

# 特别放送 | 回头看：如何更好地组织代码？

2023-02-14 郑建勋 来自北京

《Go进阶·分布式爬虫实战》

[课程介绍 >](#)



讲述：郑建勋

时长 15:57 大小 14.57M



你好，我是郑建勋。

在利用 Go 语言进行系统开发时，相信你或早或晚都会思考这样一个问题：如何更好地组织代码？或者说如何更好地构建项目的目录结构？

这并不是一个容易回答的问题，因为这通常不会有标准的答案。开发者对 Go 语言和软件开发的理解不同、面临的业务场景不同，代码的组织方式也会截然不同。

然而，如果我们对这个问题不管不顾，会发现代码书写起来越来越别扭。代码越来越难做大的调整了。因此，在完成对爬虫系统的开发后，让我们重新来思考一下如何更好地组织代码。

[领资料](#)

## 按照功能划分组织代码

我们之前设计的爬虫系统有对代码进行组织吗？

当然是有的，依靠我们软件开发的经验，代码中有很多增加代码扩展性的设计。包括 Option 模式、大量使用接口解耦依赖、在 main 函数中统一注入依赖等等。从整体上来看，我们也使用了扁平化的目录结构，没有嵌套太深的层次结构。同时，为了便于对代码进行管理，我们根据功能对代码结构进行了划分。例如，auth 负责权限验证，collect 包用于网络爬取，storage 用于数据的存储。上面的这些设计保证了我们对这个爬虫系统仍然有较强的控制力。

通过功能对代码结构进行划分是一种比较常见的也比较容易想到的模式。但是在开发过程中，我们会遇到不少的难题，这些问题也存在于我们开发爬虫系统的过程中，**比较严重的两个就是命名问题和循环依赖问题。**

先来看命名问题。

我们之前抽象出了一个 Fetcher 接口来采集网站信息，并且实现了直接爬取与模拟浏览器访问两种模式。对于网站信息采集这一重要的功能，我们很容易想到要使用一个 package 单独管理它，也很容易想到将 package 命名为 fetch。这时问题就出现了，我们要引用 fetch 包时，命名就变成了 fetch.Fetcher，fetch.BrowserFetch，类似的命名还有 task.Task 等等。这些命名包含着重复语义代码，让人感到很别扭。

 复制代码

```
1 package fetch
2
3 import (
4     "xxx/task"
5     "net/http"
6 )
7
8 type Fetcher interface {
9     Get(url *Request) ([]byte, error)
10 }
11
12 type Request struct {
13     Task *task.Task
14     URL  string
15 }
16
17 type BrowserFetch struct{}
18
19 func (b BrowserFetch) Get(request *Request) ([]byte, error) {
20     req, _ := http.NewRequest("GET", request.URL, nil)
21     if len(request.Task.Cookie) > 0 {
22         req.Header.Set("Cookie", request.Task.Cookie)
23     }
24     // ...
```

领资料

```
25     return nil, nil
26 }
```

我们再来看看循环依赖问题。假设我们又新建了一个 Task 的 package 用于处理和任务相关的工作，很快我们就会发现出现了循环依赖问题。在 fetcher 包中的 Request 结构体引用了 Task 包中的 Task 结构。而 Task 结构中包含了规则树 RuleTree，RuleTree 中引用了 fetch.Request。这导致 fetch package 与 task package 相互依赖。

 复制代码

```
1 package task
2
3 import (
4     "xxx/fetch"
5     "sync"
6 )
7
8 type Task struct {
9     Visited      map[string]bool
10    VisitedLock sync.Mutex
11    Cookie       string
12    Rule         RuleTree
13 }
14
15 type RuleTree struct {
16     Root func() ([]*fetch.Request, error) // 根节点(执行入口)
17 }
```

可以看到，简单地按照功能组织代码，命名问题和循环依赖问题都是非常让人头疼的。

## 按照单体划分组织代码

为了解决上面的问题，我们能否把所有的代码都放入同一个 package 中呢？这样就不存在命名的问题了，同时也就没有了循环依赖的问题。但是这种方式一般只适合于小型的应用程序，一旦代码数量超过一定大小（例如 1 万行代码），阅读和管理代码都会越来越困难。

 领资料

## 按照层级划分组织代码

在实践中，我们还经常看到分层的代码组织方式。比较经典的分层模式是 MVC 模式。MVC 模式一般用于 Web 服务和桌面端应用程序，在典型的 MVC 模型中，代码被分为了视图层（View）、控制层（Controller）和模型层（Model）。

其中，**View** 层用于数据或者页面展示，而比较少进行逻辑处理。**Controller** 层负责在模型和视图之间进行通信。**Model** 层用于管理应用程序的数据和业务规则，并负责与数据库进行交互。典型的 **MVC** 目录结构如下所示。

 复制代码

```
1 app/  
2   models/  
3     user.go  
4     course.go  
5   controllers/  
6     user.go  
7     course.go  
8   views/  
9     user.go  
10    course.go
```

分层的架构能够提供统一的开发模式，让开发者容易理解。但是使用分层架构意味着，只要添加或更改业务功能就几乎要涉及到所有层级，这样一来，要修改单个功能就变得非常麻烦了。

分层架构还面临着和功能架构相同的问题。例如，出现 `controller.UserController` 这样不太优雅命名。如果多个层之间共用了同一个结构，这个结构应该放在哪里呢？随着代码量的上涨，层与层之间，甚至是同层的不同子模块之间的依赖关系会非常混乱，依然容易出现循环依赖的问题。

我举一个例子，**Model** 层提供了一个用于存储数据的 **Storage** 接口，**NewStorage** 函数会根据 **storageType** 类型的不同，决定是使用数据库存储引擎还是内存存储引擎。

 复制代码

```
1 package models  
2 import "xxx/storage"  
3  
4 type Storage interface {  
5     SaveBeer(...Beer) error  
6     SaveReview(...Review) error  
7     FindBeer(Beer) ([]*Beer, error)  
8     FindReview(Review) ([]*Review, error)  
9     FindBeers() []Beer  
10    FindReviews() []Review  
11 }  
12  
13 func NewStorage(storageType StorageType) error {  
14     var err error
```

领资料




```

15
16     switch storageType {
17     case Memory:
18         DB = new(storage.Memory)
19
20     case JSON:
21         DB, err = storage.NewJSON("./data/")
22         if err != nil {
23             return err
24         }
25     }
26
27     return nil
28 }

```

数据库引擎的具体实现位于 `storage` 包中。在 `storage` 包中，又引用了 `model` 包中定义的数据类型，这就导致了循环依赖问题。

 复制代码

```

1 package storage
2
3 import "xxx/models"
4
5 type Memory struct {
6     cellar []models.Beer
7     reviews []models.Review
8 }
9
10 func (s *Memory) SaveBeer(beers ...models.Beer) error {
11     for _, beer := range beers {
12         var err error
13     }

```

这说明在 Go 中，简单地按照分层思想来组织代码仍然不足以让人满意。

## 按照领域驱动设计组织代码

为了应对上面这些挑战，我们来继续探索一下用领域驱动设计（DDD，Domain-Driven Design）组织代码。

DDD 是 Eric Evans 于 2004 年提出的一种软件设计方法和理念。DDD 的核心思想是围绕核心的业务概念，定义领域模型，指导业务与应用的设计和实现。它主张开发人员与业务人员持续



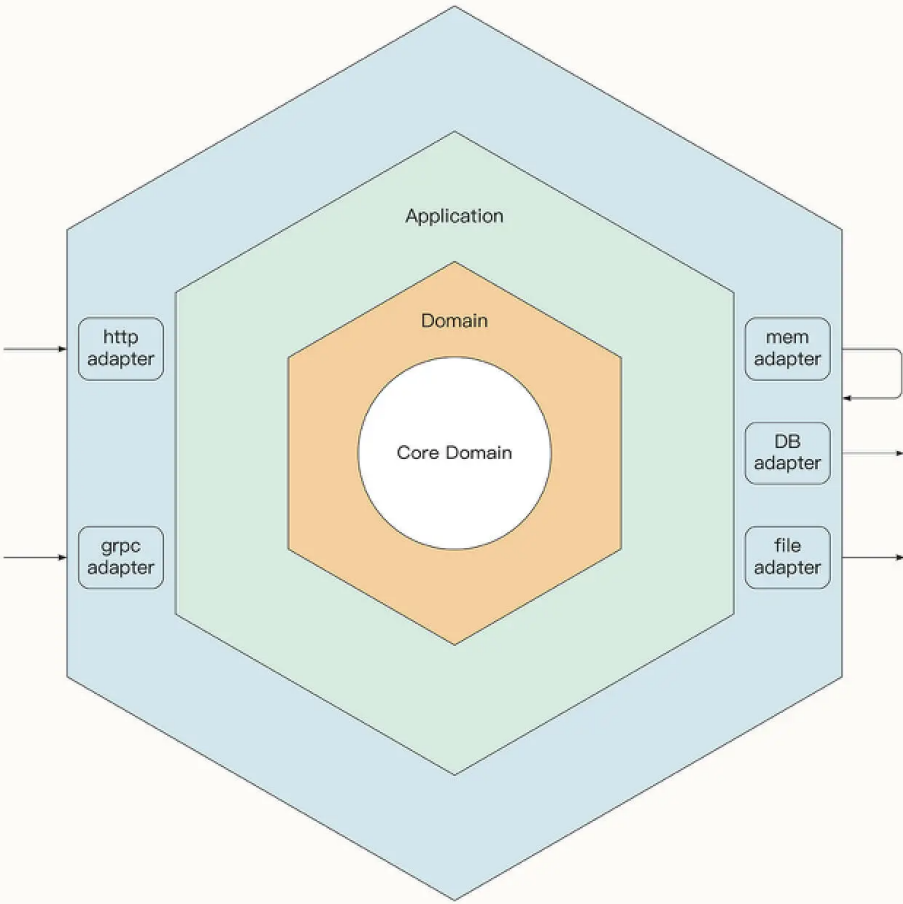


地沟通并持续迭代模型，保证业务模型与代码实现的一致性，最终有效管理业务复杂度，优化软件设计。

DDD 拥有比较陡峭的学习曲线，因为它有许多新的术语，例如领域（Domain）、限界上下文（Bounded Vontext）、实体（Entity）、值对象（Value Object）、聚合（Aggregate）、聚合根（Aggregate Root）等等。我们需要对这些概念有深入的理解，才能够在不同的应用场景中灵活地使用 DDD 进行设计。

## 六边形架构

DDD 是一种软件开发方法论，实践中比较流行的做法是将 DDD 与六边形架构（Hexagonal）结合起来，用 DDD 来指导六边形架构的设计。那什么是六边形架构呢？如下图所示，**这是一种由内而外的分层架构，每一层都有特定的职责，并与其他层级进行隔离。**



领资料

极客时间

六边形架构的最内层即为核心的领域模型（Domain Model），它定义应用程序中使用的实体和关系。应用层（Application）依赖内层的领域模型，负责实现应用程序的核心业务逻辑，例如验证用户输入的数据并处理这些数据。

而外层的适配器用于接收不同形式的输入与输出。例如我们可以有 HTTP 或者 GRPC 等多种适配器来接收不同的外部输入，但是它们最终会由相同的内部处理逻辑来处理。不同形式的输出包括了服务注册、将数据发送到消息队列、持久化存储等。而每一种输出形式都可能有多多个适配器，例如，持久化存储可以有多种形式的适配器，控制把数据存储到 MySQL、MongoDB 还是内存中。这样，程序就有了比较强的扩展性，也方便进行 Mock 测试。

到这里你可能会问，DDD 指导六边形架构为什么能够解决我们之前面临的问题呢？

首先，现在同一个领域中的相关结构（实体、值对象）等都聚合在同一个 package 中。例如我们可以把爬虫领域的相关代码放置到名为 spider 的 package 中，现在要引用该领域中的结构就变为了 spider.Task, spider.Request，这就避免了命名问题。

同时，该架构还能够解决循环依赖问题。因为在六边形架构中，外层可以依赖内层的结构，但是内层不能够依赖外层的结构，这保证了依赖关系的单向性。层级之间一般是通过 Go 接口来交流的，这不仅提高了扩展性，也实现了信息的隐藏。

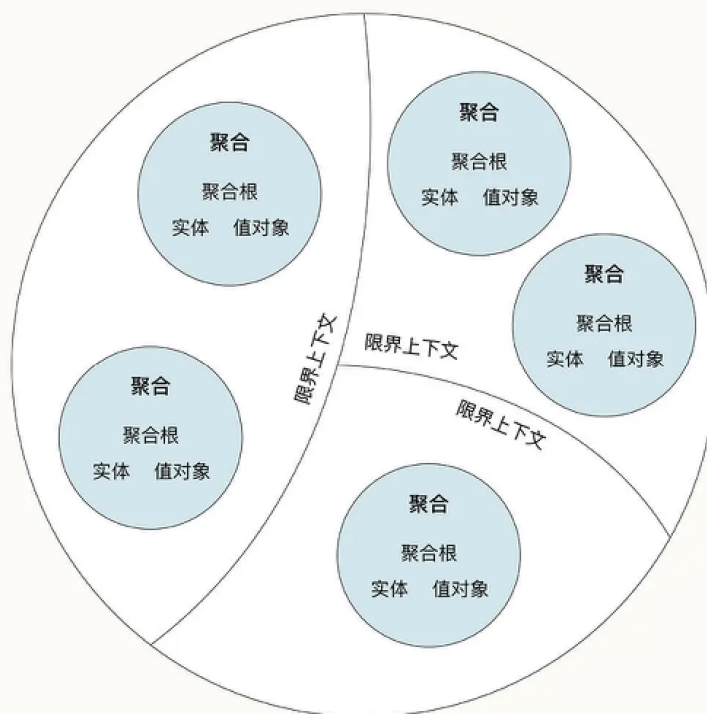
下面就让我们试着用六边形架构重构一下我们的爬虫系统，不过在这之前，我还需要简单解释一下 DDD 的术语。

## 领域与子域

**领域指的是从事一种专门活动或事业的范围。**在 DDD 中，领域可以指业务的整个范围。而业务的整个范围又可以划分为多个子领域，核心的子领域可以被叫做核心域，核心域是业务成功的关键，是公司的核心竞争力，也是需要重点关注的领域。

被多个子域使用的通用功能子域叫做通用域，而不是核心功能也不是通用功能的领域被叫做支撑域，子域还可以细分为更小的问题域。





## 限界上下文

如果我们的领域划分得足够好，会发现某些子域之间形成了天然的边界，**限定在边界中的上下文被称为限界上下文（Bounded context）**。在限界上下文中，业务人员和开发人员可以使用无歧义的统一语言来对话。

举一个例子你就明白了，我们的任务在不同的上下文中，它的含义会有变化。在 **Worker** 进行网站采集的时候，任务指的是包含了指定规则的描述爬虫任务的实例。而在 **Master** 添加任务时，任务只是一个能够唯一用名字标识的资源。再举一个例子，一个到酒吧买醉的人，在外人的眼中就是酒鬼，而在酒吧老板的眼中他其实是一个顾客。

所以，相同的事物在不同的上下文当中具有不同的内涵和外延。限界上下文将事物限制在一定的上下文当中进行讨论，这可以让代码更真实地反映业务，让开发人员与业务人员有共同的语言。同时，限界上下文的边界通常还是进行微服务拆分的基础。

我们当前项目的核心就是开发的爬虫系统，核心域是爬虫域。我们将爬虫域的相关代码放置到 **spider package** 中，并开始在爬虫域的限界上下文中构建业务模型。最终我们会构建出实体、



值对象、聚合等对象。让我们挨个来看一看。

## 实体

首先我们来构建实体。

**DDD 中的实体是在相同限界上下文中具有唯一标识的领域模型，它内部的属性可能发生变化。**在爬虫域中，一个爬虫的请求以及一个爬虫任务可以作为一个实体，我们将其抽象为 Request 结构体和 Task 结构体。ResourceSpec 结构体也可以作为一个实体，它可以描述调度的资源对象，但是否要把它放入到爬虫域中是值得推敲的，我们也许可以单独将其放入到叫做调度域的领域中。

对实体进行建模时，我们可以采取头脑风暴的方式，由开发人员与业务人员反复思考、推敲与迭代，最后根据具体的场景构建比较合适的业务模型。

 复制代码

```
1 package spider
2
3 type Request struct {
4     Task      *Task
5     URL       string
6     Method    string
7     Depth     int64
8     Priority  int64
9     RuleName  string
10    TempData  *Temp
11 }
12
13 // 请求唯一标识
14 func (r *Request) Unique() string {
15     block := md5.Sum([]byte(r.URL + r.Method))
16     return hex.EncodeToString(block[:])
17 }
18
19 func (r *Request) Check() error {
20     if r.Depth > r.Task.MaxDepth {
21         return errors.New("max depth limit reached")
22     }
23
24     if r.Task.Closed {
25         return errors.New("task has Closed")
26     }
27
28     return nil
29 }
```

领资料

```

30 type Task struct {
31     Visited      map[string]bool
32     VisitedLock sync.Mutex
33     //
34     Closed bool
35     Rule RuleTree
36     Options
37 }
38
39 type ResourceSpec struct {
40     ID          string
41     Name        string
42     AssignedNode string
43     CreationTime int64
44 }
45

```

除了添加对应实体的结构体，我们还可以添加实体的方法，用它来处理与实体相关的业务行为。例如，`Request.Check` 方法可以检查请求的有效性。这些方法我之前也已经开发好了，只是迁移到了指定的目录中，这里我就不一一列出了。

## 值对象

说完实体，我们再来看看值对象。**值对象是一种特殊的领域模型，它内部的值是一个整体，所以我们可以通过值判断同一性。**例如，`Request` 结构体中的 `URL` 字段与 `Method` 字段都是一个值对象，它描述了实体的一种属性。

要注意的是，值对象不一定只是字符串、整数这样的基础类型，它还可能是一个结构体。关键在于我们需要把值对象看做一个整体，当我们需要替换实体中的值对象时，需要把值对象的整体都加以替换。所以说，我们可以把爬虫规则树 `RuleTree` 看做一个值对象，因为 `RuleTree` 是 `Task` 的一个属性，并且不能单独被修改。

复制代码

```

1 type RuleTree struct {
2     Root func() ([]*Request, error) // 根节点(执行入口)
3     Trunk map[string]*Rule          // 规则哈希表
4 }

```

领资料

## 聚合

接着我们来看看聚合。

聚合是一组生命周期强一致，同时修改规则强关联的实体和值对象的集合，它表达的是统一的业务意义。比如，一个订单中有多个订单项，订单的总价是根据订单项目计算而来的。由于这些订单项和订单总价是密不可分的，因此可以把它们组合起来作为一个聚合。

在我们的爬虫项目中，用于存储爬虫任务的全局变量 `taskStore` 就可以看作一个聚合，而描述爬虫存储单元的 `DataCell` 和上下文 `Context` 也是一个聚合。聚合和实体一样可以有对应的方法来描述业务行为。

 复制代码

```
1 type taskStore struct {
2     List []*Task
3     Hash map[string]*Task
4 }
5
6 type Context struct {
7     Body []byte
8     Req  *Request
9 }
10
11 type DataCell struct {
12     Task *Task
13     Data map[string]interface{}
14 }
15
16 func (d *DataCell) GetTableName() string {
17     return d.Data["Task"].(string)
18 }
19
20 func (d *DataCell) GetTaskName() string {
21     return d.Data["Task"].(string)
22 }
```

## 服务

上面我们构建了爬虫域中的实体、值对象、聚合以及这些对象对应的方法，但是并不是所有的操作都适合放在这些对象的方法中。例如，爬取网站数据的行为就不适合放在 `spider.Request` 的方法中，因为爬取网站数据属于对 `spider.Request` 的操作，而不是 `spider.Request` 具有的行为。

另外，我们还希望能够灵活地使用不同的采集方式，例如直接访问、代理访问或者模拟浏览器访问。因此我们还需要构建一个 DDD 中重要的对象：服务（Service）。服务是领域模型的操作者，负责领域内业务规则的实现。

领资料

服务的结构体中一般不嵌套具体的实体、值对象和聚合，但是会在方法中串联业务逻辑。同时，服务一般会抽象出接口，这样其他的服务或者对象就可以通过依赖接口使用服务的方法，而不用管服务具体的实现了。我们将爬取网站数据的行为迁移到 `fetchservice.go` 文件中，代码如下所示。

 复制代码

```
1 package spider
2
3 type Fetcher interface {
4     Get(url *Request) ([]byte, error)
5 }
6
7 type BaseFetch struct{}
8
9 func (BaseFetch) Get(req *Request) ([]byte, error){
10     ...
11 }
12
13 type BrowserFetch struct {
14     Timeout time.Duration
15     Proxy    proxy.Func
16     Logger   *zap.Logger
17 }
18
19 // 模拟浏览器访问
20 func (b BrowserFetch) Get(request *Request) ([]byte, error){
21     ...
22 }
```

根据服务的定义，我们可以发现之前设计不太合理的地方。例如，`BrowserFetch` 包含了 `Timeout` 与 `Proxy` 字段用于保存请求相关属性。其实，控制 HTTP 超时的 `Timeout` 以及控制代理的 `Proxy` 都可以放置到描述爬虫任务的 `Task` 当中。同时，在 `service` 文件中，一般还需要有一个函数 `NewXxxService` 来返回应该使用哪一种接口的具体实现，在我们的代码中函数 `NewFetchService` 用于返回具体的采集实现。改造后的代码如下所示。

 复制代码

```
1 package spider
2
3 type FetchType int
4
5 const (
6     BaseFetchType FetchType = iota
7     BrowserFetchType
8 )
```

领资料



```

9
10 type Fetcher interface {
11     Get(url *Request) ([]byte, error)
12 }
13
14 func NewFetchService(typ FetchType) Fetcher {
15     switch typ {
16     case BaseFetchType:
17         return &baseFetch{}
18     case BrowserFetchType:
19         return &browserFetch{}
20     default:
21         return &browserFetch{}
22     }
23 }
24
25 type baseFetch struct{}
26
27 func (*baseFetch) Get(req *Request) ([]byte, error) {
28     ...
29 }
30
31 type browserFetch struct{}
32
33 // 模拟浏览器访问
34 func (b *browserFetch) Get(request *Request) ([]byte, error) {
35     ...
36 }

```

在这段改造后的代码中，我们将具体实现的结构体变为了小写，外部服务或对象只能够通过暴露的 `NewFetchService` 方法与采集服务交互。

## 仓储

服务本身可能依赖其他的多个服务和仓储，并完成更高维度的业务串联，这是我们实现 `Worker` 逻辑的基础。因为我们的爬虫 `Worker` 需要完成接收请求、爬取、存储等业务逻辑。那问题来了，什么是仓储呢？

**仓储（Repository）**是以持久化领域模型为职责的对象。仓储的目的是增加持久化基础设施的可扩展性。在核心域的 `datarepository` 文件中，我们定义了持久化存储的接口。



复制代码

```

1 package spider
2
3 type DataRepository interface {

```



```

4   Save(datas ...*DataCell) error
5 }
6
7 type DataCell struct {
8     Task *Task
9     Data map[string]interface{}
10 }
11
12 func (d *DataCell) GetTableName() string {
13     return d.Data["Task"].(string)
14 }
15
16 func (d *DataCell) GetTaskName() string {
17     return d.Data["Task"].(string)
18 }

```

**DataRepository** 是一个接口，可以有多种实现了该接口的适配器，位于六边形架构的最外层。因此我们的持久化实现不能放置到核心域，而需要放置到外层的 **sqlstorage package** 中。

## 服务串联业务逻辑

在完成了 **fetch Service** 和 **data Repository** 的建模之后，我们需要更高维度的 **Service** 完成复杂的业务逻辑的串联。我们把业务串联与请求调度相关的代码放置到核心域的子文件夹中，命名为 **workerengine**。

之前我们串联业务逻辑时，是在 **Crawler** 结构体的方法中完成的。那我们能够复用这个结构体吗？我们之前的 **Crawler** 结构体中还包含了 **visited**、**failures**、**resources** 等存储资源的容器。

 复制代码

```

1 type Crawler struct {
2     id          string
3     out         chan spider.ParseResult
4     Visited     map[string]bool
5     VisitedLock sync.Mutex
6
7     failures    map[string]*spider.Request // 失败请求id -> 失败请求
8     failureLock sync.Mutex
9
10    resources map[string]*master.ResourceSpec
11    rlock     sync.Mutex
12
13    etcdCli *clientv3.Client
14    options
15 }

```

领资料

但是这并不符合 **Service** 的定义。**Service** 应该只负责调度，不具有存储等逻辑。因此，我们需要重新思考一下设计方法。我们抽象出两个 **Repository**，其中 **ReqHistoryRepository** 用于保存 **Request** 的历史记录，保存访问过的网址或者失败的网址。

 复制代码

```
1 type ReqHistoryRepository interface {
2     AddVisited(reqs ...*Request)
3     DeleteVisited(req *Request)
4     AddFailures(req *Request) bool
5     DeleteFailures(req *Request)
6     HasVisited(req *Request) bool
7 }
8
9 type reqHistory struct {
10     Visited      map[string]bool
11     VisitedLock  sync.Mutex
12
13     failures      map[string]*Request // 失败请求id -> 失败请求
14     failureLock  sync.Mutex
15 }
```

**ResourceRepository** 用于存储资源，**Resource** 是我们用于调度的实例。

 复制代码

```
1 type ResourceRepository interface {
2     Set(req map[string]*ResourceSpec)
3     Add(req *ResourceSpec)
4     Delete(name string)
5     HasResource(name string) bool
6 }
7
8 type resourceRepository struct {
9     resources map[string]*ResourceSpec
10    rlock      sync.Mutex
11 }
```

 领资料

**WorkerService** 是我们对 **Worker** 服务的抽象，而 **workerService** 是具体的实现。从这里我们可以看出，**workerService** 大量地依赖了外部的服务例如 **spider.Fetcher**、**Scheduler**；也依赖了外部的 **Repository**，例如 **spider.DataRepository**、**spider.ReqHistoryRepository**、**spider.ResourceRepository**。

```

1
2 type WorkerService interface {
3     Run(cluster bool)
4     LoadResource() error
5     WatchResource()
6 }
7
8 type workerService struct {
9     out      chan spider.ParseResult
10    rlock     sync.Mutex
11    etcdCli   *clientv3.Client
12    options
13 }
14
15 type options struct {
16     Fetcher      spider.Fetcher
17     Storage       spider.DataRepository
18     Logger        *zap.Logger
19     scheduler     Scheduler
20     reqRepository spider.ReqHistoryRepository
21     resourceRepository spider.ResourceRepository
22 }
23
24 func (c *workerService) Run(cluster bool) {
25     if !cluster {
26         c.handleSeeds()
27     }
28     c.LoadResource()
29     go c.WatchResource()
30     go c.scheduler.Schedule()
31     for i := 0; i < c.WorkCount; i++ {
32         go c.CreateWork()
33     }
34     c.HandleResult()
35 }
36
37 func (c *workerService) CreateWork() {
38     ...
39     for {
40         // 调度器中获取请求
41         req := c.scheduler.Pull()
42
43         // 检查
44         if err := req.Check(); err != nil {
45             c.Logger.Debug("check failed",
46                 zap.Error(err),
47             )
48             continue
49         }
50         if !req.Task.Reload && c.reqRepository.HasVisited(req) {
51             c.Logger.Debug("request has visited",
52                 zap.String("url:", req.URL),

```

```

53     )
54     continue
55 }
56
57 // 请求放入reqRepository
58 c.reqRepository.AddVisited(req)
59
60 // 爬取请求
61 body, err := req.Task.Fetcher.Get(req)
62 rule := req.Task.Rule.Trunk[req.RuleName]
63 ctx := &spider.Context{
64     Body: body,
65     Req:  req,
66 }
67
68 // 规则解析
69 result, err := rule.ParseFunc(ctx)
70 if err != nil {
71     c.Logger.Error("ParseFunc failed ",
72         zap.Error(err),
73         zap.String("url", req.URL),
74     )
75     continue
76 }
77
78 // 请求放入调度器中
79 if len(result.Requesrts) > 0 {
80     go c.scheduler.Push(result.Requesrts...)
81 }
82
83 // 数据存储
84 c.out <- result
85 }
86 }

```

实际上，`workerService` 还可以被进一步优化。因为我们这里是依赖 `etcd client` 实现资源的监听与加载，但是我们很容易再抽象出一个适配器来对接不同的注册中心，所以我将与注册中心相关的代码放置到了 `resourceregistry.go` 中。

其实将 `ResourceRegistry` 接口的具体实现放置到外层的 `package` 也是可以的，可以把它作为六边形架构的最外层。



复制代码

```

1 type EventType int
2
3 const (
4     EventTypeDelete EventType = iota

```

```

5     EventTypePut
6
7     RESOURCEPATH = "/resources"
8 )
9
10 type ResourceRegistry interface {
11     GetResources() ([]*ResourceSpec, error)
12     WatchResources() WatchChan
13 }
14
15 type WatchResponse struct {
16     Typ      EventType
17     Res      *ResourceSpec
18     Canceled bool
19 }
20
21 type WatchChan chan WatchResponse
22
23 type EtcdRegistry struct {
24     etcdCli *clientv3.Client
25 }
26
27 func NewEtcdRegistry(endpoints []string) (ResourceRegistry, error) {
28     cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
29     return &EtcdRegistry{cli}, err
30 }
31
32 func (e *EtcdRegistry) GetResources() ([]*ResourceSpec, error) {
33     resp, err := e.etcdCli.Get(context.Background(), RESOURCEPATH, clientv3.WithP
34     if err != nil {
35         return nil, err
36     }
37     resources := make([]*ResourceSpec, 0)
38     for _, kv := range resp.Kvs {
39         r, err := Decode(kv.Value)
40         if err == nil && r != nil {
41             resources = append(resources, r)
42         }
43     }
44     return resources, nil
45 }
46
47 func (e *EtcdRegistry) WatchResources() WatchChan {
48     ch := make(WatchChan)
49     go func() {
50         ...
51     }()
52
53     return ch
54
55 }

```



最后，我们将所有的依赖的显式的注入到 `main` 函数中，也就是我们项目中的 `cmd/worker.go` 文件的 `Run` 函数中。

 复制代码

```
1 func Run() {
2     ...
3     // init log
4     logText := cfg.Get("logLevel").String("INFO")
5     logLevel, err := zapcore.ParseLevel(logText)
6     if err != nil {
7         panic(err)
8     }
9     plugin := log.NewStdoutPlugin(logLevel)
10    logger = log.NewLogger(plugin)
11
12    // init fetcher
13    proxyURLs := cfg.Get("fetcher", "proxy").StringSlice([]string{})
14    timeout := cfg.Get("fetcher", "timeout").Int(5000)
15    logger.Sugar().Info("proxy list: ", proxyURLs, " timeout: ", timeout)
16    if p, err = proxy.RoundRobinProxySwitcher(proxyURLs...); err != nil {
17        logger.Error("RoundRobinProxySwitcher failed", zap.Error(err))
18    }
19    f := spider.NewFetchService(spider.BrowserFetchType)
20
21    // init storage
22    sqlURL := cfg.Get("storage", "sqlURL").String("")
23    if storage, err = sqlstorage.New(
24        sqlstorage.WithSQLURL(sqlURL),
25        sqlstorage.WithLogger(logger.Named("sqlDB")),
26        sqlstorage.WithBatchCount(2),
27    ); err != nil {
28        logger.Error("create sqlstorage failed", zap.Error(err))
29        panic(err)
30        return
31    }
32
33    // init etcd registry
34    reg, err := spider.NewEtcdRegistry([]string{sconfig.RegistryAddress})
35    if err != nil {
36        logger.Error("init EtcdRegistry failed", zap.Error(err))
37    }
38
39    s, err := engine.NewWorkerService(
40        engine.WithFetcher(f),
41        engine.WithLogger(logger),
42        engine.WithWorkCount(5),
43        engine.WithSeeds(seeds),
44        engine.WithScheduler(engine.NewSchedule()),
45        engine.WithStorage(storage),
46        engine.WithID(id),
```

领资料

```
47     engine.WithReqRepository(spider.NewReqHistoryRepository()),
48     engine.WithResourceRepository(spider.NewResourceRepository()),
49     engine.WithResourceRegistry(reg),
50 )
51
52 if err != nil {
53     panic(err)
54 }
55
56 // worker start
57 go s.Run(cluster)
58 }
```


至此，我们就基于 DDD 实现了爬虫 Worker 的代码重构，虽然这里仍然有许多值得推敲与优化的地方，但现在代码确实变得更加清晰和优雅了，扩展性也更强了。相关的代码我放置到了 [v0.4.3 分支](#)，你可以对照学习。你也可以在此基础上，尝试着将 Master 也重构一下，相信重构的过程一定会非常有趣。

## 总结

在 Go 语言中，并没有一种代码组织方式适用于所有的应用场景。这节课，我们总结了按照单体、层次、或者功能来组织代码会面临的困难，并最终探索出了使用领域驱动设计和六边形架构这条路。它可以解决命名和循环依赖问题，使我们的程序具备极强的扩展性。

领域驱动设计具有较陡峭的学习曲线，且需要不断地迭代业务与领域建模。相信你能够在理论与实践感受到领域驱动设计的好处，改善自己的代码，让编程更加高效、有趣。

分享给需要的人，Ta 购买本课程，你将得 20 元

 生成海报并分享

 赞 1

 提建议

 领资料

© 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](#) 特别放送 | Go 泛型：用法、原理与最佳实践

[下一篇](#) 结束语 | 登高望远，迈向新的高峰

学习推荐

全新汇总

# Go 面试必考 300+ 题

面试真题 | 进阶实战 | 专题视频 | 学习路线

限时免费

仅限 99 名

## 精选留言 (1)

写留言



Realm

2023-02-14 来自浙江

文章中提供本项目的系统架构图，以及各模块之间的逻辑关联图，会让阅读代码体验更好。



1

领资料