



下载APP



## 19 | 在分布式环境中，Leader选举、互斥锁和读写锁该如何实现？

2020-11-23 晁岳攀

Go 并发编程实战课

[进入课程 >](#)**讲述：安晓辉**

时长 12:33 大小 11.50M



你好，我是鸟窝。

在前面的课程里，我们学习的并发原语都是在进程内使用的，也就是我们常见的一个运行程序为了控制共享资源、实现任务编排和进行消息传递而提供的控制类型。在接下来的这两节课里，我要讲的是几个分布式的并发原语，它们控制的资源或编排的任务分布在不同进程、不同机器上。

分布式的并发原语实现更加复杂，因为在分布式环境中，网络状况、服务状态都是不可控的。不过还好有相应的软件系统去做这些事情。这些软件系统会专门去处理这些节点之间的协调和异常情况，并且保证数据的一致性。我们要做的就是在它们的基础上实现我们的业务。



常用来做协调工作的软件系统是 Zookeeper、etcd、Consul 之类的软件，Zookeeper 为 Java 生态群提供了丰富的分布式并发原语（通过 Curator 库），但是缺少 Go 相关的并发原语库。Consul 在提供分布式并发原语这件事儿上不是很积极，而 etcd 就提供了非常好的分布式并发原语，比如分布式互斥锁、分布式读写锁、Leader 选举，等等。所以，今天，我就以 etcd 为基础，给你介绍几种分布式并发原语。

既然我们依赖 etcd，那么，在生产环境中要有一个 etcd 集群，而且应该保证这个 etcd 集群是 7\*24 工作的。在学习过程中，你可以使用一个 etcd 节点进行测试。

这节课我要介绍的就是 Leader 选举、互斥锁和读写锁。

## Leader 选举

Leader 选举常常用在主从架构的系统中。主从架构中的服务节点分为主（Leader、Master）和从（Follower、Slave）两种角色，实际节点包括 1 主 n 从，一共是 n+1 个节点。

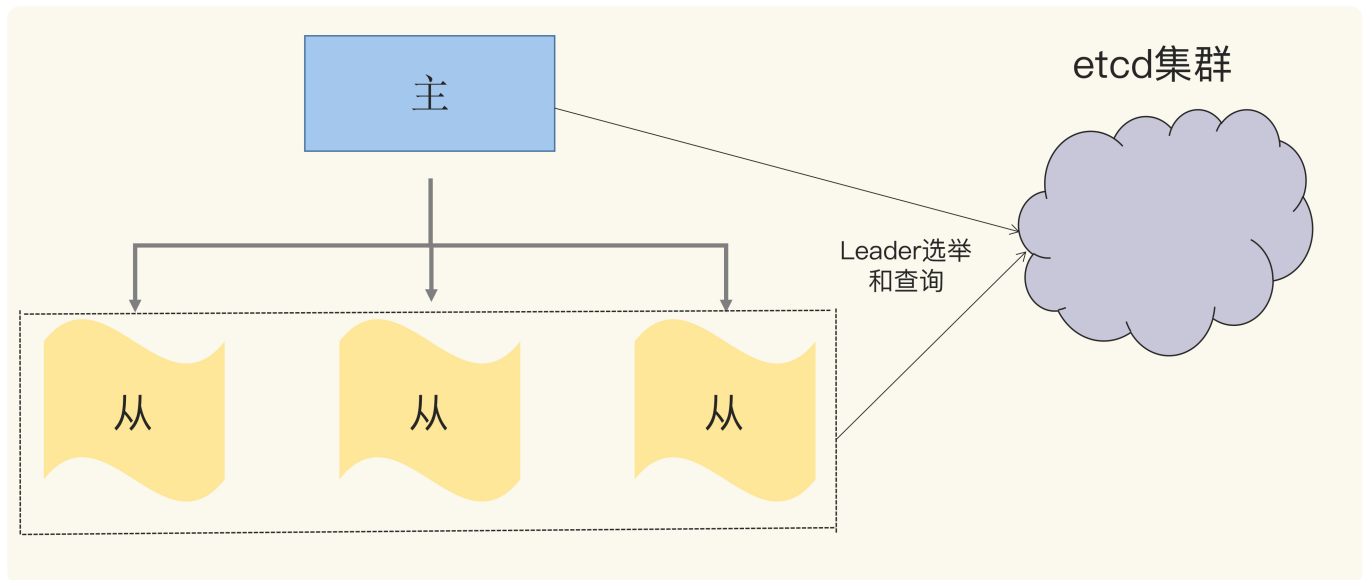
主节点常常执行写操作，从节点常常执行读操作，如果读写都在主节点，从节点只是提供一个备份功能的话，那么，主从架构就会退化成主备模式架构。

主从架构中最重要的是如何确定节点的角色，也就是，到底哪个节点是主，哪个节点是从？

**在同一时刻，系统中不能有两个主节点，否则，如果两个节点都是主，都执行写操作的话，就有可能出现数据不一致的情况，所以，我们需要一个选主机制，选择一个节点作为主节点，这个过程就是 Leader 选举。**

当主节点宕机或者是不可用时，就需要新一轮的选举，从其它的从节点中选择一个节点，让它作为新主节点，宕机的原主节点恢复后，可以变为从节点，或者被摘掉。

我们可以通过 etcd 基础服务来实现 leader 选举。具体点说，我们可以将 Leader 选举的逻辑交给 etcd 基础服务，这样，我们只需要把重心放在业务开发上。etcd 基础服务可以通过多节点的方式保证 7\*24 服务，所以，我们也不用担心 Leader 选举不可用的问题。如下图所示：



接下来，我会给你介绍业务开发中跟 Leader 选举相关的选举、查询、Leader 变动监控等功能。

我要先提醒你一句，如果你想运行我下面讲到的测试代码，就要先部署一个 etcd 的集群，或者部署一个 etcd 节点做测试。

首先，我们来实现一个测试分布式程序的框架：它会先从命令行中读取命令，然后再执行相应的命令。你可以打开两个窗口，模拟不同的节点，分别执行不同的命令。

这个测试程序如下：

复制代码

```
1 package main
2
3 // 导入所需的库
4 import (
5     "bufio"
6     "context"
7     "flag"
8     "fmt"
9     "log"
10    "os"
11    "strconv"
12    "strings"
13
14    "github.com/coreos/etcd/clientv3"
15    "github.com/coreos/etcd/clientv3/concurrency"
16 )
17
18 // 可以设置一些参数，比如节点ID
```

```
19 var (  
20     nodeID      = flag.Int("id", 0, "node ID")  
21     addr        = flag.String("addr", "http://127.0.0.1:2379", "etcd addresses")  
22     electName   = flag.String("name", "my-test-elect", "election name")  
23 )  
24  
25 func main() {  
26     flag.Parse()  
27  
28     // 将etcd的地址解析成slice of string  
29     endpoints := strings.Split(*addr, ",")  
30  
31     // 生成一个etcd的client  
32     cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})  
33     if err != nil {  
34         log.Fatal(err)  
35     }  
36     defer cli.Close()  
37  
38     // 创建session,如果程序宕机导致session断掉, etcd能检测到  
39     session, err := concurrency.NewSession(cli)  
40     defer session.Close()  
41  
42     // 生成一个选举对象。下面主要使用它进行选举和查询等操作  
43     // 另一个方法ResumeElection可以使用既有的leader初始化Election  
44     e1 := concurrency.NewElection(session, *electName)  
45  
46     // 从命令行读取命令  
47     consolescanner := bufio.NewScanner(os.Stdin)  
48     for consolescanner.Scan() {  
49         action := consolescanner.Text()  
50         switch action {  
51             case "elect": // 选举命令  
52                 go elect(e1, *electName)  
53             case "proclaim": // 只更新leader的值  
54                 proclaim(e1, *electName)  
55             case "resign": // 辞去leader,重新选举  
56                 resign(e1, *electName)  
57             case "watch": // 监控leader的变动  
58                 go watch(e1, *electName)  
59             case "query": // 查询当前的leader  
60                 query(e1, *electName)  
61             case "rev":  
62                 rev(e1, *electName)  
63             default:  
64                 fmt.Println("unknown action")  
65         }  
66     }  
67 }
```

部署完以后，我们就可以开始选举了。

## 选举

如果你的业务集群还没有主节点，或者主节点宕机了，你就需要发起新一轮的选主操作，主要会用到 **Campaign** 和 **Proclaim**。如果你需要主节点放弃主的角色，让其它从节点有机会成为主节点，就可以调用 **Resign** 方法。

这里我提到了三个和选主相关的方法，下面我来介绍下它们的用法。

**第一个方法是 Campaign**。它的作用是，把一个节点选举为主节点，并且会设置一个值。它的签名如下所示：

```
1 func (e *Election) Campaign(ctx context.Context, val string) error
```

[复制代码](#)

需要注意的是，这是一个阻塞方法，在调用它的时候会被阻塞，直到满足下面的三个条件之一，才会取消阻塞。

1. 成功当选为主；
2. 此方法返回错误；
3. ctx 被取消。

**第二个方法是 Proclaim**。它的作用是，重新设置 Leader 的值，但是不会重新选主，这个方法会返回新值设置成功或者失败的信息。方法签名如下所示：

```
1 func (e *Election) Proclaim(ctx context.Context, val string) error
```

[复制代码](#)

**第三个方法是 Resign**：开始新一次选举。这个方法会返回新的选举成功或者失败的信息。它的签名如下所示：

[复制代码](#)

```
1 func (e *Election) Resign(ctx context.Context) (err error)
```

这三个方法的测试代码如下。你可以使用测试程序进行测试，具体做法是，启动两个节点，执行和这三个方法相关的命令。

[复制代码](#)

```
1 var count int
2 // 选主
3 func elect(e1 *concurrency.Election, electName string) {
4     log.Println("acampaigning for ID:", *nodeID)
5     // 调用Campaign方法选主,主的值为value-<主节点ID>-<count>
6     if err := e1.Campaign(context.Background(), fmt.Sprintf("value-%d-%d", *no
7         log.Println(err)
8     }
9     log.Println("campaigned for ID:", *nodeID)
10    count++
11 }
12 // 为主设置新值
13 func proclaim(e1 *concurrency.Election, electName string) {
14     log.Println("proclaiming for ID:", *nodeID)
15     // 调用Proclaim方法设置新值,新值为value-<主节点ID>-<count>
16     if err := e1.Proclaim(context.Background(), fmt.Sprintf("value-%d-%d", *no
17         log.Println(err)
18     }
19     log.Println("proclaimed for ID:", *nodeID)
20     count++
21 }
22 // 重新选主, 有可能另外一个节点被选为了主
23 func resign(e1 *concurrency.Election, electName string) {
24     log.Println("resigning for ID:", *nodeID)
25     // 调用Resign重新选主
26     if err := e1.Resign(context.TODO()); err != nil {
27         log.Println(err)
28     }
29     log.Println("resigned for ID:", *nodeID)
30 }
```

## 查询

除了选举 Leader，程序在启动的过程中，或者在运行的时候，还有可能需要查询当前的主节点是哪一个节点？主节点的值是什么？版本是多少？不光是主从节点需要查询和知道哪一个节点，在分布式系统中，还有其它一些节点也需要知道集群中的哪一个节点是主节点，哪一个节点是从节点，这样它们才能把读写请求分别发往相应的主从节点上。

etcd 提供了查询当前 Leader 的方法 **Leader**，如果当前还没有 Leader，就返回一个错误，你可以使用这个方法查询主节点信息。这个方法的签名如下：

[复制代码](#)

```
1 func (e *Election) Leader(ctx context.Context) (*v3.GetResponse, error)
```

每次主节点的变动都会生成一个新的版本号，你还可以查询版本号信息（**Rev** 方法），了解主节点变动情况：

[复制代码](#)

```
1 func (e *Election) Rev() int64
```

你可以在测试完选主命令后，测试查询命令（query、rev），代码如下：

[复制代码](#)

```
1 // 查询主的信息
2 func query(e1 *concurrency.Election, electName string) {
3     // 调用Leader返回主的信息，包括key和value等信息
4     resp, err := e1.Leader(context.Background())
5     if err != nil {
6         log.Printf("failed to get the current leader: %v", err)
7     }
8     log.Println("current leader:", string(resp.Kvs[0].Key), string(resp.Kvs[0]
9 }
10 // 可以直接查询主的rev信息
11 func rev(e1 *concurrency.Election, electName string) {
12     rev := e1.Rev()
13     log.Println("current rev:", rev)
14 }
```

## 监控

有了选举和查询方法，我们还需要一个监控方法。毕竟，如果主节点变化了，我们需要得到最新的主节点信息。

我们可以通过 **Observe** 来监控主的变化，它的签名如下：



```
1 func (e *Election) Observe(ctx context.Context) <-chan v3.GetResponse
```

[复制代码](#)

它会返回一个 chan，显示主节点的变动信息。需要注意的是，它不会返回主节点的全部历史变动信息，而是只返回最近的一条变动信息以及之后的变动信息。

它的测试代码如下：

```
1 func watch(e1 *concurrency.Election, electName string) {
2     ch := e1.Observe(context.TODO())
3
4
5     log.Println("start to watch for ID:", *nodeID)
6     for i := 0; i < 10; i++ {
7         resp := <-ch
8         log.Println("leader changed to", string(resp.Kvs[0].Key), string(resp.
9     }
10 }
```

[复制代码](#)

etcd 提供了选主的逻辑，而你要做的就是利用这些方法，让它们为你的业务服务。在使用的过程中，你还需要做一些额外的设置，比如查询当前的主节点、启动一个 goroutine 阻塞调用 Campaign 方法，等等。虽然你需要做一些额外的工作，但是跟自己实现一个分布式的选主逻辑相比，大大地减少了工作量。

接下来，我们继续看 etcd 提供的分布式并发原语：互斥锁。

## 互斥锁

互斥锁是非常常用的一种并发原语，我专门花了 4 讲的时间，重点介绍了互斥锁的功能、原理和易错场景。

不过，前面说的互斥锁都是用来保护同一进程内的共享资源的，今天，我们要掌握的是分布式环境中的互斥锁。**我们要重点学习下分布在不同机器中的不同进程内的 goroutine，如何利用分布式互斥锁来保护共享资源。**

互斥锁的应用场景和主从架构的应用场景不太一样。**使用互斥锁的不同节点是没有主从这样的角色的，所有的节点都是一样的，只不过在同一时刻，只允许其中的一个节点持有**




锁。

下面，我们就来学习下互斥锁相关的两个原语，即 Locker 和 Mutex。


## Locker

etcd 提供了一个简单的 Locker 原语，它类似于 Go 标准库中的 sync.Locker 接口，也提供了 Lock/Unlock 的机制：

 复制代码

```
1 func NewLocker(s *Session, pfx string) sync.Locker
```

可以看到，它的返回值是一个 sync.Locker，因为你对标准库的 Locker 已经非常了解了，而且它只有 Lock/Unlock 两个方法，所以，接下来使用这个锁就非常容易了。下面的代码是一个使用 Locker 并发原语的例子：

 复制代码

```
1 package main
2
3 import (
4     "flag"
5     "log"
6     "math/rand"
7     "strings"
8     "time"
9
10    "github.com/coreos/etcd/clientv3"
11    "github.com/coreos/etcd/clientv3/concurrency"
12 )
13
14 var (
15     addr      = flag.String("addr", "http://127.0.0.1:2379", "etcd addresses")
16     lockName = flag.String("name", "my-test-lock", "lock name")
17 )
18
19 func main() {
20     flag.Parse()
21
22     rand.Seed(time.Now().UnixNano())
23     // etcd地址
24     endpoints := strings.Split(*addr, ",")
25     // 生成一个etcd client
26     cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
```

```
27     if err != nil {
28         log.Fatal(err)
29     }
30     defer cli.Close()
31     useLock(cli) // 测试锁
32 }
33
34 func useLock(cli *clientv3.Client) {
35     // 为锁生成session
36     s1, err := concurrency.NewSession(cli)
37     if err != nil {
38         log.Fatal(err)
39     }
40     defer s1.Close()
41     //得到一个分布式锁
42     locker := concurrency.NewLocker(s1, *lockName)
43
44     // 请求锁
45     log.Println("acquiring lock")
46     locker.Lock()
47     log.Println("acquired lock")
48
49     // 等待一段时间
50     time.Sleep(time.Duration(rand.Intn(30)) * time.Second)
51     locker.Unlock() // 释放锁
52
53     log.Println("released lock")
54 }
```

你可以同时两个终端中运行这个测试程序。可以看到，它们获得锁是有先后顺序的，一个节点释放了锁之后，另外一个节点才能获取到这个分布式锁。

## Mutex

事实上，刚刚说的 Locker 是基于 Mutex 实现的，只不过，Mutex 提供了查询 Mutex 的 key 的信息的功能。测试代码也类似：

 复制代码

```
1 func useMutex(cli *clientv3.Client) {
2     // 为锁生成session
3     s1, err := concurrency.NewSession(cli)
4     if err != nil {
5         log.Fatal(err)
6     }
7     defer s1.Close()
8     m1 := concurrency.NewMutex(s1, *lockName)
9 }
```

```
10 //在请求锁之前查询key
11 log.Printf("before acquiring. key: %s", m1.Key())
12 // 请求锁
13 log.Println("acquiring lock")
14 if err := m1.Lock(context.TODO()); err != nil {
15     log.Fatal(err)
16 }
17 log.Printf("acquired lock. key: %s", m1.Key())
18
19 //等待一段时间
20 time.Sleep(time.Duration(rand.Intn(30)) * time.Second)
21
22 // 释放锁
23 if err := m1.Unlock(context.TODO()); err != nil {
24     log.Fatal(err)
25 }
26 log.Println("released lock")
27 }
```

可以看到，Mutex 并没有实现 `sync.Locker` 接口，它的 `Lock/Unlock` 方法需要提供一个 `context.Context` 实例做参数，这也就意味着，在请求锁的时候，你可以设置超时时间，或者主动取消请求。

## 读写锁

学完了分布式 Locker 和互斥锁 Mutex，你肯定会联想到读写锁 RWMutex。是的，etcd 也提供了分布式的读写锁。不过，互斥锁 Mutex 是在 [github.com/coreos/etcd/clientv3/concurrency](https://github.com/coreos/etcd/clientv3/concurrency) 包中提供的，读写锁 RWMutex 却是在 [github.com/coreos/etcd/contrib/recipes](https://github.com/coreos/etcd/contrib/recipes) 包中提供的。

etcd 提供的分布式读写锁的功能和标准库的读写锁的功能是一样的。只不过，**etcd 提供的读写锁，可以在分布式环境中的不同的节点使用**。它提供的方法也和标准库中的读写锁的方法一致，分别提供了 `RLock/RUnlock`、`Lock/Unlock` 方法。下面的代码是使用读写锁的例子，它从命令行中读取命令，执行读写锁的操作：

 复制代码

```
1 package main
2
3
4 import (
5     "bufio"
6     "flag"
```

```
7     "fmt"
8     "log"
9     "math/rand"
10    "os"
11    "strings"
12    "time"
13
14    "github.com/coreos/etcd/clientv3"
15    "github.com/coreos/etcd/clientv3/concurrency"
16    recipe "github.com/coreos/etcd/contrib/recipes"
17 )
18
19 var (
20     addr      = flag.String("addr", "http://127.0.0.1:2379", "etcd addresses")
21     lockName = flag.String("name", "my-test-lock", "lock name")
22     action    = flag.String("rw", "w", "r means acquiring read lock, w means ac
23 )
24
25
26 func main() {
27     flag.Parse()
28     rand.Seed(time.Now().UnixNano())
29
30     // 解析etcd地址
31     endpoints := strings.Split(*addr, ",")
32
33     // 创建etcd的client
34     cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
35     if err != nil {
36         log.Fatal(err)
37     }
38     defer cli.Close()
39     // 创建session
40     s1, err := concurrency.NewSession(cli)
41     if err != nil {
42         log.Fatal(err)
43     }
44     defer s1.Close()
45     m1 := recipe.NewRWMutex(s1, *lockName)
46
47     // 从命令行读取命令
48     consolescanner := bufio.NewScanner(os.Stdin)
49     for consolescanner.Scan() {
50         action := consolescanner.Text()
51         switch action {
52             case "w": // 请求写锁
53                 testWriteLocker(m1)
54             case "r": // 请求读锁
55                 testReadLocker(m1)
56             default:
57                 fmt.Println("unknown action")
58         }
```

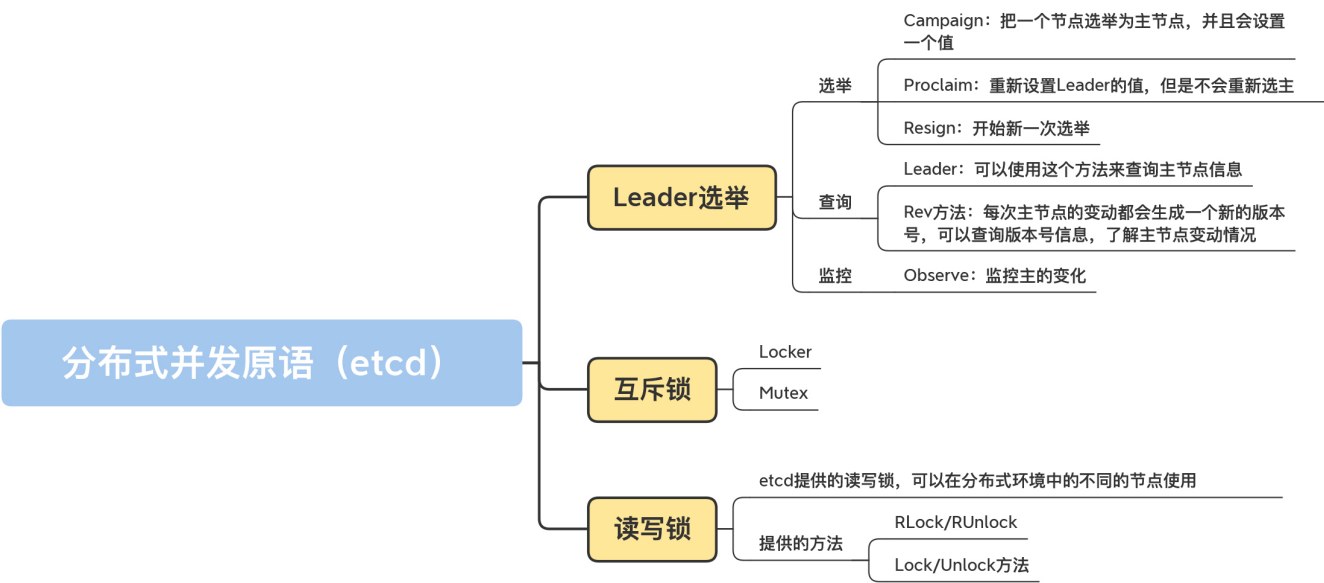
```
59     }
60 }
61
62 func testWriteLocker(m1 *recipe.RWMutex) {
63     // 请求写锁
64     log.Println("acquiring write lock")
65     if err := m1.Lock(); err != nil {
66         log.Fatal(err)
67     }
68     log.Println("acquired write lock")
69
70     // 等待一段时间
71     time.Sleep(time.Duration(rand.Intn(10)) * time.Second)
72
73     // 释放写锁
74     if err := m1.Unlock(); err != nil {
75         log.Fatal(err)
76     }
77     log.Println("released write lock")
78 }
79
80 func testReadLocker(m1 *recipe.RWMutex) {
81     // 请求读锁
82     log.Println("acquiring read lock")
83     if err := m1.RLock(); err != nil {
84         log.Fatal(err)
85     }
86     log.Println("acquired read lock")
87
88     // 等待一段时间
89     time.Sleep(time.Duration(rand.Intn(10)) * time.Second)
90
91     // 释放写锁
92     if err := m1.RUnlock(); err != nil {
93         log.Fatal(err)
94     }
95     log.Println("released read lock")
96 }
```

## 总结

自己实现分布式环境的并发原语，是相当困难的一件事，因为你需要考虑网络的延迟和异常、节点的可用性、数据的一致性等多种情况。

所以，我们可以借助 etcd 这样成熟的框架，基于它提供的分布式并发原语处理分布式的场景。需要注意的是，在使用这些分布式并发原语的时候，你需要考虑异常的情况，比如网

络断掉等。同时，分布式并发原语需要网络之间的通讯，所以会比使用标准库中的并发原语耗时更长。



好了，这节课就到这里，下节课，我会带你继续学习其它的分布式并发原语，包括队列、栅栏和 STM，敬请期待。

### 思考题

- 1. 如果持有互斥锁或者读写锁的节点意外宕机了，它持有的锁会不会被释放？
- 2. etcd 提供的读写锁中的读和写有没有优先级？

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得有所收获，也欢迎你  
把今天的内容分享给你的朋友或同事。

提建议

# Go并发编程实战课

## 鸟窝带你攻克并发编程难题

晁岳攀 (鸟窝)

前微博技术专家

知名微服务框架 rpcx 的作者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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 18 | 分组操作：处理一组子任务，该用什么并发原语？

下一篇 20 | 在分布式环境中，队列、栅栏和STM该如何实现？

### 精选留言 (3)

 写留言



鸟窝 置顶

2020-11-25

这一讲和下一讲的代码在 <https://github.com/smallnest/distributed>



1



那一刻

2020-11-23

关于思考题，

如果持有互斥锁或者读写锁的节点意外宕机了，从调用接口来看，与当前节点启动的session有关系，节点宕机之后，感觉应该有与该session相关的处理，比如超时机制，所以它持有的锁会被释放。

etcd 提供的读写锁，按照rwmutex的实现写锁应该比读锁优先级高，但是在分布式环境...  
展开





**myrfy**

2020-11-23

还没来得及去看etcd库的代码，盲猜一下。

第一个问题，我觉得要看场景，如果被锁住的资源可以被重新分配，我相信etcd能检测到持有锁的节点断开，concurrent包里应该有相关的实现把锁释放。但是，如果被锁住的资源非常重要，影响到整个系统的状态，必须要人工介入才能把破损的数据修复，那这个时候自动释放锁反而可能完成更大规模的损失。...

展开 ✓

