

48 | 完善核心能力：Master请求转发与Worker资源管理

2023-01-31 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 07:50 大小 7.16M



你好，我是郑建勋。

这节课，让我们继续优化 Master 服务，实现 Master 请求转发和并发情况下的资源保护，同时实现 Worker 对分配资源的监听。

将 Master 请求转发到 Leader

首先我们需要考虑一下，当 Master 是 Follower 状态，同时还接收到了请求的情形。在之前的设计中，为了避免并发处理时可能出现的异常情况，我们只打算让 Leader 来处理请求。所以，当 Master 节点接收到请求时，如果当前节点不是 Leader，我们可以直接报错，由客户端选择正确的 Leader 节点。如下所示。

复制代码

```
1 func (m *Master) AddResource(ctx context.Context, req *proto.ResourceSpec, resp
2     if !m.IsLeader() {
```

```
3     return errors.New("no leader")
4 }
5 }
```



天下无鱼

<https://shikey.com/>

我们还可以采用另外一种更常见的方式：将接收到的请求转发给 Leader。要实现这一点，首先所有 Master 节点要在 Leader 发生变更时，将当前最新的 Leader 地址保存到 leaderID 中。

 复制代码

```
1 func (m *Master) Campaign() {
2     select {
3     case resp := <-leaderChange:
4         m.logger.Info("watch leader change", zap.String("leader:", string(resp.Kv
5             m.leaderID = string(resp.Kvs[0].Value)
6     }
7     for {
8         select {
9         case err := <-leaderCh:
10             m.leaderID = m.ID
11         case resp := <-leaderChange:
12             if len(resp.Kvs) > 0 {
13                 m.logger.Info("watch leader change", zap.String("leader:", string(res
14                     m.leaderID = string(resp.Kvs[0].Value)
15             }
16         }
17     }
```

在处理请求前，首先判断当前 Master 的状态，如果它不是 Leader，就获取 Leader 的地址并完成请求的转发。注意，这里如果不指定 Leader 的地址，go-micro 就会随机选择一个地址进行转发。

 复制代码

```
1 func (m *Master) AddResource(ctx context.Context, req *proto.ResourceSpec, resp
2     if !m.IsLeader() && m.leaderID != "" && m.leaderID != m.ID {
3         addr := getLeaderAddress(m.leaderID)
4         nodeSpec, err := m.forwardCli.AddResource(ctx, req, client.WithAddress(addr
5         resp.Id = nodeSpec.Id
6         resp.Address = nodeSpec.Address
7         return err
8     }
9
10    nodeSpec, err := m.addResources(&ResourceSpec{Name: req.Name})
11    if nodeSpec != nil {
12        resp.Id = nodeSpec.Node.Id
```

```
13     resp.Address = nodeSpec.Node.Address
14 }
15 return err
16 }
```



在转发时，我们使用了 micro 生成的 GRPC 客户端，这是通过在初始化时导入 micro GRPC client 的插件实现的。SetForwardCli 方法将生成的 GRPC client 注入到了 Master 结构体中。

复制代码

```
1 import (
2     grpccli "github.com/go-micro/plugins/v4/client/grpc"
3 )
4
5 func RunGRPCServer(m *master.Master, logger *zap.Logger, reg registry.Registry,
6     service := micro.NewService(
7     ...
8     micro.Client(grpccli.NewClient()),
9 )
10
11 cl := proto.NewCrawlerMasterService(cfg.Name, service.Client())
12 m.SetForwardCli(cl)
13 }
```

接下来，我们来验证一下服务是否能够正确地转发。

首先启动一个 Worker 和一个 Master 服务，当前的 Leader 会变成 master2，IP 地址为 192.168.0.105:9091。

复制代码

```
1 » go run main.go worker --pprof=:9983
2 » go run main.go master --id=2 --http=:8081 --grpc=:9091
```

现在我们启动一个新的 Master 服务 master3。

复制代码

```
1 » go run main.go master --id=3 --http=:8082 --grpc=:9092 --pprof=:9982
```

接着访问 master3 服务暴露的 HTTP 接口。虽然 master3 并不是 Leader，但是访问 master3 添加资源时，操作仍然能够成功。



复制代码

```
1 » curl --request POST 'http://localhost:8082/crawler/resource' --header 'Content-Type: application/json' --data '{\"id\": \"go.micro.server.worker-1\", \"Address\": \"192.168.0.105:9090\"}'
```

同时，我们在 Leader 服务的日志中能够看到请求信息，验证成功。

复制代码

```
1 {\"level\": \"INFO\", \"ts\": \"2022-12-29T17:23:55.792+0800\", \"caller\": \"master/master.go:100\", \"msg\": \"Resource added successfully\"}
```

资源保护

由于 Worker 节点与 Resource 资源一直在动态变化当中，因此如果不考虑数据的并发安全，在复杂线上场景下，就可能出现很多难以解释的现象。

为了避免数据的并发安全问题，我们之前利用了通道来进行协程间的通信，但如果我们现在希望保护 Worker 节点与 Resource 资源，其实当前场景下更好的方式是使用原生的互斥锁。这是因为我们只希望在关键位置加锁，其他的逻辑仍然是并行的。如果我们在读取一个变量时还要用通道来通信，代码会变得不优雅。

我们来看下使用原生互斥锁的操作是怎样的。如下，在 Master 中添加 sync.Mutex 互斥锁，用于资源的并发安全。

复制代码

```
1 type Master struct {
2     ...
3     ID          string
4     rlock       sync.Mutex
5
6     options    []Option
7 }
```

我们可以在资源更新（资源加载与增删查改）、Worker 节点更新、资源分配的阶段都加入互斥锁如下所示。



```
1 func (m *Master) DeleteResource(ctx context.Context, spec *proto.ResourceSpec,
2     m.rlock.Lock()
3     defer m.rlock.Unlock()
4
5     r, ok := m.resources[spec.Name]
6     ...
7 }
8
9 func (m *Master) AddResource(ctx context.Context, req *proto.ResourceSpec, resp
10     ...
11     m.rlock.Lock()
12     defer m.rlock.Lock()
13     nodeSpec, err := m.addResources(&ResourceSpec{Name: req.Name})
14     if nodeSpec != nil {
15         resp.Id = nodeSpec.Node.Id
16         resp.Address = nodeSpec.Node.Address
17     }
18     return err
19 }
20
21 func (m *Master) updateWorkNodes() {
22     services, err := m.registry.GetService(worker.ServiceName)
23     if err != nil {
24         m.logger.Error("get service", zap.Error(err))
25     }
26
27     m.rlock.Lock()
28     defer m.rlock.Unlock()
29     ...
30     m.workNodes = nodes
31 }
32
33 func (m *Master) loadResource() error {
34     resp, err := m.etcdCli.Get(context.Background(), RESOURCEPATH, clientv3.WithP
35     ...
36     resources := make(map[string]*ResourceSpec)
37     m.rlock.Lock()
38     defer m.rlock.Unlock()
39     m.resources = resources
40 }
41
42 func (m *Master) reAssign() {
43     rs := make([]*ResourceSpec, 0, len(m.resources))
44
45     m.rlock.Lock()
46     defer m.rlock.Unlock()
47
48     for _, r := range m.resources {...}
49
50     for _, r := range rs {
51         m.addResources(r)
```

```
52     }  
53 }
```



天下无鱼

<https://shikey.com/>

当外部访问 **Leader** 的 **HTTP** 接口时，实际上服务端会开辟一个协程并发处理请求。通过使用互斥锁，我们消除了并发访问同一资源可能出现的问题。在实践中，需要合理地使用互斥锁，尽量让锁定的范围足够小，锁定的资源足够少，减少锁等待的时间。

Worker 单机模式

接下来我们回到 **Worker**。**Worker** 可以有两种模式，集群模式与单机模式。我们可以在 **Worker** 中加一个 **flag** 来切换 **Worker** 运行的模式。

对于少量的任务，可以直接用单机版的 **Worker** 来处理，种子节点来自于配置文件。而对于集群版的 **Worker**，任务将来自 **Master** 的分配。

要切换 **Worker** 模式只要判断一个 **flag** 值 **cluster** 即可做到。如下所示，在启动 **Worker** 时，如果 **cluster** 为 **false**，代表为单机模式。如果 **cluster** 为 **true**，代表是集群模式。

 复制代码

```
1  
2 WorkerCmd.Flags().BoolVar(  
3     &cluster, "cluster", true, "run mode")  
4 var cluster bool  
5  
6 func (c *Crawler) Run(cluster bool) {  
7     if !cluster {  
8         c.handleSeeds()  
9     }  
10  
11     go c.Schedule()  
12     for i := 0; i < c.WorkCount; i++ {  
13         go c.CreateWork()  
14     }  
15     c.HandleResult()  
16 }
```

Worker 集群模式

在集群模式下，我们还需要书写 **Worker** 加载和监听 **etcd** 资源这一重要的功能。首先来看看初始化时的资源加载，在初始时，我们生成了 **etcd client**，并注入到 **Crawler** 结构中。

```

1 endpoints := []string{e.registryURL}
2 cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
3 if err != nil {
4     return nil, err
5 }
6 e.etcdCli = cli

```



资源加载

`Crawler.loadResource` 方法用于从 `etcd` 中加载资源。我们调用 `etcd Get` 方法，获取前缀为 `/resources` 的全量资源列表。解析这些资源，查看当前资源分配的节点是否为当前节点。如果分配的节点和当前节点匹配，意味着当前资源是分配给当前节点的，不是当前节点的资源将会被直接忽略。

```

1 func (c *Crawler) loadResource() error {
2     resp, err := c.etcdCli.Get(context.Background(), master.RESOURCEPATH, clientv3.
3     if err != nil {
4         return fmt.Errorf("etcd get failed")
5     }
6
7     resources := make(map[string]*master.ResourceSpec)
8     for _, kv := range resp.Kvs {
9         r, err := master.Decode(kv.Value)
10        if err == nil && r != nil {
11            id := getID(r.AssignedNode)
12            if len(id) > 0 && c.id == id {
13                resources[r.Name] = r
14            }
15        }
16    }
17    c.Logger.Info("leader init load resource", zap.Int("length", len(resources)))
18    c.rlock.Lock()
19    defer c.rlock.Unlock()
20    c.resources = resources
21    for _, r := range c.resources {
22        c.runTasks(r.Name)
23    }
24
25    return nil
26 }

```

资源加载完毕后，分配给当前节点的任务会执行 `runTask` 方法，通过任务名从全局任务池中获取爬虫任务，调用 `t.Rule.Root()` 获取种子请求，并放入到调度器中执行。



 复制代码

```
1 func (c *Crawler) runTasks(taskName string) {
2     t, ok := Store.Hash[taskName]
3     if !ok {
4         c.Logger.Error("can not find preset tasks", zap.String("task name", taskName))
5         return
6     }
7     res, err := t.Rule.Root()
8
9     if err != nil {
10        c.Logger.Error("get root failed",
11            zap.Error(err),
12        )
13        return
14    }
15
16    for _, req := range res {
17        req.Task = t
18    }
19    c.scheduler.Push(res...)
20 }
```

资源监听

除了加载资源，在初始化时我们还需要开辟一个新的协程 `c.watchResource` 来监听资源的变化。

 复制代码

```
1 func (c *Crawler) Run(id string, cluster bool) {
2     c.id = id
3     if !cluster {
4         c.handleSeeds()
5     }
6     go c.loadResource()
7     go c.watchResource()
8     go c.Schedule()
9     for i := 0; i < c.WorkCount; i++ {
10        go c.CreateWork()
11    }
12    c.HandleResult()
13 }
```


如下所示，我在 `watchResource` 函数中书写了一个监听新增资源的功能。`watchResource` 借助 `etcd client` 的 `Watch` 方法监听资源的变化。`Watch` 返回值是一个通道，当 `etcd client` 监听到 `etcd` 中前缀为 `/resources` 的资源发生变化时，就会将信息写入到通道 `Watch` 中。通过通道返回的信息，不仅能够得到当前有变动的资源最新的值，还可以得知当前资源变动的事件是新增、更新还是删除。如果是新增事件，那就调用 `runTasks` 启动该资源对应的爬虫任务。

 复制代码

```
1 func (c *Crawler) watchResource() {
2     watch := c.etcdCli.Watch(context.Background(), master.RESOURCEPATH, clientv3.
3     for w := range watch {
4         if w.Err() != nil {
5             c.Logger.Error("watch resource failed", zap.Error(w.Err()))
6             continue
7         }
8         if w.Canceled {
9             c.Logger.Error("watch resource canceled")
10            return
11        }
12        for _, ev := range w.Events {
13            spec, err := master.Decode(ev.Kv.Value)
14            if err != nil {
15                c.Logger.Error("decode etcd value failed", zap.Error(err))
16            }
17
18            switch ev.Type {
19            case clientv3.EventTypePut:
20                if ev.IsCreate() {
21                    c.Logger.Info("receive create resource", zap.Any("spec", spec))
22                } else if ev.IsModify() {
23                    c.Logger.Info("receive update resource", zap.Any("spec", spec))
24                }
25                c.runTasks(spec.Name)
26            case clientv3.EventTypeDelete:
27                c.Logger.Info("receive delete resource", zap.Any("spec", spec))
28            }
29        }
30    }
31 }
32 }
```

现在让我们来验证一下新增资源的功能，启动 `Master` 与 `Worker` 节点。

 复制代码

```
1 » go run main.go master --id=3 --http=:8082 --grpc=:9092 --pprof=:9982
2 » go run main.go worker --pprof=:9983
```

紧接着调用 Master 的添加资源接口。



复制代码

```
1 » curl -H "content-type: application/json" -d '{"id":"zjx","name": "douban_book"
2 {"id":"go.micro.server.worker-1", "Address":"192.168.0.105:9090"}
```

可以看到，Worker 日志中任务开始正常地执行了，验证成功。完整代码你可以查看 [v0.4.1 分支](#)。

复制代码

```
1 {"level":"DEBUG","ts":"2022-12-30T21:06:56.743+0800","caller":"doubanbook/book.
2 {"level":"DEBUG","ts":"2022-12-30T21:07:00.532+0800","caller":"doubanbook/book.
3 {"level":"DEBUG","ts":"2022-12-30T21:07:04.240+0800","caller":"doubanbook/book.
```

资源删除

接下来让我们继续看看如何删除一个爬虫任务。

我们需要在 Watch 的选项中设置 `clientv3.WithPrevKV()`，这样，当监听到资源的删除时，就能够获取当前删除的资源信息，接着就可以调用 `c.deleteTasks` 来删除任务了。

复制代码

```
1 func (c *Crawler) watchResource() {
2     watch := c.etcdCli.Watch(context.Background(), master.RESOURCEPATH, clientv3.
3     for w := range watch {
4         for _, ev := range w.Events {
5             switch ev.Type {
6                 ...
7             case clientv3.EventTypeDelete:
8                 spec, err := master.Decode(ev.PrevKv.Value)
9                 c.Logger.Info("receive delete resource", zap.Any("spec", spec))
10                if err != nil {
11                    c.Logger.Error("decode etcd value failed", zap.Error(err))
12                }
13                c.rlock.Lock()
14                c.deleteTasks(spec.Name)
15                c.rlock.Unlock()
16            }
17        }
18    }
```

```
19 }
```



`deleteTasks` 会删除 `c.resources` 中存储的当前 Task，并且将 Task 的 `Closed` 变量设置为 `true`。

复制代码

```
1 func (c *Crawler) deleteTasks(taskName string) {
2     t, ok := Store.Hash[taskName]
3     if !ok {
4         c.Logger.Error("can not find preset tasks", zap.String("task name", taskName))
5         return
6     }
7     t.Closed = true
8     delete(c.resources, taskName)
9 }
```

我们在 `Task` 中设计了一个新的变量 `Closed` 用于标识当前的任务是否已经被删除了。这是因为被删除的任务可能现在还在运行当中，我们通过该变量确认它已经不再运行了。

在一些场景中，我们也可以将标识任务是否已经结束的变量设计为通道类型或者 `context.Context`，然后与 `select` 语句结合起来实现多路复用。我们在 `HTTP` 标准库中也经常看到这种用法，它可以判断通道的事件与其他事件哪一个先发生。

复制代码

```
1 type Task struct {
2     Visited      map[string]bool
3     VisitedLock sync.Mutex
4
5     //
6     Closed bool
7
8     Rule RuleTree
9     Options
10 }
```

不过我们这里使用一个标识任务是否关闭的 `bool` 类型就足够了。在任务流程的核心位置，我们都需要检测该变量。检测到任务关闭时，就不再执行后续的流程。


具体操作是在 `request.Check` 方法中，加入对任务是否关闭的判断。

```
1 func (r *Request) Check() error {
2     if r.Depth > r.Task.MaxDepth {
3         return errors.New("max depth limit reached")
4     }
5
6     if r.Task.Closed {
7         return errors.New("task has Closed")
8     }
9
10    return nil
11 }
```



接着，在任务的采集和调度的两个核心位置检测任务的有效性。一旦发现任务已经被关闭，它所有的请求将不再被调度和采集。

```
1 func (c *Crawler) CreateWork() {
2     for {
3         req := c.scheduler.Pull()
4         if err := req.Check(); err != nil {
5             c.Logger.Debug("check failed",
6                 zap.Error(err),
7             )
8
9             continue
10        }
11    }
12
13    func (s *Schedule) Schedule() {
14        var ch chan *spider.Request
15
16        var req *spider.Request
17
18        for {
19            ...
20            // 请求校验
21            if req != nil {
22                if err := req.Check(); err != nil {
23                    zap.S().Debug("check failed",
24                        zap.Error(err),
25                    )
26                    req = nil
27                    ch = nil
28                    continue
29                }
30            }
31        }
```

下面让我们来验证一下任务的删除功能是否正常，首先启动一个 Master 和一个 Worker 服务。 <https://shikey.com/>

 复制代码

```
1 » go run main.go master --id=3 --http=:8082 --grpc=:9092 --pprof=:9982
2 » go run main.go worker --pprof=:9983
```

紧接着，调用 Master 的添加资源接口，可以看到爬虫任务是正常执行的。

 复制代码

```
1 » curl -H "content-type: application/json" -d '{"id":"zjx","name": "douban_book"
2 {"id":"go.micro.server.worker-1", "Address":"192.168.0.105:9090"}
```

然后，我们调用 Master 的删除资源接口，从 Worker 中的日志可以看到，Worker 监听到了删除资源的事件。在日志打印出 "task has Closed" 的错误信息之后，删除的爬虫任务将不再运行。

 复制代码

```
1 {"level":"INFO","ts":"2022-12-31T14:06:33.528+0800","caller":"engine/schedule.g
2 {"level":"DEBUG","ts":"2022-12-31T14:06:33.845+0800","caller":"engine/schedule.
3 {"level":"DEBUG","ts":"2022-12-31T14:06:33.845+0800","caller":"engine/schedule.
```

此后，当我们再次调用 Master 的添加资源接口时，爬虫任务又将恢复如初。删除功能验证成功。

总结

好了，这节课，我们设计了将 Master 请求转发到 Leader 的功能，让所有的 Master 都具备了接收请求的能力。此外，我们还使用了原生的互斥锁解决了并发安全问题。因为通道并不总是解决并发安全问题的最佳方式，在这里如果我们使用通道会减慢程序的并发性，使代码变得不优雅。

最后，我们还实现了在 **Worker** 集群模式下任务的加载与监听。在初始化时，我们通过加载 **etcd** 中属于当前节点的资源获取了全量的爬虫任务。我们还启动了对 **etcd** 资源的监听，实现了资源的动态添加和删除。至此，**Master** 与 **Worker** 的核心功能与交互都已经能够正常工作了。

课后题

最后，还是给你留一道思考题。

在我们的设计中，默认一个爬虫任务是不能够被添加多次的。那有没有一种场景，可以让同一个爬虫任务添加多次，也就是让多个 **Worker** 可以同时执行同一个爬虫任务呢？如果有这样的场景，我们应该如何修改设计？

欢迎你在留言区与我交流讨论，我们下节课再见！

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

 赞 2  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 47 | 故障容错：如何在Worker崩溃时进行重新调度？

下一篇 49 | 服务治理：如何进行限流、熔断与认证？

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。