



下载APP

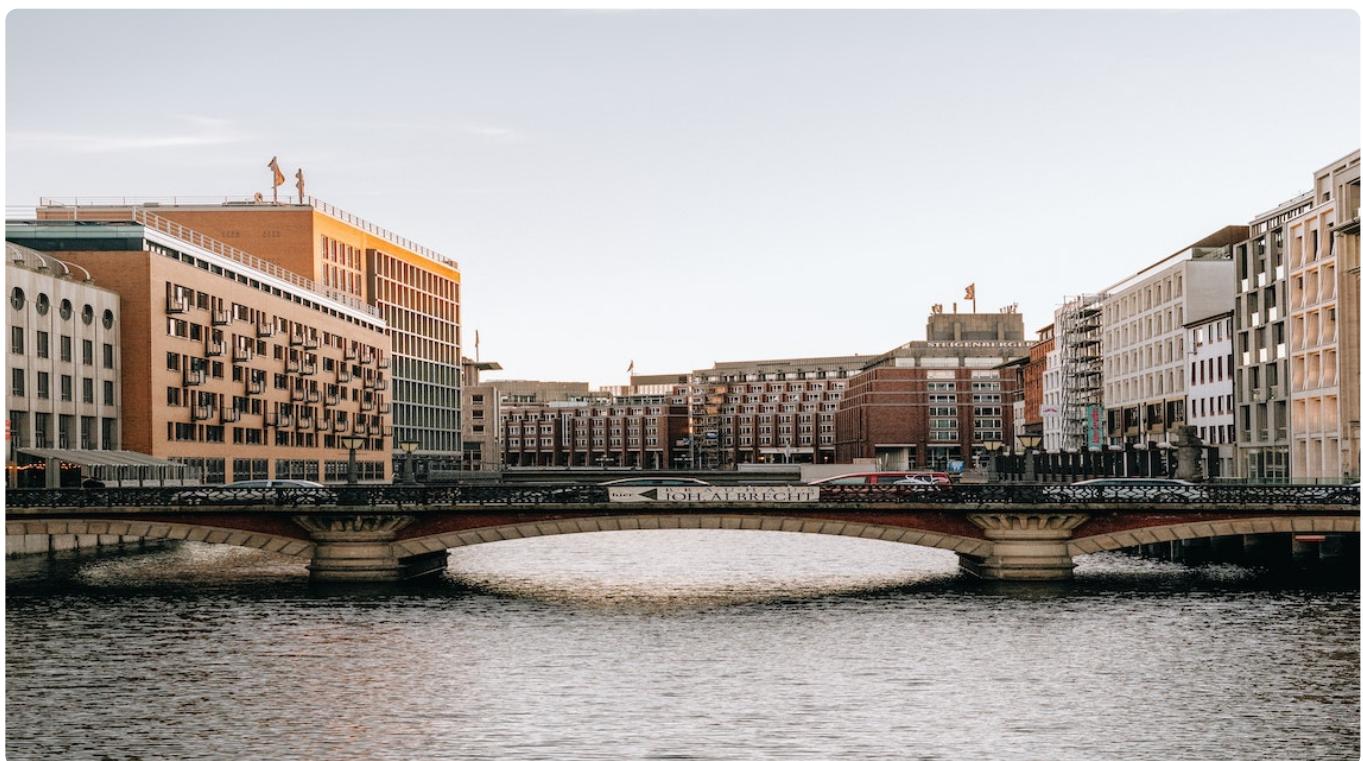


20 | 在分布式环境中，队列、栅栏和STM该如何实现？

2020-11-25 鬼岳攀

Go 并发编程实战课

[进入课程 >](#)



讲述：安晓辉

时长 14:01 大小 12.85M



你好，我是鸟窝。

上一讲，我已经带你认识了基于 etcd 实现的 Leader 选举、互斥锁和读写锁，今天，我们来学习下基于 etcd 的分布式队列、栅栏和 STM。

只要你学过计算机算法和数据结构相关的知识，队列这种数据结构你一定不陌生，它是一种先进先出的类型，有出队 (dequeue) 和入队 (enqueue) 两种操作。在 [第 12 讲](#) 中，我专门讲到了一种叫做 lock-free 的队列。队列在单机的应用程序中常常使用，但在分布式环境中，多节点如何并发地执行入队和出队的操作呢？这一讲，我会带你认识一下基于 etcd 实现的分布式队列。

除此之外，我还会讲用分布式栅栏编排一组分布式节点同时执行的方法，以及简化多个 key 的操作并且提供事务功能的 STM (Software Transactional Memory，软件事务内存)。

分布式队列和优先级队列

前一讲我也讲到，我们并不是从零开始实现一个分布式队列，而是站在 etcd 的肩膀上，利用 etcd 提供的功能实现分布式队列。

etcd 集群的可用性由 etcd 集群的维护者来保证，我们不用担心网络分区、节点宕机等问题。我们可以把这些通通交给 etcd 的运维人员，把我们自己的关注点放在使用上。

下面，我们就来了解下 etcd 提供的分布式队列。etcd 通过 github.com/coreos/etcd/contrib/recipes 包提供了分布式队列这种数据结构。

创建分布式队列的方法非常简单，只有一个，即 NewQueue，你只需要传入 etcd 的 client 和这个队列的名字，就可以了。代码如下：

 复制代码

```
1 func NewQueue(client *v3.Client, keyPrefix string) *Queue
```

这个队列只有两个方法，分别是出队和入队，队列中的元素是字符串类型。这两个方法的签名如下所示：

 复制代码

```
1 // 入队
2 func (q *Queue) Enqueue(val string) error
3 // 出队
4 func (q *Queue) Dequeue() (string, error)
```

需要注意的是，如果这个分布式队列当前为空，调用 Dequeue 方法的话，会被阻塞，直到有元素可以出队才返回。

既然是分布式的队列，那就意味着，我们可以在一个节点将元素放入队列，在另外一个节点把它取出。

在我接下来讲的例子中，你就可以启动两个节点，一个节点往队列中放入元素，一个节点从队列中取出元素，看看是否能正常取出来。etcd 的分布式队列是一种多读多写的队列，所以，你也可以启动多个写节点和多个读节点。

下面我们来借助代码，看一下如何实现分布式队列。

首先，我们启动一个程序，它会从命令行读取你的命令，然后执行。你可以输入push <value>，将一个元素入队，输入pop，将一个元素弹出。另外，你还可以使用这个程序启动多个实例，用来模拟分布式的环境：

复制代码

```
1 package main
2
3
4 import (
5     "bufio"
6     "flag"
7     "fmt"
8     "log"
9     "os"
10    "strings"
11
12
13    "github.com/coreos/etcd/clientv3"
14    recipe "github.com/coreos/etcd/contrib/recipes"
15 )
16
17
18 var (
19     addr      = flag.String("addr", "http://127.0.0.1:2379", "etcd addresses")
20     queueName = flag.String("name", "my-test-queue", "queue name")
21 )
22
23
24 func main() {
25     flag.Parse()
26
27
28     // 解析etcd地址
29     endpoints := strings.Split(*addr, ",")
30
31
32     // 创建etcd的client
33     cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
34     if err != nil {
35         log.Fatal(err)
```

```

36     }
37     defer cli.Close()
38
39
40     // 创建/获取队列
41     q := recipe.NewQueue(cli, *queueName)
42
43
44     // 从命令行读取命令
45     consoleScanner := bufio.NewScanner(os.Stdin)
46     for consoleScanner.Scan() {
47         action := consoleScanner.Text()
48         items := strings.Split(action, " ")
49         switch items[0] {
50             case "push": // 加入队列
51                 if len(items) != 2 {
52                     fmt.Println("must set value to push")
53                     continue
54                 }
55                 q.Enqueue(items[1]) // 入队
56             case "pop": // 从队列弹出
57                 v, err := q.Dequeue() // 出队
58                 if err != nil {
59                     log.Fatal(err)
60                 }
61                 fmt.Println(v) // 输出出队的元素
62             case "quit", "exit": //退出
63                 return
64             default:
65                 fmt.Println("unknown action")
66             }
67         }
68     }

```

我们可以打开两个终端，分别执行这个程序。在第一个终端中执行入队操作，在第二个终端中执行出队操作，并且观察一下出队、入队是否正常。

除了刚刚说的分布式队列，etcd 还提供了优先级队列（PriorityQueue）。

它的用法和队列类似，也提供了出队和入队的操作，只不过，在入队的时候，除了需要把一个值加入到队列，我们还需要提供 `uint16` 类型的一个整数，作为此值的优先级，优先级高的元素会优先出队。

优先级队列的测试程序如下，你可以在一个节点输入一些不同优先级的元素，在另外一个节点读取出来，看看它们是不是按照优先级顺序弹出的：

 复制代码

```
1 package main
2
3
4 import (
5     "bufio"
6     "flag"
7     "fmt"
8     "log"
9     "os"
10    "strconv"
11    "strings"
12
13
14    "github.com/coreos/etcd/clientv3"
15    recipe "github.com/coreos/etcd/contrib/recipes"
16 )
17
18
19 var (
20     addr      = flag.String("addr", "http://127.0.0.1:2379", "etcd addresses")
21     queueName = flag.String("name", "my-test-queue", "queue name")
22 )
23
24
25 func main() {
26     flag.Parse()
27
28
29     // 解析etcd地址
30     endpoints := strings.Split(*addr, ",")
31
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
40
41     // 创建/获取队列
42     q := recipe.NewPriorityQueue(cli, *queueName)
43
44
45     // 从命令行读取命令
46     consoleScanner := bufio.NewScanner(os.Stdin)
47     for consoleScanner.Scan() {
48         action := consoleScanner.Text()
49         items := strings.Split(action, " ")
50         switch items[0] {
```

```

51     case "push": // 加入队列
52         if len(items) != 3 {
53             fmt.Println("must set value and priority to push")
54             continue
55         }
56         pr, err := strconv.Atoi(items[2]) // 读取优先级
57         if err != nil {
58             fmt.Println("must set uint16 as priority")
59             continue
60         }
61         q.Enqueue(items[1], uint16(pr)) // 入队
62     case "pop": // 从队列弹出
63         v, err := q.Dequeue() // 出队
64         if err != nil {
65             log.Fatal(err)
66         }
67         fmt.Println(v) // 输出出队的元素
68     case "quit", "exit": // 退出
69         return
70     default:
71         fmt.Println("unknown action")
72     }
73 }
74 }
```

你看，利用 etcd 实现分布式队列和分布式优先队列，就是这么简单。所以，在实际项目中，如果有这类需求的话，你就可以选择用 etcd 实现。

不过，在使用分布式并发原语时，除了需要考虑可用性和数据一致性，还需要考虑分布式设计带来的性能损耗问题。所以，在使用之前，你一定要做好性能的评估。

分布式栅栏

在 [第 17 讲](#) 中，我们学习了循环栅栏 CyclicBarrier，它和 [第 6 讲](#) 的标准库中的 WaitGroup，本质上是同一类并发原语，都是等待同一组 goroutine 同时执行，或者是等待同一组 goroutine 都完成。

在分布式环境中，我们也会遇到这样的场景：一组节点协同工作，共同等待一个信号，在信号未出现前，这些节点会被阻塞住，而一旦信号出现，这些阻塞的节点就会同时开始继续执行下一步的任务。

etcd 也提供了相应的分布式并发原语。

Barrier: 分布式栅栏。如果持有 Barrier 的节点释放了它，所有等待这个 Barrier 的节点就不会被阻塞，而是会继续执行。

DoubleBarrier: 计数型栅栏。在初始化计数型栅栏的时候，我们就必须提供参与节点的数量，当这些数量的节点都 Enter 或者 Leave 的时候，这个栅栏就会放开。所以，我们把它称为计数型栅栏。

Barrier: 分布式栅栏

我们先来学习下分布式 Barrier。

分布式 Barrier 的创建很简单，你只需要提供 etcd 的 Client 和 Barrier 的名字就可以了，如下所示：

 复制代码

```
1 func NewBarrier(client *v3.Client, key string) *Barrier
```

Barrier 提供了三个方法，分别是 Hold、**Release** 和 **Wait**，代码如下：

 复制代码

```
1 func (b *Barrier) Hold() error
2 func (b *Barrier) Release() error
3 func (b *Barrier) Wait() error
```

Hold 方法是创建一个 Barrier。如果 Barrier 已经创建好了，有节点调用它的 Wait 方法，就会被阻塞。

Release 方法是释放这个 Barrier，也就是打开栅栏。如果使用了这个方法，所有被阻塞的节点都会被放行，继续执行。

Wait 方法会阻塞当前的调用者，直到这个 Barrier 被 release。如果这个栅栏不存在，调用者不会被阻塞，而是会继续执行。

学习并发原语最好的方式就是使用它。下面我们就来借助一个例子，来看看 Barrier 该怎么用。

你可以在一个终端中运行这个程序，执行"hold""release"命令，模拟栅栏的持有和释放。在另外一个终端中运行这个程序，不断调用"wait"方法，看看是否能正常地跳出阻塞继续执行：

 复制代码

```
1 package main
2
3
4 import (
5     "bufio"
6     "flag"
7     "fmt"
8     "log"
9     "os"
10    "strings"
11
12
13    "github.com/coreos/etcd/clientv3"
14    recipe "github.com/coreos/etcd/contrib/recipes"
15 )
16
17
18 var (
19     addr         = flag.String("addr", "http://127.0.0.1:2379", "etcd addresses")
20     barrierName = flag.String("name", "my-test-queue", "barrier name")
21 )
22
23
24 func main() {
25     flag.Parse()
26
27
28     // 解析etcd地址
29     endpoints := strings.Split(*addr, ",")
30
31
32     // 创建etcd的client
33     cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
34     if err != nil {
35         log.Fatal(err)
36     }
37     defer cli.Close()
38
39
40     // 创建/获取栅栏
41     b := recipe.NewBarrier(cli, *barrierName)
42
43
44     // 从命令行读取命令
```

```

45  consolescanner := bufio.NewScanner(os.Stdin)
46  for consolescanner.Scan() {
47      action := consolescanner.Text()
48      items := strings.Split(action, " ")
49      switch items[0] {
50          case "hold": // 持有这个barrier
51              b.Hold()
52              fmt.Println("hold")
53          case "release": // 释放这个barrier
54              b.Release()
55              fmt.Println("released")
56          case "wait": // 等待barrier被释放
57              b.Wait()
58              fmt.Println("after wait")
59          case "quit", "exit": //退出
60              return
61          default:
62              fmt.Println("unknown action")
63      }
64  }
65 }
```

DoubleBarrier：计数型栅栏

etcd 还提供了另外一种栅栏，叫做 DoubleBarrier，这也是一种非常有用的栅栏。这个栅栏初始化的时候需要提供一个计数 count，如下所示：

 复制代码

```
1 func NewDoubleBarrier(s *concurrency.Session, key string, count int) *DoubleBa
```

同时，它还提供了两个方法，分别是 Enter 和 Leave，代码如下：

 复制代码

```
1 func (b *DoubleBarrier) Enter() error
2 func (b *DoubleBarrier) Leave() error
```

我来解释下这两个方法的作用。

当调用者调用 Enter 时，会被阻塞住，直到一共有 count (初始化这个栅栏的时候设定的值) 个节点调用了 Enter，这 count 个被阻塞的节点才能继续执行。所以，你可以利用它编排一组节点，让这些节点在同一个时刻开始执行任务。

同理，如果你想让一组节点在同一个时刻完成任务，就可以调用 `Leave` 方法。节点调用 `Leave` 方法的时候，会被阻塞，直到有 `count` 个节点，都调用了 `Leave` 方法，这些节点才能继续执行。

我们再来看一下 DoubleBarrier 的使用例子。你可以起两个节点，同时执行 Enter 方法，看看这两个节点是不是先阻塞，之后才继续执行。然后，你再执行 Leave 方法，也观察一下，是不是先阻塞又继续执行的。

复制代码

```

39  defer cli.Close()
40  // 创建session
41  s1, err := concurrency.NewSession(cli)
42  if err != nil {
43      log.Fatal(err)
44  }
45  defer s1.Close()
46
47
48  // 创建/获取栅栏
49  b := recipe.NewDoubleBarrier(s1, *barrierName, *count)
50
51
52  // 从命令行读取命令
53  consoleScanner := bufio.NewScanner(os.Stdin)
54  for consoleScanner.Scan() {
55      action := consoleScanner.Text()
56      items := strings.Split(action, " ")
57      switch items[0] {
58      case "enter": // 持有这个barrier
59          b.Enter()
60          fmt.Println("enter")
61      case "leave": // 释放这个barrier
62          b.Leave()
63          fmt.Println("leave")
64      case "quit", "exit": //退出
65          return
66      default:
67          fmt.Println("unknown action")
68      }
69  }
70 }
```

好了，我们先来简单总结一下。我们在第 17 讲学习的循环栅栏，控制的是同一个进程中的不同 goroutine 的执行，而**分布式栅栏和计数型栅栏控制的是不同节点、不同进程的执行**。当你需要协调一组分布式节点在某个时间点同时运行的时候，可以考虑 etcd 提供的这组并发原语。

STM

提到事务，你肯定不陌生。在开发基于数据库的应用程序的时候，我们经常用到事务。事务就是要保证一组操作要么全部成功，要么全部失败。

在学习 STM 之前，我们要先了解一下 etcd 的事务以及它的问题。

etcd 提供了在一个事务中对多个 key 的更新功能，这一组 key 的操作要么全部成功，要么全部失败。etcd 的事务实现方式是基于 CAS 方式实现的，融合了 Get、Put 和 Delete 操作。

etcd 的事务操作如下，分为条件块、成功块和失败块，条件块用来检测事务是否成功，如果成功，就执行 Then(...)，如果失败，就执行 Else(...):

复制代码

```
1 Txn().If(cond1, cond2, ...).Then(op1, op2, ...).Else(op1', op2', ...)
```

我们来看一个利用 etcd 的事务实现转账的小例子。我们从账户 from 向账户 to 转账 amount，代码如下：

复制代码

```
1 func doTxnXfer(etcd *v3.Client, from, to string, amount uint) (bool, error) {
2     // 一个查询事务
3     getresp, err := etcd.Txn(ctx.TODO()).Then(OpGet(from), OpGet(to)).Commit()
4     if err != nil {
5         return false, err
6     }
7     // 获取转账账户的值
8     fromKV := getresp.Responses[0].GetRangeResponse().Kvs[0]
9     toKV := getresp.Responses[1].GetRangeResponse().Kvs[1]
10    fromV, toV := toUInt64(fromKV.Value), toUInt64(toKV.Value)
11    if fromV < amount {
12        return false, fmt.Errorf("insufficient value")
13    }
14    // 转账事务
15    // 条件块
16    txn := etcd.Txn(ctx.TODO()).If(
17        v3.Compare(v3.ModRevision(from), "=", fromKV.ModRevision),
18        v3.Compare(v3.ModRevision(to), "=", toKV.ModRevision))
19    // 成功块
20    txn = txn.Then(
21        OpPut(from, fromUInt64(fromV - amount)),
22        OpPut(to, fromUInt64(toV + amount)))
23    // 提交事务
24    putresp, err := txn.Commit()
25    // 检查事务的执行结果
26    if err != nil {
27        return false, err
28    }
29    return putresp.Succeeded, nil
30 }
```

从刚刚的这段代码中，我们可以看到，虽然可以利用 etcd 实现事务操作，但是逻辑还是比较复杂的。

因为事务使用起来非常麻烦，所以 etcd 又在这些基础 API 上进行了封装，新增了一种叫做 STM 的操作，提供了更加便利的方法。

下面我们来看一看 STM 怎么用。

要使用 STM，你需要先编写一个 apply 函数，这个函数的执行是在一个事务之中的：

 复制代码

```
1 apply func(STM) error
```

这个方法包含一个 STM 类型的参数，它提供了对 key 值的读写操作。

STM 提供了 4 个方法，分别是 Get、Put、Receive 和 Delete，代码如下：

 复制代码

```
1 type STM interface {
2     Get(key ...string) string
3     Put(key, val string, opts ...v3.OpOption)
4     Rev(key string) int64
5     Del(key string)
6 }
```

使用 etcd STM 的时候，我们只需要定义一个 apply 方法，比如说转账方法 exchange，然后通过 concurrency.NewSTM(cli, exchange)，就可以完成转账事务的执行了。

STM 作用呢？我们还是借助一个例子来学习下。

下面这个例子创建了 5 个银行账号，然后随机选择一些账号两两转账。在转账的时候，要把源账号一半的钱要转给目标账号。这个例子启动了 10 个 goroutine 去执行这些事务，每个 goroutine 要完成 100 个事务。

为了确认事务是否出错了，我们最后要校验每个账号的钱数和总钱数。总钱数不变，就代表执行成功了。这个例子的代码如下：

复制代码

```
1 package main
2
3
4 import (
5     "context"
6     "flag"
7     "fmt"
8     "log"
9     "math/rand"
10    "strings"
11    "sync"
12
13
14    "github.com/coreos/etcd/clientv3"
15    "github.com/coreos/etcd/clientv3/concurrency"
16 )
17
18
19 var (
20     addr = flag.String("addr", "http://127.0.0.1:2379", "etcd addresses")
21 )
22
23
24 func main() {
25     flag.Parse()
26
27
28     // 解析etcd地址
29     endpoints := strings.Split(*addr, ",")
30
31
32     cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
33     if err != nil {
34         log.Fatal(err)
35     }
36     defer cli.Close()
37
38
39     // 设置5个账户，每个账号都有100元，总共500元
40     totalAccounts := 5
41     for i := 0; i < totalAccounts; i++ {
42         k := fmt.Sprintf("accts/%d", i)
43         if _, err = cli.Put(context.TODO(), k, "100"); err != nil {
44             log.Fatal(err)
45         }
46     }
```

```

47
48
49 // STM的应用函数，主要的事务逻辑
50 exchange := func(stm concurrency.STM) error {
51     // 随机得到两个转账账号
52     from, to := rand.Intn(totalAccounts), rand.Intn(totalAccounts)
53     if from == to {
54         // 自己不和自己转账
55         return nil
56     }
57     // 读取账号的值
58     fromK, toK := fmt.Sprintf("accts/%d", from), fmt.Sprintf("accts/%d", t
59     fromV, toV := stm.Get(fromK), stm.Get(toK)
60     fromInt, toInt := 0, 0
61     fmt.Sscanf(fromV, "%d", &fromInt)
62     fmt.Sscanf(toV, "%d", &toInt)
63
64
65     // 把源账号一半的钱转账给目标账号
66     xfer := fromInt / 2
67     fromInt, toInt = fromInt-xfer, toInt+xfer
68
69
70     // 把转账后的值写回
71     stm.Put(fromK, fmt.Sprintf("%d", fromInt))
72     stm.Put(toK, fmt.Sprintf("%d", toInt))
73     return nil
74 }
75
76
77 // 启动10个goroutine进行转账操作
78 var wg sync.WaitGroup
79 wg.Add(10)
80 for i := 0; i < 10; i++ {
81     go func() {
82         defer wg.Done()
83         for j := 0; j < 100; j++ {
84             _, serr := concurrency.NewSTM(cli, exchange); serr != nil {
85                 log.Fatal(serr)
86             }
87         }
88     }()
89 }
90 wg.Wait()
91
92
93 // 检查账号最后的数目
94 sum := 0
95 accts, err := cli.Get(context.TODO(), "accts/", clientv3.WithPrefix()) //
96 if err != nil {
97     log.Fatal(err)
98 }

```

```
99     for _, kv := range accts.Kvs { // 遍历账号的值
100         v := 0
101         fmt.Sscanf(string(kv.Value), "%d", &v)
102         sum += v
103         log.Printf("account %s: %d", kv.Key, v)
104     }
105
106
107     log.Println("account sum is", sum) // 总数
108 }
```

总结一下，当你利用 etcd 做存储时，是可以利用 STM 实现事务操作的，一个事务可以包含多个账号的数据更改操作，事务能够保证这些更改要么全成功，要么全失败。

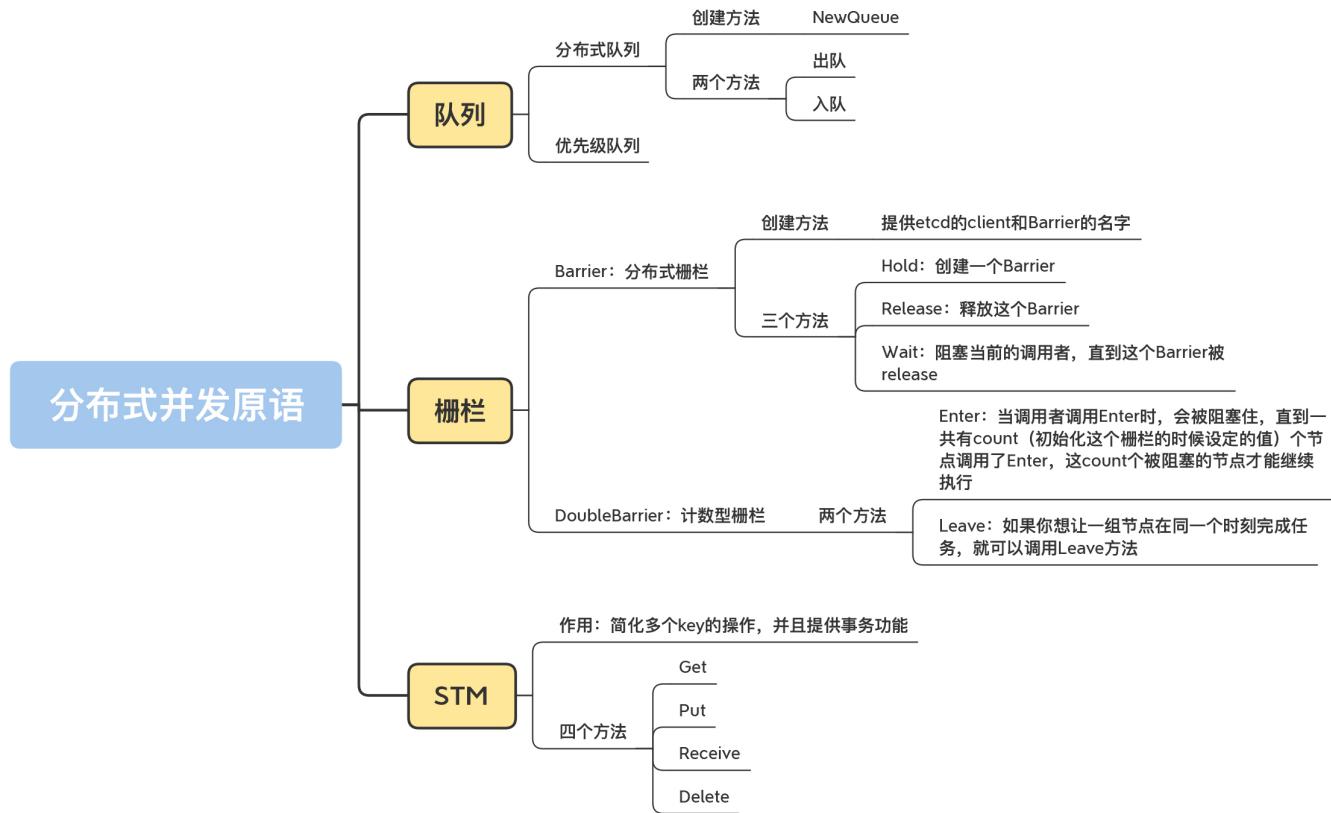
总结

如果我们把眼光放得更宽广一些，其实并不只是 etcd 提供了这些并发原语，比如我上节课一开始就提到了，Zookeeper 很早也提供了类似的并发原语，只不过只提供了 Java 的库，并没有提供合适的 Go 库。另外，根据 Consul 官方的反馈，他们并没有开发这些并发原语的计划，所以，从目前来看，etcd 是个不错的选择。

当然，也有一些其它不太知名的分布式原语库，但是活跃度不高，可用性低，所以我们也不需要去了解了。

其实，你也可以使用 Redis 实现分布式锁，或者是基于 MySQL 实现分布式锁，这也是常用的选择。对于大厂来说，选择起来是非常简单的，只需要看看厂内提供了哪个基础服务，哪个更稳定些。对于没有 etcd、Redis 这些基础服务的公司来说，很重要的一点，就是自己搭建一套这样的基础服务，并且运维好，这就需要考察你们对 etcd、Redis、MySQL 的技术把控能力了，哪个用得更顺手，就用哪个。

一般来说，我不建议你自己去实现分布式原语，最好是直接使用 etcd、Redis 这些成熟的软件提供的功能，这也意味着，我们将程序的风险转嫁到了这些基础服务上，这些基础服务必须要能够提供足够的服务保障。



思考题

- 部署一个3节点的etcd集群，测试一下分布式队列的性能。
- etcd提供的STM是分布式事务吗？

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 19 | 在分布式环境中，Leader选举、互斥锁和读写锁该如何实现？

下一篇 结束语 | 再聊Go并发编程的价值和精进之路

精选留言 (2)

 写留言



橙子888

2020-11-26

打卡。

展开 ▼



myrfy

2020-11-25

感觉是包装了一层最基础的乐观锁，离分布式事务应该还差不少吧

