

46 | Master任务调度：服务发现与资源管理

2023-01-24 郑建勋 来自北京



天下无鱼

<https://shikey.com/>

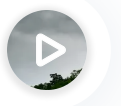
《Go进阶·分布式爬虫实战》

[课程介绍 >](#)



讲述：郑建勋

时长 05:35 大小 5.10M



你好，我是郑建勋。

在上一节课程中，我们实现了 Master 的选主，这一节课，我们继续深入 Master 的开发，实现一下 Master 的服务发现与资源的管理。

Master 服务发现

首先我们来实现一下 Master 对 Worker 的服务发现。

Master 需要监听 Worker 节点的信息，感知到 Worker 节点的注册与销毁。和服务的注册一样，我们的服务发现也使用 micro 提供的 registry 功能，代码如下所示。

m.WatchWorker 方法调用 registry.Watch 监听 Worker 节点的变化，watch.Next() 会堵塞等待节点的下一个事件，当 Master 收到节点变化事件时，将事件发送到 workerNodeChange 通

道。m.Campaign 方法接收到变化事件后，会用日志打印出变化的信息。



天下无鱼

<https://mkey.com/>

```
1
2 func (m *Master) Campaign() {
3     ...
4     workerNodeChange := m.WatchWorker()
5
6     for {
7         select {
8             ...
9             case resp := <-workerNodeChange:
10                m.logger.Info("watch worker change", zap.Any("worker:", resp))
11        }
12    }
13 }
14
15 func (m *Master) WatchWorker() chan *registry.Result {
16     watch, err := m.registry.Watch(registry.WatchService(worker.ServiceName))
17     if err != nil {
18         panic(err)
19     }
20     ch := make(chan *registry.Result)
21     go func() {
22         for {
23             res, err := watch.Next()
24             if err != nil {
25                 m.logger.Info("watch worker service failed", zap.Error(err))
26                 continue
27             }
28             ch <- res
29         }
30     }()
31     return ch
32 }
33 }
```

Master 中的 etcd registry 对象是我们在初始化时注册到 go-micro 中的。

 复制代码

```
1 // cmd/master/master.go
2 reg := etcd.NewRegistry(registry.Addrs(sconfig.RegistryAddress))
3 master.New(
4     masterID,
5     master.WithLogger(logger.Named("master")),
6     master.WithGRPCAddress(GRPCListenAddress),
7     master.WithregistryURL(sconfig.RegistryAddress),
```

```
8     master.WithRegistry(reg),
9     master.WithSeeds(seeds),
10 )
```



深入 go-micro registry 接口

go-micro 提供的 registry 接口提供了诸多 API，其结构如下所示。

复制代码

```
1 type Registry interface {
2     Init(...Option) error
3     Options() Options
4     Register(*Service, ...RegisterOption) error
5     Deregister(*Service, ...DeregisterOption) error
6     GetService(string, ...GetOption) ([]*Service, error)
7     ListServices(...ListOption) ([]*Service, error)
8     Watch(...WatchOption) (Watcher, error)
9     String() string
10 }
```

对于 Master 的服务发现，我们借助了 registry.Watch 方法。Watch 方法借助 client.Watch 实现了对特定 Key 的监听，并封装了 client.Watch 返回的结果。

复制代码

```
1 func (e *etcdRegistry) Watch(opts ...registry.WatchOption) (registry.Watcher, error) {
2     return newEtcdWatcher(e, e.options.Timeout, opts...)
3 }
4
5 func newEtcdWatcher(r *etcdRegistry, timeout time.Duration, opts ...registry.WatchOption) (registry.Watcher, error) {
6     var wo registry.WatchOptions
7     for _, o := range opts {
8         o(&wo)
9     }
10    watchPath := prefix
11    if len(wo.Service) > 0 {
12        watchPath = servicePath(wo.Service) + "/"
13    }
14    return &etcdWatcher{
15        stop:    stop,
16        w:       r.client.Watch(ctx, watchPath, clientv3.WithPrefix(), clientv3.WithWatch()),
17        client:  r.client,
18        timeout: timeout,
19    }, nil
20 }
```

registry.Watch 方法返回了 Watcher 接口，Watcher 接口中有 Next 方法用于完成事件的迭代。



复制代码

```
1 type Watcher interface {
2     // Next 堵塞调用
3     Next() (*Result, error)
4     Stop()
5 }
```

go-micro 的 etcd 插件库实现的 Next 方法也比较简单，只要监听 client.Watch 返回的通道，并将事件信息封装后返回即可。

复制代码

```
1 func (ew *etcdWatcher) Next() (*registry.Result, error) {
2     for wresp := range ew.w {
3         if wresp.Err() != nil {
4             return nil, wresp.Err()
5         }
6         if wresp.Canceled {
7             return nil, errors.New("could not get next")
8         }
9         for _, ev := range wresp.Events {
10             service := decode(ev.Kv.Value)
11             var action string
12             switch ev.Type {
13             case clientv3.EventTypePut:
14                 if ev.IsCreate() {
15                     action = "create"
16                 } else if ev.IsModify() {
17                     action = "update"
18                 }
19             case clientv3.EventTypeDelete:
20                 action = "delete"
21                 // get service from prevKv
22                 service = decode(ev.PrevKv.Value)
23             }
24             if service == nil {
25                 continue
26             }
27             return &registry.Result{
28                 Action: action,
29                 Service: service,
30             }, nil
31         }
32     }
```

```
33     return nil, errors.New("could not get next")
34 }
```



天下无鱼

<https://shikey.com/>

另外，Worker 节点也利用了 registry 接口的 Register 方法实现了服务的注册。如下所示，Register 方法最终调用了 clientv3 的 Put 方法，将包含节点信息的键值对写入了 etcd 中。

 复制代码

```
1 func (e *etcdRegistry) Register(s *registry.Service, opts ...registry.RegisterC
2     // register each node individually
3     for _, node := range s.Nodes {
4         err := e.registerNode(s, node, opts...)
5         if err != nil {
6             gerr = err
7         }
8     }
9     return gerr
10 }
11
12 func (e *etcdRegistry) registerNode(s *registry.Service, node *registry.Node, c
13     service := &registry.Service{
14         Name:      s.Name,
15         Version:    s.Version,
16         Metadata:   s.Metadata,
17         Endpoints:  s.Endpoints,
18         Nodes:      []*registry.Node{node},
19     }
20     ...
21     // create an entry for the node
22     if lgr != nil {
23         _, err = e.client.Put(ctx, nodePath(service.Name, node.Id), encode(service)
24     } else {
25         _, err = e.client.Put(ctx, nodePath(service.Name, node.Id), encode(service)
26     }
27     if err != nil {
28         return err
29     }
30 }
```

现在让我们来看一看服务发现的效果。首先，启动 Master 服务。

 复制代码

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

接着启动 Worker 服务。

```
1 » go run main.go worker --id=2 --http=:8079 --grpc=:9089
```



Worker 启动后，在 Master 的日志中会看到变化的事件。其中，"Action": "create" 表明当前的事件为节点的注册。

```
1 {"level": "INFO", "ts": "2022-12-12T16:55:42.798+0800", "logger": "master", "caller":
```

中止 Worker 节点后，我们还会看到 Master 的信息。其中，"Action": "delete" 表明当前的事件为节点的删除。

```
1 {"level": "INFO", "ts": "2022-12-12T16:58:31.985+0800", "logger": "master", "caller":
```

维护 Worker 节点信息

完成服务发现之后，让我们更进一步，维护 Worker 节点的信息。在 updateWorkNodes 函数中，我们利用 registry.GetService 方法获取当前集群中全量的 Worker 节点，并将它最新的状态保存起来。

```
1 func (m *Master) Campaign() {
2     ...
3     workerNodeChange := m.WatchWorker()
4
5     for {
6         select {
7             ...
8             case resp := <-workerNodeChange:
9                 m.logger.Info("watch worker change", zap.Any("worker:", resp))
10        }
11    }
12 }
13
14 type Master struct {
15     ...
16     workNodes map[string]*registry.Node
17 }
```

```

18 func (m *Master) updateWorkNodes() {
19     services, err := m.registry.GetService(worker.ServiceName)
20     if err != nil {
21         m.logger.Error("get service", zap.Error(err))
22     }
23
24     nodes := make(map[string]*registry.Node)
25     if len(services) > 0 {
26         for _, spec := range services[0].Nodes {
27             nodes[spec.Id] = spec
28         }
29     }
30
31     added, deleted, changed := workNodeDiff(m.workNodes, nodes)
32     m.logger.Sugar().Info("worker joined: ", added, ", leaved: ", deleted, ", cha
33
34     m.workNodes = nodes
35
36 }
37

```



我们还可以使用 `workNodeDiff` 函数比较集群中新旧节点的变化。

 复制代码

```

1 func workNodeDiff(old map[string]*registry.Node, new map[string]*registry.Node)
2     added := make([]string, 0)
3     deleted := make([]string, 0)
4     changed := make([]string, 0)
5     for k, v := range new {
6         if ov, ok := old[k]; ok {
7             if !reflect.DeepEqual(v, ov) {
8                 changed = append(changed, k)
9             }
10        } else {
11            added = append(added, k)
12        }
13    }
14    for k := range old {
15        if _, ok := new[k]; !ok {
16            deleted = append(deleted, k)
17        }
18    }
19    return added, deleted, changed
20 }

```

当节点发生变化时，可以打印出日志。


```

1 {"level":"INFO","ts":"2022-12-12T16:55:42.810+0800","logger":"master","caller":
2 {"level":"INFO","ts":"2022-12-12T16:58:32.026+0800","logger":"master","caller":

```

Master 资源管理

下一步，让我们来看看对爬虫任务的管理。

爬虫任务也可以理解为一种资源。和 Worker 一样，Master 中可以有一些初始化的爬虫任务存储在配置文件中。初始化时，程序通过读取配置文件将爬虫任务注入到 Master 中。这节课我们先将任务放置到配置文件中，下节课我们还会构建 Master 的 API 来完成任务的增删查改。

```

1 seeds := worker.ParseTaskConfig(logger, nil, nil, tcfg)
2 master.New(
3     masterID,
4     master.WithLogger(logger.Named("master")),
5     master.WithGRPCAddress(GRPCListenAddress),
6     master.WithRegistryURL(sconfig.RegistryAddress),
7     master.WithRegistry(reg),
8     master.WithSeeds(seeds),
9 )

```

在初始化 Master 时，调用 `m.AddSeed` 函数完成资源的添加。`m.AddSeed` 会首先调用 `etcdCli.Get` 方法，查看当前任务是否已经写入到了 etcd 中。如果没有，则调用 `m.AddResource` 将任务存储到 etcd，存储在 etcd 中的任务的 Key 为 `/resources/xxxx`。

```

1 func (m *Master) AddSeed() {
2     rs := make([]*ResourceSpec, 0, len(m.Seeds))
3     for _, seed := range m.Seeds {
4         resp, err := m.etcdCli.Get(context.Background(), getResourcePath(seed.Name))
5         if err != nil {
6             m.logger.Error("etcd get failed", zap.Error(err))
7             continue
8         }
9         if len(resp.Kvs) == 0 {
10             r := &ResourceSpec{
11                 Name: seed.Name,
12             }
13             rs = append(rs, r)
14         }
15     }
16 }

```



```

15     }
16
17     m.AddResource(rs)
18 }
19
20 const (
21     RESOURCEPATH = "/resources"
22 )
23
24 func getResourcePath(name string) string {
25     return fmt.Sprintf("%s/%s", RESOURCEPATH, name)
26 }

```



在添加资源的时候，我们可以设置资源的 ID、创建时间等。在这里我借助了第三方库 [Snowflake](https://github.com/twitter/snowflake)，使用雪花算法来为资源生成了一个单调递增的分布式 ID。

复制代码

```

1 func (m *Master) AddResource(rs []*ResourceSpec) {
2     for _, r := range rs {
3         r.ID = m.IDGen.Generate().String()
4         ns, err := m.Assign(r)
5         if err != nil {
6             m.logger.Error("assign failed", zap.Error(err))
7             continue
8         }
9         r.AssignedNode = ns.Id + "|" + ns.Address
10        r.CreationTime = time.Now().UnixNano()
11        m.logger.Debug("add resource", zap.Any("specs", r))
12
13        _, err = m.etcdCli.Put(context.Background(), getResourcePath(r.Name), encoded)
14        if err != nil {
15            m.logger.Error("put etcd failed", zap.Error(err))
16            continue
17        }
18        m.resources[r.Name] = r
19    }
20 }

```

Snowflake 利用雪花算法生成了一个 64 位的唯一 ID，其结构如下。

复制代码

```

1
2 +-----+
3 | 1 Bit Unused | 41 Bit Timestamp | 10 Bit NodeID | 12 Bit Sequence ID |
4 +-----+

```

其中，41 位用于存储时间戳；10 位用于存储 NodeID，在这里就是我们的 Master ID；最后 12 位为序列号。如果我们的程序打算在同一个毫秒内生成多个 ID，那么每生成一个新的 ID，序列号会递增 1，这意味着每个节点每毫秒最多能够产生 4096 个不同的 ID，这已经能满足我们当前的场景了。雪花算法确保了我们生成的资源 ID 是全局唯一的。

添加资源时，还有一步很重要，那就是调用 `m.Assign` 计算出当前的资源应该被分配到哪一个节点上。在这里，我们先用随机的方式选择一个节点，后面还会再优化调度逻辑。

 复制代码

```
1 func (m *Master) Assign(r *ResourceSpec) (*registry.Node, error) {
2     for _, n := range m.workNodes {
3         return n, nil
4     }
5     return nil, errors.New("no worker nodes")
6 }
```

设置好资源的 ID 信息、分配信息之后，调用 `etcdCli.Put`，将资源的 KV 信息存储到 `etcd` 中。其中，存储到 `etcd` 中的 Value 需要是 `string` 类型，所以我们书写了 JSON 的序列化与反序列化函数，用于存储信息的序列化和反序列化。

 复制代码

```
1 func encode(s *ResourceSpec) string {
2     b, _ := json.Marshal(s)
3     return string(b)
4 }
5
6 func decode(ds []byte) (*ResourceSpec, error) {
7     var s *ResourceSpec
8     err := json.Unmarshal(ds, &s)
9     return s, err
10 }
```

最后一步，当 Master 成为新的 Leader 后，我们还要全量地获取一次 `etcd` 中当前最新的资源信息，并把它保存到内存中，核心逻辑位于 `loadResource` 函数中。

 复制代码

```

2 func (m *Master) BecomeLeader() error {
3     if err := m.loadResource(); err != nil {
4         return fmt.Errorf("loadResource failed:%w", err)
5     }
6     atomic.StoreInt32(&m.ready, 1)
7     return nil
8 }
9
10 func (m *Master) loadResource() error {
11     resp, err := m.etcdCli.Get(context.Background(), RESOURCEPATH, clientv3.WithS
12     if err != nil {
13         return fmt.Errorf("etcd get failed")
14     }
15
16     resources := make(map[string]*ResourceSpec)
17     for _, kv := range resp.Kvs {
18         r, err := decode(kv.Value)
19         if err == nil && r != nil {
20             resources[r.Name] = r
21         }
22     }
23     m.logger.Info("leader init load resource", zap.Int("length", len(m.resources)))
24     m.resources = resources
25     return nil
26 }

```



天下无鱼

<https://shikey.com/>

验证 Master 资源分配结果

最后让我们实战验证一下 Master 的资源分配结果。

首先我们需要启动 Worker。要注意的是，如果先启动了 Master，初始的任务将会由于没有对应的 Worker 节点而添加失败。

复制代码

```
1 » go run main.go worker --id=2 --http=:8079 --grpc=:9089
```

接着启动 Master 服务。

复制代码

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

现在查看 `etcd` 的信息会发现，当前两个爬虫任务都已经设置到 `etcd` 中，并且 `Master` 为他们分配的 `Worker` 节点为 `"go.micro.server.worker-2|192.168.0.107:9089"`，说明 `Master` 的资源分配成功了。



复制代码

```
1 » docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl get --prefix /
2 /micro/registry/go.micro.server.master/go.micro.server.master-2
3 {"name":"go.micro.server.master","version":"latest","metadata":null,"endpoints"
4
5 /micro/registry/go.micro.server.worker/go.micro.server.worker-2
6 {"name":"go.micro.server.worker","version":"latest","metadata":null,"endpoints"
7
8 /resources/douban_book_list
9 {"ID":"1602250527540776960","Name":"douban_book_list","AssignedNode":"go.micro.
10
11 /resources/election/3f3584fc571ae9d0
12 master2-192.168.0.107:9091
13
14 /resources/xxx
15 {"ID":"1602250527570137088","Name":"xxx","AssignedNode":"go.micro.server.worker
```

总结

这节课。我们实现了 `Master` 的两个重要功能：服务发现与资源管理。

对于服务发现，我们借助了 `micro registry` 提供的接口，实现了节点的注册、发现和状态获取。`micro` 的 `registry` 接口是一个插件，这意味着我们可以轻松使用不同插件与不同的注册中心交互。在这里我们使用的仍然是 `go-micro` 的 `etcd` 插件，借助 `etcd clientv3` 的 `API` 实现了服务发现与注册的相关功能。

而对于资源管理，这节课我们为资源加上了必要的 `ID` 信息，我们使用了分布式的雪花算法来保证生成 `ID` 全局唯一。同时，我们用随机的方式为资源分配了其所属的 `Worker` 节点并验证了分配的效果。在下一节课程中，我们还会继续实现负载均衡的资源分配。

课后题

学完这节课，给你留两道思考题。

1. 我们什么时候需要全量拉取资源？什么时候需要使用事件监听机制？你认为监听机制是可靠的吗？

2. 我们前面提到的 Snowflake 雪花算法生成的分布式 ID，在什么场景下是不适用的？

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

 赞 1  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 45 | Master高可用：怎样借助etcd实现服务选主？

下一篇 47 | 故障容错：如何在Worker崩溃时进行重新调度？

精选留言 (1)

 写留言



Realm

2023-02-01 来自浙江

1 “当 Master 成为新的 Leader 后，我们还要全量地获取一次 etcd 中当前最新的资源信息，并把它保存到内存中”，
当添加单个task任务时，使用事件监听；
事件监听机制在服务异常时可能丢信息？
望勋哥指点。

2 雪花算法生成的id有可能出现重复。
如一个节点时，把服务器时钟回拨的情况；
多个节点时候，假如服务器的标志位一样，同一毫秒不同的节点可能产生的id相同；

