

28 | 调度引擎：负载均衡与调度器实战

2022-12-13 郑建勋 来自北京



天下无鱼

<https://shikey.com/>

《Go进阶·分布式爬虫实战》

[课程介绍 >](#)



讲述：郑建勋

时长 11:44 大小 10.72M



你好，我是郑建勋。

在上一节课程中，我们实战了广度优先搜索算法，不过我们对网站的爬取都是在一个协程中进行的。在真实的实践场景中，我们常常需要爬取多个初始网站，我们希望能够同时爬取这些网站。这就需要合理调度和组织爬虫任务了。因此，这节课的重点就是实战任务调度的高并发模型，使资源得到充分的利用。

实战调度引擎

首先，我们新建一个文件夹 `engine` 用于存储调度引擎的代码，核心的调度逻辑位于 `ScheduleEngine.Run` 中。这部分的完整代码位于 [🔗 tag v0.1.4](#)，你可以对照代码进行查看。

调度引擎主要目标是完成下面几个功能：

1. 创建调度程序，接收任务并将任务存储起来；
2. 执行调度任务，通过一定的调度算法将任务调度到合适的 worker 中执行；
3. 创建指定数量的 worker，完成实际任务的处理；
4. 创建数据处理协程，对爬取到的数据进行进一步处理。



复制代码

```
1 func (s *ScheduleEngine) Run() {
2     requestCh := make(chan *collect.Request)
3     workerCh := make(chan *collect.Request)
4     out := make(chan collect.ParseResult)
5     s.requestCh = requestCh
6     s.workerCh = workerCh
7     s.out = out
8     go s.Schedule()
9 }
```

Run 方法首先初始化了三个通道。其中，requestCh 负责接收请求，workerCh 负责分配任务给 worker，out 负责处理爬取后的数据，完成下一步的存储操作。schedule 函数会创建调度程序，负责的是调度的核心逻辑。

第一步我们来看看 schedule 函数如何接收任务并完成任务的调度。

schedule 函数如下所示，其中，requestCh 通道接收来自外界的请求，并将请求存储到 reqQueue 队列中。workerCh 通道负责传送任务，后面每一个 worker 将获取该通道的内容，并执行对应的操作。

在这里，我们使用了 for 语句，让调度器循环往复地获取外界的爬虫任务，并将任务分发到 worker 中。如果任务队列 reqQueue 大于 0，意味着有爬虫任务，这时我们获取队列中第一个任务，并将其剔除出队列。最后 ch <- req 会将任务发送到 workerCh 通道中，等待 worker 接收。

复制代码

```
1 func (s *Schedule) Schedule() {
2     var reqQueue = s.Seeds
3     go func() {
4         for {
5             var req *collect.Request
6             var ch chan *collect.Request
```

```

7     if len(reqQueue) > 0 {
8         req = reqQueue[0]
9         reqQueue = reqQueue[1:]
10        ch = s.workerCh
11    }
12    select {
13    case r := <-s.requestCh:
14        reqQueue = append(reqQueue, r)
15
16        case ch <- req:
17        }
18    }
19 }()
20 }
21 }

```



通道还有一个特性，就是我们往 `nil` 通道中写入数据会陷入到堵塞的状态。因此，如果 `reqQueue` 为空，这时 `req` 和 `ch` 都是 `nil`，当前协程就会陷入到堵塞的状态，直到接收到新的请求才会被唤醒。

我们可以用一个例子来验证这一特性：

复制代码

```

1 func main() {
2     var ch chan *int
3     go func() {
4         <-ch
5     }()
6     select {
7     case ch <- nil:
8         fmt.Println("it's time")
9     }
10 }

```

在这个例子中，运行后会出现死锁，因为当前程序全部陷入到了无限堵塞的状态。

复制代码

```

1 fatal error: all goroutines are asleep - deadlock!

```

调度引擎除了启动 `schedule` 函数，还需要安排多个实际干活的 `worker` 程序。

下一步，让我们创建指定数量的 worker，完成实际任务的处理。其中 WorkCount 为执行任务的数量，可以灵活地去配置。

 复制代码

```
1 func (s *ScheduleEngine) Run() {
2     ...
3     go s.Schedule()
4     for i := 0; i < s.WorkCount; i++ {
5         go s.CreateWork()
6     }
7 }
```

这里的 CreateWork 创建出实际处理任务的函数，它又细分为下面几个步骤：

- \leftarrow s.workerCh 接收到调度器分配的任务；
- s.fetcher.Get 访问服务器，r.ParseFunc 解析服务器返回的数据；
- s.out \leftarrow result 将返回的数据发送到 out 通道中，方便后续的处理。

 复制代码

```
1 func (s *Schedule) CreateWork() {
2     for {
3         r :=  $\leftarrow$ s.workerCh
4         body, err := s.Fetcher.Get(r)
5         if err != nil {
6             s.Logger.Error("can't fetch ",
7                 zap.Error(err),
8             )
9             continue
10        }
11        result := r.ParseFunc(body, r)
12        s.out  $\leftarrow$  result
13    }
14 }
```

最后一步，我们需要单独安排一个函数来处理爬取并解析后的数据结构，完整的函数如下：

 复制代码

```
1 func (s *Schedule) HandleResult() {
2     for {
3         select {
```

```

4     case result := <-s.out:
5         for _, req := range result.Requests {
6             s.requestCh <- req
7         }
8         for _, item := range result.Items {
9             // todo: store
10            s.Logger.Sugar().Info("get result", item)
11        }
12    }
13 }
14 }

```



在 `HandleResult` 函数中，`<-s.out` 接收所有 worker 解析后的数据，其中要进一步爬取的 `Requests` 列表将全部发送回 `s.requestCh` 通道，而 `result.Items` 里包含了我们实际希望得到的结果，所以我们先用日志把结果打印出来。

最后，让我们用之前介绍的爬取豆瓣小组的例子来验证调度器的功能。

在 `main` 函数中，生成初始网址列表作为种子任务。构建 `engine.Schedule`，设置 worker 的数量，采集器 `Fetcher` 和日志 `Logger`，并调用 `s.Run()` 运行调度器。

复制代码

```

1 func main(){
2     var seeds []*collect.Request
3     for i := 0; i <= 0; i += 25 {
4         str := fmt.Sprintf("<https://www.douban.com/group/szsh/discussion?start=%d", i)
5         seeds = append(seeds, &collect.Request{
6             Url:      str,
7             WaitTime: 1 * time.Second,
8             Cookie:   "xxx",
9             ParseFunc: doubangroup.ParseURL,
10        })
11    }
12
13    var f collect.Fetcher = &collect.BrowserFetch{
14        Timeout: 3000 * time.Millisecond,
15        Logger:  logger,
16        Proxy:   p,
17    }
18
19    s := engine.Schedule{
20        WorkCount: 5,
21        Logger:    logger,
22        Fetcher:   f,
23        Seeds:     seeds,
24    }

```

```
25     s.Run()
26 }
27
```



天下无鱼

<https://shikey.com/>

输出结果为:

 复制代码

```
1 {"level":"INFO","ts":"2022-10-19T00:55:54.281+0800","caller":"engine/schedule.g
2 {"level":"INFO","ts":"2022-10-19T00:55:54.355+0800","caller":"engine/schedule.g
```

函数式选项模式

在上面的例子中，我们初始化 `engine.Schedule` 时将一系列的参数传递到了结构体当中。在实践中可能会有几十个参数等着我们赋值，从面向对象的角度来看，不同参数的灵活组合可能会带来不同的调度器类型。在实践中为了方便使用，开发者可能会创建非常多的 **API** 来满足不同场景的需要，如下所示：

 复制代码

```
1 // 基本调度器
2 func NewBaseSchedule() *Schedule {
3     return &Schedule{
4         WorkCount: 1,
5         Fetcher: baseFetch,
6     }
7 }
8 // 多worker调度器
9 func NewMultiWorkSchedule(workCount int) *Schedule {
10    return &Schedule{
11        WorkCount: workCount,
12        Fetcher: baseFetch,
13    }
14 }
15
16 // 代理调度器
17 func NewProxySchedule(proxy string) *Schedule {
18    return &Schedule{
19        WorkCount: 1,
20        Fetcher: proxyFetch(proxy),
21    }
22 }
```

随着参数的不断增多，这种 **API** 会变得越来越来多，这就增加了开发者的心理负担。

另一种使用方式就是传递一个统一的 **Config** 配置结构，如下所示。这种方式只需要创建单个 **API**，但是需要在内部对所有的变量进行判断，繁琐且不优雅。对于使用者来说，也很难确定自己需要使用哪一个字段。



复制代码

```
1 type Config struct {
2     WorkCount int
3     Fetcher   collect.Fetcher
4     Logger    *zap.Logger
5     Seeds     []*collect.Request
6 }
7
8 func NewSchedule(c *Config) *Schedule {
9     var s = &Schedule{}
10    if c.Seeds != nil {
11        s.Seeds = c.Seeds
12    }
13    if c.Fetcher != nil {
14        s.Fetcher = c.Fetcher
15    }
16
17    if c.Logger != nil {
18        s.Logger = c.Logger
19    }
20    ...
21    return s
22 }
```

那么有没有方法可以更加优雅地处理这种多参数配置问题呢？

Rob Pike 在 2014 年的 [一篇博客中](#)提到了一种优雅的处理方法叫做**函数式选项模式 (Functional Options)**。这种模式展示了闭包函数的有趣用途，目前在很多开源库中都能看到它的身影，我们项目中使用的日志库 **Zap** 也使用了这种模式。下面我以上面 **schedule** 的配置为例来说明函数式选项模式（完整代码请查看 [tag v0.1.5](#)）。

第一步，我们要对 **schedule** 结构进行改造，把可以配置的参数放入到 **options** 结构中：

复制代码

```
1 type Schedule struct {
2     requestCh chan *collect.Request
3     workerCh  chan *collect.Request
4     out       chan collect.ParseResult
5     options
```

```
6 }
7
8 type options struct {
9     WorkCount int
10    Fetcher    collect.Fetcher
11    Logger     *zap.Logger
12    Seeds      []*collect.Request
13 }
```



第二步，我们需要书写一系列的闭包函数，这些函数的返回值是一个参数为 `options` 的函数：

 复制代码

```
1
2 type Option func(opts *options)
3
4 func WithLogger(logger *zap.Logger) Option {
5     return func(opts *options) {
6         opts.Logger = logger
7     }
8 }
9
10 func WithFetcher(fetcher collect.Fetcher) Option {
11     return func(opts *options) {
12         opts.Fetcher = fetcher
13     }
14 }
15
16 func WithWorkCount(workCount int) Option {
17     return func(opts *options) {
18         opts.WorkCount = workCount
19     }
20 }
21
22 func WithSeeds(seed []*collect.Request) Option {
23     return func(opts *options) {
24         opts.Seeds = seed
25     }
26 }
```

第三步，创建一个生成 `schedule` 的新函数，函数参数为 `Option` 的可变参数列表。

`defaultOptions` 为默认的 `Option`，代表默认的参数列表，然后循环遍历可变函数参数列表并执行。

 复制代码


```
1 func NewSchedule(opts ...Option) *Schedule {
2     options := defaultOptions
3     for _, opt := range opts {
4         opt(&options)
5     }
6     s := &Schedule{}
7     s.options = options
8     return s
9 }
```



第四步，在 `main` 函数中调用 `NewSchedule`。让我们来看看函数式选项模式的效果：

```
1 func main(){
2     s := engine.NewSchedule(
3         engine.WithFetcher(f),
4         engine.WithLogger(logger),
5         engine.WithWorkCount(5),
6         engine.WithSeeds(seeds),
7     )
8     s.Run()
9 }
```

复制代码

从这个例子中，我们可以看到函数式选项模式的好处：

- API 具有可扩展性，高度可配置化，新增参数不会破坏现有代码；
- 参数列表非常简洁，并且可以使用默认的参数；
- `option` 函数使参数的含义非常清晰，易于开发者理解和使用；
- 如果将 `options` 结构中的参数设置为小写，还可以限制这些参数的权限，防止这些参数在 `package` 外部使用。

刚才，我们实战了 `fan-in` 和 `fan-out` 高并发模型，并深度使用了通道和 `select` 的机制。接下来让我们更进一步，看一下实现通道和 `select` 机制的原理，掌握这种高并发模型的底层图像。

通道的底层原理

通道的实现并没有想象中复杂。它利用互斥锁实现了并发安全，只不过 `Go` 运行时为我们屏蔽了底层的细节。**通道包括两种类型，一种是无缓冲的通道，另一种是带缓冲区的通道。**通道的结构如下：



通道字段	含义
qcount	通道队列中的数据个数
dataqsiz	通道队列中的数据大小
buf	存放实际数据的指针
elemsize	通道类型大小
closed	通道是否关闭
elemtype	通道类型
sendx	记录发送者在buf中的序号
recvx	记录接受者在buf中的序号
recvq	读取的阻塞协程队列
sendq	写入的阻塞协程队列
lock	锁，并发保护

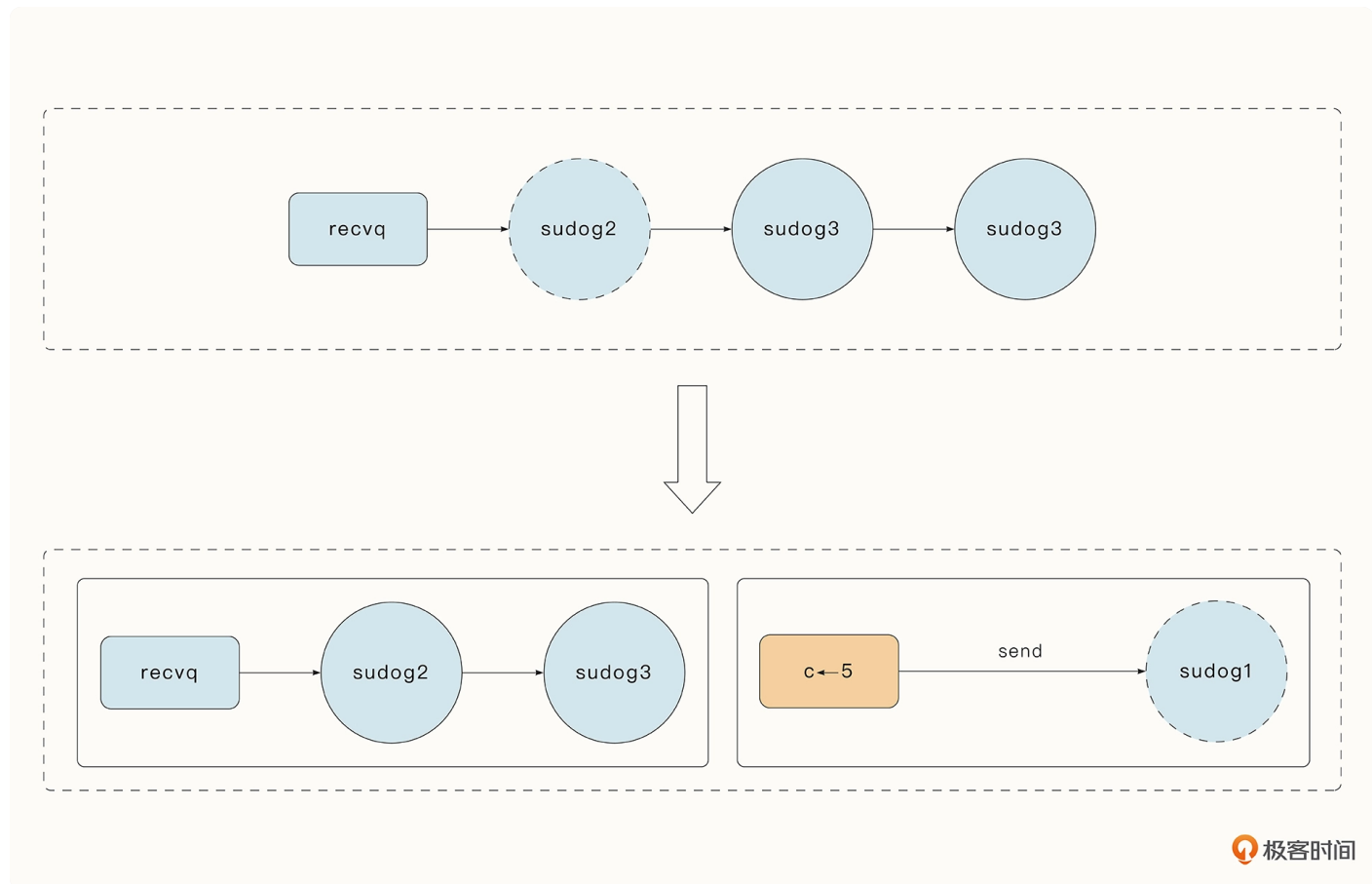
可以看到，通道中包含了数据的类型、大小、数量，堵塞协程队列，以及用于缓存区的诸多字段。

我们先来看看无缓冲区的通道是怎么实现的。

无缓冲区的通道

通道需要有多个协程分别完成读和写的功能，这样才能保证数据传输是顺畅的。对于无缓冲区的通道来说，如果有一个协程正在将数据写入通道，但是当前没有协程读取数据，那么写入协程将立即陷入到休眠状态。写入协程堵塞之前协程会被封装到 `sudog` 结构中，并存储到写入的堵塞队列 `sendq` 中，之后协程陷入休眠。

之前我们介绍过，协程的堵塞是位于用户态的，协程切换时，运行时保存当前协程的状态，并调用 `gopark` 函数切换到 `g0` 完成新一轮的调度。如果之后有协程读取数据，那么读取协程会立即读取 `sendq` 队列中第一个等待的协程，并将该协程对应的元素拷贝到读取协程中，同时调用 `goready` 唤醒写入协程，将写入协程放入到可运行队列中等待被调度器调度。



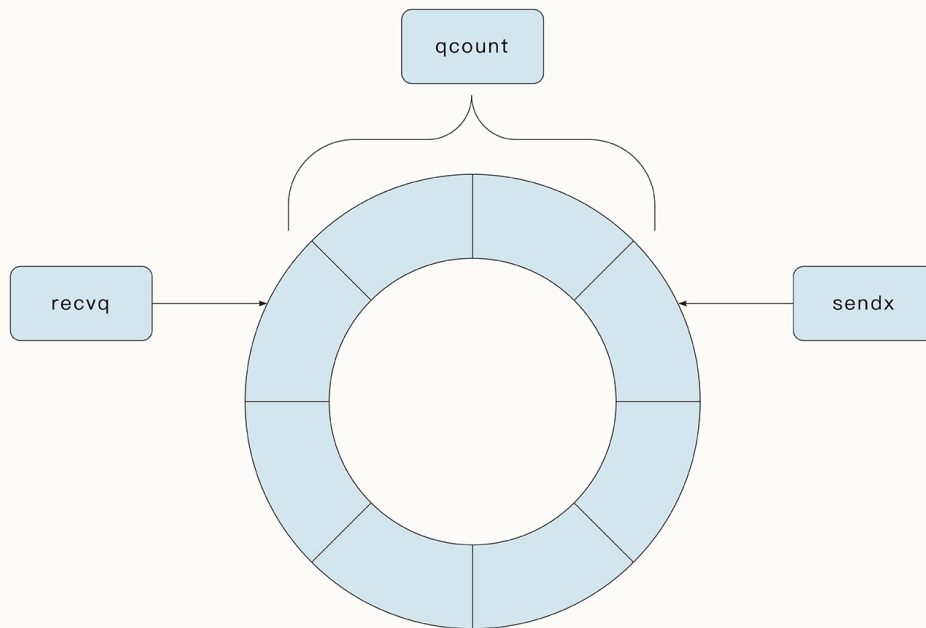
带缓冲区的通道

而对于带缓冲区的通道来说，假设缓存队列的数量为 N ，那么如果写入的数据量不大于 N ，写入协程就不会陷入到休眠状态，所有数据都会存储在缓冲队列中。

缓冲队列可以在一定程度上削峰填谷，加快处理速度。但是如果写入速度始终大于读取数据，那么缓冲区迟早也有写满的时候，到时候仍然会陷入堵塞，只是延迟了问题的暴露并带来内存的浪费。因此缓冲区的容量不可以过大，我们可以根据实际情况给出一个经验值。例如上面的爬虫案例中，我们就可以给接收任务的 `requestCh` 通道加上缓存区，先将缓存区设置为 500，这样就不会频繁堵塞住调度器了。

对于有缓存的通道，存储在 `buf` 中的数据虽然是线性的数组，但是这些数组和序号 `recvx`、`recvq` 模拟了一个环形队列。`recvx` 可以定位到 `buf` 是从哪个位置读取的通道中的元素，而

sendx 则能够找到写入时放入 buf 的位置，这样做主要是为了再次利用已经使用过的空间。从 recvx 到 sendx 的距离 qcount 就是通道队列中的元素数量。



Select 机制的底层原理

在前面的实战案例中，我们还看到了大量 channel 与 select 结合使用的场景。

受到通道特性的限制，如果单个通道被堵塞，协程就无法继续执行了。那有没有一种机制可以像网络中的多路复用一样，监听多个通道，使后续处理协程能够及时地运行？

其实就和网络中把 select 用于 Socket 的多路复用机制一样，Go 中也可以用 select 语句实现这样的多路复用机制。select 语句中的每一个 case 都对应着一个待处理的读取或写入通道。举个简单实用的例子，下面的程序如果 800 毫秒之后也接受不到通道 c 中的数据，定时器 time.After 就会接收到数据，从而打印 timeout。

复制代码

```
1 select {
2     case <-c:
3         fmt.Println("random 01")
4     case <-time.After(800 * time.Millisecond):
5         fmt.Println("timeout")
6 }
```

借助 `select` 可以实现许多有表现力的设计，那 `select` 是如何工作的呢？

`Select` 底层调用了 `selectgo` 函数，它的工作可以分为三个部分：



- 第一部分涉及到遍历。`selectgo` 首先循环查找当前准备就绪的通道，如果能够找到，则正常进行后续处理。在具体的实现方式上，由于 `select` 内部的 `scases` 数组存储了所有需要管理的通道，所以很容易想到循环遍历 `scases`，最终找到可用的通道。不过这可能导致一个问题，那就是如果前面的通道始终有数据，后面的通道将永远得不到执行的机会。为了解决这一问题，Go 语言为 `select` 加入了随机性，会利用洗牌算法随机打散数组顺序，保证了所有通道都有执行的机会。
- 第二部分涉及到协程的休眠。如果 `select` 找不到准备就绪的通道，这时和单个协程的堵塞一样，它会将当前协程放入到所有通道的等待队列中，并陷入到休眠状态。
- 第三部分涉及到协程的唤醒。如果有任意一个通道准备就绪，当前的协程将会被唤醒，并到准备就绪的 `case` 处继续执行。要注意的一点是，最后 `selectgo` 会将 `sudog` 结构体从其他通道的等待队列中移出，因为当前协程已经能够正常运行，不再需要被其他通道唤醒了。

总结


这节课，我们用 `fan-in`、`fan-out` 并发模式实战了爬虫的任务调度器。在实战中，我们频繁使用了通道与 `select` 结合的方式，还深入底层看到了通道与 `select` 的原理。最后我们还学习了函数选项模式在构建 API 时的优势。在后面的项目中，我们还会频繁地用到这种特性。

课后题

在我们的课程中，`schedule` 函数其实有一个 `bug`，您能看出来吗？你觉得可以用什么方式找出这样的 `Bug`？

欢迎你在留言区与我交流讨论，我们下节课见！

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

上一篇 27 | 掘地三尺：实战深度与广度优先搜索算法

下一篇 29 | 细节决定成败：切片与哈希表的陷阱与原理

精选留言 (6)

 写留言



shuff1e

2022-12-15 来自北京

bug应该是，会丢失发给worker的任务。

case r := <-s.requestCh:的情况下，如果req不是nil，应该把req再添加到reqQueue头部



Realm

2022-12-14 来自浙江

...

```
func (s *ScheduleEngine) Schedule() {  
    var reqQueue = s.Seeds  
    go func() {  
        for {  
            var req *collect.Request  
            //var ch chan *collect.Request  
            ch := make(chan *collect.Request)  
            ...  
        }  
    }  
}
```

使用make创建ch，这样ch就不是nil了，即使reqQueue为空的时候，case ch <- req:就不是往nil通道中写数据了。



老猫

2022-12-13 来自江苏

```
// s.requestCh = make(chan *collect.Request,100)  
// s.workerCh = make(chan *collect.Request,500)  
func (s *ScheduleEngine) Schedule() {  
    var reqQueue = s.Seeds  
    go func() {  
        for _, req := range reqQueue {
```

```
s.workerCh <- req
```

```
}
```

```
for {
```

```
  select {
```

```
    case r := <-s.requestCh:
```

```
      s.workerCh <- r
```

```
  }
```

```
}
```

```
}()
```

```
}
```



天下无鱼

<https://shikey.com/>



拾掇拾掇

2022-12-13 来自浙江

开启go run 的datarace参数吗?



Realm

2022-12-13 来自浙江

Seeds

|

req

ParseFunc(req)

HandleResult()

requestCh----> reqQueue ----> workerCh -----> out-----> result:

^

- item ==> 存储

|

- req |

|-----<-----<-----<-----|



Geek_a9ea01

2022-12-13 来自广东

```
for { var req *collect.Request var ch chan *collect.Request if len(reqQueue) > 0 { req = reqQueue[0] reqQueue = reqQueue[1:] ch = s.workerCh } select { case r := <-s.requestCh: reqQueue = append(reqQueue, r) case ch <- req: } }
```

有个问题:

如果ch堵塞了 这时候又有requestCh请求上来; 会不会导致ch数据丢失?



