

# 45 | Master高可用：怎样借助etcd实现服务选主？

2023-01-21 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 06:49 大小 6.23M



你好，我是郑建勋。

上一节课，我们搭建起了 Master 的基本框架。这一节课，让我们接着实现分布式 Master 的核心功能：选主。

## etcd 选主 API

我们在讲解架构设计时提到过，可以开启多个 Master 来实现分布式服务的故障容错。其中，只有一个 Master 能够成为 Leader，只有 Leader 能够完成任务的分配，只有 Leader 能够处理外部访问。当 Leader 崩溃时，其他的 Master 将竞争上岗成为 Leader。

实现分布式的选主并没有想象中那样复杂，在我们的项目中，只需要借助分布式协调服务 etcd 就能实现。🔗 [etcd clientv3](#) 已经为我们封装了对分布式选主的实现，核心的 API 如下。

```

1 // client/v3/concurrency
2 func NewSession(client *v3.Client, opts ...SessionOption) (*Session, error)
3 func NewElection(s *Session, pfx string) *Election
4 func (e *Election) Campaign(ctx context.Context, val string) error
5 func (e *Election) Leader(ctx context.Context) (*v3.GetResponse, error)
6 func (e *Election) Observe(ctx context.Context) <-chan v3.GetResponse
7 func (e *Election) Resign(ctx context.Context) (err error)

```



我来解释一下这些 API 的含义。

- NewSession 函数：创建一个与 etcd 服务端带租约的会话。
- NewElection 函数：创建一个选举对象 Election，Election 有许多方法。
- Election.Leader 方法可以查询当前集群中的 Leader 信息。
- Election.Observe 可以接收到当前 Leader 的变化
- Election.Campaign 方法：开启选举，该方法会阻塞住协程，直到调用者成为 Leader。

## 实现 Master 选主与故障容错

现在让我们在项目中实现分布式选主算法，核心逻辑位于 Master.Campaign 方法中，完整的代码位于 [v0.3.5 分支](#)。

```

1 func (m *Master) Campaign() {
2     endpoints := []string{m.registryURL}
3     cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
4     if err != nil {
5         panic(err)
6     }
7
8     s, err := concurrency.NewSession(cli, concurrency.WithTTL(5))
9     if err != nil {
10         fmt.Println("NewSession", "error", "err", err)
11     }
12     defer s.Close()
13
14     // 创建一个新的etcd选举election
15     e := concurrency.NewElection(s, "/resources/election")
16     leaderCh := make(chan error)
17     go m.elect(e, leaderCh)
18     leaderChange := e.Observe(context.Background())
19     select {

```

```

20 case resp := <-leaderChange:
21     m.logger.Info("watch leader change", zap.String("leader:", string(resp.Kvs[
22     ]
23
24     for {
25         select {
26             case err := <-leaderCh:
27                 if err != nil {
28                     m.logger.Error("leader elect failed", zap.Error(err))
29                     go m.elect(e, leaderCh)
30                 } else {
31                     m.logger.Info("master change to leader")
32                     m.leaderID = m.ID
33                     if !m.IsLeader() {
34                         m.BecomeLeader()
35                     }
36                 }
37             case resp := <-leaderChange:
38                 if len(resp.Kvs) > 0 {
39                     m.logger.Info("watch leader change", zap.String("leader:", string(resp.
40                     ]
41                 }
42             }
43 }

```



天下无鱼

<https://shikey.com/>

我们一步步来解析这段分布式选主的代码。

1. 第 3 行调用 `clientv3.New` 函数创建一个 `etcd clientv3` 的客户端。
2. 第 15 行, `concurrency.NewElection(s, "/resources/election")` 意为创建一个新的 `etcd` 选举对象。其中的第二个参数就是所有 Master 都在抢占的 Key, 抢占到该 Key 的 Master 将变为 Leader。

在 `etcd` 中, 一般都会选择这种目录形式的结构作为 Key, 这种方式可以方便我们进行前缀查找。例如, Kubernetes 资源在 `etcd` 中的存储格式为 `prefix/资源类型/namespace/资源名称`。

 复制代码

```

1 /registry/clusterrolebindings/system:coredns
2 /registry/clusterroles/system:coredns
3 /registry/configmaps/kube-system/coredns
4 /registry/deployments/kube-system/coredns
5 /registry/replicasets/kube-system/coredns-7fdd6d65dc
6 /registry/secrets/kube-system/coredns-token-hpqbtt
7 /registry/serviceaccounts/kube-system/coredns

```

3. 第 17 行 `go m.select(e, leaderCh)` 代表开启一个新的协程，让当前的 **Master** 进行 **Leader** 的选举。如果集群中已经有了其他的 **Leader**，当前协程将陷入到堵塞状态，如果当前 **Master** 选举成功，成为了 **Leader**，`e.Campaign` 方法会被唤醒，我们将其返回的消息传递到 `ch` 通道中。

复制代码

```
1 func (m *Master) elect(e *concurrency.Election, ch chan error) {
2     // 堵塞直到选取成功
3     err := e.Campaign(context.Background(), m.ID)
4     ch <- err
5 }
```

`e.Campaign` 方法的第二个参数为 **Master** 成为 **Leader** 后，设置到 **Key** 中的 **Value** 值。在这里，我们将 **Master ID** 作为 **Value** 值。**Master** 的 **ID** 是初始化时设置的，它当前包含了 **Master** 的序号、**Master** 的 **IP** 地址和监听的 **GRPC** 地址。

其实，**Master** 的 **ID** 就足够标识唯一的 **Master** 了，但这里还存储了 **Master IP**，是为了方便后续其他的 **Master** 拿到 **Leader** 的 **IP** 地址，从而对 **Leader** 进行访问。

复制代码

```
1 // master/master.go
2 func New(id string, opts ...Option) (*Master, error) {
3     m := &Master{}
4
5     options := defaultOptions
6     for _, opt := range opts {
7         opt(&options)
8     }
9     m.options = options
10
11     ipv4, err := getLocalIP()
12     if err != nil {
13         return nil, err
14     }
15     m.ID = genMasterID(id, ipv4, m.GRPCAddress)
16     m.logger.Sugar().Debugln("master_id:", m.ID)
17     go m.Campaign()
18
19     return &Master{}, nil
20 }
21
22 type Master struct {
```

```

23 ID string
24 ready int32
25 leaderID string
26 workNodes map[string]*registry.Node
27 options
28 }
29
30 func genMasterID(id string, ipv4 string, GRPCAddress string) string {
31     return "master" + id + "-" + ipv4 + GRPCAddress
32 }

```



获取本机的 IP 地址有一个很简单的方式，那就是遍历所有网卡，找到第一个 IPv4 地址，代码如下所示。

复制代码

```

1 func getLocalIP() (string, error) {
2     var (
3         addrs []net.Addr
4         err error
5     )
6     // 获取所有网卡
7     if addrs, err = net.InterfaceAddrs(); err != nil {
8         return "", err
9     }
10    // 取第一个非lo的网卡IP
11    for _, addr := range addrs {
12        if ipNet, isIpNet := addr.(*net.IPNet); isIpNet && !ipNet.IP.IsLoopback() {
13            if ipNet.IP.To4() != nil {
14                return ipNet.IP.String(), nil
15            }
16        }
17    }
18
19    return "", errors.New("no local ip")
20 }

```

4. 当 Master 并行进行选举的同时（第 18 行），调用 `e.Observe` 监听 Leader 的变化。  
`e.Observe` 函数会返回一个通道，当 Leader 状态发生变化时，会将当前 Leader 的信息发送到通道中。在这里我们初始化时首先堵塞读取了一次 `e.Observe` 返回的通道信息。因为只有成功收到 `e.Observe` 返回的消息，才意味着集群中已经存在 Leader，表示集群完成了选举。
5. 第 24 行，我们在 for 循环中使用 `select` 监听了多个通道的变化，其中通道 `leaderCh` 负责监听当前 Master 是否当上了 Leader，而 `leaderChange` 负责监听当前集群中 Leader 是否

发生了变化。

书写好 Master 的选主逻辑之后，接下来让我们执行 Master 程序，完整的代码位于 <https://shikekey.com/v0.3.5/> 分支。

 复制代码

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

由于当前只有一个 Master，因此当前 Master 一定会成为 Leader。我们可以看到打印出的当前 Leader 的信息：master2-192.168.0.107:9091。

 复制代码

```
1 {"level":"INFO","ts":"2022-12-07T18:23:28.494+0800","logger":"master","caller":
2 {"level":"INFO","ts":"2022-12-07T18:23:28.494+0800","logger":"master","caller":
3 {"level":"INFO","ts":"2022-12-07T18:23:28.494+0800","logger":"master","caller":
4 {"level":"DEBUG","ts":"2022-12-07T18:23:38.500+0800","logger":"master","caller"
```

如果这时我们查看 etcd 的信息，会看到自动生成了 /resources/election/xxx 的 Key，并且它的 Value 是我们设置的 master2-192.168.0.107:9091。

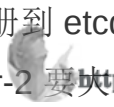
 复制代码

```
1 » docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl get --prefix /
2
3 /micro/registry/go.micro.server.master/go.micro.server.master-2
4 {"name":"go.micro.server.master","version":"latest","metadata":null,"endpoints"
5
6 /resources/election/3f3584fc571ae898
7 master2-192.168.0.107:9091
```

如果我们再启动一个新的 Master 程序，会发现当前获取到的 Leader 仍然是 master2-192.168.0.107:9091。

 复制代码

```
1 » go run main.go master --id=3 --http=:8082 --grpc=:9092
2 {"level":"DEBUG","ts":"2022-12-07T18:23:52.371+0800","logger":"master","caller"
3 {"level":"INFO","ts":"2022-12-07T18:23:52.387+0800","logger":"master","caller":
4 {"level":"DEBUG","ts":"2022-12-07T18:24:02.393+0800","logger":"master","caller"
```

再次查看 etcd 的信息，会发现 go.micro.server.master-3 也成功注册到 etcd 中了，并且 c 在 /resources/election 下方注册了自己的 Key，但是该 Key 比 master-2 要大。

 复制代码

```
1 /micro/registry/go.micro.server.master/go.micro.server.master-2
2 {"name":"go.micro.server.master","version":"latest","metadata":null,"endpoints"
3 /micro/registry/go.micro.server.master/go.micro.server.master-3
4 {"name":"go.micro.server.master","version":"latest","metadata":null,"endpoints"
5 /resources/election/3f3584fc571ae898
6 master2-192.168.0.107:9091
7 /resources/election/3f3584fc571ae8a9
8 master3-192.168.0.107:9092
```

到这里，我们就实现了 Master 的选主操作，所有的 Master 都只认定一个 Leader。当我们终止 master-2 程序，在 master-3 程序中会立即看到如下日志，说明当前的 Leader 已经顺利完成了切换，master-3 当选为了新的 Leader。

 复制代码

```
1 {"level":"INFO","ts":"2022-12-12T00:46:58.288+0800","logger":"master","caller":
2 {"level":"INFO","ts":"2022-12-12T00:46:58.289+0800","logger":"master","caller":
3 {"level":"DEBUG","ts":"2022-12-12T00:47:18.296+0800","logger":"master","caller"
```

当我们再次查看 etcd，发现 /resources/election/ 路径下只剩下 master-3 程序的注册信息了，证明 Master 的选举成功。

 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-3
3 {"name":"go.micro.server.master","version":"latest","metadata":null,"endpoints"
4 /resources/election/3f3584fc571ae8a9
5 master3-192.168.0.107:9092
```

## etcd 选主原理

经过上面的实践，我们可以看到，借助 etcd，分布式选主变得非常容易了，现在我们来看一看 etcd 实现分布式选主的原理。它的核心代码位于 Election.Campaign 方法中，如下所示，下面



代码做了简化，省略了对异常情况的处理。



```
1 func (e *Election) Campaign(ctx context.Context, val string) error {
2     s := e.session
3     client := e.session.Client()
4
5     k := fmt.Sprintf("%s%x", e.keyPrefix, s.Lease())
6     txn := client.Txn(ctx).If(v3.Compare(v3.CreateRevision(k), "=", 0))
7     txn = txn.Then(v3.OpPut(k, val, v3.WithLease(s.Lease())))
8     txn = txn.Else(v3.OpGet(k))
9     resp, err := txn.Commit()
10    if err != nil {
11        return err
12    }
13    e.leaderKey, e.leaderRev, e.leaderSession = k, resp.Header.Revision, s
14    _, err = waitDeletes(ctx, client, e.keyPrefix, e.leaderRev-1)
15    if err != nil {
16        // clean up in case of context cancel
17        select {
18            case <-ctx.Done():
19                e.Resign(client.Ctx())
20            default:
21                e.leaderSession = nil
22        }
23        return err
24    }
25    e.hdr = resp.Header
26
27    return nil
28 }
```

Campaign 首先用了一个事务操作在要抢占的 `e.keyPrefix` 路径下维护一个 Key。其中，`e.keyPrefix` 是 Master 要抢占的 etcd 路径，在我们的项目中为 `/resources/election/`。这段事务操作会首先判断当前生成的 Key（例如 `/resources/election/3f3584fc571ae8a9`）是否已经在 etcd 中了。如果不存在，才会创建该 Key。这样，每一个 Master 都会在 `/resources/election/` 下维护一个 Key，并且当前的 Key 是带租约的。

Campaign 第二步会调用 `waitDeletes` 函数堵塞等待，直到自己成为 Leader 为止。那什么时候当前 Master 会成为 Leader 呢？

`waitDeletes` 函数会调用 `client.Get` 获取到当前争抢的 `/resources/election/` 路径下具有最大版本号 Key，并调用 `waitDelete` 函数等待该 Key 被删除。而 `waitDelete` 会调用 `client.Watch`



来完成对特定版本 Key 的监听。

当前 Master 需要监听这个最大版本号 Key 的删除事件。当这个特定的 Key 被删除，就意味着已经没有了比当前 Master 创建的 Key 更早的 Key 了，因此当前的 Master 理所当然就排队成为了 Leader。

 复制代码

```
1 func waitDeletes(ctx context.Context, client *v3.Client, pfx string, maxCreateRev
2   getOpts := append(v3.WithLastCreate(), v3.WithMaxCreateRev(maxCreateRev))
3   for {
4     resp, err := client.Get(ctx, pfx, getOpts...)
5     if err != nil {
6       return nil, err
7     }
8     if len(resp.Kvs) == 0 {
9       return resp.Header, nil
10    }
11    lastKey := string(resp.Kvs[0].Key)
12    if err = waitDelete(ctx, client, lastKey, resp.Header.Revision); err != nil
13      return nil, err
14    }
15  }
16 }
17
18 func waitDelete(ctx context.Context, client *v3.Client, key string, rev int64)
19   cctx, cancel := context.WithCancel(ctx)
20   defer cancel()
21
22   var wr v3.WatchResponse
23   wch := client.Watch(cctx, key, v3.WithRev(rev))
24   for wr = range wch {
25     for _, ev := range wr.Events {
26       if ev.Type == mvccpb.DELETE {
27         return nil
28       }
29     }
30   }
31   if err := wr.Err(); err != nil {
32     return err
33   }
34   if err := cctx.Err(); err != nil {
35     return err
36   }
37   return fmt.Errorf("lost watcher waiting for delete")
38 }
```

这种监听方式还避免了惊群效应，因为当 Leader 崩溃后，并不会唤醒所有在选举中的 Master。只有当队列中的前一个 Master 创建的 Key 被删除，当前的 Master 才会被唤醒。也就是说，每一个 Master 都在排队等待着前一个 Master 退出，这样 Master 就以最小的代价实现了 Key 的争抢。

## 总结

这节课，我们借助 etcd 实现了分布式 Master 的选主，确保了在同一时刻只能存在一个 Leader。此外，我们还实现了 Master 的故障容错能力。

etcd clientv3 为我们封装了选主的实现，它的实现方式也很简单，通过监听最近的 Key 的 DELETE 事件，我们实现了所有的节点对同一个 Key 的抢占，同时还避免了集群可能出现的惊群效应。在实践中，我们也可以使用其他的分布式协调组件（例如 ZooKeeper、Consul）帮助我们实现选主，它们的实现原理都和 etcd 类似。


## 课后题

最后，给你留一道思考题吧。

其实，利用 MySQL 也可以实现分布式的选主，你知道如何实现吗？（参考：  
<http://code.openark.org/blog/mysql/leader-election-using-mysql>）

欢迎你给我留言交流讨论，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 3  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 44 | 一个程序多种功能：构建子命令与 flags

下一篇 46 | Master 任务调度：服务发现与资源管理

## 精选留言 (2)



**Realm**

2023-01-26 来自浙江

请问老师：

“当前 Master 需要监听这个最大版本号 Key 的删除事件。当这个特定的 Key 被删除，就意味着已经没有比当前 Master 创建的 Key 更早的 Key 了，因此当前的 Master 理所当然就排队成为了 Leader。”

1 是所有master监听的内容都相同吗？

2 这里如何避免惊群？

共 2 条评论 >



**Geek\_7e6c5e**

2023-01-23 来自陕西

太酷了，etcd让普通程序员也有了开发分布式系统的能力

