- 新的编程思维
- 泛型的限制

资料

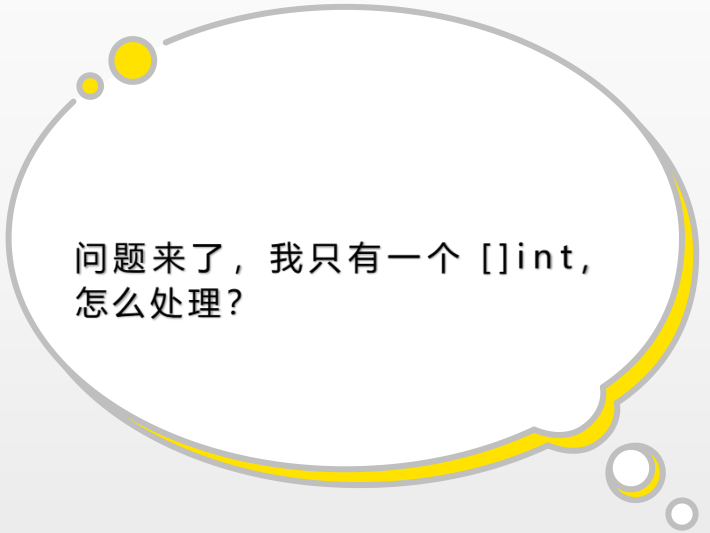
- https://go2goplay.golang.org/ Go2线上实验网站（官方维护）
- https://github.com/gotomicro/ego-kit 实例代码
- https://www.yuque.com/docs/share/9589face-4791-4cae-b2a2-67351cc90d5b?# 《如何在 Goland 里体验 go 范型》
- https://www.yuque.com/docs/share/4f7e0285-5b23-4844-a732-702e56030a86?# 《快速使用泛型》

GO 为什么需要泛型

- 基本类型的数学运算
- 通用的切片、map 逻辑
- 样板代码和生成代码
- 遍布项目的 interface{} 与 类型转换
- …

GO 为什么需要泛型

```go
func Sum(values []int64) int64 {
    var res int64 = 0
    for _, value := range values {
        res += value
    }
    return res
}
```



GO 为什么需要泛型

```go
func SumInt(values []int) int {
    res := 0
    for _, value := range values {
        res += value
    }
    return res
}
```

```go
func TestSum(t *testing.T) {
    values := []int {1, 2, 3, 4}

    valuesInt64 := make([]int64, 0, len(values))
    for _, value := range values {
        valuesInt64 = append(valuesInt64, int64(value))
    }
    sum := Sum(valuesInt64)
    print(sum)
}
```

```go
type Addable interface {
    Add(val Addable) Addable
}

type addInt int

func (a addInt) Add(val Addable) Addable {
    return a + val.(addInt)
}
```

GO 泛型简介 —— 结构体和指针作为类型参数

```
13
14 func Print[T any](s []T) {
15         for _, v := range s {
16                 fmt.Print(v)
17         }
18 }
19
20 type User struct {
21     Name string
22 }
23
24 func main() {
25         Print([]string{"Hello, ", "playground\n"})
26         u := &User{
27             Name: "Tom",
28         }
29         Print[*User]([]*User{u})
30 }
31
32
```

GO 泛型简介 —— 定义约束

```
13
14  type Numeric interface {
15      type int, int64, int32
16  }
17
18  func Sum[T Numeric](values []T) T {
19      var res T
20      for _, val := range values {
21          res = res + val
22      }
23      return res
24  }
```

GO 泛型简介 —— 加强约束

```go
type Request interface {
    Value(key string)
}

type RpcRequest interface {
    Request
    ServiceName() string
}

type Filter[R Request] func(req R)

type RpcFilter[R RpcRequest] func(req R)

func DubboFilter(req DubboRpcRequest) {
}

var _ Filter[DubboRpcRequest] = DubboFilter
var _ RpcFilter[DubboRpcRequest] = DubboFilter
```

GO 泛型——新的编程思维

- 数字类型的数学运算
- 内置类型的辅助方法（slice, map）
- 集合类型
- Steam API 与 Map Reduce
- 设计模式
- DAO 编程和调用第三方
- 其它

GO 泛型—— 内置类型的辅助方法

- slice
  - Add, Delete, Concat等结构修改操作
  - Max, Min 等查找类操作
- map
  - PutIfAbsent
  - GetOrDefault
  - Keys 和 Values
  - Merge

GO 泛型—— map 辅助方法

```go
func PutIfAbsent[K any, V any](m map[K]V, key K, val V) (V, bool) {
    old, ok := m[key]
    if ok {
        return old, false
    }
    m[key] = val
    return val, true
}
```

泛型限制—— map 的 key 不能是 any

```go
func PutIfAbsent[K comparable, V any](m map[K]V, key K, val V) (V, bool) {
    old, ok := m[key]
    if ok {
        return old, false
    }
    m[key] = val
    return val, true
}
```

```
type checking failed for maps
maps.go2:8:38: incomparable map key type K (missing comparable constraint)
```

GO 泛型—— slice 辅助方法

```go
func Add[T any] (values []T, val T, index int) ([]T, error) {
    if index < 0 || index > len(values)-1 {
        return nil, errors.New( text: "index out of range")
    }
    res := make([]T, 0, len(values)+1)
    res = append(res, values[:index]...)
    res = append(res, val)
    res = append(res, values[index:]...)

    return res, nil
}
```

```go
type Slice[T] []T

func (s Slice[T]) Add(val T, index int) (Slice[T], error) {
    //
}
```

基于泛型的集合类型

- Golang 内置类型种类不够丰富，无法满足需要
  - Set，有序Set 和 有序 Map
  - 特殊类型作为 Map 和 Set 的 Key （基于 map 的实现都受制于此）
  - 链表
  - 堆、栈
  - 队列、优先级队列、并发阻塞队列
- 更加友好的 API
- 已有的集合框架，都是使用 interface{} 作为类型，需要类型转换
- 并发安全集合类型不足

基于泛型的集合类型—— List

```
type List[E any] interface {
    Len() int
    Empty() bool
    Clear()
    Add(val E, index int)
    Set(val E, index int)
    Remove(index int)
    Append(val E)
    ForEach(f func(e E))
    Get(index int) (E, bool)
    Traverse() []E
}
```

```
type Queue[E any] interface {
    Front() E
    Back() E
    Push(e E)
    Pop() E
}

type Stack[E any] interface {
    Push(e E)
    Pop() E
    Peek() E
}
```

基于泛型的集合类型—— List

```go
// node of linkedlist
type linkedListNode[E any] struct {
    Val        E
    Prev, Next *linkedListNode[E]
}

// linkedlist is a circular doubly linked list
// linkedlist has two dummy nodes, one is head dummyNode, the other is tail dummyNode
type linkedList[E any] struct {
    head, tail *linkedListNode[E]
    //length of linkedlist
    len int
}

func CreateLinkedList[E any]() *linkedList[E] {
    h, t := &linkedListNode[E]{}, &linkedListNode[E]{}
    h.Prev, h.Next, t.Prev, t.Next = t, t, h, h
    return &linkedList[E]{
        head: h,
        tail: t,
        len: 0,
    }
}
```

```go
type stack[E any] struct {
    l *linkedList[E]
}
```

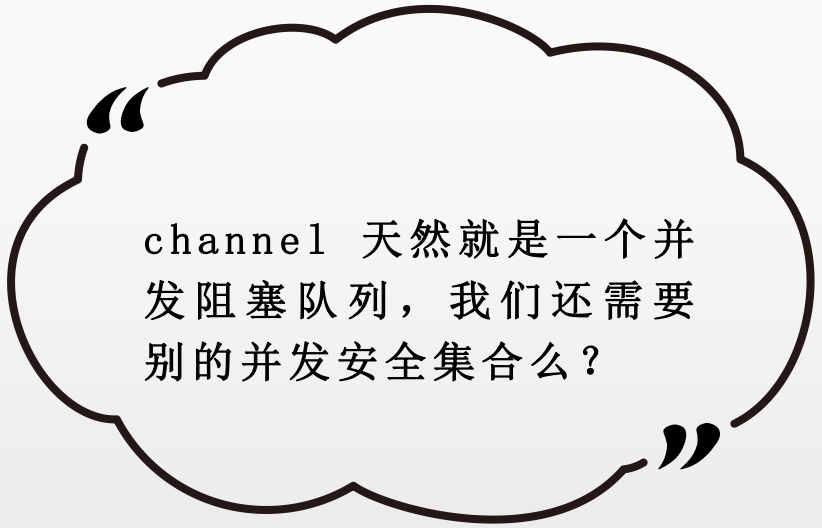
```go
type queue[E any] struct {
    l *linkedList[E]
}
```

基于泛型的集合类型——Map 和 Set

```go
type Map[K any, V any] interface {
    Put(key K, value V)
    PutIfAbsent(key K, value V) (V, bool)

    GetOrDefault(key K, defProvider func() V) V
    Get(key K) (V, bool)

    Delete(key K) (V, bool)
}

type Hashable interface {
    HashCode() int
}

func NewMap[K any, V any] (withHashCode bool) Map[K, V] {
    return nil
}
```

1. 普通类型，直接基于内置 map 实现
2. 复杂类型，需要实现 HashCode 方法

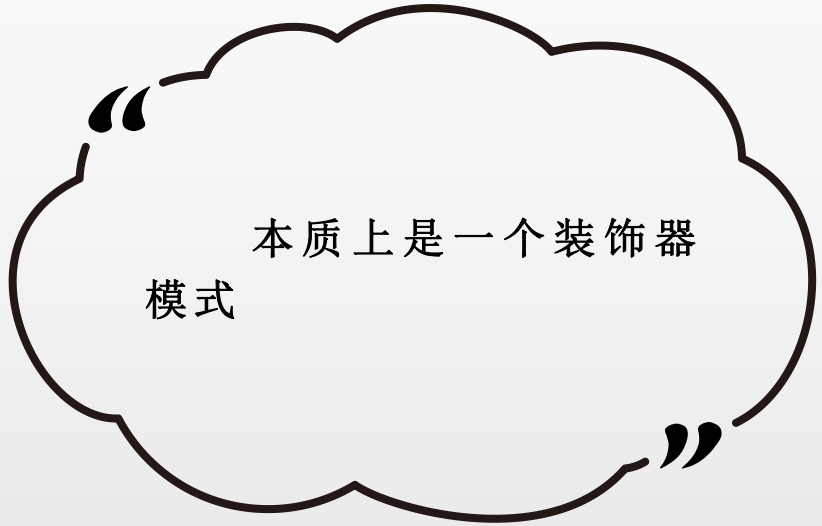
并发安全集合 —— channel 够么?



- 场景1：我们需要一个并发安全的优先级队列。比如说生产者生成不同时间执行的任务，消费者取出任务执行；
- 场景2：遍历队列中的元素。比如说从中找到某个尚未执行的任务；
- 场景3：随机访问队列中的元素

线程安全集合 —— 基于锁的简单实现

```go
type SafeList[E any] struct {
    list List[E]
    mutex sync.RWMutex
}

func (s *SafeList[E]) Len() int {
    s.mutex.RLock()
    defer s.mutex.RUnlock()
    return s.list.Len()
}

func (s *SafeList[E]) Empty() bool {
    s.mutex.RLock()
    defer s.mutex.RUnlock()
    return s.list.Empty()
}
```



Stream API

- 目标：操作切片、数组、map 和集合类型
- 支持复杂的过滤、查找、转化、拼接、聚合运算
- 声明式 API，提高编程效率和代码可读性
- 延迟求值
- 并行 Stream
- 不可变性：Stream 本身不可变（考虑中）

Stream API

```
type Stream[E any] interface {

    Distinct(c comparator[E]) Stream[E]
    Sort(c comparator[E]) Stream[E]
    Limit(offset int, limit int) Stream[E]
    Skip(num int) Stream[E]
    Concat(tail Stream[E]) Stream[E]
    ConcatArray(tail []E) Stream[E]

    ForEach(f func(e E)) Stream[E]
    ToSlice() []E

    Max(c comparator[E]) (E, error)
    Min(c comparator[E]) (E, error)
    Count() int

    AnyMatch(m match[E]) bool
    AllMatch(m match[E]) bool
    NoneMatch(m match[E]) bool

    OrElse(e E) Stream[E]
    Filter(m match[E]) Stream[E]
    FindFirst(m match[E]) (E, error)
    FindLast(m match[E]) (E, error)
    FindAny(m match[E]) (E, error)
    FindNth(n int, m match[E]) (E, error)
}
```

- 查找类 API
- 结构调整类 API
- 聚合类 API
- 迭代类 API

Stream API —— 例子

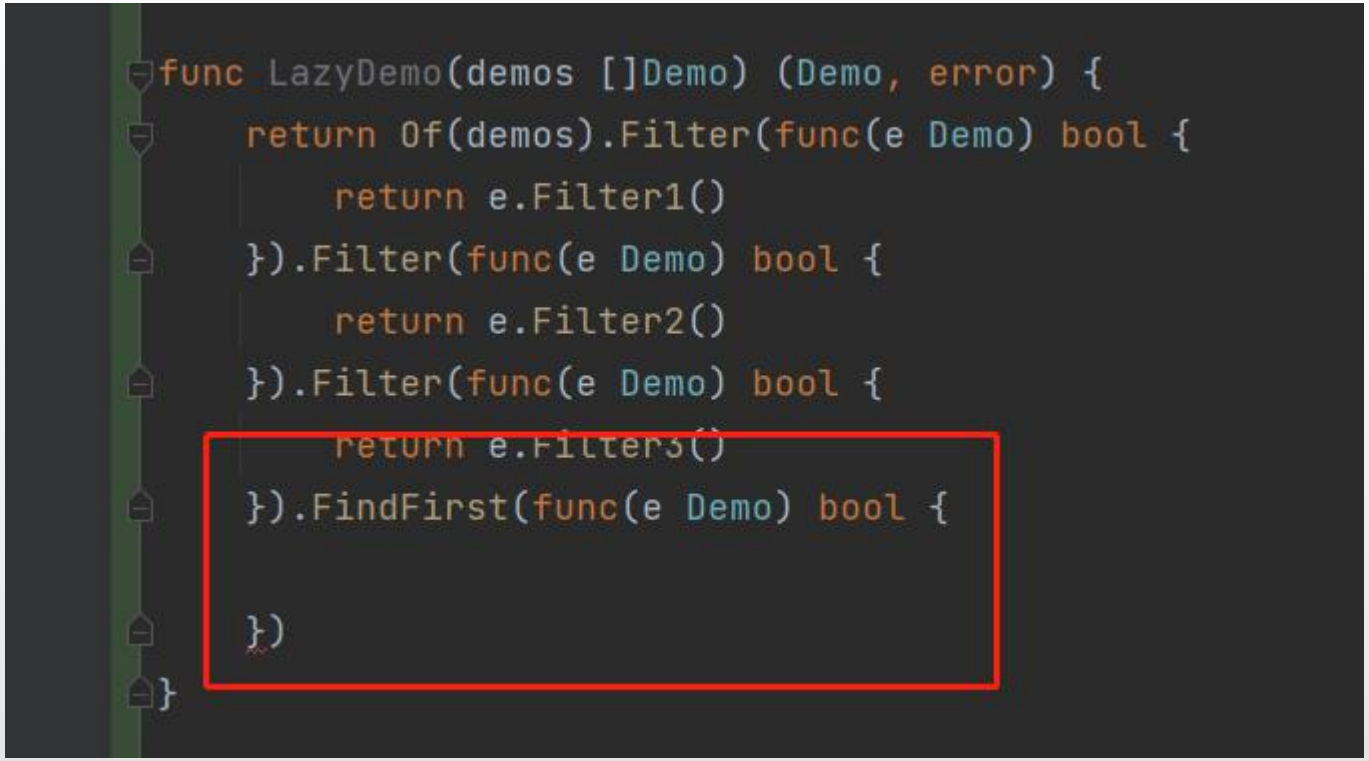
```go
func FindOrder() []Order {
    // 假如这里我们拿到了所有的order
    orders := []Order{}

    res := make([]Order, 0, len(orders))
    for _, order := range orders {
        // 准备支付
        if order.Payable() {
            res = append(res, order)
        }
    }
    sort.Sort(OrderSlice(res))
    return res
}
```

```go
type OrderSlice []Order

func (o OrderSlice) Len() int {
    panic( v: "implement me")
}

func (o OrderSlice) Less(i, j int) bool {
    panic( v: "implement me")
}

func (o OrderSlice) Swap(i, j int) {
    panic( v: "implement me")
}
```

Stream API —— 例子

```
func FindOrderV2() []Order {
    orders := []Order{}

    return Of(orders).Filter(func(e Order) bool {
        return e.Payable()
    }).Sort(func(e1, e2 Order) int {
        return int(e1.CreateTime - e2.CreateTime)
    }).ToSlice()
}
```

Stream API —— 延迟求值



```
func LazyDemo(demos []Demo) (Demo, error) {
    return Of(demos).Filter(func(e Demo) bool {
        return e.Filter1()
    }).Filter(func(e Demo) bool {
        return e.Filter2()
    }).Filter(func(e Demo) bool {
        return e.Filter3()
    }).FindFirst(func(e Demo) bool {

    })
}
```

Stream API —— 延迟求值

- 中间操作：不会执行，只会暂存，如`Filter`操作；
- 终结操作：执行。在执行前会执行所有的中间操作，最终生成结果；

```go
func (s *SequentialStream[E]) FindNth(n int, m match[E]) (E, error) {
    filters := append(s.filters, m)
    cnt := 0
    for _, e := range s.eles {
        if s.matchAll(filters, e) {
            if cnt == n {
                return e, nil
            }
            cnt ++
        }
    }
    return s.def, ErrNotFound
}
```

Stream API —— 并行 Stream

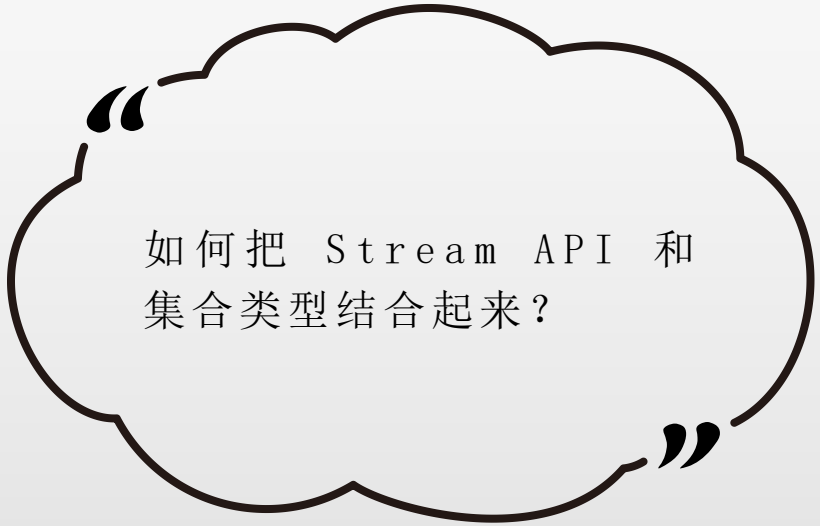
```go
func (p *ParallelStream[E]) FindAny(m match[E]) (E, error) {
    filters := append(p.filters, m)
    ch := make(chan interface{})
    wg := sync.WaitGroup{}
    for _, e := range p.eles {
        wg.Add( delta: 1)
        go func(ele E) {
            if p.matchAll(filters, ele) {
                ch <- ele
            }
            wg.Done()
        }(e)
    }

    go func() {
        wg.Wait()
        ch <- ErrNotFound
    }()

    select {
    case data := <- ch:
        if res, ok := data.(E); ok {
            return res, nil
        }
        return p.def, ErrNotFound
    }
}
```

```go
func (p *ParallelStream[E]) ForEach(f func(e E)) {
    wg := sync.WaitGroup{}
    wg.Add(len(p.eles))
    for _, e := range p.eles {
        go func(ele E) {
            if p.matchAll(p.filters, ele) {
                f(ele)
            }
            wg.Done()
        }(e)
    }
    wg.Wait()
}
```

Stream API —— 并行 Stream

```go
func (p *ParallelStream[E]) FindAny(m match[E]) (E, error) {
    filters := append(p.filters, m)
    ch := make(chan interface{})
    wg := sync.WaitGroup{}
    for _, e := range p.eles {
        wg.Add( delta: 1)
        go func(ele E) {
            if p.matchAll(filters, ele) {
                ch <- ele
            }
            wg.Done()
        }(e)
    }

    go func() {
        wg.Wait()
        ch <- ErrNotFound
    }()

    select {
    case data := <- ch:
        if res, ok := data.(E); ok {
            return res, nil
        }
        return p.def, ErrNotFound
    }
}
```

- 如何中断执行？
- 如何控制 goroutine 的数量？



Map Reduce API

- Map方法：将一个切片转化为另外一个切片

```go
func MapSlice[S, T](s []S, mapper Mapper[S, T]) []T {
    res := make([]T, 0, len(s))
    for _, ele := range s {
        res = append(res, mapper(ele))
    }
    return res
}
```

- Reduce方法：将一个切片转化为一个特定的值

```go
func ReduceSlice[E, R](slice []E, base R, accumulator Accumulator) R {
    res := base
    for _, ele := range slice {
        res = accumulator(res, ele)
    }
    return res
}
```

泛型限制——结构体方法不能额外有泛型参数

```
41
```

```
type checking failed for main
prog.go2:15:25: methods cannot have type parameters


Go build failed.
```

```go
type Demo[T any] struct {
    t T
}

func (d Demo[T]) Process[K any](key K) T {
    return d.t
}
```

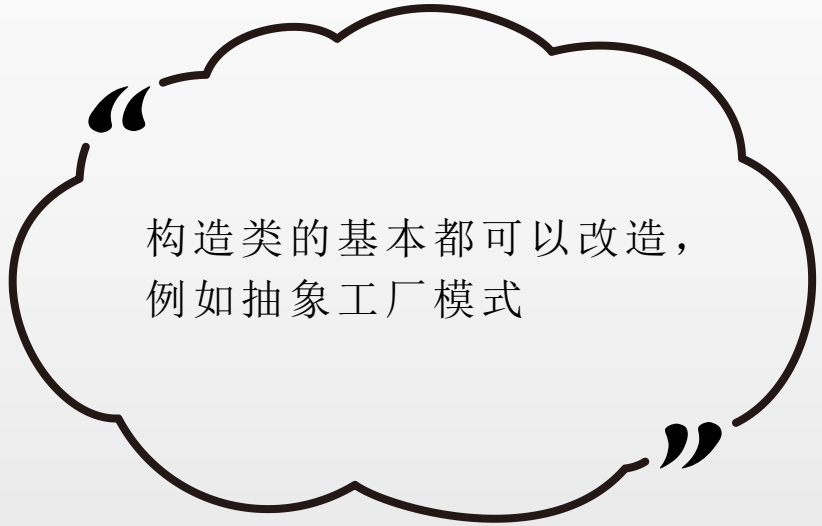
```go
type Stream[E any] interface {
    Map[T any](func(e E) T)Stream[T]
    Reduce[T any](base T, opr func(acc T, e E) T) T
}
```

Steam 支持 Map Reduce API ？

```
    MapToInt(m func(e E) int) Stream[int]
    MapToString(m func(e E) string) Stream[string]
    MapToInterface(m func(e E)interface{}) Stream[interface{}]

    ReduceToInt(base int, r func(acc int, e E) int) int
    ReduceToString(base string, r func(acc string, e E) string) string
    ReduceToInterface(base interface{}, r func(acc interface{}, e E)) interface{}
}
```

基于泛型的设计模式 —— Builder 模式

```go
type Builder[T Animal] interface {
    Part1() Builder[T]
    Part2() Builder[T]
    Part3() Builder[T]
    Build() T
}

var _ Builder[Cat] = &CatBuilder{}
type Cat struct {
}

type CatBuilder struct {

}

func (c *CatBuilder) Part1() Builder[Cat] {
    panic( v: "implement me")
}
```

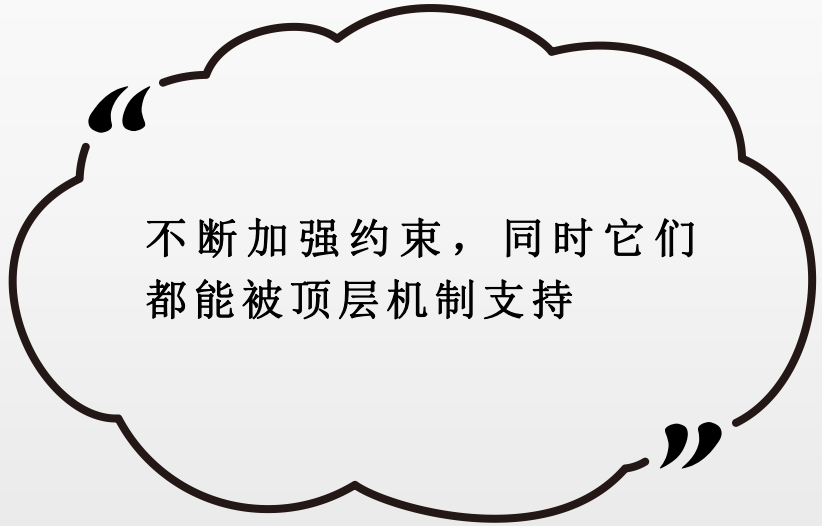


基于泛型的设计模式—— 责任链

```go
type Filter[R Request] func(req R)

type Chain[r Request] func(next Filter[R]) Filter[R]

type FilterChainBuilder[R Request] struct {
    chains []Chain[R]
}

func (f FilterChainBuilder[R]) Add(chain Chain[R]){
    f.chains = append(f.chains, chain)
}

func (f *FilterChainBuilder[R]) Root() Filter[R] {
    root := f.chains[0]( next: nil)
    for i := 1; i < len(f.chains); i++ {
        root = f.chains[i](root)
    }
    return root
}
```

```go
type HttpRequest struct {

}

func (h *HttpRequest) Value(key string) {
    panic( v: "implement me")
}

type RpcRequest struct {

}

func (r *RpcRequest) Value(key string) {
    panic( v: "implement me")
}
```

```go
type Request interface {
    Value(key string)
}

type RpcRequest interface {
    Request
    ServiceName() string
}

type Filter[R Request] func(req R)

type RpcFilter[R RpcRequest] func(req R)

func DubboFilter(req DubboRpcRequest) {
}

var _ Filter[DubboRpcRequest] = DubboFilter
var _ RpcFilter[DubboRpcRequest] = DubboFilter
```



基于泛型的设计模式

- 用泛型设计机制
- 加强约束具体化机制
- 具体类型表达策略
- 具体类型策略有不同实现

基于泛型的设计模式—— Proxy 模式

```go
func New[T any](handler InvocationHandler[T], opts ...proxyOpt[T]) *T {
    obj := new(T)
    objValue := reflect.ValueOf(obj)
    objElem := objValue.Elem()
    num := objElem.NumField()

    for i := 0; i < num; i++ {...}
    return obj
}
```

基于泛型的设计模式—— Proxy 模式

```go
func TestNewProxy(t *testing.T) {
    ho := New[HelloObj](&LogInvocationHandler[HelloObj]{})
    err := ho.Hello("Tom")
    println(err.Error())
}

type LogInvocationHandler[T any] struct {

}

func (l *LogInvocationHandler[T]) Handle(target *T, inv *Invocation) []reflect.Value {
    println( args...: "before")
    println("running " + inv.Method.Name)
    // res := inv.Invoke()
    println( args...: "after")
    return []reflect.Value{reflect.ValueOf(errors.New( text: "this is error"))}
}

type HelloObj struct {
    Hello func(name string) error
}
```

数据库编程

- 延续已有的 ORM 设计
- 采用 DAO 设计

数据库编程—— ORM 泛型

```
)

type Orm[T any] interface {
    Insert(ctx context.Context, t *T) (int64, error)
    Update(ctx context.Context, t *T, cols ...string) (int64, error)
    Delete(ctx context.Context, t *T) (int64, error)
    Find(ctx context.Context, t *T) ([]*T, error)
    FindOne(ctx context.Context, t *T) ([]*T, error)
    Get(ctx context.Context, id int64) (*T, error)
}

type ormBasedOnOrm struct {
    o orm.Ormer        非泛型的原始实现
}
```

只能定义有限的方法，无法无缝衔接所有用户的需求；
实现简单，过渡自然，形如装饰器模式——一层泛型的装饰

数据库编程 —— DAO 设计

```
148
149  type UserDAO struct{
150      BaseDAO[User]
151      FindByEmail func(ctx context.Context, email string) (*User, error) `sql:"SELECT * FROM USER WHERE EMAIL = ?"`
152  }
153
154  type BaseDAO[T any] struct {
155      Insert func(ctx context.Context, t *T) (int64, error)
156      Update func(ctx context.Context, t *T, cols ...string) (int64, error)
157      Delete func(ctx context.Context, t *T) (int64, error)
158      Find func(ctx context.Context, t *T) ([]*T, error)
159      FindOne func(ctx context.Context, t *T) ([]*T, error)
160      Get func(ctx context.Context, id int64) (*T, error)
161  }
```

泛型限制——泛型组合

```
13
14  type Sub[T any] struct {
15      Name string
16  }
17
18  func (s Sub[T]) Hello() {
19      println( args...: "hello, sub")
20  }
21
22  type Parent[T any] struct {
23      Sub[T]
24  }
25
26  func TestHello(t *testing.T) {
27      p := Parent[string]{
28          Sub[string]{},
29      }
30      p.Hello()
31  }
```

```
//line slice.go2:17
type instantiate○○Parent○string struct {
//line slice_test.go2:22
 instantiate○○Sub○string
//line slice_test.go2:24
}
//line slice_test.go2:24
type instantiate○○Sub○string struct {
//line slice_test.go2:15
 Name string
}

func (s instantiate○○Sub○string,) Hello() {
    println("hello, sub")
}
```

```
38 }
39

# _/tmp/sandbox3786386347
./prog.go2:36: p.Sub undefined (type instantiate○○Parent○string has no field or method Sub)

Go build failed.
```

数据库编程 —— DAO 设计

```go
type UserDAO struct{
    Base *BaseDAO[User]
    FindByEmail func(ctx context.Context, email string) (*User, error) `sql:"SELECT * FROM USER WHERE EMAIL = ?"`
}

type BaseDAO[T any] struct {
    Insert func(ctx context.Context, t *T) (int64, error)
    Update func(ctx context.Context, t *T, cols ...string) (int64, error)
    Delete func(ctx context.Context, t *T) (int64, error)
    Find func(ctx context.Context, t *T) ([]*T, error)
    FindOne func(ctx context.Context, t *T) ([]*T, error)
    Get func(ctx context.Context, id int64) (*T, error)
}
```

没太大意义，和 ORM 差不多

第三方调用，涉及序列化与反序列化过程

```
43  +
44  + // Post Send a POST request and try to give its result value
45  + func (c *Client) Post(value interface{}, path string, body interface{}, opts ...Bee
46  +       req := Post(c.Endpoint + path)
47  +       c.customReq(req, opts)
48  +       if body != nil {
49  +               req = req.Body(body)
50  +       }
51  +       return c.handleResponse(value, req)
52  + }
```

第三方调用，涉及序列化与反序列化过程

```go
type Client[T any] struct {

}

func (c *Client[T]) Post(path string, body interface{}) (T, error){
    // TODO
}
```

其它 —— Pair, Triplet

```
16
17  type Pair[K any, V any] struct {
18      Key K
19      Val V
20  }
21
22  type Triplet[V1 any, V2 any, V3 any] struct {
23      Val1 V1
24      Val2 V2
25      Val3 V3
26  }
```

其它 —— 序列化与反序列化

```go
func JsonUnmarshal[T any] (data []byte) (*T, error) {
    t := new(T)
    err := json.Unmarshal(data, t)
    return t, err
}

type User struct {
    Name string `json:"name"`
}

func main()  {
    data := []byte(`
{
    "name": "Tom"
}
`)
    u, _ := JsonUnmarshal[User](data)
    println(u.Name)
}
```

泛型限制——结构体方法不能额外有泛型参数

```go
type Demo[T any] struct {
    t T
}

func (d Demo[T]) Process[K any](key K) T {
    return d.t
}
```

```
41

type checking failed for main
prog.go2:15:25: methods cannot have type parameters

Go build failed.
```

```go
type Stream[E any] interface {
    Map[T any](func(e E) T)Stream[T]
    Reduce[T any](base T, opr func(acc T, e E) T) T
}
```

```go
type Container struct {

}

func (c *Container) Get[T any](key string) (t T, err error){
    return
}
```

泛型限制——结构体方法不能额外有泛型参数

```
71
72      type Container struct {
73          m map[string]interface{}
74      }
75
76      func (c *Container) Get[T any](key string) (T, error){
77          val, ok := m[key]
78          if ok {
79              return val.(T), nil
80          }
81          var t T
82          return t, errors.New( text: "not found")
83      }
```

泛型限制——泛型组合

```go
type Sub[T any] struct {
    Name string
}

func (s Sub[T]) Hello() {
    println( args...: "hello, sub")
}

type Parent[T any] struct {
    Sub[T]
}

func TestHello(t *testing.T) {
    p := Parent[string]{
        Sub[string]{},
    }
    p.Hello()
}
```

```go
//line slice.go2:17
type instantiate○○Parent○string struct {
//line slice_test.go2:22
 instantiate○○Sub○string
//line slice_test.go2:24
}
//line slice_test.go2:24
type instantiate○○Sub○string struct {
//line slice_test.go2:15
 Name string
}

func (s instantiate○○Sub○string,) Hello() {
    println("hello, sub")
}
```

```
38 }
39

# _/tmp/sandbox3786386347
./prog.go2:36: p.Sub undefined (type instantiate○○Parent○string has no field or method Sub)

Go build failed.
```

泛型限制—— map 的 key 不能是 any

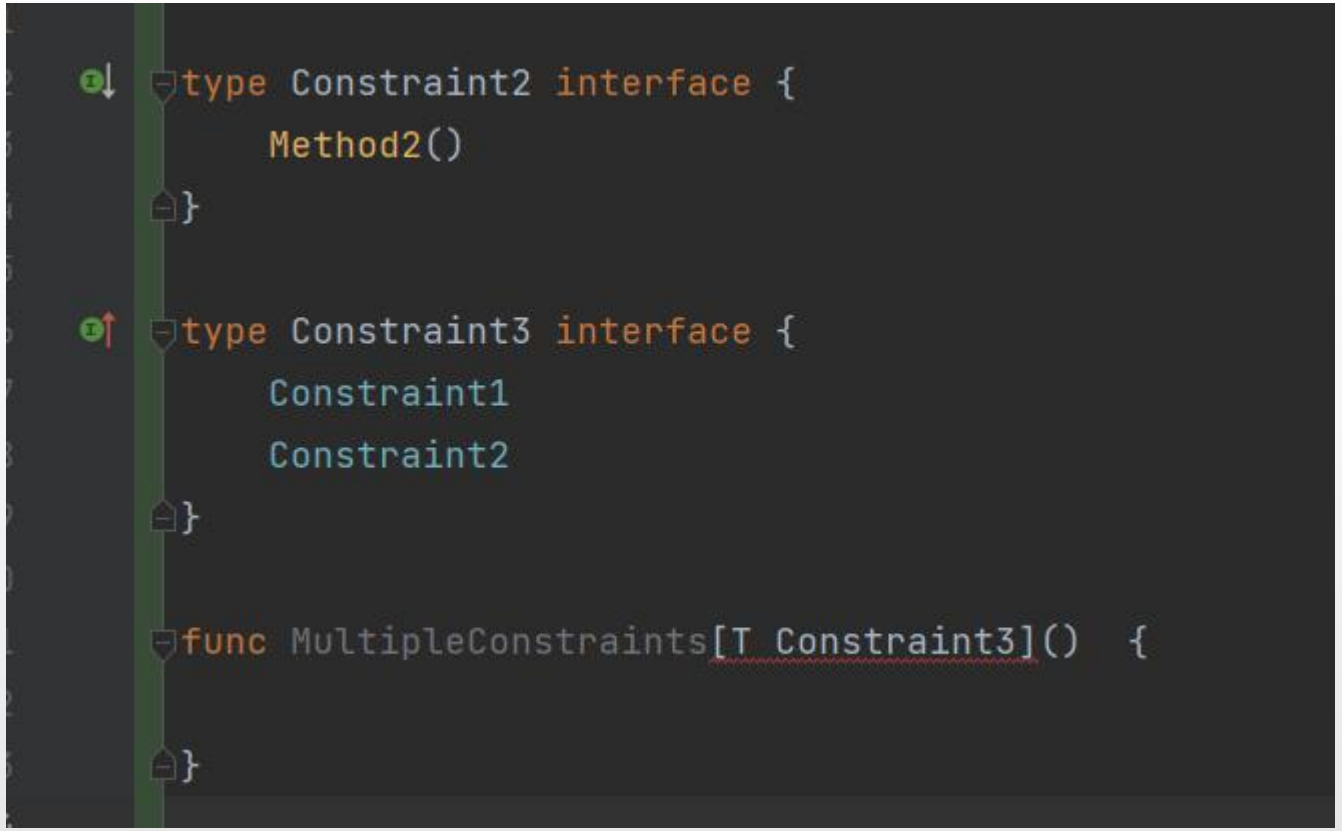
```go
func PutIfAbsent[K comparable, V any](m map[K]V, key K, val V) (V, bool) {
    old, ok := m[key]
    if ok {
        return old, false
    }
    m[key] = val
    return val, true
}
```

```
type checking failed for maps
maps.go2:8:38: incomparable map key type K (missing comparable constraint)
```

泛型限制—— 无法同时声明多个约束

```go
type Constraint1 interface {
    Method1()
}

type Constraint2 interface {
    Method2()
}
func MultipleConstraints[T Constraint1 & Constraint2]()  {

}
```

泛型限制—— 无法同时声明多个约束



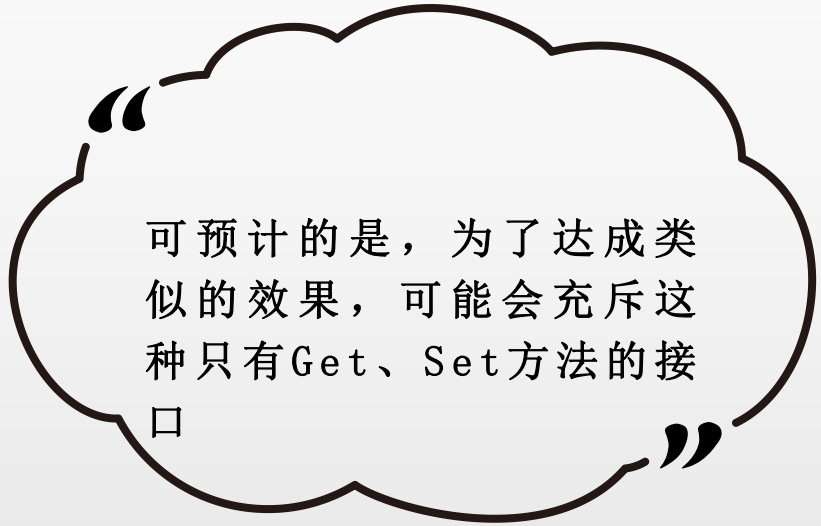
泛型限制—— 约束必须是接口

```go
type Constraint struct {
    Name string
}

func Generic[T Constraint](t T) {
    println(t.Name)
}
```

这是因为 GOLANG 是没有继承的；
也因此限制了所有需要访问字段的泛型用法；

泛型限制—— 约束必须是接口

```
type Getter interface {
    GetName() string
}

type Constraint struct {
    Name string
}

func Generic[T Getter](t T) {
    println(t.GetName())
}
```



总结

- 泛型很好用，但是限制也很强
- 泛型**能极大提升研发效率和代码内聚**
- 泛型出来之后，标准库和第三方库都会引来一波修改浪潮
- 非泛型接口转为泛型接口会带来兼容性问题

扫码提问