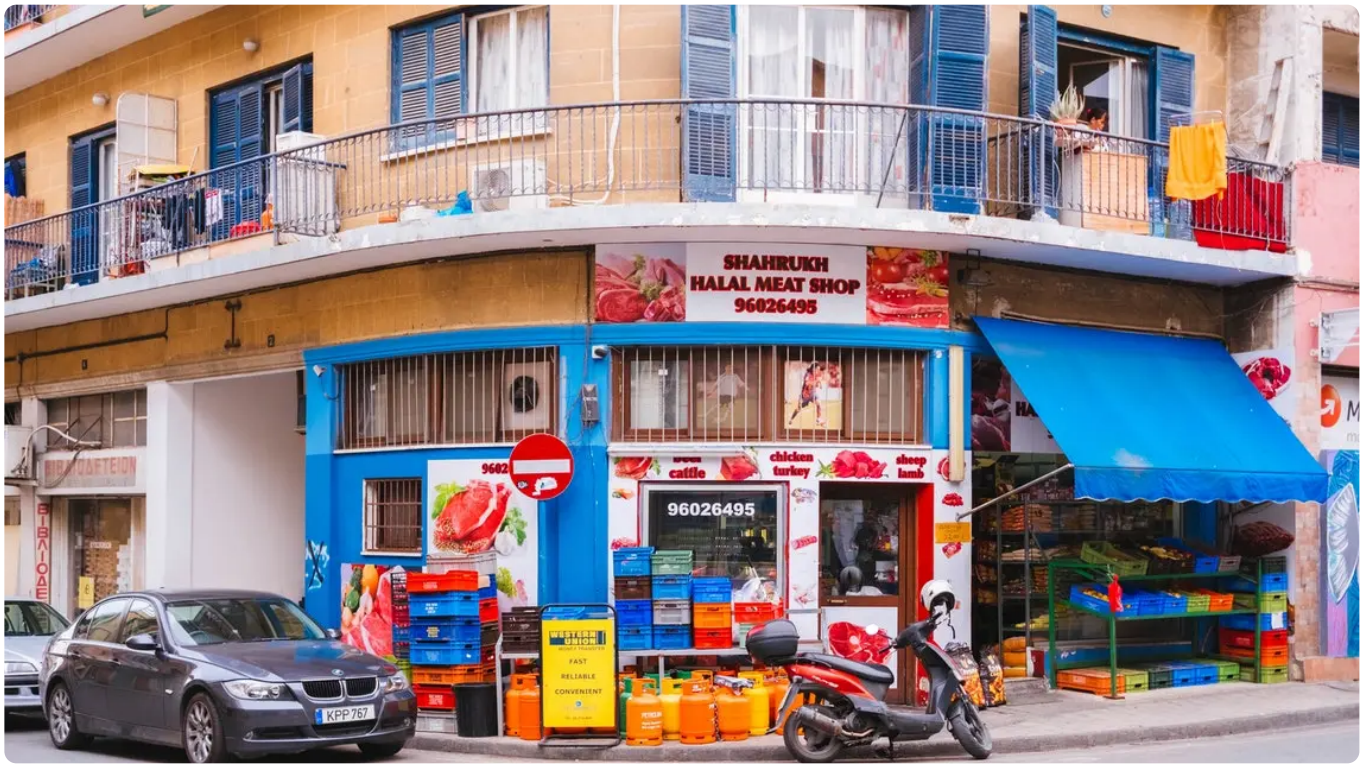
大咖助阵｜大明：Go泛型，泛了，但没有完全泛

2022-02-11 大明

《Tony Bai · Go语言第一课》 课程介绍 >



讲述：大荣

时长 13:28 大小 12.34M

你好，我是大明，一个专注于中间件研发的开源爱好者。

我们都知道，Go 泛型已经日渐成熟，距离发布正式版本已经不远了。目前已经有很多的开发者开始探索泛型会对 Go 编程带来什么影响。比如说，目前我们比较肯定的是泛型能够解决这一类的痛点：

数学计算：写出通用的方法来操作 `int`、`float` 类型；

集合类型：例如用泛型来写堆、栈、队列等。虽然大部分类似的需求可以通过 `sli` `l` `channel` 来解决，但是始终有一些情况难以避免要设计特殊的集合类型，如有序 `Set` 和优先级队列；

`slice` 和 `map` 的辅助方法：典型的如 map-reduce API；

但是至今还是没有人讨论 Go 泛型的限制，以及这些限制会如何影响我们解决问题。

所以今天我将重点讨论这个问题，不过因为目前我主要是在设计和开发中间件，所以我会侧重于中间件来进行讨论，当然也会涉及业务开发的内容。你可以结合自己了解的 Go 泛型和现有的编程模式来学习这些限制，从而在将来 Go 泛型正式发布之后避开这些限制，写出优雅的 Go 泛型代码。

话不多说，我们现在开始讨论第一点：Go 泛型存在哪些局限。

Go 泛型的局限

在早期泛型还处于提案阶段的时候，我就尝试过利用泛型来设计中间件。当然，即便到现在，我对泛型的应用依旧提留在尝试阶段。根据我一年多的不断尝试，以及我自己对中间件开发的理解，目前我认为影响最大的三个局限是：

Go 接口和结构体不支持泛型方法；

泛型约束不能作为类型声明；

泛型约束只能是接口，而不能是结构体。

接下来我们就来逐个分析。

Go 接口和结构体不支持泛型方法

这里需要我们注意的是，虽然 Go 接口或者结构体不允许声明泛型方法，但 Go 接口或者结构体可以是泛型。

在原来没有泛型的时候，如果要设计一个可以处理任意类型的接口，我们只能使用 `interface{}` ，例如：

复制代码

```
type Orm interface {
    Insert(data ...interface{}) (sql.Result, error)
}
```

在这种模式下，用户可以输入任何数据。但是如果用户混用不同类型，例如 `Insert(&User{},&Order{})` ，就会导致插入失败，而编译器并不能帮助用户检测到这种错误。

从利用 ORM 读写数据库的场景来看，我们是希望限制住 data 参数只能是单一类型的多个实例。那么在引入了泛型之后，我们可以声明一个泛型接口来达成这种约束：

复制代码

```
type Orm[T any] interface {
    Insert(data ...T) (sql.Result, error)
}
```

在这个接口里面，我们声明了一个泛型接口 `Orm` ，它含有一个类型参数 `T` ，通过 T 我们确保在 `Insert` 里面，用户传入的都是同一个类型的不同实例。

然而这种设计也是有问题的：**我们需要创建很多个`Orm`实例**。例如插入 `User` 的实例和插入 `Order` 的实例：

复制代码

```
var userOrm Orm[User]
var orderOrm Orm[Order]
```

也就是说，我们的应用有多少个模型，就要声明多少个 `Orm` 的实例。

这显然是不可接受的。因为我们认为 `Orm` 应该是一个可以操作任意模型的接口，也就是一个 `Orm` 实例既可以用于操作 `User` ，也可以用于操作 `Order` 。换言之，我们并不能把类型参数声明在 `Orm` 这样一个接口上。

那么我们应该声明在哪里呢？ 显然，我们应该声明在方法上：

复制代码

```
type Orm interface {
    Insert[T any](data ...T) (sql.Result, error)
}
```

乍一看，这种声明方式完全能够达到我们的目标，用户用起来只需要：

复制代码

```
orm.Insert[*User](&User{}, &User{})
orm.Insert[*Order](&Order{}, &Order{})
```

然而，如果我们尝试编译，就会得到错误：

复制代码

```
interface method cannot have type parameters
```

也不仅仅是接口会这样，即便我们直接做成结构体：

复制代码

```
type orm struct {
}
func (o orm) Insert[T any](data ...T) (sql.Result, error) {
    //...
}
```

我们依旧会得到一个类似的错误：

复制代码

```
invalid AST: method must have no type parameters
```

实际上，操作任意类型的接口很常见，特别是对于提供客户端功能的中间件来说，尤其常见。例如，如果我们设计一个 HTTP 客户端的中间件，我们可能希望是：

复制代码

```
type HttpClient interface {
   Get[T any](url string) (T, error)
}
```

这样，用户可以用 `client.Get[User]("/user/123")` 得到一个 `User` 的实例。

又比如，如果我们要设计一个缓存客户端：

复制代码

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

但是，显然它们都无法通过编译。

因此，我们可以说，**这个限制对所有的客户端类应用都很不友好**。这些客户端包括 Redis、Kafka 等各种中间件，也包括这提到的 HTTP、ORM 等框架。

泛型约束不能作为类型声明

在泛型里面，有一个很重要的概念“约束”。我们使用约束来声明类型参数要满足的条件。一个典型的 Go 约束，可以是一个普通的接口，也可以是多个类型的组合，例如：

复制代码

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

看到这种语法，我们自然会想到在中间件开发中，经常会有这么一种情况：我们某个方法接收多种类型的的参数，但是它们又没有实现共同的接口。

例如，我们现在要设计一个 SQL 的 `Builder`，用于构造 `SQL`。那么在设计 `SELECT XXX` 这个部分的时候，最基本，也是最基本的做法就是直接传 `string`：

复制代码

```
type Selector struct {
    columns []string
}
func (s *Selector) Select(cols ...string) *Selector {
    s.columns = cols
    return s
}

```

但是这种设计存在一个问题，就是**用户如果要使用聚合函数的话，需要自己手动拼接**，例如 `Select("AVG(age)")` 。而实际上，我们希望能够帮助用户完成这个过程，将聚合函数也变成一个方法调用：

复制代码

```
type Aggregate struct {
    fun   string
    col   string
    alias string
}
func Avg(col string) Aggregate {
    return Aggregate{
        fun: "AVG",
        col: col,
    }
}
func (a Aggregate) As(alias string) Aggregate {
    return Aggregate{
        fun:   a.fun,
        col:   a.col,
        alias: alias,
    }
}
```

使用起来如： `Select(Avg("age").As("avg_age"), "name")` ，这样我们就能帮助检测传入的列名是否正确，而且用户可以避开字符串拼接之类的问题。

在这种情况下， `Select` 方法必须要接收两种输入： `string` 和 `Aggregate` 。在没有泛型的情况下，大多数时候我们都是直接使用 `interface` 来作为参数，并且结合 `switch-case` 来判断：

复制代码

```
type Selector struct {
    columns []Aggregate
}
func (s *Selector) Select(cols ...interface{}) *Selector {
    for _, col := range cols {
        switch c := col.(type) {
        case string:
            s.columns = append(s.columns, Aggregate{col: c})
        case Aggregate:
            s.columns = append(s.columns, c)
        default:
```

```
12            panic("invalid type")
13        }
14    }
15    return s
16 }
```

但是这种用法存在一个问题，就是无法在编译期检查用户输入的类型是否正确，只有在运行期 default 分支发生 panic 时我们才能知道。

特别是，如果用户本意是传一个 var cols []string ，结果写成了 Select(cols) 而不是 Select(cols...) ，那么就会出现 panic ，而编译器丝毫不能帮我们避免这种低级错误。

那么，结合我们的泛型约束，似乎我们可以考虑写成这样：

复制代码

```
type Selectable interface {
    string | Aggregate
}
type Selector struct {
    columns []Selectable
}
func (s *Selector) Select(cols ...Selectable) *Selector {
    panic("implement me")
}
```

利用 Selectable 约束，我们限制住了类型只能是 string 或者 Aggregate 。那么用户可以直接使用 string ，例如 Select("name", Avg("age")) 。

看起来非常完美，用户享受到了编译期检查，又不会出现 panic 的问题。

然而，这依旧是不行的，编译的时候会直接报错：

复制代码

```
interface contains type constraints
```

也就是说，泛型约束不能被用于做参数，它只能和泛型结合在一起使用，**这就导致我们并不能用泛型的约束，来解决某个接口可以处理有限多种类型输入的问题**。所以长期来看，`interface{}` 这种参数类型还会广泛存在于所有中间件的设计中。

泛型约束只能是接口，而不能是结构体

众所周知，Go 里面有一个特性是组合。因此在泛型引入的时候，我们可能会考虑能否用泛型的约束来限制具体类型必须组合了某个类型。

例如：

复制代码

```
type BaseEntity struct {
   Id int64
}
func Insert[Entity BaseEntity](e *Entity) {

}
type myEntity struct {
   BaseEntity
   Name string
}
```

在实际中这也是一个很常见的场景，即我们在 ORM 操作的时候希望实体类必须组合 `BaseEntity` ，这个 `BaseEntity` 上会定义一些公共字段和公共方法。

不过，同样的，Go 泛型不支持这种用法。Go 泛型约束必须是一个接口，而不能是一个结构体，因此上面这段代码会报错：

复制代码

```
myEntity does not implement BaseEntity
```

不过，目前泛型还没有完全支持好，所以这个报错信息并不够准确，更加准确的信息应该是指出 `BaseEntity` 只能为接口。

这个限制影响也很大，**它直接堵死了我们设计共享字段的泛型方法的道路**。而且，它对于业务开发的影响要比对中间件开发的影响更大，因为业务开发会经常遇到必须要共享某些

数据的场景，例如这里的 ORM 例子，还有前端接收参数的场景等。

绕开限制的思路

从前面的分析来看，这些限制影响广泛，而且限制了泛型的进一步应用。但是，我们可以尝试使用一些别的手段来绕开这些限制。

Builder 模式

前面提到，因为 Go 泛型限制了接口或者结构体，让它们不能有泛型方法，所以对客户端类的中间件 API 设计很不友好。

但是，我们可以尝试用 Builder 模式来解决这个问题。现在我们回到前面说的 HTTP 客户端的例子试试看，这个例子我们可以设计成：

复制代码

```
type HttpClient struct {
    endpoint string
}
type GetBuilder[T any] struct {
    client *HttpClient
    path   string
}
func (g *GetBuilder[T]) Path(path string) *GetBuilder[T] {
    g.path = path
    return g
}
func (g *GetBuilder[T]) Do() T {
    // 真实发出 HTTP 请求
    url := g.client.endpoint + g.path
}
func NewGetRequest[T any](client *HttpClient) *GetBuilder {
    return &GetBuilder[T]{client: client}
}
```

而最开始想利用泛型的时候，我们是希望将泛型定义在方法级别上：

复制代码

```
type HttpClient interface {
    Get[T any](url string) (T, error)
}
```

两者最大的不同就在于NewGetRequest 是一种过程式的设计。在 Builder 设计之下，HttpClient 被视作各种配置的载体，而不是一个真实发出请求的客户端。如果你有 Java 之类面向对象的编程语言的使用背景，那么你会很不习惯这种写法。

当然我们可以将 HttpClient 做成真的客户端，而把 GetBuilder 看成是一层泛型的皮：

复制代码

```
func (g *GetBuilder[T]) Do() T {
    var t T
    g.client.get(g.path, &t)
    return t
}
```

无论哪一种，本质上都是利用了过程式的写法，核心都是 Client + Builder。那么在将来设计各种客户端中间件的时候，你就可以考虑尝试这种解决思路。

标记接口

标记接口这种方案，可以用来解决泛型约束不能用作类型声明的限制。顾名思义，标记接口（tag interface 或者 marker interface）就是打一个标记，本身并不具备意义。这种思路在别的语言里面也很常见，比如说 Java shardingsphere 里面就有一个 OptionalSPI 的标记接口：

复制代码

```
public interface OptionalSPI {
}
```

它什么方法都没有，只是说实现了这个接口的 SPI 都是“可选的”。

Go 里面就不能用空接口作为标记接口，否则所有结构体都可以被认为实现了标签接口，Go 中必须要至少声明一个私有方法。例如我们前面讨论的 SELECT 既可以是列，也可以是聚合函数的问题，我们就可以尝试声明一个接口：

复制代码

```
type Selectable interface {
    aggr()
}
type Column string
func (c Column) aggr() {}
type Aggregate struct {}
func (c Aggregate) aggr() {}

func (s *Selector) Select(cols ...Selectable) *Selector {
    panic("implement me")
}
```

这里， Selectable 就是一个标记接口。它的方法 aggr 没有任何的含义，它就是用于限定 Select 方法只能接收特定类型的输入。

但是用起来其实也不是很方便，比如你看这句代码： Select(Column("name"), Avg("avg_id"))。即便你完全不用聚合函数，你也得用 Column 来传入列，而不能直接传递字符串。但是相比之前接收 interface{} 作为输入，就要好很多了，至少受到了编译器对类型的检查。

Getter/Setter 接口

Getter/Setter 可以用于解决泛型约束只能是接口而不能是结构体的限制，但是它并不那么优雅。

核心思路在于，我们给所有的字段都加上 Get/Set 方法，并且提取出来作为一个接口。例如：

复制代码

```
type Entity interface {
    Id() int64
    SetId(id int64)
}
type BaseEntity struct {
    id int64
}
func (b *BaseEntity) Id() int64 {
    return b.id
}
func (b *BaseEntity) SetId(id int64) {
    b.id = id
```

```
13 }
```

那么我们所有的 ORM 操作都可以限制到该类型上：

复制代码

```
type myEntity struct {
    BaseEntity
}
func Insert[E Entity](e *E) {
}
```

看着这段代码，Java 背景的同学应该会觉得很眼熟，但是在 Go 里面更加习惯于直接访问字段，而不是使用 Getter/Setter。

但是如果只是少量使用，并且结合组合特性，那么效果也还算不错。例如在我们的例子里面，通过组合 `BaseEntity` ，我们解决了所有的结构体都要实现一遍 `Entity` 接口的问题。

总结

我们这里讨论了三个泛型的限制：

Go 接口和结构体不支持泛型方法；

泛型约束不能作为类型声明；

泛型约束只能是接口，而不能是结构体。

并且讨论了三个对应的、可行的解决思路：

Builder 模式；

标记接口；

Getter/Setter 接口。

从我个人看来，这些解决思路只能算是效果不错，但是不够优雅。所以我还是很期盼 Go 将来能够放开这种限制。毕竟在这三个约束之下，Go 泛型使用场景过于受限。

从我个人的工作经历出发，我觉得对于大多数中间件来说，可以使用泛型来提供对用户友好的 API，但是对于内部实现来说，使用泛型的收益非常有限。如果现在已有的代码风格就是过程式的，那么你就可以尝试用泛型将它完全重构一番。

总的来说，我对 Go 泛型的普及持有一种比较悲观的态度。我预计泛型从出来到大规模应用还有一段非常遥远的距离，甚至有些公司可能在长时间内为了保持代码风格统一，而禁用泛型特性。

思考题

1. 如果你是中间件开发，你觉得 Go 泛型出来之后，你会用泛型来改造你维护的项目吗？
2. 如果你用了一些开源软件，你会希望它们的维护者暴露泛型 API 给你吗？
3. 从你的个人经验出发，你会希望 Go 泛型放开这些限制吗？

欢迎在留言区分享你的看法，我们一起讨论。

上一篇 加餐｜聊聊最近大热的Go泛型

精选留言 (1)

写留言

罗杰 神

2022-02-11

看起来限制还是非常多，不过对我而言，泛型几乎都没有使用场景。