锁

# 数据库锁

数据库锁

乐观锁

悲观锁

共享锁

独占锁

写锁

行锁/更新锁

表锁

读锁

数据库锁

乐观锁

悲观锁：互斥锁和共享锁 （其中互斥锁又称独占锁，共享锁包括读锁）

互斥锁包括：写锁、行锁、表锁

## 理解一（精简）：

**悲观锁 和 乐观锁**

**（1）悲观锁：**顾名思义，就是很悲观，**每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁**，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

**（2）乐观锁：** 顾名思义，就是很乐观，**每次去拿数据的时候都认为别人不会修改，所以不会上锁**，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

**（3）悲观锁 和 乐观锁的区别：**

两种锁各有优缺点，不可认为一种好于另一种，像**乐观锁适用于写比较少的情况下**，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但**如果经常产生冲突**，上层应用会不断的进行retry，这样反倒是降低了性能，**所以这种情况下用悲观锁就比较合适**。

**（4）行锁 和 表锁**

主要是针对锁粒度划分的。

一般分为：行锁、表锁、库锁

（1）行锁：访问数据库的时候，锁定整个行数据，防止并发错误。

（2）表锁：访问数据库的时候，锁定整个表数据，防止并发错误。

**行锁 和 表锁 的区别：**

表锁： 开销小，加锁快；不会出现死锁；锁定力度大，发生锁冲突概率高，并发度最低

行锁： 开销大，加锁慢；会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高

**（5）行锁加锁方法**：

**行级锁**是一种排他锁，防止其他事务修改此行；

在使用以下语句时，Mysql / Oracle会自动应用行级锁：  
INSERT、UPDATE、DELETE、SELECT … FOR UPDATE [OF columns] [WAIT n | NOWAIT];  
SELECT … FOR UPDATE语句允许用户一次锁定多条记录进行更新  
使用COMMIT或ROLLBACK语句释放锁

注意：

InnoDB引擎的行锁是通过给索引项加锁来实现的，这一点mysql和[oracle](https://www.baidu.com/s?wd=oracle&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)不同，

后者是通过在数据库中对相应的数据行加锁来实现的，

InnoDB这种行级锁决定，只有通过索引条件来检索数据，才能使用行级锁，否则，直接使用表级锁。

特别注意:使用行级锁一定要使用索引

**（6）表级锁**又分为5类：

行共享 (ROW SHARE) – 禁止排他锁定表  
行排他(ROW EXCLUSIVE) – 禁止使用排他锁和共享锁  
共享锁(SHARE) - 锁定表，对记录只读不写，多个用户可以同时在同一个表上应用此锁  
共享行排他(SHARE ROW EXCLUSIVE) – 比共享锁更多的限制，禁止使用共享锁及更高的锁  
排他(EXCLUSIVE) – 限制最强的表锁，仅允许其他用户查询该表的行。禁止修改和锁定表。

## 理解二（详细）：

独占锁、共享锁、更新锁，乐观锁、悲观锁

1、锁的两种分类方式

### 从数据库系统的角度看，锁分为以下三种类型：

独占锁（Exclusive Lock）  
      独占锁锁定的资源只允许进行锁定操作的程序使用，其它任何对它的操作均不会被接受。执行数据更新命令，即INSERT、 UPDATE 或DELETE 命令时，SQL Server 会自动使用独占锁。但当对象上有其它锁存在时，无法对其加独占锁。独占锁一直到事务结束才能被释放。  
 共享锁（Shared Lock）  
      共享锁锁定的资源可以被其它用户读取，但其它用户不能修改它。在SELECT 命令执行时，SQL Server 通常会对对象进行共享锁锁定。通常加共享锁的数据页被读取完毕后，共享锁就会立即被释放。  
 更新锁（Update Lock行锁）  
      更新锁是为了防止死锁而设立的。当SQL Server 准备更新数据时，它首先对数据对象作更新锁锁定，这样数据将不能被修改，但可以读取。等到SQL Server 确定要进行更新数据操作时，它会自动将更新锁换为独占锁。但当对象上有其它锁存在时，无法对其作更新锁锁定。

### 从程序员的角度看，锁分为以下两种类型：

悲观锁（Pessimistic Lock）  
      悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

乐观锁（Optimistic Lock）  
      相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。  
      而乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

## 、数据库中如何使用锁

首先从悲观锁开始说。在SqlServer等其余很多数据库中，数据的锁定通常采用页级锁的方式，也就是说对一张表内的数据是一种串行化的更新插入机制，在任何时间同一张表只会插1条数据，别的想插入的数据要等到这一条数据插完以后才能依次插入。带来的后果就是性能的降低，在多用户并发访问的时候，当对一张表进行频繁操作时，会发现响应效率很低，数据库经常处于一种假死状态。而Oracle用的是行级锁，只是对想锁定的数据才进行锁定，其余的数据不相干，所以在对Oracle表中并发插数据的时候，基本上不会有任何影响。

注：对于悲观锁是针对并发的可能性比较大，而一般在我们的应用中用乐观锁足以。

### 悲观锁使用

Oracle的悲观锁需要利用一条现有的连接，分成两种方式，从SQL语句的区别来看，就是一种是for update，一种是for update nowait的形式。

比如我们看一个例子。首先建立测试用的数据库表：

CREATE TABLE TEST(ID,NAME,LOCATION,VALUE,CONSTRAINT test\_pk PRIMARY KEY(ID))AS SELECT deptno, dname, loc, **1** FROM scott.dept

这里我们利用了Oracle的Sample的scott用户的表，把数据copy到我们的test表中。  
  
（1）for update 形式介绍  
  
然后我们看一下for update锁定方式。我们执行如下的select for update语句：

select \* from test where id = **10** for update

通过这条检索语句锁定以后，再开另外一个sql\*plus窗口进行操作，再把上面这条sql语句执行一便，你会发现sqlplus好像死在那里了，好像检索不到数据的样子，但是也不返回任何结果，就属于卡在那里的感觉。这个时候是什么原因呢，就是一开始的第一个Session中的select for update语句把数据锁定住了。由于这里锁定的机制是wait的状态(只要不表示nowait那就是wait)，所以第二个Session(也就是卡住的那个sql\*plus)中当前这个检索就处于等待状态。当第一个session最后commit或者rollback之后，第二个session中的检索结果就是自动跳出来，并且也把数据锁定住。  
  
不过如果你第二个session中你的检索语句如下所示：select \* from test where id = 10，也就是没有for update这种锁定数据的语句的话，就不会造成阻塞了。

（2）for update nowait 形式介绍

另外一种情况，就是当数据库数据被锁定的时候，也就是执行刚才for update那条sql以后，我们在另外一个session中执行for update nowait后又是什么样呢。  
比如如下的sql语句:

select \* from test where id = **10** for update nowait

由于这条语句中是制定采用nowait方式来进行检索，所以当发现数据被别的session锁定中的时候，就会迅速返回ORA-00054错误，内容是资源正忙, 但指定以 NOWAIT 方式获取资源。所以在程序中我们可以采用nowait方式迅速判断当前数据是否被锁定中，如果锁定中的话，就要采取相应的业务措施进行处理。  
  
那这里另外一个问题，就是当我们锁定住数据的时候，我们对数据进行更新和删除的话会是什么样呢。  
  
比如同样，我们让第一个Session锁定住id=10的那条数据，我们在第二个session中执行如下语句：

update test set value=**2** where id = **10**

这个时候我们发现update语句就好像select for update语句一样也停住卡在这里，当你第一个session放开锁定以后update才能正常运行。当你update运行后，数据又被你update 语句锁定住了，这个时候只要你update后还没有commit，别的session照样不能对数据进行锁定更新等等。  
  
总之，Oracle中的悲观锁就是利用Oracle的Connection对数据进行锁定。在Oracle中，用这种行级锁带来的性能损失是很小的，只是要注意程序逻辑，不要给你一不小心搞成死锁了就好。而且由于数据的及时锁定，在数据提交时候就不呼出现冲突，可以省去很多恼人的数据冲突处理。缺点就是你必须要始终有一条数据库连接，就是说在整个锁定到最后放开锁的过程中，你的数据库联接要始终保持住。  
  
与悲观锁相对的，我们有了乐观锁。乐观锁一开始也说了，就是一开始假设不会造成数据冲突，在最后提交的时候再进行数据冲突检测。

### 乐观锁使用，

我们有3种常用的做法来实现：

a. 在数据取得的时候把整个数据都copy到应用中，在进行提交的时候比对当前数据库中的数据和开始的时候更新前取得的数据。  
    当发现两个数据一模一样以后，就表示没有冲突可以提交，否则则是并发冲突，需要去用业务逻辑进行解决。

b. 乐观锁的做法就是采用版本戳，这个在Hibernate中得到了使用。  
    采用版本戳的话，首先需要在你有乐观锁的数据库table上建立一个新的column，比如为number型，当你数据每更新一次的时候，版本数就会往上增加1。  
    比如同样有2个session同样对某条数据进行操作。两者都取到当前的数据的版本号为1，当第一个session进行数据更新后，在提交的时候查看到当前数据的版本还为1，和自己一开始取到的版本相同。就正式提交，然后把版本号增加1，这个时候当前数据的版本为2。当第二个session也更新了数据提交的时候，发现数据库中版本为2，和一开始这个session取到的版本号不一致，就知道别人更新过此条数据，这个时候再进行业务处理，比如整个Transaction都Rollback等等操作。  
    在用版本戳的时候，可以在应用程序侧使用版本戳的验证，也可以在数据库侧采用Trigger(触发器)来进行验证。不过数据库的Trigger的性能开销还是比较的大，所以能在应用侧进行验证的话还是推荐不用Trigger。  
  
c. 第三种做法和第二种做法有点类似，就是也新增一个Table的Column，不过这次这个column是采用timestamp型，存储数据最后更新的时间。  
    在Oracle9i以后可以采用新的数据类型，也就是timestamp with time zone类型来做时间戳。这种Timestamp的数据精度在Oracle的时间类型中是最高的，精确到微秒(还没与到纳秒的级别)，一般来说，加上数据库处理时间和人的思考动作时间，微秒级别是非常非常够了，其实只要精确到毫秒甚至秒都应该没有什么问题。  
    和刚才的版本戳类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。如果不想把代码写在程序中或者由于别的原因无法把代码写在现有的程序中，也可以把这个时间戳乐观锁逻辑写在Trigger或者存储过程中。

# 分布式锁

目前几乎很多大型网站及应用都是分布式部署的，分布式场景中的数据一致性问题一直是一个比较重要的话题。

分布式的CAP理论告诉我们“任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。”

所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

在很多场景中，我们为了保证数据的最终一致性，需要很多的技术方案来支持，比如分布式事务、分布式锁等。有的时候，我们需要保证一个方法在同一时间内只能被同一个线程执行。在单机环境中，Java中其实提供了很多并发处理相关的API，但是这些API在分布式场景中就无能为力了。也就是说单纯的Java Api并不能提供分布式锁的能力。所以针对分布式锁的实现目前有多种方案。

针对分布式锁的实现，目前比较常用的有以下几种方案：

1 基于数据库实现分布式锁

2 基于缓存（redis，memcached，tair）实现分布式锁

3 基于Zookeeper实现分布式锁

在分析这几种实现方案之前我们先来想一下，我们需要的分布式锁应该是怎么样的？（这里以方法锁为例，资源锁同理）

分布式锁要有那写特性：

* 阻塞锁 ->如果不行就是用while循环
* 可释放 ->这个不好整
* 可重入 ->如果不行，就添加主机和资源信息对比，相同可获取锁
* 高可用 ->数据库集群、Zookeeper集群、Redis集群

1 可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。

2 这把锁要是一把可重入锁（避免死锁）

3 这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）

4 有高可用的获取锁和释放锁功能

5获取锁和释放锁的性能要好

## 基于数据库

### 基于数据库的表数据

要实现分布式锁，最简单的方式可能就是直接创建一张锁表，然后通过操作该表中的数据来实现了。

当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

创建这样一张数据库表：



当我们想要锁住某个方法时，执行以下SQL：

https://images2017.cnblogs.com/blog/883340/201712/883340-20171215162325324-1574794414.png

因为我们对method\_name做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

当方法执行完毕之后，想要释放锁的话，需要执行以下Sql:

https://images2017.cnblogs.com/blog/883340/201712/883340-20171215162408918-407807720.png

上面这种简单的实现有以下几个问题：

1、这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。

2、这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。

3、这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。

4、这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

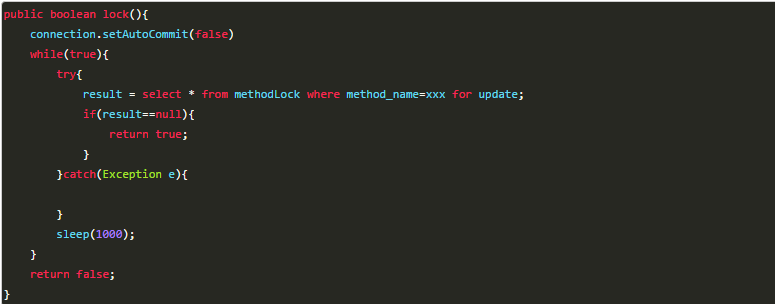
当然，我们也可以有其他方式解决上面的问题。

* 数据库是单点？ 搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。
* 没有失效时间？ 只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。
* 非阻塞的？ 搞一个while循环，直到insert成功再返回成功。
* 非重入的？ 在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

### 基于数据库的锁机制（互斥锁/行锁/更新锁）

除了可以通过增删操作数据表中的记录以外，其实还可以借助数据中自带的锁来实现分布式的锁。

我们还用刚刚创建的那张数据库表。可以通过数据库的排他锁来实现分布式锁。 基于MySql的InnoDB引擎，可以使用以下方法来实现加锁操作：



在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁（这里再多提一句，InnoDB引擎在加锁的时候，只有通过索引进行检索的时候才会使用行级锁，否则会使用表级锁。这里我们希望使用行级锁，就要给method\_name添加索引，值得注意的是，这个索引一定要创建成唯一索引，否则会出现多个重载方法之间无法同时被访问的问题。重载方法的话建议把参数类型也加上。）。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。

我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，再通过以下方法解锁：

https://images2017.cnblogs.com/blog/883340/201712/883340-20171215162527355-711937824.png

通过connection.commit()操作来释放锁。

这种方法可以有效的解决上面提到的无法释放锁和阻塞锁的问题。

* 阻塞锁？ for update语句会在执行成功后立即返回，在执行失败时一直处于阻塞状态，直到成功。
* 锁定之后服务宕机，无法释放？使用这种方式，服务宕机之后数据库会自己把锁释放掉。

但是还是无法直接解决数据库单点和可重入问题。

这里还可能存在另外一个问题，虽然我们对method\_name 使用了唯一索引，并且显示使用for update来使用行级锁。但是，MySql会对查询进行优化，即便在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫效率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 将使用表锁，而不是行锁。如果发生这种情况就悲剧了。。。

还有一个问题，就是我们要使用排他锁来进行分布式锁的lock，那么一个排他锁长时间不提交，就会占用数据库连接。一旦类似的连接变得多了，就可能把数据库连接池撑爆

### 总结

总结一下使用数据库来实现分布式锁的方式：

1 这两种方式都是依赖数据库的一张表，

2 一种是通过表中的记录的存在情况确定当前是否有锁存在，

3 另外一种是通过数据库的排他锁来实现分布式锁。

数据库实现分布式锁的优点

直接借助数据库，容易理解。

数据库实现分布式锁的缺点

会有各种各样的问题，在解决问题的过程中会使整个方案变得越来越复杂。

操作数据库需要一定的开销，性能问题需要考虑。

使用数据库的行级锁并不一定靠谱，尤其是当我们的锁表并不大的时候。

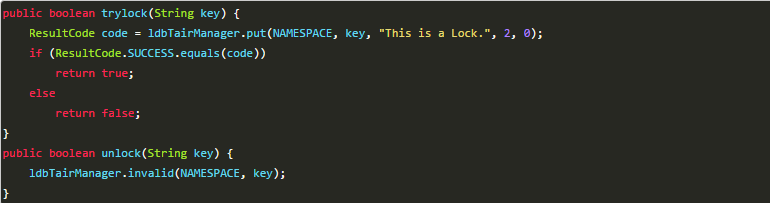
## 基于Redis缓存

相比较于基于数据库实现分布式锁的方案来说，基于缓存来实现在性能方面会表现的更好一点。而且很多缓存是可以集群部署的，可以解决单点问题。

目前有很多成熟的缓存产品，包括Redis，memcached以及我们公司内部的Tair。

这里以Tair为例来分析下使用缓存实现分布式锁的方案。关于Redis和memcached在网络上有很多相关的文章，并且也有一些成熟的框架及算法可以直接使用。

基于Tair的实现分布式锁其实和Redis类似，其中主要的实现方式是使用TairManager.put方法来实现。



以上实现方式同样存在几个问题：

1、这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在tair中，其他线程无法再获得到锁。

2、这把锁只能是非阻塞的，无论成功还是失败都直接返回。

3、这把锁是非重入的，一个线程获得锁之后，在释放锁之前，无法再次获得该锁，因为使用到的key在tair中已经存在。无法再执行put操作。

当然，同样有方式可以解决。

* 没有失效时间？ tair的put方法支持传入失效时间，到达时间之后数据会自动删除。
* 非阻塞？ while重复执行。
* 非可重入？ 在一个线程获取到锁之后，把当前主机信息和线程信息保存起来，下次再获取之前先检查自己是不是当前锁的拥有者。

但是，失效时间我设置多长时间为好？如何设置的失效时间太短，方法没等执行完，锁就自动释放了，那么就会产生并发问题。如果设置的时间太长，其他获取锁的线程就可能要平白的多等一段时间。这个问题使用数据库实现分布式锁同样存在

### 总结

可以使用缓存来代替数据库来实现分布式锁，这个可以提供更好的性能，同时，很多缓存服务都是集群部署的，可以避免单点问题。并且很多缓存服务都提供了可以用来实现分布式锁的方法，比如Tair的put方法，redis的setnx方法等。并且，这些缓存服务也都提供了对数据的过期自动删除的支持，可以直接设置超时时间来控制锁的释放。

使用缓存实现分布式锁的优点

性能好，实现起来较为方便。

使用缓存实现分布式锁的缺点

通过超时时间来控制锁的失效时间并不是十分的靠谱。

## 基于Zookeeper分布式锁

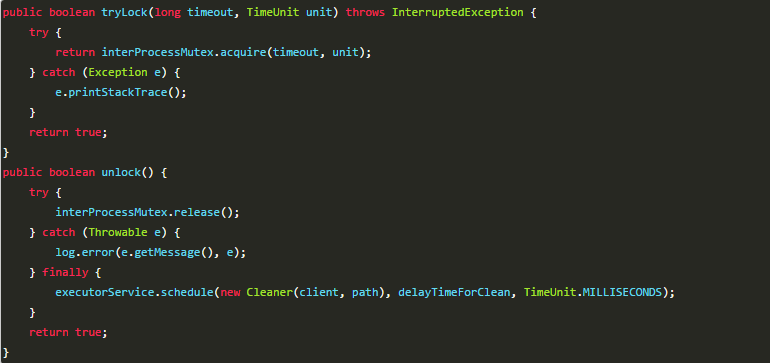
基于zookeeper临时有序节点可以实现的分布式锁。

大致思想即为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

来看下Zookeeper能不能解决前面提到的问题。

* 锁无法释放？使用Zookeeper可以有效的解决锁无法释放的问题，因为在创建锁的时候，客户端会在ZK中创建一个临时节点，一旦客户端获取到锁之后突然挂掉（Session连接断开），那么这个临时节点就会自动删除掉。其他客户端就可以再次获得锁。
* 非阻塞锁？使用Zookeeper可以实现阻塞的锁，客户端可以通过在ZK中创建顺序节点，并且在节点上绑定监听器，一旦节点有变化，Zookeeper会通知客户端，客户端可以检查自己创建的节点是不是当前所有节点中序号最小的，如果是，那么自己就获取到锁，便可以执行业务逻辑了。
* 不可重入？使用Zookeeper也可以有效的解决不可重入的问题，客户端在创建节点的时候，把当前客户端的主机信息和线程信息直接写入到节点中，下次想要获取锁的时候和当前最小的节点中的数据比对一下就可以了。如果和自己的信息一样，那么自己直接获取到锁，如果不一样就再创建一个临时的顺序节点，参与排队。
* 单点问题？使用Zookeeper可以有效的解决单点问题，ZK是集群部署的，只要集群中有半数以上的机器存活，就可以对外提供服务。

可以直接使用zookeeper第三方库[Curator](https://curator.apache.org/)客户端，这个客户端中封装了一个可重入的锁服务。



Curator提供的InterProcessMutex是分布式锁的实现。acquire方法用户获取锁，release方法用于释放锁。

使用ZK实现的分布式锁好像完全符合了本文开头我们对一个分布式锁的所有期望。但是，其实并不是，Zookeeper实现的分布式锁其实存在一个缺点，那就是性能上可能并没有缓存服务那么高。因为每次在创建锁和释放锁的过程中，都要动态创建、销毁瞬时节点来实现锁功能。ZK中创建和删除节点只能通过Leader服务器来执行，然后将数据同不到所有的Follower机器上。

其实，使用Zookeeper也有可能带来并发问题，只是并不常见而已。考虑这样的情况，由于网络抖动，客户端可ZK集群的session连接断了，那么zk以为客户端挂了，就会删除临时节点，这时候其他客户端就可以获取到分布式锁了。就可能产生并发问题。这个问题不常见是因为zk有重试机制，一旦zk集群检测不到客户端的心跳，就会重试，Curator客户端支持多种重试策略。多次重试之后还不行的话才会删除临时节点。（所以，选择一个合适的重试策略也比较重要，要在锁的粒度和并发之间找一个平衡。）

### 总结

使用Zookeeper实现分布式锁的优点

有效的解决单点问题，不可重入问题，非阻塞问题以及锁无法释放的问题。实现起来较为简单。

使用Zookeeper实现分布式锁的缺点

性能上不如使用缓存实现分布式锁。 需要对ZK的原理有所了解。

## 三种方案的比较

上面几种方式，哪种方式都无法做到完美。就像CAP一样，在复杂性、可靠性、性能等方面无法同时满足，所以，根据不同的应用场景选择最适合自己的才是王道。

### 从理解的难易程度角度（从低到高）

数据库 > 缓存 > Zookeeper

### 从实现的复杂性角度（从低到高）

Zookeeper >= 缓存 > 数据库

### 从性能角度（从高到低）

缓存 > Zookeeper >= 数据库

### 从可靠性角度（从高到低）

Zookeeper > 缓存 > 数据库

# Redis分布式锁具体实现

Redis是线程安全的：所以一下三个方案中，所说的安全问题是不能存在的。方案一就可以了。

redis是单线程服务器。它的设计并不能从多核cpu中受益。人们可以安装多个redis实例来提高多核cpu的利用率。拿redis更多线程存储服务器做对比是不公平的。每个请求被缓存在一个线程中，一个时间只能有一个线程在处理请求。

## 我的测试代码：

### 利用Redis的原子性插入方法：setNX();

没有考虑lock的过期时间，不过这个相对安全

|  |
| --- |
| **package** com.taotao.content.jedis;  **import** java.util.concurrent.CountDownLatch;  **import** org.springframework.util.StringUtils;  **import** redis.clients.jedis.Jedis;  /\*\*  \*  \* **@Description** 测试redis客户端jedis  \* **@ClassName** JedisTest  \* **@Author** [浅醉]  \* **@Date** 2019年1月14日 下午10:56:14  \*/  **public** **class** JedisTest2 {  // 测试数据  **private** **static** **int** *num*;  // Reids客户端连接  **private** Jedis jedis = **new** Jedis("192.168.153.136",6379,60000);  // 当前线程id  **private** **int** id;  **public** JedisTest2(**int** id){  **this**.id = id;  }  // 数据加一  **public** **void** numInc(){  System.***out***.println("【线程"+id+"】执行任务");  *num*++;  }  // 获取锁  **public** **boolean** getLock(){  // 原子性插入 ： 只有当不存在这个key时，才能插入成功  Long lock = jedis.setnx("lock", "someone get the lock");  // 成功返回 1 ，失败返回 0  **if**(lock == 0){  **return** **false**;  }**else**{  System.***out***.println("【线程"+id+"】获取锁");  **return** **true**;  }  }  // 释放锁，删除key，关闭连接  **public** **void** unLock(){  System.***out***.println("【线程"+id+"】释放锁");  jedis.del("lock");  jedis.close();  }    **public** **static** **void** main(String[] args) **throws** InterruptedException {    // 确保所有线程就绪后，同时抢夺资源  **final** CountDownLatch cdl = **new** CountDownLatch(1000);  // 确保所有线程结束后，在执行最后的输出结果  **final** CountDownLatch cd2 = **new** CountDownLatch(1000);  **for**(**int** i = 0;i<1000;i++){  **final** **int** id = i+1;  **new** Thread(**new** Runnable() {  **public** **void** run() {  JedisTest2 jt = **new** JedisTest2(id);  cdl.countDown();  **try** {  System.***out***.println("【线程"+id+"】就绪");  cdl.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  // 自旋直到获取锁  **while**(!jt.getLock()){  // 自旋直到获取锁  //System.out.println("【线程"+id+"】尝试获取锁");  }  jt.numInc();  jt.unLock();  cd2.countDown();  }  }).start();  }  // 先在此等待，等线程执行全部结束再往下走  cd2.await();  System.***out***.println(JedisTest2.*num*);  }  } |

### 测试代码二 采用获取值，判断是否为空，为空则获取锁，并插入值，不为空则无法获取锁

很不安全： 当a读取到值为空时，b也再a设值之前，读取到为空，并获得锁。导致 多个客户端都获取到锁。

同一把锁同一时间可能被不同客户端获取到。仔细分析一下getLock方法，该方法并不是原子性的，当一个客户端检查到某个锁不存在，并在执行setKey方法之前，别的客户端可能也会检查到该锁不存在，并也会执行setKey方法，这样一来，同一把锁就有可能被不同的客户端获取到了。

100个并发就会出问题了。

|  |
| --- |
| **package** com.taotao.content.jedis;  **import** java.util.concurrent.CountDownLatch;  **import** org.springframework.util.StringUtils;  **import** redis.clients.jedis.Jedis;  /\*\*  \*  \* **@Description** 测试redis客户端jedis  \* **@ClassName** JedisTest  \* **@Author** [浅醉]  \* **@Date** 2019年1月14日 下午10:56:14  \*/  **public** **class** JedisTest2 {  // 测试数据  **private** **static** **int** *num*;  // Reids客户端连接  **private** Jedis jedis = **new** Jedis("192.168.153.136",6379,60000);  // 当前线程id  **private** **int** id;  **public** JedisTest2(**int** id){  **this**.id = id;  }  // 数据加一  **public** **void** numInc(){  System.***out***.println("【线程"+id+"】执行任务");  *num*++;  }  // 获取锁  **public** **boolean** getLock(){  // 获取key为lock的值  String lock = jedis.get("lock");  // 值存在，说明锁被别人获取了  **if**(!StringUtils.*isEmpty*(lock)){  **return** **false**;  }**else**{  // 值不存在，则获取锁，并插入值  System.***out***.println("【线程"+id+"】获取锁");  // 很不安全： 当a读取到值为空时，b也再a设值之前，读取到为空，并获得锁。导致 多个客户端都获取到锁。  jedis.set("lock","someone get the lock");  **return** **true**;  }  }  **public** **boolean** getLock2(){  // 原子性插入 ： 只有当不存在这个key时，才能插入成功  Long lock = jedis.setnx("lock", "someone get the lock");  // 成功返回 1 ，失败返回 0  **if**(lock == 0){  **return** **false**;  }**else**{  System.***out***.println("【线程"+id+"】获取锁");  **return** **true**;  }  }  // 释放锁，删除key，关闭连接  **public** **void** unLock(){  System.***out***.println("【线程"+id+"】释放锁");  jedis.del("lock");  jedis.close();  }    **public** **static** **void** main(String[] args) **throws** InterruptedException {  **final** Person p = **new** Person();  // 确保所有线程就绪后，同时抢夺资源  **final** CountDownLatch cdl = **new** CountDownLatch(500);  // 确保所有线程结束后，在执行最后的输出结果  **final** CountDownLatch cd2 = **new** CountDownLatch(500);  **for**(**int** i = 0;i<500;i++){  **final** **int** id = i+1;  **new** Thread(**new** Runnable() {  **public** **void** run() {  JedisTest2 jt = **new** JedisTest2(id);  cdl.countDown();  **try** {  System.***out***.println("【线程"+id+"】就绪");  cdl.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  // 自旋直到获取锁  **while**(!jt.getLock2()){  // 自旋直到获取锁  //System.out.println("【线程"+id+"】尝试获取锁");  }  **int** count = p.getCount();  p.setCount(count+1);  jt.unLock();  cd2.countDown();  }  }).start();  }  // 先在此等待，等线程执行全部结束再往下走  cd2.await();  System.***out***.println(p.getCount());  }    }  **class** Person{  **private** **static** **int** *count* = 0;  **public** **int** getCount() {  **return** *count*;  }  **public** **void** setCount(**int** count) {  **this**.*count* = count;  }  } |

## Redis分布式锁方案一

使用Redis实现分布式锁最简单的方案是在获取锁之前先查询一下以该锁为key对应的value存不存在，如果存在，则说明该锁被其他客户端获取了，否则的话就尝试获取锁，获取锁的方法很简单，只要以该锁为key，设置一个随机的值就行了。比如，我们有一批任务需要由多个分布式线程处理，每个任务都有一个taskId，为了保证每个任务只被执行一次，在工作线程执行任务之前，先获取该任务的锁，锁的key可以为taskId。因此，获取锁的过程可以用如下伪代码实现：



上述就是最简单的获取锁的方案了，但是大家可以想想这个方案有什么问题呢？有没有什么潜在的坑？在分析这种方案的优缺点之前，先说一下获取锁后我们一般是怎么使用锁，并且又是如何释放锁的，以Java语言为例，我们一般获取锁后会将释放锁的代码放在finally块中，这样做的好处是即使在使用锁的过程中出现异常，也能顺利将锁释放掉。用伪代码描述如下：



其中，getLock方法的伪代码上文已经给出，releaseLock方法是释放锁的方法，在该方案中，只是简单地删除掉key，就不给出伪代码了。

上述使用锁的代码咋一看是没有什么问题的，学过Java的人都知道，在try...finally...代码块中，即使try代码块中抛出异常，最终也会执行finally代码块，然而这样就能保证锁一定会被释放吗？考虑这样一种情况：代码执行到doSomething()方法的时候，服务器宕机了，这个时候finally代码块就没法被执行了，因此在这种情况下，该锁不会被正常释放，在上述案例中，可能会导致任务漏算。因此，这种方案的第一个问题是会出现锁无法正常释放的风险，解决这个问题的方法也很简单，Redis设置key的时候可以指定一个过期时间，只要获取锁的时候设置一个合理的过期时间，那么即使服务器宕机了，也能保证锁被正确释放。

该方案的另外一个问题是，获取到的锁不一定是排他锁，也就是说同一把锁同一时间可能被不同客户端获取到。仔细分析一下getLock方法，该方法并不是原子性的，当一个客户端检查到某个锁不存在，并在执行setKey方法之前，别的客户端可能也会检查到该锁不存在，并也会执行setKey方法，这样一来，同一把锁就有可能被不同的客户端获取到了。

既然这种方案有以上缺点，那么该如何改进呢？且听我慢慢道来。

## Redis分布式锁方案二

上一小节的方案有2个缺点，一个是获取的锁可能无法释放，另一个是同一把锁在同一时间可能被不同线程获取到。通过查看Redis文档，可以找到Redis提供了一个只有在某个key不存在的情况下才会设置key的值的原子命令，该命令也能设置key值过期时间，因此使用该命令，不存在上述方案出现的问题，该命令为：

SET my\_key my\_value NX PX milliseconds

其中，NX表示只有当键key不存在的时候才会设置key的值，PX表示设置键key的过期时间，单位是毫秒。

如此一来，获取锁的过程可以用如下伪代码描述：

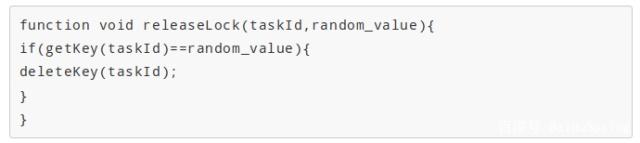


其中，setKeyOnlyIfNotExists方法表示的是原子命令SET my\_key my\_value NX PX milliseconds。

如此一来，获取锁的代码应该就没什么问题了，但是这种方案还是会有其他问题。大家再仔细研究下释放锁的代码。因为现在我们设置key的时候也设置了过期时间，所以原来的释放锁的代码现在看来就有问题了。考虑这样一种情况：客户端A获取锁的时候设置了key的过期时间为2秒，然后客户端A在获取到锁之后，业务逻辑方法doSomething执行了3秒（大于2秒），当执行完业务逻辑方法的时候，客户端A获取的锁已经被Redis过期机制自动释放了，因此客户端A在获取锁经过2秒之后，该锁可能已经被其他客户端获取到了。当客户端A执行完doSomething方法之后接下来就是执行releaseLock方法释放锁了，由于前面说了，该锁可能已经被其他客户端获取到了，因此这个时候释放锁就有可能释放的是其他客户端获取到的锁。

## Redis分布式锁方案三

既然方案二可能会出现释放了别的客户端申请的锁的问题，那么该如何进行改进呢？有一个很简单的方法是，我们设置key的时候，将value设置为一个随机值r，当释放锁，也就是删除key的时候，不是直接删除，而是先判断该key对应的value是否等于先前设置的随机值，只有当两者相等的时候才删除该key，由于每个客户端产生的随机值是不一样的，这样一来就不会误释放别的客户端申请的锁了。新的释放锁的方案用伪代码描述如下：

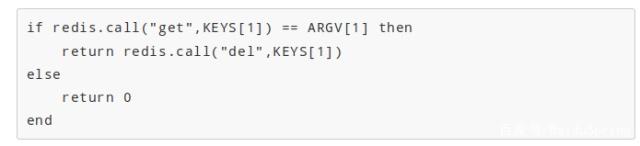


其中，getKey方法就是Redis的查询key值的方法，deleteKey就是Redis的删除key值的方法，在此不给出伪代码了。

那么这种方案就没有问题了吗？很遗憾地说，这种方案也是有问题的。原因在于上述释放锁的操作不是原子性的，不是原子性操作意味着当一个客户端执行完getKey方法并在执行deleteKey方法之前，也就是在这2个方法执行之间，其他客户端是可以执行其他命令的。考虑这样一种情况，在客户端A执行完getKey方法，并且该key对应的值也等于先前的随机值的时候，接下来客户端A将会执行deleteKey方法。假设由于网络或其他原因，客户端A执行getKey方法之后过了1秒钟才执行deleteKey方法，那么在这1秒钟里，该key有可能也会因为过期而被Redis清除了，这样一来另一个客户端，姑且称之为客户端B，就有可能在这期间获取到锁，然后接下来客户端A就执行到deleteKey方法了，如此一来就又出现误释放别的客户端申请的锁的问题了。

## Redis分布式锁方案四

既然方案三的问题是因为释放锁的方法不是原子操作导致的，那么我们只要保证释放锁的代码是原子性的就能解决该问题了。很遗憾的是，查阅Redis开发文档，并没有发现相关的原子操作。不过幸运的是，在Redis中执行原子操作不止有通过官方提供的命令的方式，还有另外一种方式，就是Lua脚本。因此，方案三中的释放锁的代码可以用以下Lua脚本来实现：



其中ARGV[1]表示设置key时指定的随机值。

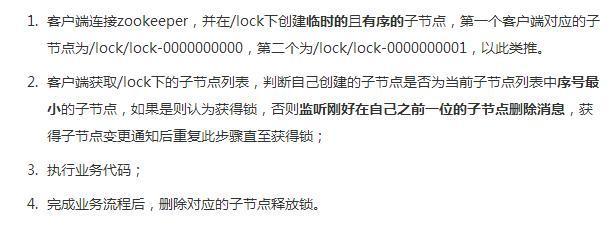
由于Lua脚本的原子性，在Redis执行该脚本的过程中，其他客户端的命令都需要等待该Lua脚本执行完才能执行，所以不会出现方案三所说的问题。至此，使用Redis实现分布式锁的方案就相对完善了。

**总结**

上述分布式锁的实现方案中，都是针对单节点Redis而言的，然而在生产环境中，我们使用的通常是Redis集群，并且每个主节点还会有从节点。由于Redis的主从复制是异步的，因此上述方案在Redis集群的环境下也是有问题的。关于在Redis集群中如何优雅地实现分布式锁，后续再写文章详述

# Zookeeper 锁具体实现：

整体思路：



## 使用第三方插件实现

package cn.qianzui.zk\_lock;  
  
import org.apache.curator.RetryPolicy;  
import org.apache.curator.framework.CuratorFramework;  
import org.apache.curator.framework.CuratorFrameworkFactory;  
import org.apache.curator.framework.recipes.locks.InterProcessMutex;  
import org.apache.curator.retry.ExponentialBackoffRetry;  
  
public class ZkLockTest {  
  
 public static void main(String[] args) {  
 // 创建重试策略 RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000,3);  
//创建zookeeper客户端

CuratorFramework client = CuratorFrameworkFactory.*newClient*("192.168.153.136:2181",retryPolicy);  
 client.start();  
  
 //创建分布式锁 ，锁空间接口为/curator/lock  
 InterProcessMutex lock = new InterProcessMutex(client,"/curator/lock");  
 try {  
 // 获取锁  
 lock.acquire();  
 System.*out*.println("执行任务");  
 // 释放锁  
 lock.release();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 client.close();  
 }  
}

报错：KeeperErrorCode = Unimplemented for /curator/lock/\_c\_5d4cc11e-5fdb-4e79-a294

使用Curator时报错，原因是因为版本问题  
zookeeper版本：zookeeper-3.4.8  
cuartor版本：3.2.0  
看Curator官网的声明：  
地址：[http://curator.apache.org/](https://link.jianshu.com?t=http:/curator.apache.org/)  
The are currently two released versions of Curator, 2.x.x and 3.x.x:

Curator 2.x.x - compatible with both ZooKeeper 3.4.x and ZooKeeper 3.5.x  
Curator 3.x.x - compatible only with ZooKeeper 3.5.x and includes support for new features such as dynamic reconfiguration, etc.

## 原始zookeeper 实现

package cn.qianzui.zk\_lock;  
  
  
import org.apache.zookeeper.\*;  
import org.apache.zookeeper.data.Stat;  
  
import java.io.IOException;  
import java.util.Collections;  
import java.util.List;  
import java.util.concurrent.CountDownLatch;  
  
public class MyZkLock implements Watcher {  
 // 声明链接  
 private ZooKeeper zk = null;  
  
 // 服务器地址  
 private static final String *ZKSERVER* = "192.168.153.136:2181";  
 // 超时时间  
 private static final int *SESSION\_TIMEOUT* = 10000;  
  
 // 锁节点目录  
 private static final String *HOME\_PATH*="/myLock";  
 // 锁节点子节点 的前缀  
 private static final String *LOCK\_PATH* ="/myLock/sub";  
  
 //创建字节记录自己的节点  
 private String selfPath;  
 //创建字节记录自己等待的节点  
 private String waitPath;  
  
 // 线程数  
 private static final int *THREAD\_NUM* = 10;  
 // 当前线程id  
 private int threadId;  
  
 // 保证所有节点链接成功  
 private CountDownLatch allConnection = new CountDownLatch(1);  
 // 保证所有线程完成任务  
 private static final CountDownLatch *allThread* = new CountDownLatch(*THREAD\_NUM*);  
 // 保证当前线程确实获取锁了，在往下走，否则就阻塞 ，不能执行那些获取锁之后才能做的事  
 private CountDownLatch insureGetLock = new CountDownLatch(1);  
 private String Thread\_Prefix;  
  
 public MyZkLock(int id) {  
 this.threadId = id;  
 this.Thread\_Prefix = "【第"+threadId+"个线程】";  
 }  
 public static void main(String[] args) throws IOException, KeeperException, InterruptedException {  
 for (int i= 0; i<*THREAD\_NUM*;i++){  
 // 当前线程id  
 final int threadId = i+1;  
 new Thread(new Runnable() {  
 public void run() {  
 try {  
 // 创建锁对象  
 MyZkLock myZkLock = new MyZkLock(threadId);  
 // 创建连接  
 myZkLock.createConnection();  
 // 创建根目录  
 myZkLock.createPath(*HOME\_PATH*,"this is home root",true);  
 // 获取锁  
 myZkLock.getLock();  
 System.*out*.println("【第"+threadId+"个线程】执行任务");  
 // 释放锁  
 myZkLock.unlock();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }).start();  
 }  
 *allThread*.await();  
 System.*out*.println("所有任务执行完毕。。。。。。。。");  
 }  
  
  
  
 public void createConnection() throws InterruptedException, KeeperException, IOException {  
 // 参数1 服务器地址 参数2 链接超时时间 参数3 监听器  
 zk = new ZooKeeper(*ZKSERVER*,*SESSION\_TIMEOUT*,this);  
 // 启动后等待，让所有线程都完成链接  
 allConnection.await();  
 }  
  
 public synchronized void createPath(String path,String desc,boolean useWatcher) throws KeeperException, InterruptedException {  
 // 判断该路径是否已经存在  
 // 如果不存在就创建 否则不操作  
 if (zk.exists(path,useWatcher) == null){  
 zk.create(path,desc.getBytes(),ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*PERSISTENT*);  
 }  
 }  
  
 public void getLock() throws KeeperException, InterruptedException {  
 // 创建自己的子锁节点 并保存起来  
 selfPath = zk.create(*LOCK\_PATH*,null, ZooDefs.Ids.*OPEN\_ACL\_UNSAFE*, CreateMode.*EPHEMERAL\_SEQUENTIAL*);  
 // 如果监测到自己是最小的，则获取锁成功  
 if(checkMinPath()){  
 getLockSuccess();  
 System.*out*.println(Thread\_Prefix+"获取锁成功");  
 }else {  
 // 没有获取锁，阻塞线程，不往下走 重要的一步  
 insureGetLock.await();  
 }  
 }  
 public boolean checkMinPath() throws KeeperException, InterruptedException {  
 // 获取主节点的子节点  
 List<String> list = zk.getChildren(*HOME\_PATH*, false);  
 Collections.*sort*(list);  
 // 获取自身节点  
 int index = list.indexOf(selfPath.substring(*HOME\_PATH*.length()+1));  
 switch (index){  
 case -1:{  
 //System.out.println("这个节点已经不存在了： "+selfPath);  
 return false;  
 }  
 case 0:{  
 // System.out.println("这个节点的索引是0，是最小的，可以获取锁： "+selfPath);  
 return true;  
 }  
 default:{  
 // 如果即存在又不是最小的，说明他前边有节点，获取他前边的那个节点-->为监听使用  
 waitPath = *HOME\_PATH*+"/"+list.get(index-1);  
 //System.out.println(Thread\_Prefix+" : "+waitPath+"排在前面，对他施加监听");  
 // 对这个节点进行监听 但是可能存在 前边的节点在添加监听前消失，所以对其进行异常捕捉  
 try {  
 zk.getData(waitPath,true,new Stat());  
 return false;  
 } catch (Exception e) {  
 // 判断是否真的不存在了  
 if(zk.exists(waitPath,false) == null){  
 System.*out*.println(*HOME\_PATH*+"子节点中，排在我前面的"+waitPath+"已失踪，幸福来得太突然?");  
 // 不存在后，再次进行对比分析  
 return checkMinPath();  
 }else{  
 throw e;  
 }  
 }  
 }  
 }  
 }  
  
 public void getLockSuccess() throws KeeperException, InterruptedException {  
 // 判断该节点是否还存在，防止网络波动启发的异常  
 if(zk.exists(selfPath,false) == null){  
 System.*out*.println("本节点已经不存在了");  
 return;  
 }  
 }  
 public void unlock() throws KeeperException, InterruptedException {  
 // 判断该节点是否还存在，防止网络波动启发的异常  
 if(zk.exists(selfPath,false) == null){  
 System.*out*.println(Thread\_Prefix+"本节点已经不存在了"+selfPath);  
 return;  
 }  
 // 线程睡一下，可以看的更清晰，不然嗖的一下都结束了  
 //Thread.sleep(3000);  
 // 自动删除这个节点  
 System.*out*.println( Thread\_Prefix+"释放连接");  
 zk.delete(this.selfPath, -1);  
 // 释放这个zk链接  
 releaseConnection();  
 // 该连接线程任务完成 数量减1  
 *allThread*.countDown();  
 }  
 public void releaseConnection(){  
 if ( this.zk !=null ) {  
 try {  
 this.zk.close();  
 } catch ( InterruptedException e ) {}  
 }  
 //System.out.println( Thread\_Prefix+"断开连接");  
 }  
  
 @Override  
 public void process(WatchedEvent event) {  
 if(event == null){  
 return;  
 }  
 // 获取事件拥有者的状态  
 Event.KeeperState keeperState = event.getState();  
 // 获取被监听的事件的类型  
 Event.EventType eventType = event.getType();  
 if ( Event.KeeperState.*SyncConnected* == keeperState) {  
 if ( Event.EventType.*None* == eventType ) {  
 System.*out*.println(Thread\_Prefix+"成功连接上ZK服务器" );  
 // 连接成功，倒计时减1  
 allConnection.countDown();  
 }else if (event.getType() == Event.EventType.*NodeDeleted* && event.getPath().equals(waitPath)) {  
 System.*out*.println( Thread\_Prefix+"收到情报，排我前面的家伙已挂，我是不是可以出山了？");  
 // 可以获取锁了，解开阻塞，继续往下走  
 insureGetLock.countDown();  
 // 重新获取锁  
 try {  
 if(checkMinPath()){  
 getLockSuccess();  
 }  
 } catch (KeeperException e) {  
 e.printStackTrace();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }else if ( Event.KeeperState.*Disconnected* == keeperState ) {  
 System.*out*.println( Thread\_Prefix+"与ZK服务器断开连接" );  
 } else if ( Event.KeeperState.*AuthFailed* == keeperState ) {  
 System.*out*.println( Thread\_Prefix+ "权限检查失败" );  
 } else if ( Event.KeeperState.*Expired* == keeperState ) {  
 System.*out*.println( Thread\_Prefix+ "会话失效" );  
 }  
 }  
}

· [zookeeper分布式锁](https://www.cnblogs.com/qfjavabd/p/10132885.html)  
· [自旋锁、文件锁、大内核锁](https://www.cnblogs.com/shihaochangeworld/p/5795569.html)  
· [ZooKeeper实现分布式锁](https://www.cnblogs.com/xiohao/p/9063755.html" \t "_blank)  
· [Zookeeper+Curator 分布式锁](https://www.cnblogs.com/tdws/p/5874686.html" \t "_blank)  
· [zookeeper实现分布式锁](https://www.cnblogs.com/Yulong123/p/10027505.html)

· [分布式锁](https://www.cnblogs.com/mcahkf/p/9110567.html)  
· [分布式锁](https://www.cnblogs.com/aoyihuashao/p/8762401.html)  
· [锁以及分布式锁](https://www.cnblogs.com/AlmostWasteTime/p/10134993.html)  
· [Redis分布式锁](https://www.cnblogs.com/tdws/p/5808528.html)  
· [分布式锁](https://www.cnblogs.com/chinano1/p/10016983.html)

· [进程锁](https://www.cnblogs.com/kuraki/p/9602862.html)  
· [this锁与static锁](https://www.cnblogs.com/yuluoxingkong/p/9802242.html)  
· [独占锁、共享锁、更新锁，乐观锁、悲观锁](https://www.cnblogs.com/cwfsoft/p/7759944.html)  
· [java多线程之锁--偏向锁、轻量级锁、自旋锁、重量级锁](https://www.cnblogs.com/maxigang/p/9039702.html)  
· [自旋锁、排队自旋锁、MCS锁、CLH锁。锁的进阶](https://www.cnblogs.com/bigboytiny/p/6672134.html)

· [分布式锁之一：zookeeper分布式锁1](https://www.cnblogs.com/duanxz/p/3509423.html)  
· [zookeeper分布式锁](https://www.cnblogs.com/qfjavabd/p/10132885.html)  
· [zookeeper实现分布式锁服务](https://www.cnblogs.com/Evil-Rebe/p/5098104.html)

1. [**Java锁的种类以及辨析（三）：阻塞锁**](http://ifeve.com/java_lock_see3/)
2. [**Java锁的种类以及辨析**](http://ifeve.com/java_lock_see/)
3. [**Java锁的种类以及辨析（四）：可重入锁**](http://ifeve.com/java_lock_see4/)
4. [**Java锁的种类以及辨析（二）：自旋锁的其他种类**](http://ifeve.com/java_lock_see2/)
5. [**并发网每月TOP10文章**](http://ifeve.com/ifeve-top10/)
6. [**LockSupport 源码阅读**](http://ifeve.com/locksuppor-source/)
7. [**更快的AtomicInteger**](http://ifeve.com/better_atomicinteger/)
8. [**ReentrantLock(重入锁)以及公平性**](http://ifeve.com/reentrantlock-and-fairness/)
9. [**Java8中CAS的增强**](http://ifeve.com/enhanced-cas-in-jdk8/)
10. [**《 Java并发编程从入门到精通》Thread安全与不安全**](http://ifeve.com/java-concurrency-thread-safe/)
11. [**Bug:StampedLock的中断问题导致CPU爆满**](http://ifeve.com/stampedlock-bug-cpu/)
12. [**ThreadLocal使用**](http://ifeve.com/threadlocal%e4%bd%bf%e7%94%a8/)
13. [**《Java并发编程从入门到精通》显示锁Lock和ReentrantLock**](http://ifeve.com/lock-reentrantlock/)
14. [**使用数据库悲观锁实现不可重入的分布式锁**](http://ifeve.com/dblock/)
15. [**并发集合（七）创建并发随机数**](http://ifeve.com/concurrent-collections-7/)