



下载APP



27 | 缓存服务：如何基于Redis实现封装？

2021-11-17 叶剑峰

《手把手带你写一个Web框架》

课程介绍 >

**讲述：叶剑峰**

时长 20:07 大小 18.43M



你好，我是轩脉刃。

上面两节课把数据库操作接入到 hade 框架中了，现在我们能使用容器中的 ORM 服务来操作数据库了。在实际工作中，一旦数据库出现性能瓶颈，除了优化数据库本身之外，另外一个常用的方法是使用缓存来优化业务请求。所以这节课，我们来讨论一下，hade 框架如何提供缓存支持。

现在的 Web 业务，大部分都是使用 Redis 来做缓存实现。但是，缓存的实现方式远不止 Redis 一种，比如在 Redis 出现之前，Memcached 一般是缓存首选；在单机上，还可以使用文件来存储数据，又或者直接使用进程的内存也可以进行缓存实现。



缓存服务的底层使用哪个存储方式，和具体的业务架构原型相关。我个人在不同业务场景中用过不少的缓存存储方案，不过业界用的最多的 Redis，还是优点比较突出。相比文件存储，它能集中分布式管理；而相比 Memcached，优势在于多维度的存储数据结构。所以，顺应潮流，我们 hade 框架主要也针对使用 Redis 来实现缓存服务。

我们这节课会创建两个服务，一个是 Redis 服务，提供对 Redis 的封装，另外一个缓存服务，提供一系列对“缓存”的统一操作。而这些统一操作，具体底层是由 Redis 还是内存进行驱动的，这个可以根据配置决定。


下面我们一个个来讨论吧。

Redis 服务

首先封装一个可以对 Redis 进行操作的服务。和封装 ORM 一样，我们自己并不实现 Redis 的底层传输协议和操作封装，只将 Redis “创建连接”的过程封装在 hade 中就行了。

这里我们就选择 [go-redis](#) 这个库来实现对 Redis 的连接。这个库目前也是 Golang 开源社区最常用的 Redis 库，有 12.8k 的 star 数，使用的是 BSD 协议，可以引用，可以修改，但是修改的同时要保留版权声明，这里我们并不需要修改，所以 BSD 已经足够了。这个库目前是 v8 版本，可以使用 `go get github.com/go-redis/redis/v8` 来引入它。

go-redis 的连接非常简单，我们看官网的例子，就看创建连接部分：

 复制代码

```
1 import (  
2     "context"  
3     "github.com/go-redis/redis/v8"  
4 )  
5  
6 var ctx = context.Background()  
7  
8 func ExampleClient() {  
9     // 创建连接  
10    rdb := redis.NewClient(&redis.Options{  
11        Addr:      "localhost:6379",  
12        Password: "", // no password set  
13        DB:        0, // use default DB  
14    })  
15 }
```

```
14     })
15
16     ...
17 }
```

核心的 `redis.NewClient` 方法，返回的是一个 `*redis.Client` 结构，它就相当于 Gorm 中的 DB 数据结构，就是我们要实例化 Redis 的实例。这个结构是一个封装了 300+ 个 Redis 操作的数据结构，你可以使用 `go doc github.com/go-redis/redis/v8.Client` 来观察它封装的 Redis 操作。

配置

`redis.NewClient` 方法还有一个参数：`*redis.Options` 数据结构。这个数据结构就相当于 Gorm 中的 `gorm.Config`，里面封装了实例化 `redis.Client` 的各种配置信息，来看一些重要的配置，都做了注释：

[复制代码](#)

```
1 // redis的连接配置
2 type Options struct {
3     // 网络情况
4     // Default is tcp.
5     Network string
6     // host:port 格式的地址
7     Addr string
8
9     // redis的用户名
10    Username string
11    // redis密码
12    Password string
13    // redis的database
14    DB int
15
16    // 连接超时
17    // Default is 5 seconds.
18    DialTimeout time.Duration
19    // 读超时
20    // Default is 3 seconds.
21    ReadTimeout time.Duration
22    // 写超时
23    // Default is ReadTimeout.
24    WriteTimeout time.Duration
25
26    // 最小空闲连接数
27    MinIdleConns int
28    // 最大连接时长
```

```
29     MaxConnAge time.Duration
30
31     // 空闲连接时长
32     // Default is 5 minutes. -1 disables idle timeout check.
33     IdleTimeout time.Duration
34     ...
35 }
```

这些配置项相信你也非常熟悉了，既有连接请求的配置项，也有连接池的配置项。

和 Gorm 的配置封装一样，我们想要给用户提供的一个配置即用的缓存服务，需要做如下三个事情：

自定义一个数据结构，封装 redis.Options 结构

让刚才自定义的结构能生成一个唯一标识（类似 Gorm 的 DSN）

支持通过配置文件加载这个结构，同时，支持通过 Option 可变参数来修改它

在 framework/contract/redis.go 中，我们首先定义 **RedisConfig 数据结构**，这个结构单纯封装 redis.Options 就行了，没有其他额外的参数需要设置：

```
1 // RedisConfig 为hade定义的Redis配置结构
2 type RedisConfig struct {
3     *redis.Options
```

[复制代码](#)

同时为这个 RedisConfig 定义一个唯一标识，来标识一个 redis.Client。这里我们选用了 Addr、DB、UserName、Network 四个字段值来标识。**基本上这四个字段加起来能标识“用什么账号登录哪个 Redis 地址的哪个 database”了：**

```
1 // UniqKey 用来唯一标识一个RedisConfig配置
2 func (config *RedisConfig) UniqKey() string {
3     return fmt.Sprintf("%v_%v_%v_%v", config.Addr, config.DB, config.Username,
4 }
```

[复制代码](#)

RedisConfig 结构定义完成，下面想要把它加载并支持可修改，我们要结合实例化 redis.Client 对象来说。

初始化连接

如何封装 Redis 的连接实例，这个同 Gorm 的封装一样，使用 Option 可变参数的方式。还是在 framework/contract/redis.go 中继续写入：

[复制代码](#)

```
1 package contract
2
3 ...
4
5 const RedisKey = "hade:redis"
6
7 // RedisOption 代表初始化的时候的选项
8 type RedisOption func(container framework.Container, config *RedisConfig) error
9
10 // RedisService 表示一个redis服务
11 type RedisService interface {
12     // GetClient 获取redis连接实例
13     GetClient(option ...RedisOption) (*redis.Client, error)
14 }
```

定义了一个 RedisService，表示 Redis 服务对外提供的协议，它只有一个 GetClient 方法，通过这个方法能获取到 Redis 的一个连接实例 redis.Client。

你能看到 GetClient 方法有一个可变参数 RedisOption，这个可变参数是一个函数结构，参数中带有传递进入了的 RedisConfig 指针，所以**这个 RedisOption 是有修改 RedisConfig 结构的能力的**。

那具体提供哪些 RedisOption 函数呢？和 ORM 一样，我们要提供多层次的修改方案，包括默认配置、按照配置项进行配置，以及手动配置：

GetBaseConfig 获取 redis.yaml 根目录下的 Redis 配置，作为默认配置

GetConfigPath 根据指定配置路径获取 Redis 配置

WithRedisConfig 可以直接修改 RedisConfig 中的 redis.Options 配置信息

在实现这三个函数之前，有必要先看一下我们的 Redis 配置文件 `cofig/testing/redis.yaml`：

[复制代码](#)

```
1 timeout: 10s # 连接超时
2 read_timeout: 2s # 读超时
3 write_timeout: 2s # 写超时
4
5 write:
6     host: localhost # ip地址
7     port: 3306 # 端口
8     db: 0 #db
9     username: jianfengye # 用户名
10    password: "123456789" # 密码
11    timeout: 10s # 连接超时
12    read_timeout: 2s # 读超时
13    write_timeout: 2s # 写超时
14    conn_min_idle: 10 # 连接池最小空闲连接数
15    conn_max_open: 20 # 连接池最大连接数
16    conn_max_lifetime: 1h # 连接数最大生命周期
17    conn_max_idletime: 1h # 连接数空闲时长
```

和 `database.yaml` 的配置一样，根级别的作为默认配置，二级配置作为单个 Redis 的配置，并且二级配置会覆盖默认配置。这里还有一个小心思，特意将这些配置项都和 `database.yaml` 保持一致了，这样使用者在配置的时候能减少学习成本。

这三个方法的具体实现和 Gorm 没有什么太大的区别。基本方法就是使用容器中的配置服务、读取配置信息，然后修改参数中的 `RedisConfig` 指针，你可以参考分支中的代码文件 [@framework/provider/redis/config.go](#)。

我们重点把注意力放在 **GetClient 方法的实现上**，写在 `framework/provider/redis/service.go` 中。类似 gorm 的 `GetDB` 方法，它是一个单例模式，就是一个 `RedisConfig`，只产生一个 `redis.Client`，用一个 `map` 加上一个 `lock` 来初始化 Redis 实例：

[复制代码](#)

```
1 // HadeRedis 代表hade框架的redis实现
2 type HadeRedis struct {
3     container framework.Container // 服务容器
4     clients    map[string]*redis.Client // key为uniqKey, value为redis.Client (连
5
```

```
6     lock *sync.RWMutex
7 }
```

在 GetClient 函数中，首先还是获取基本 Redis 配置 redisConfig，使用参数 opts 对 redisConfig 进行修改，最后判断当前 redisConfig 是否已经实例化了：

如果已经实例化，返回实例化 redis.Client；

如果未实例化，实例化 redis.Client，返回实例化的 redis.Client。

[复制代码](#)

```
1 // GetClient 获取Client实例
2 func (app *HadeRedis) GetClient(option ...contract.RedisOption) (*redis.Client
3     // 读取默认配置
4     config := GetBaseConfig(app.container)
5
6     // option对opt进行修改
7     for _, opt := range option {
8         if err := opt(app.container, config); err != nil {
9             return nil, err
10        }
11    }
12
13    // 如果最终的config没有设置dsn,就生成dsn
14    key := config.UniqKey()
15
16    // 判断是否已经实例化了redis.Client
17    app.lock.RLock()
18    if db, ok := app.clients[key]; ok {
19        app.lock.RUnlock()
20        return db, nil
21    }
22    app.lock.RUnlock()
23
24    // 没有实例化gorm.DB，那么就要进行实例化操作
25    app.lock.Lock()
26    defer app.lock.Unlock()
27
28    // 实例化gorm.DB
29    client := redis.NewClient(config.Options)
30
31    // 挂载到map中，结束配置
32    app.clients[key] = client
33
34    return client, nil
35 }
```


这里只讲了 Redis 服务的接口和服务实现的关键函数，其中 provider 的实现基本上和 ORM 的一致，没有什么特别，就不在这里重复列出代码了。

到这里我们就将 Redis 的服务融合进入 hade 框架了。但 Redis 只是缓存服务的一种实现，我们这节课最终目标是想实现一个缓存服务。

缓存服务

缓存服务的使用方式其实非常多，我们可以设置有超时 / 无超时的缓存，也可以使用计数器缓存，一份好的缓存接口的设计，能对应用的缓存使用帮助很大。

所以这一部分，相比缓存服务的具体实现，**缓存服务的协议设计直接影响了这个服务的可用性**，我们要重点理解对缓存协议的设计。

协议


实现一个服务的三步骤，服务协议、服务提供者、服务实例。就先从协议开始，我们希望这个缓存服务提供哪些能力呢？

首先，缓存协议一定是有两个方法，一个设置缓存、一个获取缓存。设定为 Get 方法为获取缓存，Set 方法为设置缓存。

 复制代码


```
1 // Get 获取某个key对应的值
2 Get(ctx context.Context, key string) (string, error)
3 // Set 设置某个key和值到缓存，带超时时间
4 Set(ctx context.Context, key string, val string, timeout time.Duration) error
```

同时，注意设置缓存的时候，又区分出两种需求，我们需要设置带超时时间的缓存，也需要设置不带超时时间的、永久的缓存。所以，Set 方法衍生出 Set 和 SetForever 两种。

 复制代码

```
1 // SetForever 设置某个key和值到缓存，不带超时时间
2 SetForever(ctx context.Context, key string, val string) error
```



在设置了某个 key 之后，会不会需要修改这个缓存 key 的缓存时长呢？完全是有可能的，比如将某个 key 的缓存时长加大，或者想要获取某个 key 的缓存时长，所以我们再把注意力放在缓存时长的操作上，提供对缓存时长的操作函数 SetTTL 和 GetTTL：

 复制代码

```
1 // SetTTL 设置某个key的超时时间
2 SetTTL(ctx context.Context, key string, timeout time.Duration) error
3 // GetTTL 获取某个key的超时时间
4 GetTTL(ctx context.Context, key string) (time.Duration, error)
```


再来，Get 和 Set 目前对应的 value 值为 string，但是我们希望 value 值能不仅仅是一个字符串，它还可以直接是一个对象，这样缓存服务就能存储和获取一个对象出来，能大大方便缓存需求。

所以我们定义两个 GetObj 和 SetObj 方法，来实现对象的缓存存储和获取，但是这个对象在实际存储的时候，又势必要进行序列化和反序列的过程，所以我们对存储和获取的对象再增加一个要求，让它实现官方库的 BinaryMarshaler 和 BinaryUnMarshaler 接口：

 复制代码

```
1 // GetObj 获取某个key对应的对象，对象必须实现 https://pkg.go.dev/encoding#BinaryUnM
2 GetObj(ctx context.Context, key string, model interface{}) error
3 // SetObj 设置某个key和对象到缓存，对象必须实现 https://pkg.go.dev/encoding#BinaryM
4 SetObj(ctx context.Context, key string, val interface{}, timeout time.Duration
5 // SetForeverObj 设置某个key和对象到缓存，不带超时时间，对象必须实现 https://pkg.go.de
6 SetForeverObj(ctx context.Context, key string, val interface{}) error
```

现在，我们已经可以一个 key 进行缓存获取和设置了，但是有时候要同时对多个 key 做缓存的获取和设置，来设置对多个 key 进行操作的方法 GetMany 和 SetMany：

 复制代码

```
1 // GetMany 获取某些key对应的值
2 GetMany(ctx context.Context, keys []string) (map[string]string, error)
3 // SetMany 设置多个key和值到缓存
4 SetMany(ctx context.Context, data map[string]string, timeout time.Duration) er
```

在实际业务中，我们还会有一些计数器的需求，需要将计数器存储到缓存，同时也要能对这个计数器缓存进行增加和减少的操作。可以为计数器缓存设计 Calc、Increment、Decrement 的接口：

[复制代码](#)

```
1 // Calc 往key对应的值中增加step计数
2 Calc(ctx context.Context, key string, step int64) (int64, error)
3 // Increment 往key对应的值中增加1
4 Increment(ctx context.Context, key string) (int64, error)
5 // Decrement 往key对应的值中减去1
6 Decrement(ctx context.Context, key string) (int64, error)
```

缓存的使用有一种 Cache-Aside 模式，可以提升“获取数据”的性能。可能你没有听过这个名字，但其实我们都用过，这个模式描述的就是在实际操作之前，先去缓存中查看有没有对应的数据，如果有的话，不进行操作，如果没有的话才进行实际操作生成数据，并且把数据存储在缓存中。

我们希望缓存服务也能支持这种 Cache-Aside 模式。如何支持呢？

首先，要有一个生成数据的通用方法结构，我们定义为 RememberFunc，让这个函数将服务容器传递进去，这样在具体的实现中，使用者就可以从服务容器中获取各种各样的具体注册服务了，能大大增强这个 RememberFunc 的实现能力：

[复制代码](#)

```
1 // RememberFunc 缓存的Remember方法使用，Cache-Aside模式对应的对象生成方法
2 type RememberFunc func(ctx context.Context, container framework.Container) (in
```

然后，我们为缓存服务定义一个 Remember 方法，来实现这个 Cache-Aside 模式。

[复制代码](#)

```
1 // Remember 实现缓存的Cache-Aside模式，先去缓存中根据key获取对象，如果有的话，返回，如果
2 Remember(ctx context.Context, key string, timeout time.Duration, rememberFunc
```

它的参数来仔细看下。除了 context 之外，有一个 key，代表这个缓存使用的 key，其次是 timeout 代表缓存时长，接着是前面定义的 RememberFunc 了，代表如果缓存中没有

这个 key，就调用 RememberFunc 函数来生成数据对象。

这个数据对象从哪里输出呢？就是这里的最后一个参数 model 了，当然这个 Obj 必须实现 BinaryMarshaler 和 BinaryUnmarshaler 接口。这样定义之后，Remember 的具体实现就简单了。

看这个我在单元测试代码 provider/cache/services/redis_test.go 中写的测试：

[复制代码](#)

```
1 type Bar struct {
2     Name string
3 }
4 func (b *Bar) MarshalBinary() ([]byte, error) {
5     return json.Marshal(b)
6 }
7 func (b *Bar) UnmarshalBinary(bt []byte) error {
8     return json.Unmarshal(bt, b)
9 }
10
11 Convey("remember op", func() {
12     objNew := Bar{}
13     objNewFunc := func(ctx context.Context, container framework.Container) (int
14         obj := &Bar{
15             Name: "bar",
16         }
17         return obj, nil
18     }
19     err = mc.Remember(ctx, "foo_remember", 1*time.Minute, objNewFunc, &objNew)
20     So(err, ShouldBeNil)
21     So(objNew.Name, ShouldEqual, "bar")
22 })
```

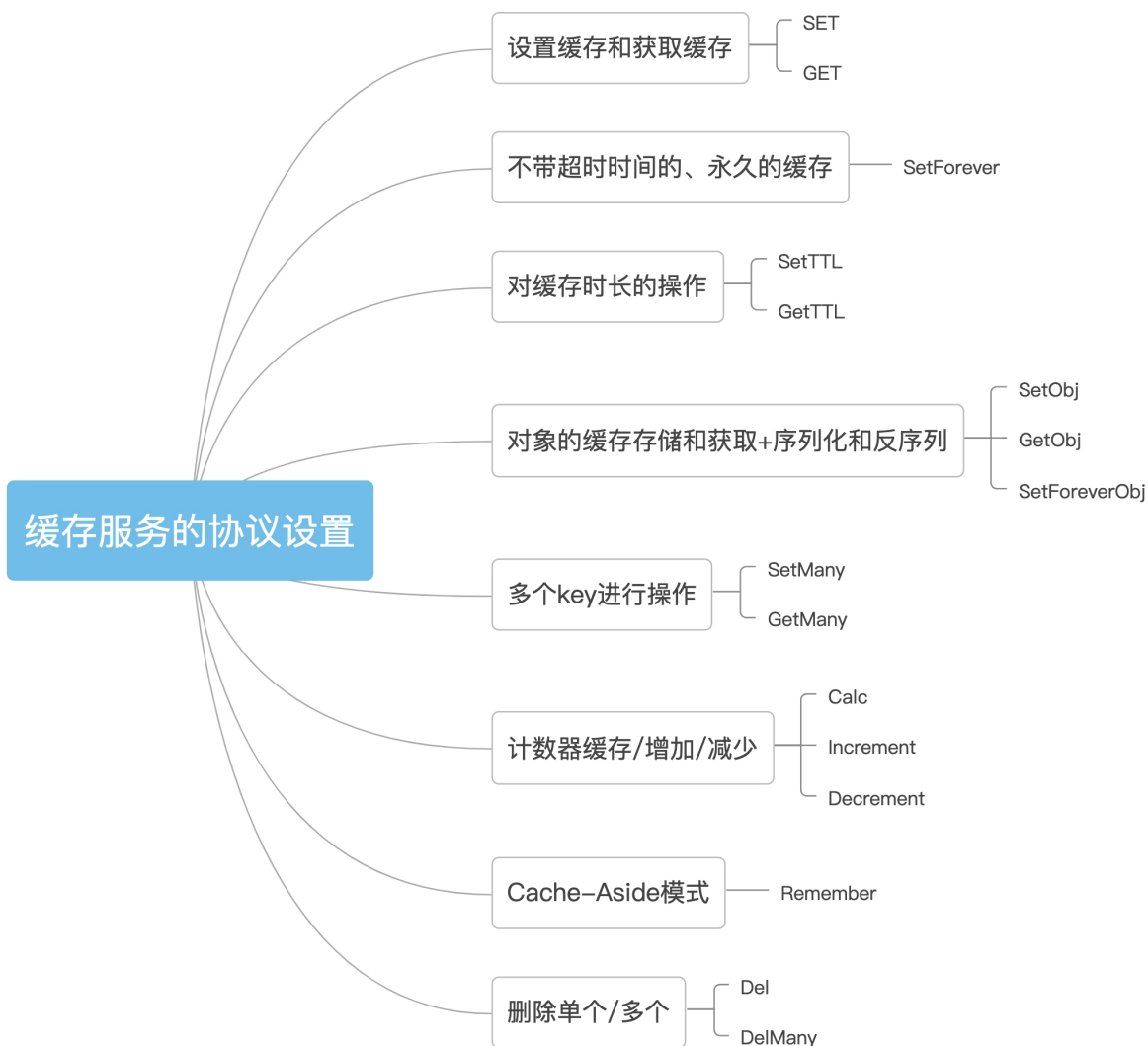
我们定义了 Bar 结构，它实现了 BinaryMarshaler 和 BinaryUnmarshaler 接口，并且定义了一个 objNewFunc 方法实现了前面我们定义的 RememberFunc。

之后可以使用 Remember 方法来为这个方法设置一个 Cache-Aside 缓存，它的 key 为 foo_remember，缓存时长为 1 分钟。

最后回看一下我们对缓存的协议定义，各种缓存的设置和获取方法都有了，还差删除缓存的方法对吧。所以来定义删除单个 key 的缓存和删除多个 key 的缓存：

```
1 // Del 删除某个key
2 Del(ctx context.Context, key string) error
3 // DelMany 删除某些key
4 DelMany(ctx context.Context, keys []string) error
```

到这里缓存协议就定义完成了，一共 16 个方法，要好好理解下这些方法的定义，还是那句话，理解如何定义协议比实现更为重要。



实现

下面来实现这个缓存服务。前面一再强调了，Redis 只是缓存的一种实现，Redis 之外，我们可以用不同的存储来实现缓存，甚至，可以使用内存来实现。目前 hade 框架支持内存

和 Redis 实现缓存，这里我们就先看看如何用 Redis 来实现缓存。

由于缓存有不同实现，所以和日志服务一样，**要使用配置文件来 cache.yaml 中的 driver 字段，来区别使用哪个缓存**。如果 driver 为 redis，表示使用 Redis 来实现缓存，如果为 memory，表示用内存来实现缓存。当然如果使用 Redis 的话，就需要同时带上 Redis 连接的各种参数，参数关键字都类似前面说的 Redis 服务的配置。

一个典型的 cache.yaml 的配置如下：

[复制代码](#)

```
1 driver: redis # 连接驱动
2 host: 127.0.0.1 # ip地址
3 port: 6379 # 端口
4 db: 0 #db
5 timeout: 10s # 连接超时
6 read_timeout: 2s # 读超时
7 write_timeout: 2s # 写超时
8
9 #driver: memory # 连接驱动
```

那对应到具体实现上，区分使用哪个缓存驱动，我们会在服务提供者 provider 中进行。在 provider 中，注意下 Register 方法，注册具体的服务实例方法时，要先读取配置中的 cache.driver 路径：

[复制代码](#)

```
1 // Register 注册一个服务实例
2 func (l *HadeCacheProvider) Register(c framework.Container) framework.NewInsta
3     if l.Driver == "" {
4         tcs, err := c.Make(contract.ConfigKey)
5         if err != nil {
6             // 默认使用console
7             return services.NewMemoryCache
8         }
9
10        cs := tcs.(contract.Config)
11        l.Driver = strings.ToLower(cs.GetString("cache.driver"))
12    }
13
14    // 根据driver的配置项确定
15    switch l.Driver {
16    case "redis":
17        return services.NewRedisCache
```

```
18     case "memory":
19         return services.NewMemoryCache
20     default:
21         return services.NewMemoryCache
22 }
23 }
```

如果是 Redis 驱动，我们使用 `service.NewRedisCache` 来初始化一个 Redis 连接，定义 `RedisCache` 结构来存储 `redis.Client`。

在初始化的时候，先确定下容器中是否已经绑定了 Redis 服务，如果没有的话，做一下绑定操作。这个行为能让我们的缓存容器更为安全。

接着使用 `cache.yaml` 中的配置，来初始化一个 `redis.Client`，这里使用的 `redisService.GetClient` 和 `redis.WithConfigPath`，都是上面设计 Redis 服务的时候刚设计实现的方法。最后将 `redis.Client` 封装到 `RedisCache` 中，返回：

[📄 复制代码](#)

```
1  import (
2      "context"
3      "errors"
4      redisv8 "github.com/go-redis/redis/v8"
5      "github.com/gohade/hade/framework"
6      "github.com/gohade/hade/framework/contract"
7      "github.com/gohade/hade/framework/provider/redis"
8      "sync"
9      "time"
10 )
11
12 // RedisCache 代表Redis缓存
13 type RedisCache struct {
14     container framework.Container
15     client    *redisv8.Client
16     lock      sync.RWMutex
17 }
18
19 // NewRedisCache 初始化redis服务
20 func NewRedisCache(params ...interface{}) (interface{}, error) {
21     container := params[0].(framework.Container)
22     if !container.IsBind(contract.RedisKey) {
23         err := container.Bind(&redis.RedisProvider{})
24         if err != nil {
25             return nil, err
26         }
27     }
```

```
28 // 获取redis服务配置，并且实例化redis.Client
29 redisService := container.MustMake(contract.RedisKey).(contract.RedisService)
30 client, err := redisService.GetClient(redis.WithConfigPath("cache"))
31 if err != nil {
32     return nil, err
33 }
34
35 // 返回RedisCache实例
36 obj := &RedisCache{
37     container: container,
38     client:    client,
39     lock:      sync.RWMutex{},
40 }
41 return obj, nil
42 }
43
```

好，有 Redis 缓存的实例了，下面来看 16 个方法的实现。


Set 系列的方法一共有 Set/SetObj/SetMany/SetForever/SetForeverObj/SetTTL 6 个，其他 5 个相对简单一些，在生成的 redis.Client 结构中都有对应实现，我们直接使用 redis.Client 调用即可，就不赘述了。其中 SetMany 方法相对复杂些，我们着重说明下。

在 Redis 中，SetMany 这种为多个 key 设置缓存的方法，一般可以遍历 key，然后一个个调用 Set 方法，但是这样效率就低了。更好的实现方式是使用 pipeline。

什么是 Redis 的 pipeline 呢？Redis 的客户端和服务端的交互，采用的是客户端 - 服务端模式，就是每个客户端的请求发送到 Redis 服务端，都会有一个完整的响应。所以，向服务端发送 n 个请求，就对应 n 次响应。那么**对于这种 n 个请求且 n 个请求没有上下文逻辑关系，我们能不能批量发送，但是只发送一次请求，然后只获取一次响应呢？**

Redis 的 pipeline 就是这个原理，它将多个请求合成为一个请求，批量发送给 Redis 服务端，并且只从服务端获取一次数据，拿到这些请求的所有结果。

我们的 SetMany 就很符合这个场景。具体的代码如下：

 复制代码

```
1 // SetMany 设置多个key和值到缓存
2 func (r *RedisCache) SetMany(ctx context.Context, data map[string]string, time
3     pipeline := r.client.Pipeline()
4     cmds := make([]*redisv8.StatusCmd, 0, len(data))
```



```
5     for k, v := range data {
6         cmds = append(cmds, pipeline.Set(ctx, k, v, timeout))
7     }
8     _, err := pipeline.Exec(ctx)
9     return err
10 }
```

先用 `redis.Client.Pipeline()` 来创建一个 pipeline 管道，然后用一个 `redis.StatusCmd` 数组来存储要发送的所有命令，最后调用一次 `pipeline.Exec` 来一次发送命令。

Set 方法就讲到这里，Get 系列的方法一共有 4 个，Get/GetObj/GetMany/GetTTL。

在实现 Get 系列方法的时候有地方需要注意下，因为 **Get 是有可能 Get 一个不存在的 key 的**，对于这种不存在的 key 是否返回 error，是一个可以稍微思考的话题。

比如 Get 这个方法，返回的是 string 和 error，如果对于一个不存在的 key，返回了空字符串 + 空 error 的组合，而对于一个设置了空字符串的 key，也返回空字符串 + 空 error 的组合，这里其实是丢失了“是否存在 key”的信息的。

所以，对于这些不存在的 key，我们设计返回一个 `ErrKeyNotFound` 的自定义 error。像 Get 函数就实现为如下：

[复制代码](#)

```
1 // Get 获取某个key对应的值
2 func (r *RedisCache) Get(ctx context.Context, key string) (string, error) {
3     val, err := r.client.Get(ctx, key).Result()
4     // 这里判断了key是否为空
5     if errors.Is(err, redisv8.Nil) {
6         return val, ErrKeyNotFound
7     }
8     return val, err
9 }
```

其他 Get 相关的实现没有什么难点。

除了 Get 系列和 Set 系列，其他的方法有 Calc、Increment、Decrement、Del、DelMany 都没有什么太复杂的逻辑，都是 `redis.Client` 的具体封装。

最后看下 Remember 这个方法：

 复制代码

```
1 // Remember 实现缓存的Cache-Aside模式，先去缓存中根据key获取对象，如果有的话，返回，如果
2 func (r *RedisCache) Remember(ctx context.Context, key string, timeout time.Du
3     err := r.GetObj(ctx, key, obj)
4     // 如果返回为nil，说明有这个key，且有数据，obj已经注入了，返回nil
5     if err == nil {
6         return nil
7     }
8
9     // 有err，但是并不是key不存在，说明是有具体的error的，不能继续往下执行了，返回err
10    if !errors.Is(err, ErrKeyNotFound) {
11        return err
12    }
13
14    // 以下是key不存在的情况，调用rememberFunc
15    objNew, err := rememberFunc(ctx, r.container)
16    if err != nil {
17        return err
18    }
19
20    // 设置key
21    if err := r.SetObj(ctx, key, objNew, timeout); err != nil {
22        return err
23    }
24    // 用GetObj将数据注入到obj中
25    if err := r.GetObj(ctx, key, obj); err != nil {
26        return err
27    }
28    return nil
29 }
```

前面说过 Remember 方法是 Cache-Aside 模式的实现，它的逻辑是先判断缓存中是否有这个 key，如果有的话，直接返回对象，如果没有的话，就调用 RememberFunc 方法来实例化这个对象，并且返回这个实例化对象。

好了，这里的 framework/provider/cache/redis.go 我们实现差不多了。

验证


来做验证，我们为缓存服务写一个简单的路由，在这个路由中：

获取缓存服务

设置 foo 为 key 的缓存，值为 bar


获取 foo 为 key 的缓存，把值打印到控制台

删除 foo 为 key 的缓存

 复制代码

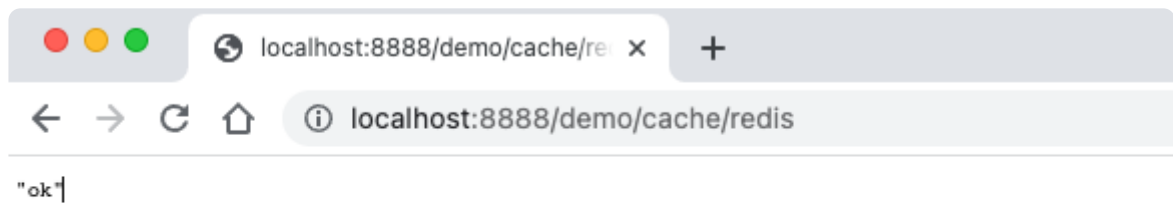
```
1 // DemoCache cache的简单例子
2 func (api *DemoApi) DemoCache(c *gin.Context) {
3     logger := c.MustMakeLog()
4     logger.Info(c, "request start", nil)
5     // 初始化cache服务
6     cacheService := c.MustMake(contract.CacheKey).(contract.CacheService)
7     // 设置key为foo
8     err := cacheService.Set(c, "foo", "bar", 1*time.Hour)
9     if err != nil {
10         c.AbortWithError(500, err)
11         return
12     }
13     // 获取key为foo
14     val, err := cacheService.Get(c, "foo")
15     if err != nil {
16         c.AbortWithError(500, err)
17         return
18     }
19     logger.Info(c, "cache get", map[string]interface{}{
20         "val": val,
21     })
22     // 删除key为foo
23     if err := cacheService.Del(c, "foo"); err != nil {
24         c.AbortWithError(500, err)
25         return
26     }
27     c.JSON(200, "ok")
28 }
```

增加对应的路由：

 复制代码

```
1 r.GET("/demo/cache/redis", api.DemoRedis)
```

在浏览器中请求地址：<http://localhost:8888/demo/cache/redis>：

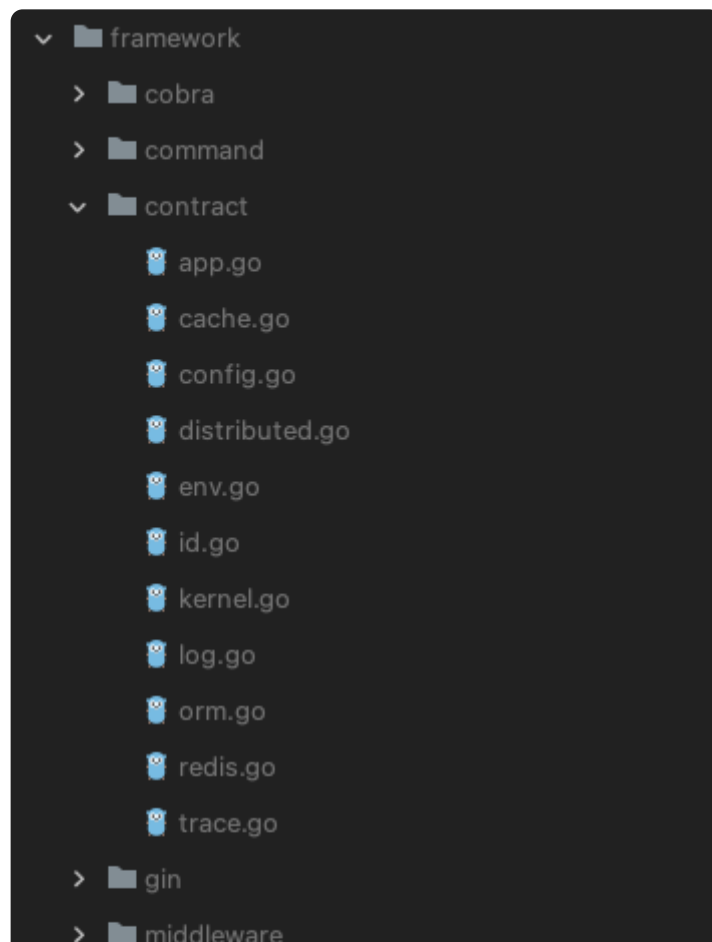


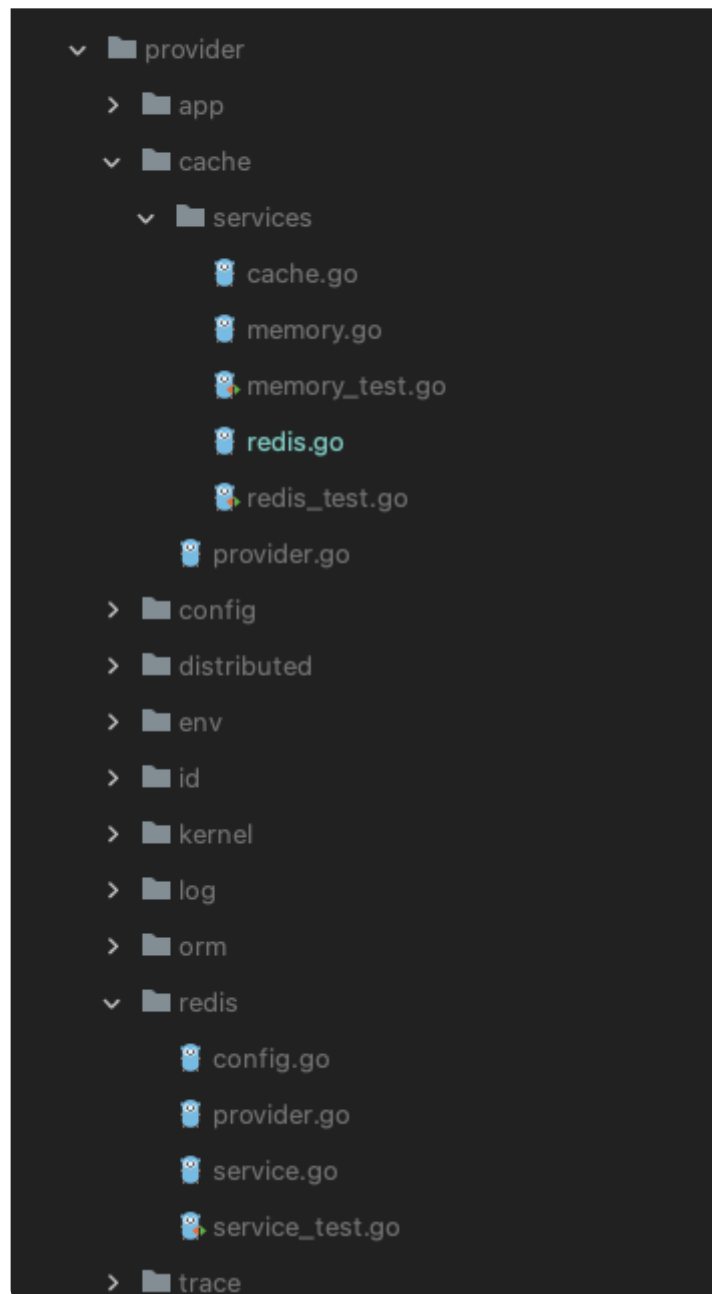
查看控制台输出的日志：

```
~/Documents/UGit/coredemo  geekbang/27  ./hade app start
[PID] 99191
app serve url: :8888
[Info] 2021-11-06T16:55:03+08:00      "request start" map[]
[Info] 2021-11-06T16:55:03+08:00      "redis get"      map[val:get foo: bar]
```

可以明显看到 `cacheService.Get` 的数据为 `bar`，打印了出来。验证正确！

本节课我们主要修改了 `framework` 目录下 `Redis` 和 `cache` 相关的代码。目录截图也放在这里供你对比查看，所有代码都已经上传到 [geekbang/27](https://github.com/geekbang/geekbang/tree/master/27) 分支了。

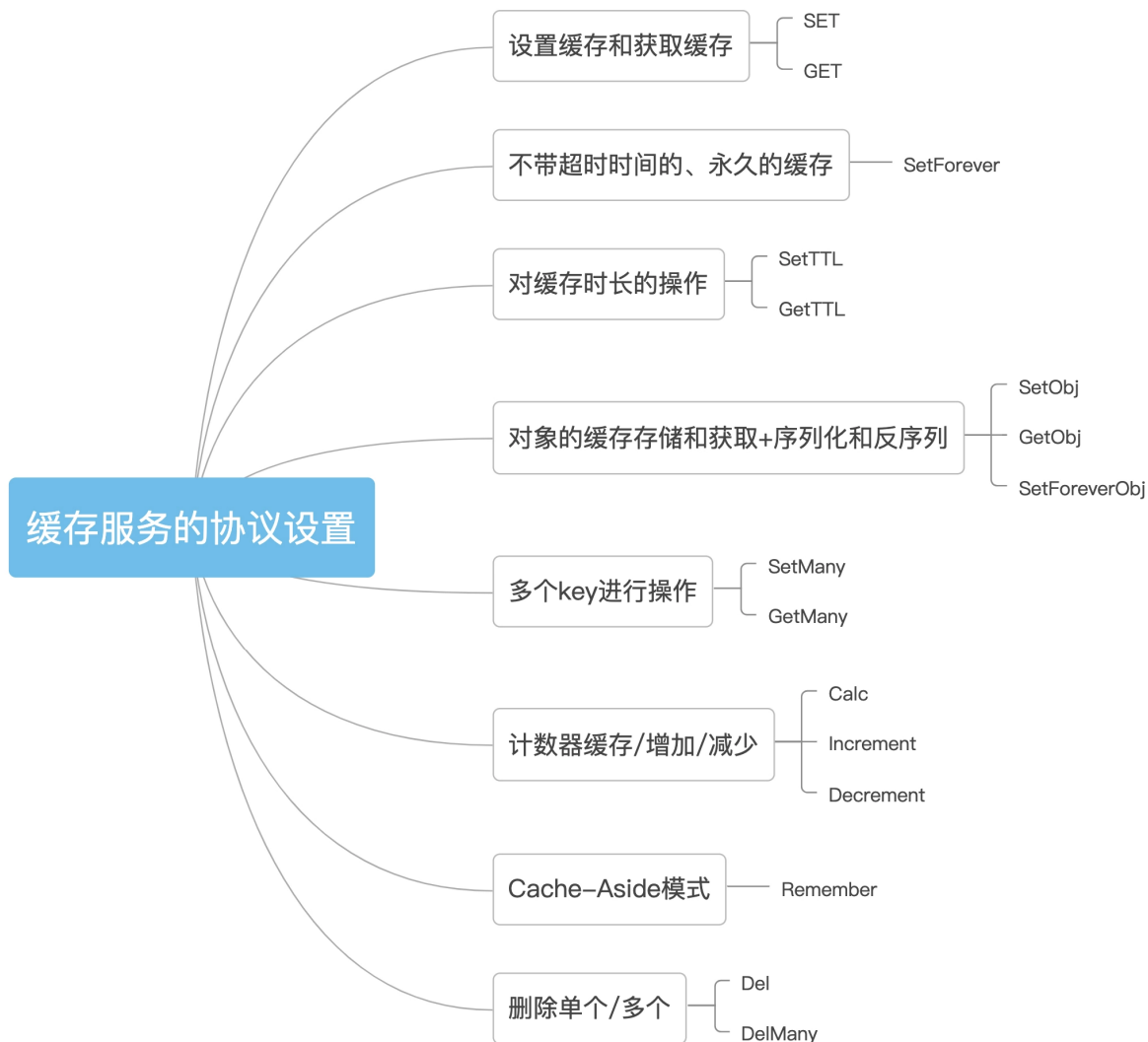




小结

除 DB 之外，缓存是我们最常使用的一个存储了，今天我们先是实现了 Redis 的服务，再用 Redis 服务实现了一个缓存服务。

第一部分的 Redis 服务，同上一节课 ORM 的逻辑一样，我们只是将 go-redis 库进行了封装，具体怎么使用，还是依赖你在实际工作中多使用、多琢磨，网上也有很多 go-redis 库的相关资料。



在第二部分实现的过程中，相信你现在能理解，**一个服务的接口设计，就是一个“我们要什么服务”的思考过程**。比如在缓存服务接口设计中，我们定义了 16 个方法，囊括了 Get/Set/Del/Remember 等一系列方法，你可以对照思维导图复习一下。但这些方法并不是随便拍脑袋出来的，是因为有设置缓存、获取缓存、删除缓存等需求，才这样设计的。

思考题

目前 hade 框架支持内存和 Redis 实现缓存，我们今天展示了 Redis 的实现。缓存服务的内存缓存如何实现呢？可以先思考一下，如果是你来实现会如何设计呢？如果有兴趣，你可以自己动手操作一下。完成之后，你可以比对 GitHub 分支上我已经实现的版本，看看有没有更好的方案。

欢迎在留言区分享你的思考。感谢你的收听，如果你觉得今天的内容对你有所帮助，也欢迎分享给你身边的朋友，邀请他一起学习。我们下节课见~

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0

 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 26 | GORM：数据库的使用必不可少（下）

下一篇 28 | SSH：如何生成发布系统让框架发布自动化？

训练营推荐

Java 学习包免费领 **NEW**

面试题答案均由大厂工程师整理

阿里、美团等
大厂真题

18 大知识点
专项练习

大厂面试
流程解析

可复用的
面试方法

面试前
要做的准备

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

