课程总结

大明

课程总结

- 课程用过的设计模式
- 我所坚持的设计原则
- 测试用例设计

课程用过的设计模式

- Option 模式
- Builder 模式
- 工厂模式
- 装饰器模式
- 洋葱模式
- singleflight 模式

Option 模式

Option 模式主要用于初始化对象。一般来说对象内部的字段分成两类：

- 必须用户指定的。
- 可选的。给予可选字段默认值，然后允许用户通过 Option 模式来修改默认值为自定义的值。

```
func Open(driver string, dataSourceName string,
    opts...DBOption) (*DB, error) {
    db, err:= sql.Open(driver, dataSourceName)
    if err != nil {
        return nil, err
    }
    return OpenDB(db, opts...)
}
```

大多数时候，用户必须指定的字段作为初始化方法的必传参数，而后跟着 Option 作为不定参数。

Builder 模式

Builder 模式一般用于分步骤构建复杂的对象。复杂一般指两个方面：

- 字段很多
- 对象的构造过程很复杂

那么 ORM 里面的 Selector 就符合上面两个特征。

相比之下，Option 模式更加适合用于构造过程比较简单的对象，比如说只是简单给字段赋值的那种。

大家只要想想 Selector 的构造过程，就可以理解 Option 模式和 Builder 模式的差异。

```
type Selector[T any] struct {  17 usages
    builder
    table TableReference
    where []Predicate
    columns []Selectable

    sess Session
}
```

```
func (s *Selector[T]) Build() (*Query, error) {
    defer func() {
        s.sb.Reset()
    }()
    if s.model == nil {...}

    s.sb.WriteString( s: "SELECT ")

    if err := s.buildColumns(); err != nil : nil

    s.sb.WriteString( s: " FROM ")

    if err := s.buildTable(s.table); err != nil

    /.../
    if len(s.where) > 0 {...}
```

工厂模式

工厂模式我个人用得不多，更加多时候喜欢把它用在提供一个一致抽象上。

在 Go 里面，我还喜欢用函数式的工厂模式。如图就是我说的提供一致抽象来构造一个对象实例，如果对象特别复杂，那么我偏好使用 Builder 模式。

```go
type Value interface {  3 usages
    Field(name string) (any, error)
    SetColumns(rows *sql.Rows) error
}

type Creator func(model *model.Model, entity any) Value  7 us
```

装饰器模式

万物皆可装饰。

装饰器模式是我用得最顺手的设计模式：为接口的实现提供无侵入式的增强功能。也可以提供额外的功能。

在我们课程中：

- 缓存模式大量使用
- 将一个非线程安全的类型转化为类型安全的类型

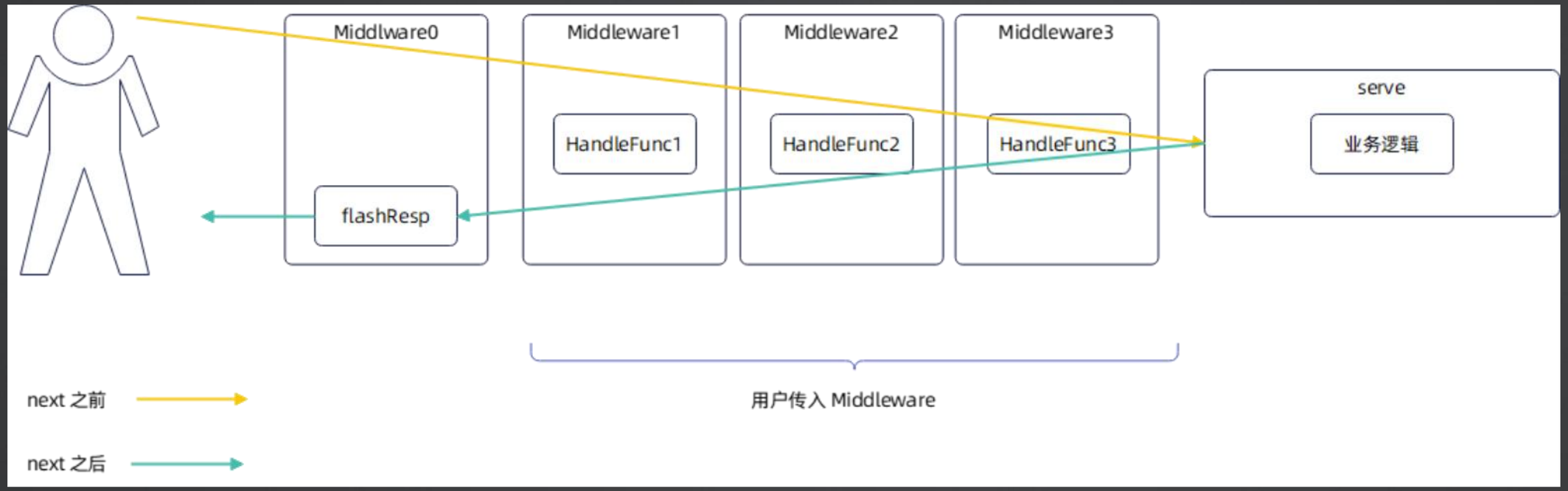
```go
// ReadThroughCache 你一定要赋值 LoadFunc 和 Expiration
// Expiration 是你的过期时间
type ReadThroughCache struct {
    Cache
    LoadFunc func(ctx context.Context, key string) (any, error)
    Expiration time.Duration
    //loadFunc func(ctx context.Context, key string) (any, error)
    //LoadFunc func(key string) (any, error)
    //logFunc func()
    //g singleflight.Group
}
```

这是非常典型的装饰器实践：

- 有一个接口
- 装饰器里面组合接口
- 装饰器增加额外字段，用于二次封装

洋葱模式

洋葱模式大量被用于 AOP 方案中，例如 Web 框架里的 middleware 的设计。

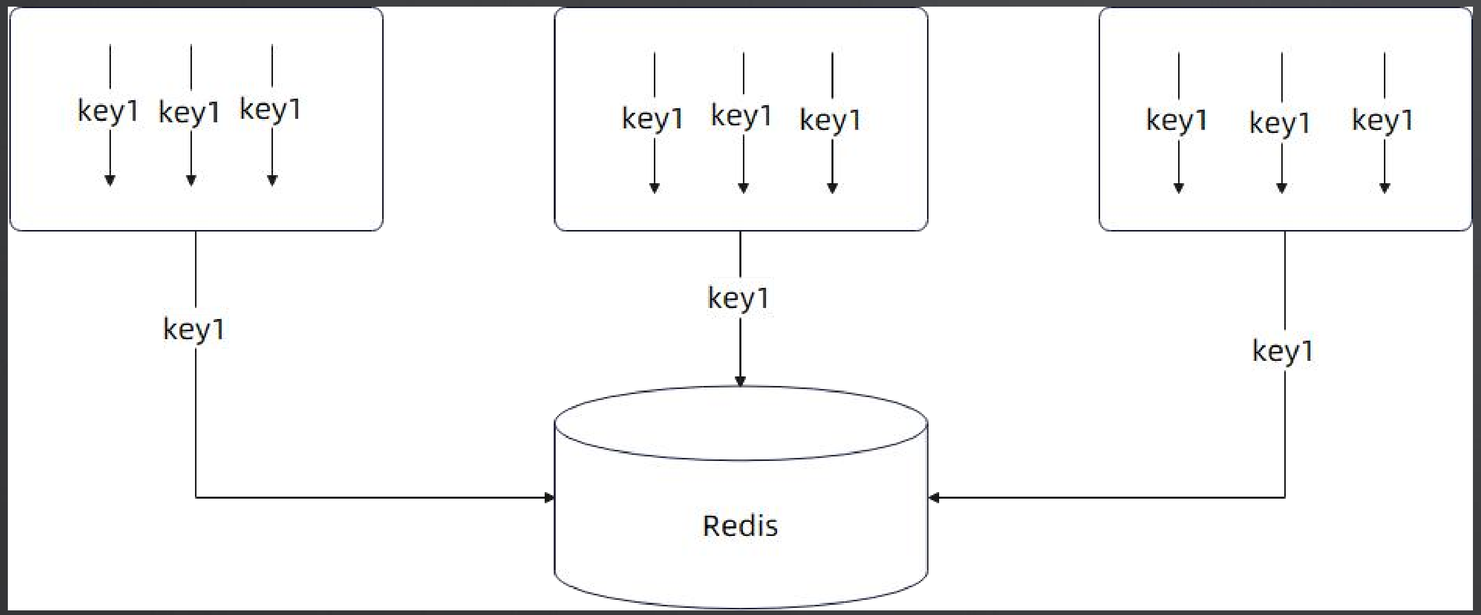


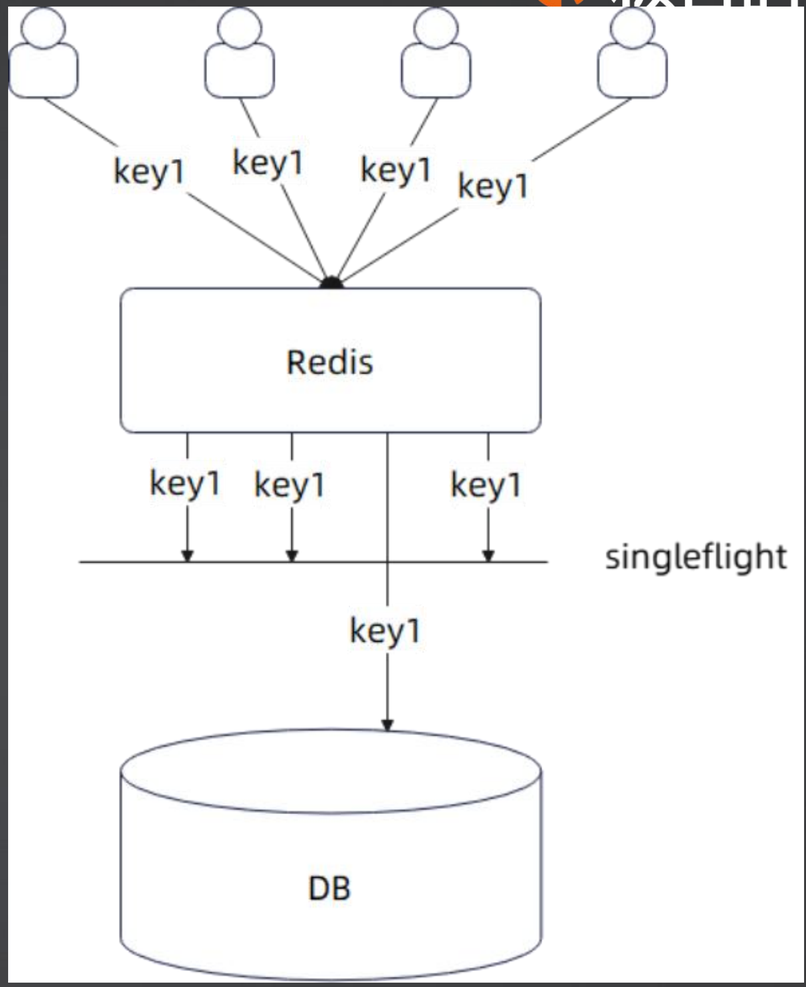
singleflight 模式

singleflight 用于优化并发性能，减少并发数。

我们课程里面在两个地方用了：

- 用 singleflight 来解决缓存穿透、击穿问题
- 用 singleflight 来减轻全局分布式锁的竞争





我所坚持的设计原则

- SOLID：
  - 单一职责原则
  - 开闭原则
  - Liskov 替换原则
  - 接口隔离原则
  - 依赖反转原则

单一功能原则

定义：一个接口就只包含一个职责，而且是最小化的职责。

典型的例子就是我们 Web 框架 Server 接口的设计。和 Gin 的比起来，我们的接口设计更加克制。

你们可以按照以下标准来衡量自己的接口：

- 核心接口方法尽可能少，如果一个方法可以挪出去，那就挪出去。
- 如无必要，不增方法，例如部分实现有某个特性，部分实现没有这个特性，那就不要定义在接口上。

```go
// IRoutes defines all router handle interface.
type IRoutes interface {
    Use(...HandlerFunc) IRoutes

    Handle(string, string, ...HandlerFunc) IRoutes
    Any(string, ...HandlerFunc) IRoutes
    GET(string, ...HandlerFunc) IRoutes
    POST(string, ...HandlerFunc) IRoutes
    DELETE(string, ...HandlerFunc) IRoutes
    PATCH(string, ...HandlerFunc) IRoutes
    PUT(string, ...HandlerFunc) IRoutes
    OPTIONS(string, ...HandlerFunc) IRoutes
    HEAD(string, ...HandlerFunc) IRoutes

    StaticFile(string, string) IRoutes
    StaticFileFS(string, string, http.FileSystem) IRoutes
    Static(string, string) IRoutes
    StaticFS(string, http.FileSystem) IRoutes
}
```

从职责上来说，这个接口只需要保留 Handle 一个方法。尤其是和静态文件处理有关的，不应该放在这里。

开闭原则

可谓是我在课堂里面千叮咛万叮嘱的一条原则。

开闭原则：对扩展开放，对修改闭合。

用通俗的话来说，就是与其修改已有的代码，不如增加新的实现。最大的优点就是因为没有修改已有的代码，所以可以确保已有的业务不会收到影响。

应该秉持这么一个原则，当实现一个新功能的时候：

- 要看看能不能通过提供一个新实现来达成
- 看看能不能通过装饰器模式为已有实现增加额外的功能

大多数的人的代码后面会变得非常难以维护，基本上就是因为每次增加功能都是增加一个 if-else 分支。

Liskov 替换原则

Likov 替换原则：派生类（子类）对象可以在程式中代替其基类（超类）对象。

在Go里面则是：具体实现在代码中可以替换其接口。如果你一直坚持面向接口编程，那么你肯定遵循了这条原则。

你可以用这条来判断你是否遵循了这条原则：如果使用了某个接口，那么不管用什么实现，你的代码都能正确运行

- 任何实现地位都是平等的：尤其是，你怎么接入你的默认实现，别人就怎么接入他的自定义实现
- 使用接口的地方没有类型断言或者向下转型

典型的就是我们 middleware 的设计，和任何实现都没有关系，可以随便替换，接入方式也一模一样。

```
opentelemetry.MiddlewareBuilder{}.Build(),
accesslog.NewBuilder().LogFunc(func(accessLog string) {
    zap.L().Info(accessLog)
}).Build(),
recover.MiddlewareBuilder{
    StatusCode: http.StatusInternalServerError,
    Data: []byte("系统异常"),
    Log: func(ctx *web.Context, err any) {
        zap.L().Error(msg:"服务 panic", zap.Any(key:"panic
            // 发生 panic 的时候，可能都还没到路由查找那里
            zap.String(key:"route", ctx.MatchedRoute))
    },
}.Build(),
prometheus.MiddlewareBuilder{
    Name: "userapp",
    Subsystem: "web",
    // 可以考虑在这里设置 instance id 之类的东西
    // ConstLabels: map[string]string{},
    Help: "userapp 的 web 统计",
} Build()
```

在 userapp 里面，我们不管是用 Web 提供的实现，还是用 userapp 提供的实现，接入方式都完全一样。

接口隔离原则

接口隔离原则和单一职责原则非常类似，定义：客户端不应该被迫依赖于它不使用的方法。

它其实就是我们在单一职责里面讲到的：

- 核心接口方法尽可能少，如果一个方法可以挪出去，那就挪出去。
- 如无必要，不增方法，例如部分实现有某个特性，部分实现没有这个特性，那就不要定义在接口上。

依赖反转原则

定义：使得高层次的模块不依赖于低层次的模块的实现细节，依赖关系被颠倒（反转），从而使得低层次模块依赖于高层次模块的需求抽象。

简单来说：

- 依赖于接口，而不依赖于实现，同时不自己初始化依赖
- 客户端需要什么接口，就定义什么接口

面向接口编程大家普遍会记得，但是如果你面向接口，又得自己初始化实现，那就不对了。比如说右边如果你自己内部初始化这个 client，那么你就依赖了具体的实现。所以依赖反转往往和依赖注入是一起出现的。

```go
//func NewRedisCacheV2(cfgBytes []byte) *RedisCache {
//  var cfg RedisConfig
//  err := json.Unmarshal(cfgBytes, &cfg)
//}

//func NewRedisCacheV1(cfg RedisConfig) *RedisCache {
//  redis.NewClient()
//}

func NewRedisCache(client redis.Cmdable) *RedisCache {
    return &RedisCache{
        client: client,
    }
}
```

单元测试用例模板

分成几个部分：

- name：名字
- mock：如果你单元测试里面需要用到一些 mock 对象和mock 数据，在这里处理。
- 测试对象：你要测试什么结构体，就在这里定义一个。不过也可以考虑在测试代码里面初始化，比如说利用 mock 来初始化。
- 输入：你要测试的具体方法有什么输入，你在这里就定义多少。如果某个参数不影响方法的行为，那么就可以不用定义。
- 输出：方法返回什么，就在这里定义什么。

```
func TestClient_TryLock(t *testing.T) {  Deng Ming
    testCases := []struct {
        name string
        mock func(ctrl *gomock.Controller) redis.Cmdable

        key string

        wantErr  error
        wantLock *Lock
    }{
        {
```

集成测试用例模板

分成几个部分：

- name：名字
- before：准备集成测试用的数据。如果所有的测试用例都需要一些共同的数据，那么可以考虑使用 BeforeTest 钩子。
- after：集成测试用例完成之后，用于校验第三方的数据、行为对不对。比如说在 Redis 相关的测试里面你要判断 Redis 里面的数据对不对。
- 测试对象：你要测试什么结构体，就在这里定义一个，也可以在测试内部初始化。
- 输入：你要测试的具体方法有什么输入，你在这里就定义多少。如果某个参数不影响方法的行为，那么就可以不用定义。
- 输出：方法返回什么，就在这里定义什么。

```
testCases := []struct{
    name string

    before func(t *testing.T)
    after func(t *testing.T)

    key string
    expiration time.Duration
    timeout time.Duration
    retry RetryStrategy

    wantLock *Lock
    wantErr error
} {
```

Q & A

THANKS