

分布式锁

知识储备: jmeter/ab压力测试工具 nginx docker redis zookeeper 微服务编程基础等

在多线程高并发场景下，为了保证资源的线程安全问题，jdk为我们提供了synchronized关键字和ReentrantLock可重入锁，但是它们只能保证一个jvm内的线程安全。在分布式集群、微服务、云原生横行的当下，如何保证不同进程、不同服务、不同机器的线程安全问题，jdk并没有给我们提供既有的解决方案。此时，我们就必须借助于相关技术手动实现了。目前主流的实现有三种方式：

1. 基于mysql关系型实现
2. 基于redis非关系型数据实现
3. 基于zookeeper实现

1. 从减库存聊起

库存在并发量较大情况下很容易发生超卖现象，一旦发生超卖现象，就会出现多成交了订单而发不了货的情况。

场景：

商品S库存余量为5时，用户A和B同时来购买一个商品S，此时查询库存数都为5，库存充足则开始减库存：

用户A: update db_stock set stock = stock - 1 where id = 1

用户B: update db_stock set stock = stock - 1 where id = 1

并发情况下，更新后的结果可能是4，而实际的最终库存量应该是3才对

1.1. 环境准备

建表语句：

```
1 CREATE TABLE `db_stock` (  
2   `id` bigint(20) NOT NULL AUTO_INCREMENT,  
3   `product_code` varchar(255) DEFAULT NULL COMMENT '商品编号',  
4   `stock_code` varchar(255) DEFAULT NULL COMMENT '仓库编号',  
5   `count` int(11) DEFAULT NULL COMMENT '库存量',  
6   PRIMARY KEY (`id`)  
7 ) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

表中数据如下：

对象 db_stock @test (centos) - 表			
开始事务	备注	筛选	排序
导入	导出		
id	product_code	stock_code	count
1	1001	001	5000

1001商品在001仓库有5000件库存。

创建分布式锁demo工程：

New Project

Project Metadata

Group:

com.atguigu

Artifact:

distributed-lock

Type:

Maven Project (Generate a Maven based project archive.)

Language:

Java

Packaging:

Jar

Java Version:

8

Version:

0.0.1-SNAPSHOT

Name:

distributed-lock

Description:

分布式锁demo工程

Package:

com.atguigu.distributedlock

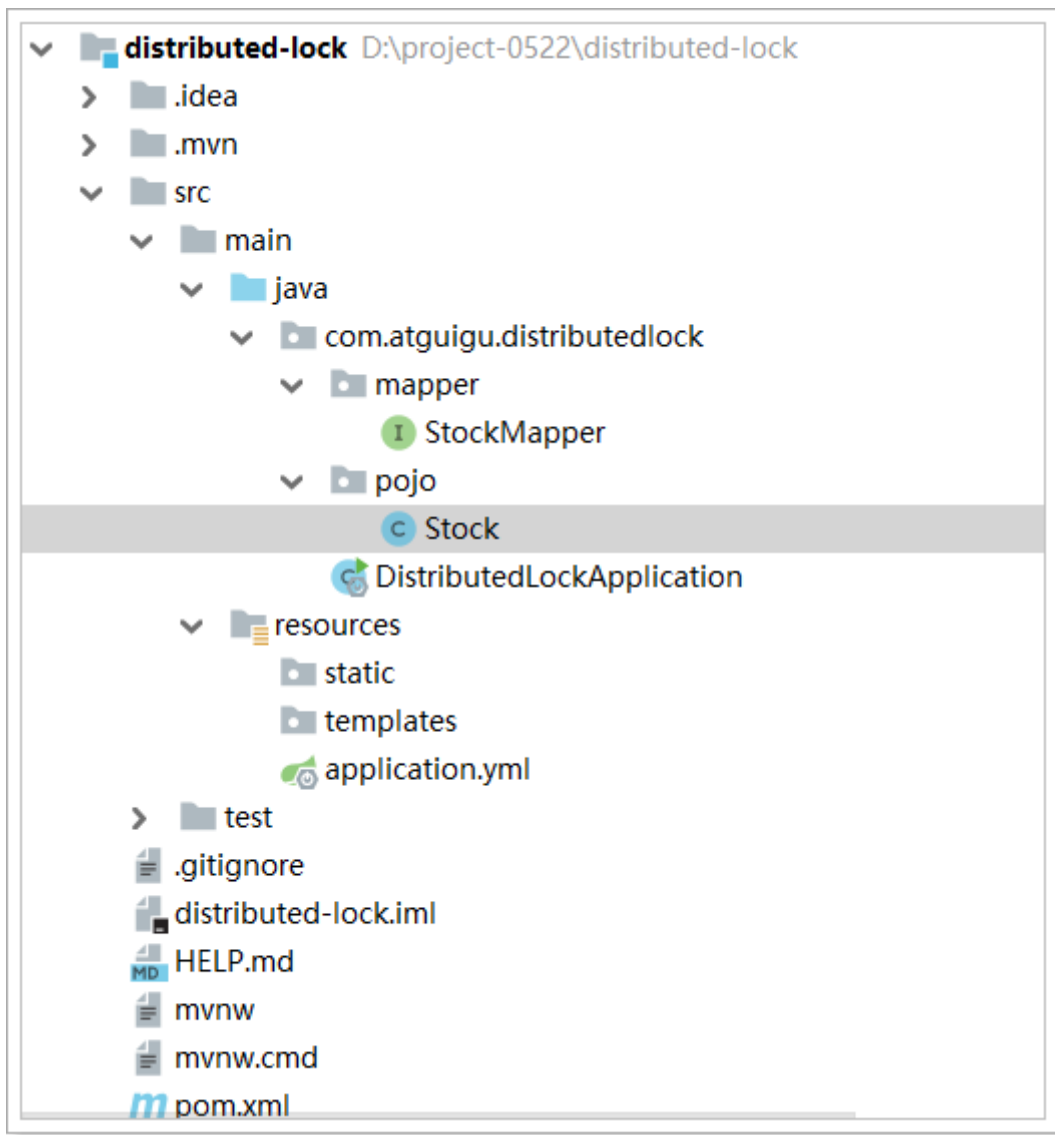
Previous

Next

Cancel

Help

创建好之后：



pom.xml如下:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>org.springframework.boot</groupId>
7         <artifactId>spring-boot-starter-parent</artifactId>
8         <version>2.2.11.RELEASE</version>
9         <relativePath/> <!-- lookup parent from repository -->
10    </parent>
11    <groupId>com.atguigu</groupId>
12    <artifactId>distributed-lock</artifactId>
13    <version>0.0.1-SNAPSHOT</version>
14    <name>distributed-lock</name>
15    <description>分布式锁demo工程</description>
16
17    <properties>
18        <java.version>1.8</java.version>
19    </properties>
20
21    <dependencies>
```

```

22     <dependency>
23         <groupId>org.springframework.boot</groupId>
24         <artifactId>spring-boot-starter-web</artifactId>
25     </dependency>
26
27     <dependency>
28         <groupId>mysql</groupId>
29         <artifactId>mysql-connector-java</artifactId>
30         <version>5.1.46</version>
31     </dependency>
32
33     <dependency>
34         <groupId>com.baomidou</groupId>
35         <artifactId>mybatis-plus-boot-starter</artifactId>
36         <version>3.4.0</version>
37     </dependency>
38
39     <dependency>
40         <groupId>org.projectlombok</groupId>
41         <artifactId>lombok</artifactId>
42         <version>1.18.16</version>
43     </dependency>
44     <dependency>
45         <groupId>org.springframework.boot</groupId>
46         <artifactId>spring-boot-starter-data-redis</artifactId>
47     </dependency>
48
49     <dependency>
50         <groupId>org.springframework.boot</groupId>
51         <artifactId>spring-boot-devtools</artifactId>
52     </dependency>
53     <dependency>
54         <groupId>org.springframework.boot</groupId>
55         <artifactId>spring-boot-starter-test</artifactId>
56         <scope>test</scope>
57         <exclusions>
58             <exclusion>
59                 <groupId>org.junit.vintage</groupId>
60                 <artifactId>junit-vintage-engine</artifactId>
61             </exclusion>
62         </exclusions>
63     </dependency>
64 </dependencies>
65
66 <build>
67     <plugins>
68         <plugin>
69             <groupId>org.springframework.boot</groupId>
70             <artifactId>spring-boot-maven-plugin</artifactId>
71         </plugin>
72     </plugins>
73 </build>
74
75 </project>

```

application.yml配置文件:

```

1  server:
2    port: 6000
3  spring:
4    datasource:
5      driver-class-name: com.mysql.jdbc.Driver
6      url: jdbc:mysql://172.16.116.100:3306/test
7      username: root
8      password: root
9    redis:
10     host: 172.16.116.100
11

```

DistributedLockApplication启动类:

```

1  @SpringBootApplication
2  @MapperScan("com.atguigu.distributedlock.mapper")
3  public class DistributedLockApplication {
4
5      public static void main(String[] args) {
6          SpringApplication.run(DistributedLockApplication.class, args);
7      }
8
9  }

```

Stock实体类:

```

1  @Data
2  @TableName("db_stock")
3  public class Stock {
4
5      @TableId
6      private Long id;
7
8      private String productCode;
9
10     private String stockCode;
11
12     private Integer count;
13 }

```

StockMapper接口:

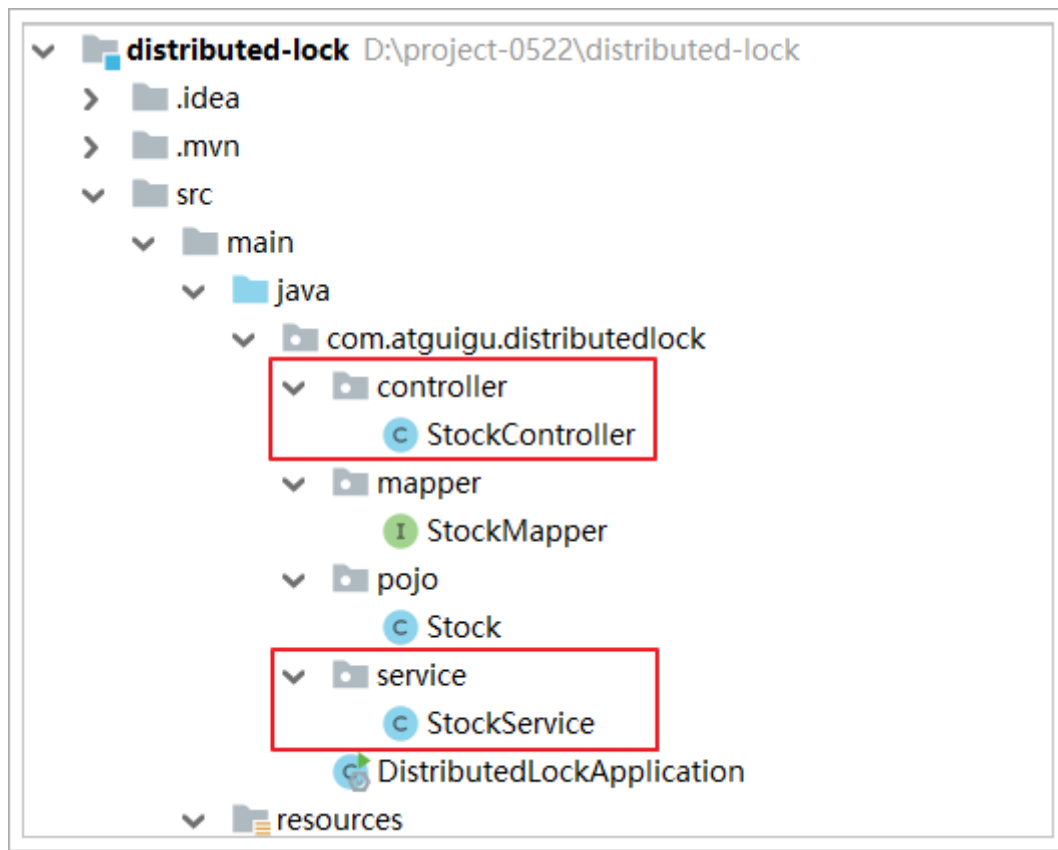
```

1  public interface StockMapper extends BaseMapper<Stock> {
2  }

```

1.2. 简单实现减库存

接下来咱们代码实操一下。



StockController:

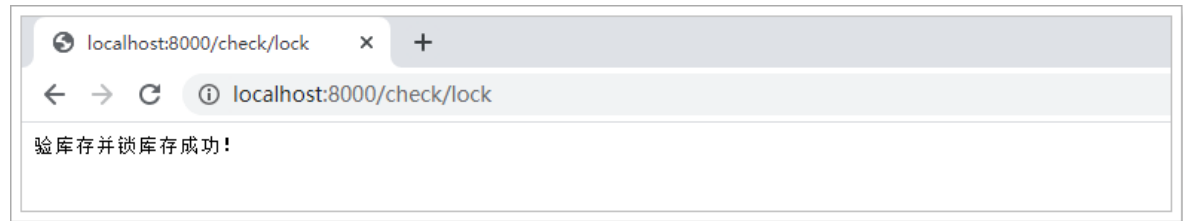
```
1  @RestController
2  public class StockController {
3
4      @Autowired
5      private StockService stockService;
6
7      @GetMapping("check/lock")
8      public String checkAndLock(){
9
10         this.stockService.checkAndLock();
11
12         return "验库存并锁库存成功! ";
13     }
14 }
```

StockService:

```
1  @Service
2  public class StockService {
3
4      @Autowired
5      private StockMapper stockMapper;
6
7      public void checkAndLock() {
8
9          // 先查询库存是否充足
10         Stock stock = this.stockMapper.selectById(1L);
11
12         // 再减库存
13         if (stock != null && stock.getCount() > 0){
```

```
14         stock.setCount(stock.getCount() - 1);
15         this.stockMapper.updateById(stock);
16     }
17 }
18 }
```

测试：



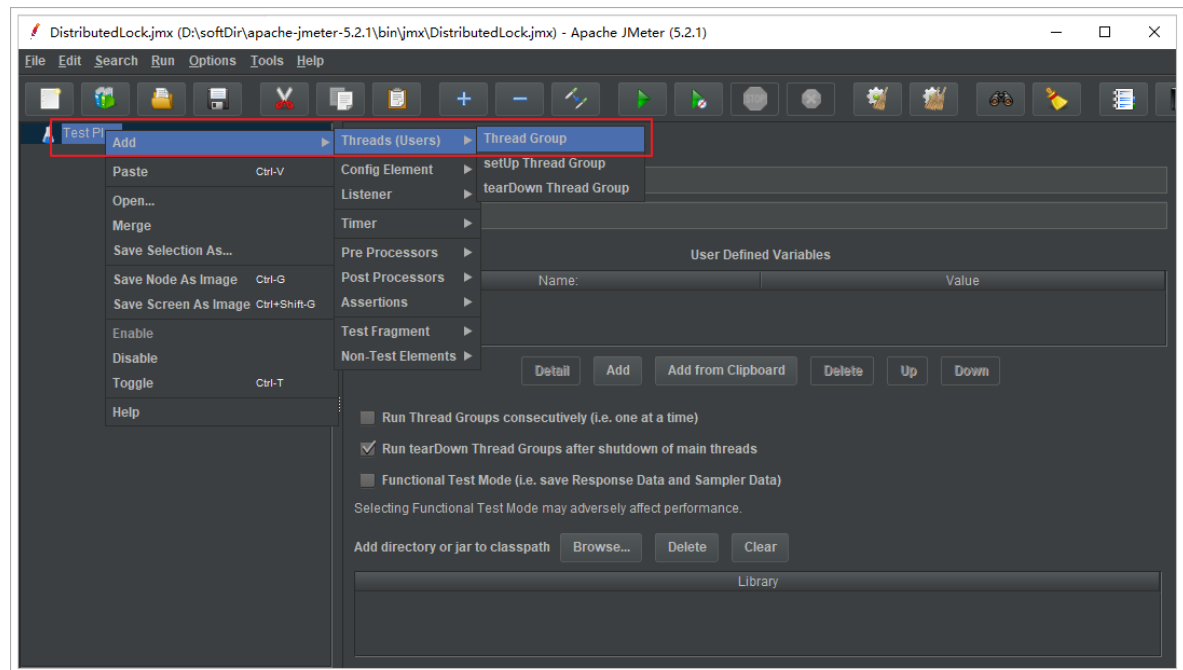
查看数据库：

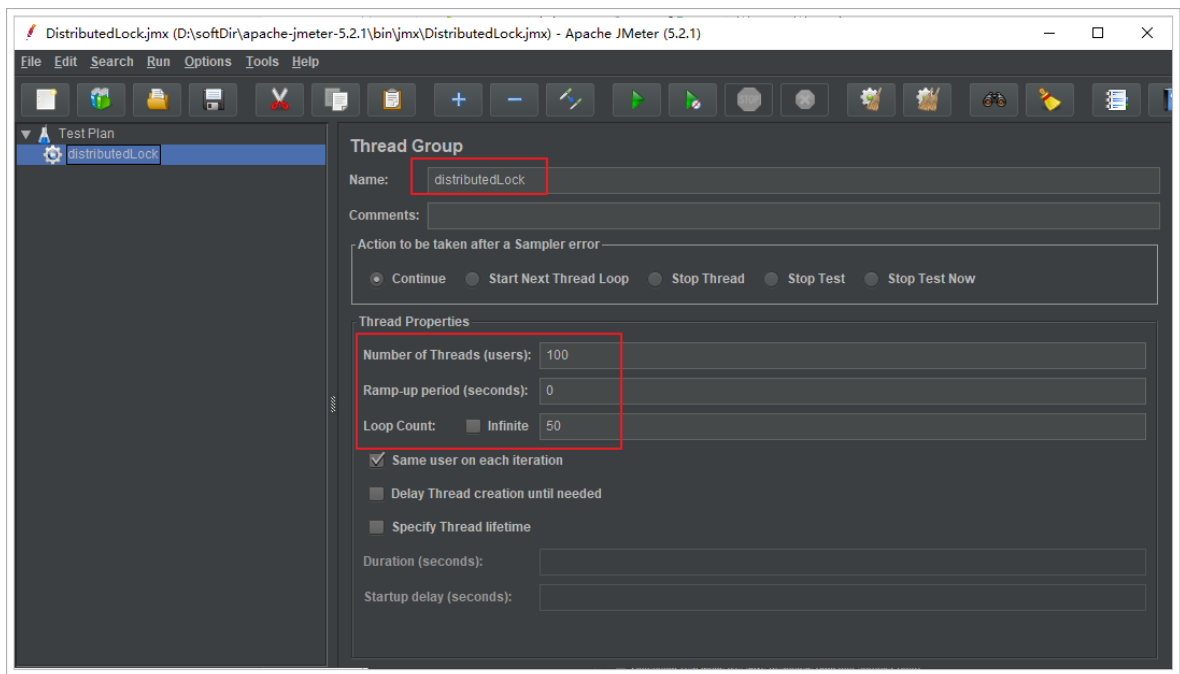
对象 db_stock @test (centos) - 表			
开始事务 备注 筛选 排序 导入 导出			
id	product_code	stock_code	count
1	1001	001	4999

在浏览器中一个一个访问时，每访问一次，库存量减1，没有任何问题。

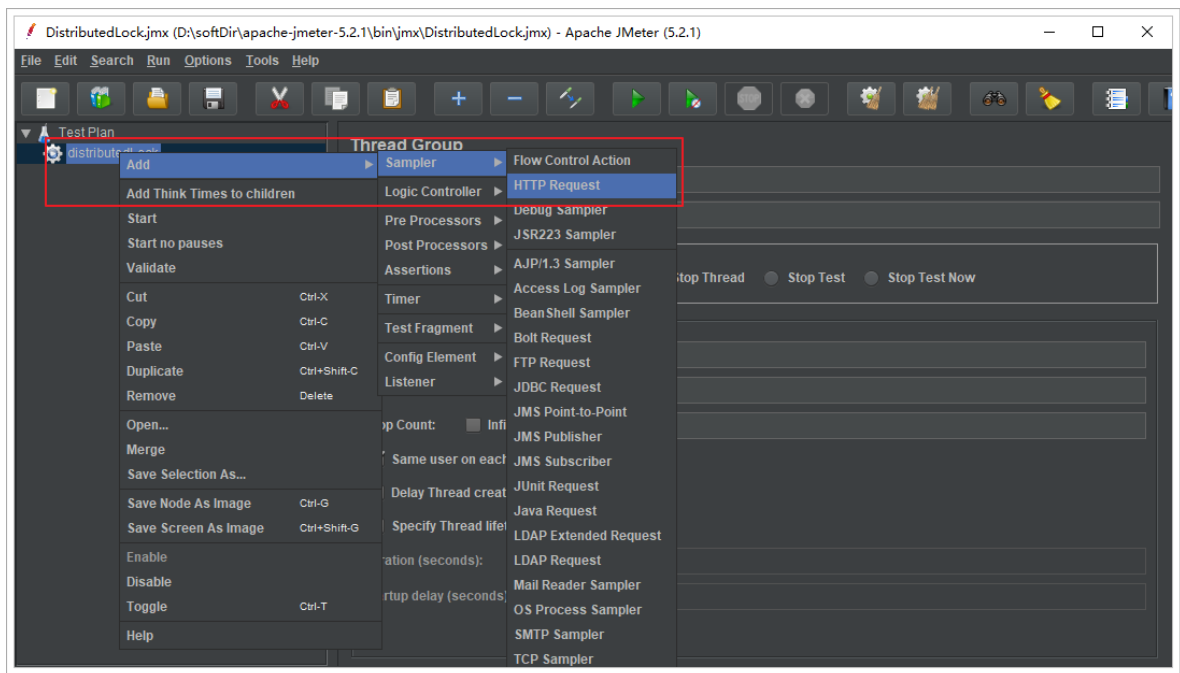
1.3. 演示超卖现象

接下来咱们使用jmeter压力测试工具，高并发下压测一下，添加线程组：并发100循环50次，即5000次请求。

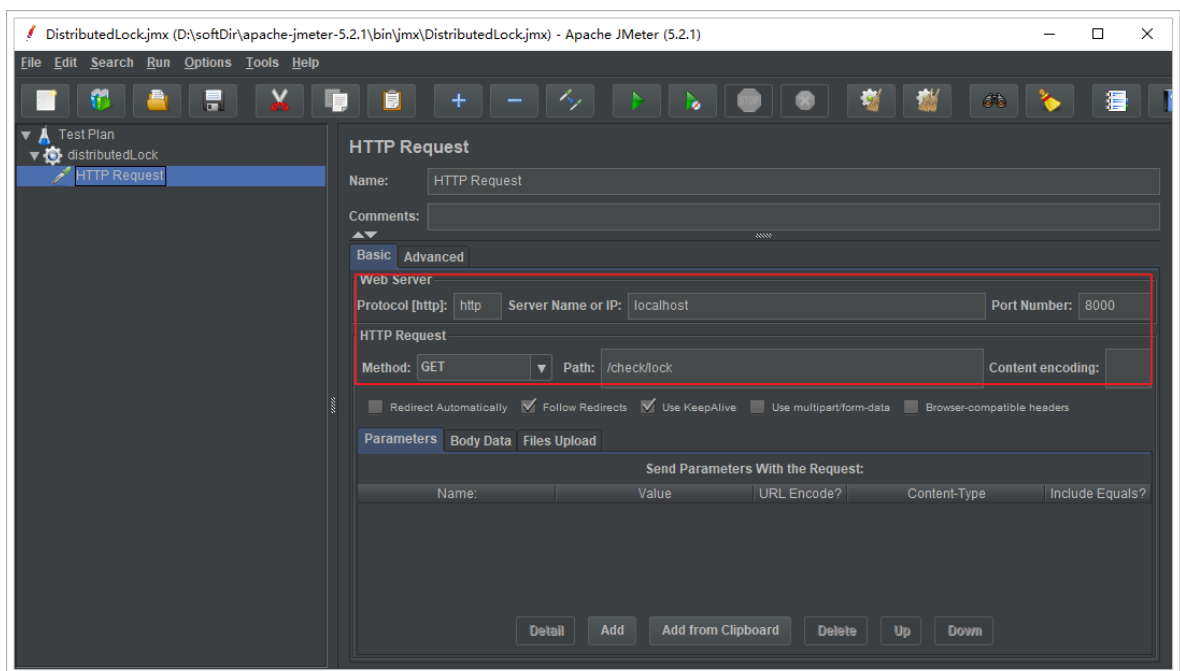




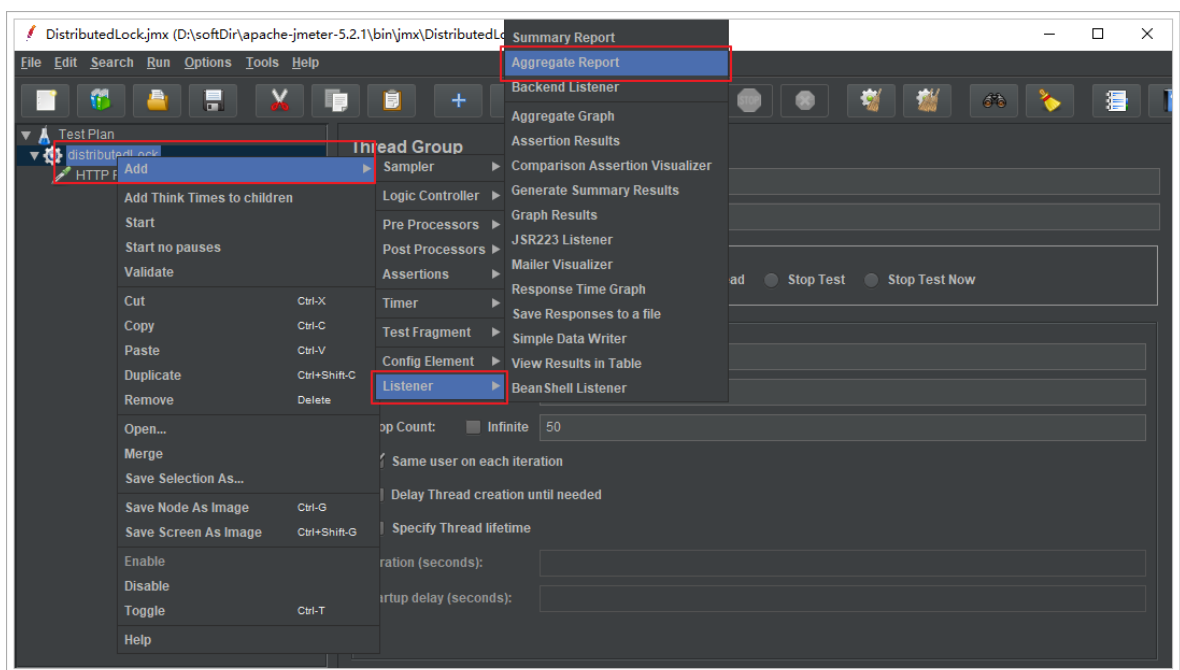
给线程组添加HTTP Request请求：



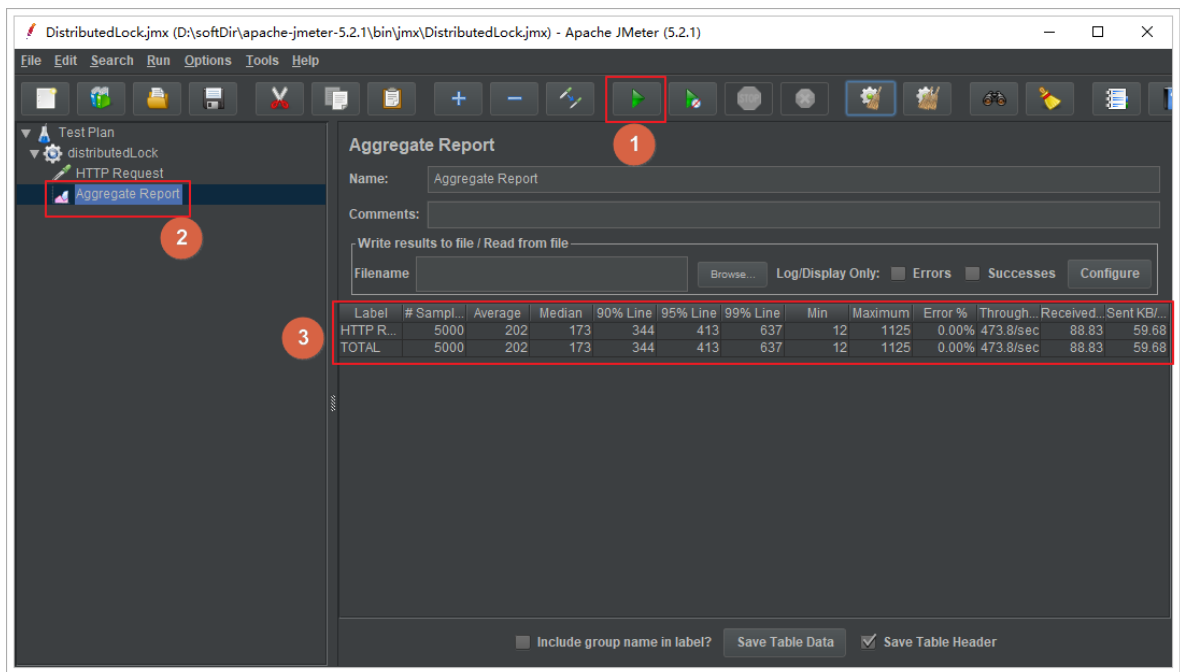
填写测试接口路径如下：



再选择你想要的测试报表，例如这里选择聚合报告：



启动测试，查看压力测试报告：



测试结果：请求总数5000次，平均请求时间202ms，中位数（50%）请求是在173ms内完成的，90%请求是在344ms内完成的，最小耗时12ms，最大耗时1125ms，错误率0%，每秒钟平均473.8次。

查看mysql数据库剩余库存数：还有4870

对象 db_stock @test (centos) - 表			
开始事务 备注 筛选 排序 导入 导出			
id	product_code	stock_code	count
1	1001	001	4870

此时如果还有人来下单，就会出现超卖现象（别人购买成功，而无货可发）。

1.4. jvm锁问题演示

1.4.1. 添加jvm锁

使用jvm锁（synchronized关键字或者ReentrantLock）试试：

```

9  @Service
10 public class StockService {
11
12     @Autowired
13     private StockMapper stockMapper;
14
15     public synchronized void checkAndLock() {
16
17         // 先查询库存是否充足
18         Stock stock = this.stockMapper.selectOne(new QueryWrapper<Sto
19
20         // 再减库存
21         if (stock != null && stock.getCount() > 0) {
22             stock.setCount(stock.getCount() - 1);
23             this.stockMapper.updateById(stock);
24         }
25     }
26 }

```

重启tomcat服务，再次使用jmeter压力测试，效果如下：

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	1179	1144	2128	2359	2925	18	3258	0.00%	82.2/sec	15.41	10.36
TOTAL	5000	1179	1144	2128	2359	2925	18	3258	0.00%	82.2/sec	15.41	10.36

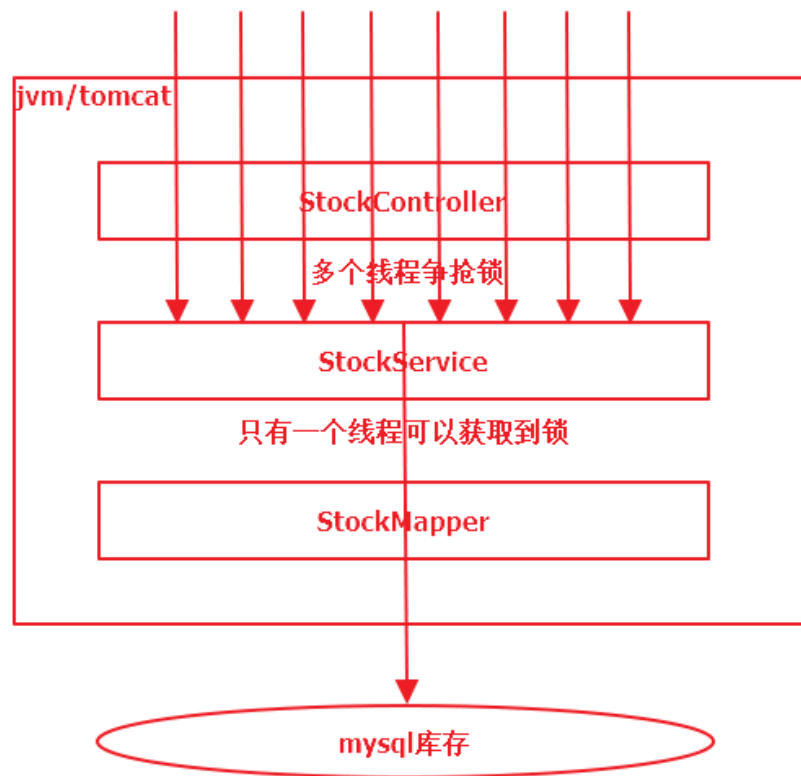
查看mysql数据库：

对象 db_stock @test (centos) - 表			
id	product_code	stock_code	count
1	1001	001	0

并没有发生超卖现象，完美解决。

1.4.2. 原理

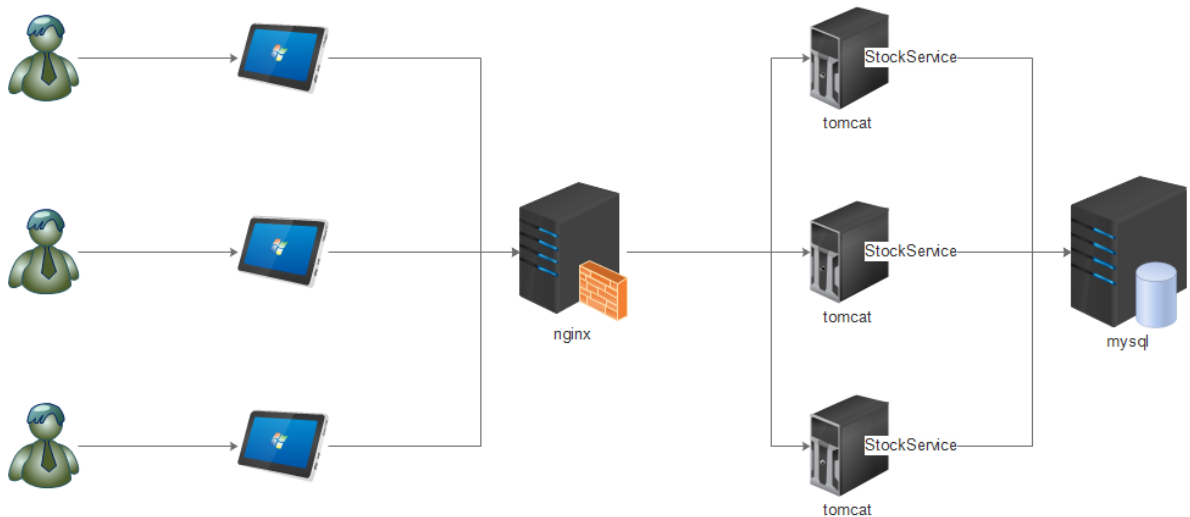
添加synchronized关键字之后，StockService就具备了对象锁，由于添加了独占的排他锁，同一时刻只有一个请求能够获取到锁，并减库存。此时，所有请求只会one-by-one执行下去，也就不会发生超卖现象。



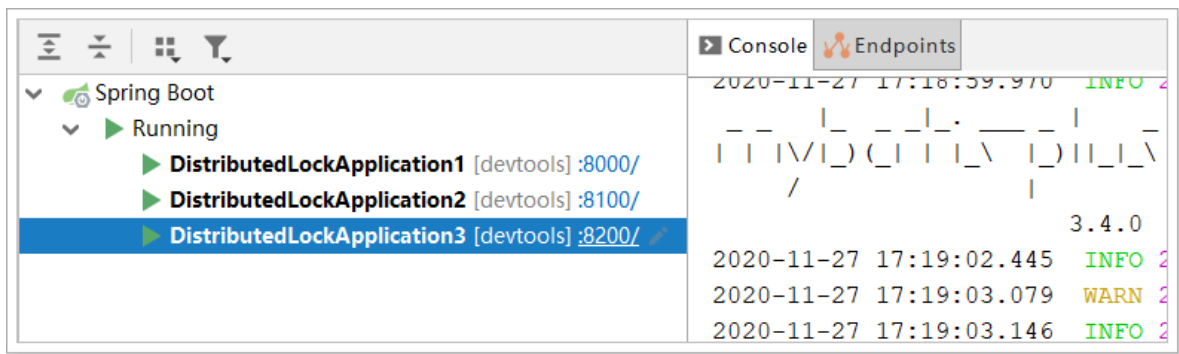
1.5. 多服务问题

使用jvm锁在单工程单服务情况下确实没有问题，但是在集群情况下会怎样？

接下启动多个服务并使用nginx负载均衡，结构如下：



启动三个服务（端口号分别8000 8100 8200），如下：



1.5.1. 安装配置nginx

基于安装nginx:

```
1 # 拉取镜像
2 docker pull nginx:latest
3 # 创建nginx对应资源、日志及配置目录
4 mkdir -p /opt/nginx/logs /opt/nginx/conf /opt/nginx/html
5 # 先在conf目录下创建nginx.conf文件，配置内容参照下方
6 # 再运行容器
7 docker run -d -p 80:80 --name nginx -v /opt/nginx/html:/usr/share/nginx/html
  -v /opt/nginx/conf/nginx.conf:/etc/nginx/nginx.conf -v
  /opt/nginx/logs:/var/log/nginx nginx
```

nginx.conf配置如下:

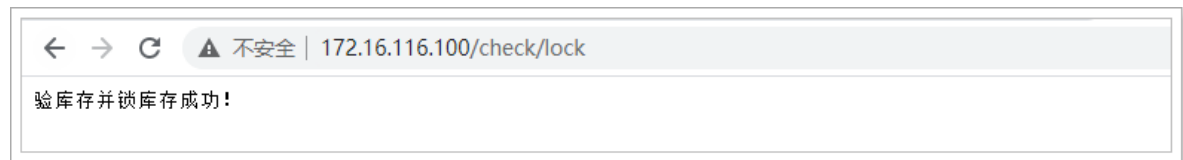
```
1 user nginx;
2 worker_processes 1;
3
4 error_log /var/log/nginx/error.log warn;
5 pid /var/run/nginx.pid;
6
7 events {
8     worker_connections 1024;
9 }
10
11 http {
12     include /etc/nginx/mime.types;
13     default_type application/octet-stream;
14
15     log_format main '$remote_addr - $remote_user [$time_local] "$request"
16     ,
17     '$status $body_bytes_sent "$http_referer" '
18     '"$http_user_agent" "$http_x_forwarded_for"';
19
20     access_log /var/log/nginx/access.log main;
21
22     sendfile on;
23     #tcp_nopush on;
24
25     keepalive_timeout 65;
26
27     #gzip on;
```

```

28     #include /etc/nginx/conf.d/*.conf;
29
30     upstream distributed {
31         server 172.16.116.10:8000;
32         server 172.16.116.10:8100;
33         server 172.16.116.10:8200;
34     }
35
36     server {
37         listen      80;
38         server_name 172.16.116.100;
39         location / {
40             proxy_pass http://distributed;
41         }
42     }
43
44 }

```

在浏览器中测试：172.16.116.100是我的nginx服务器地址

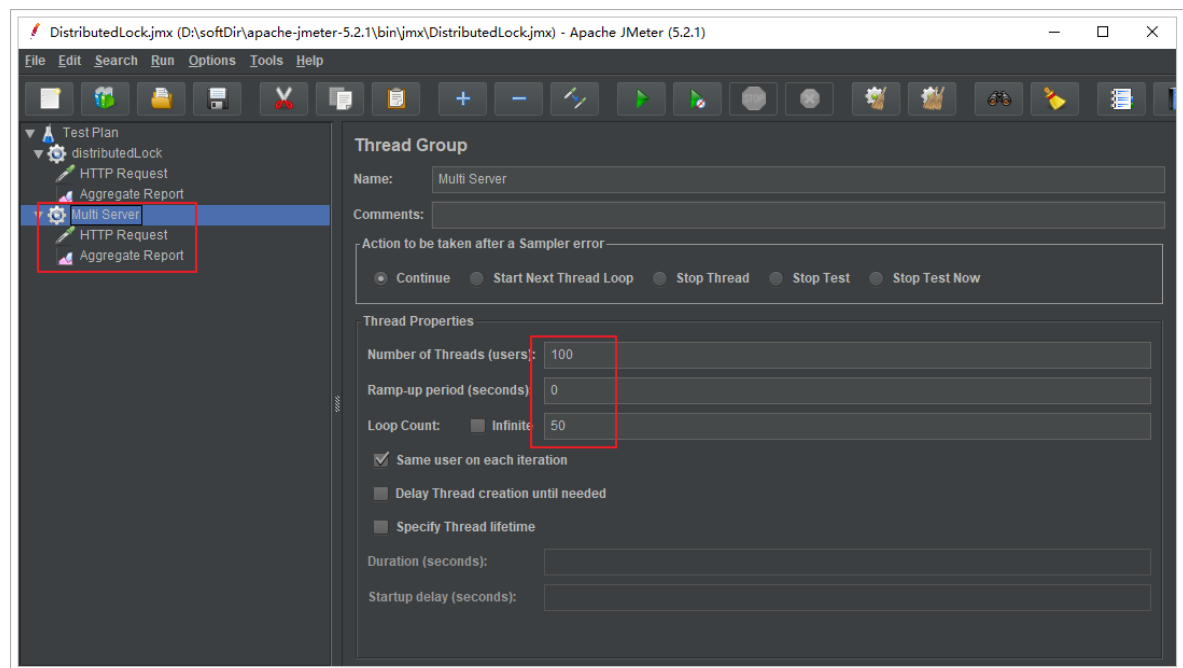


经过测试，通过nginx访问服务一切正常。

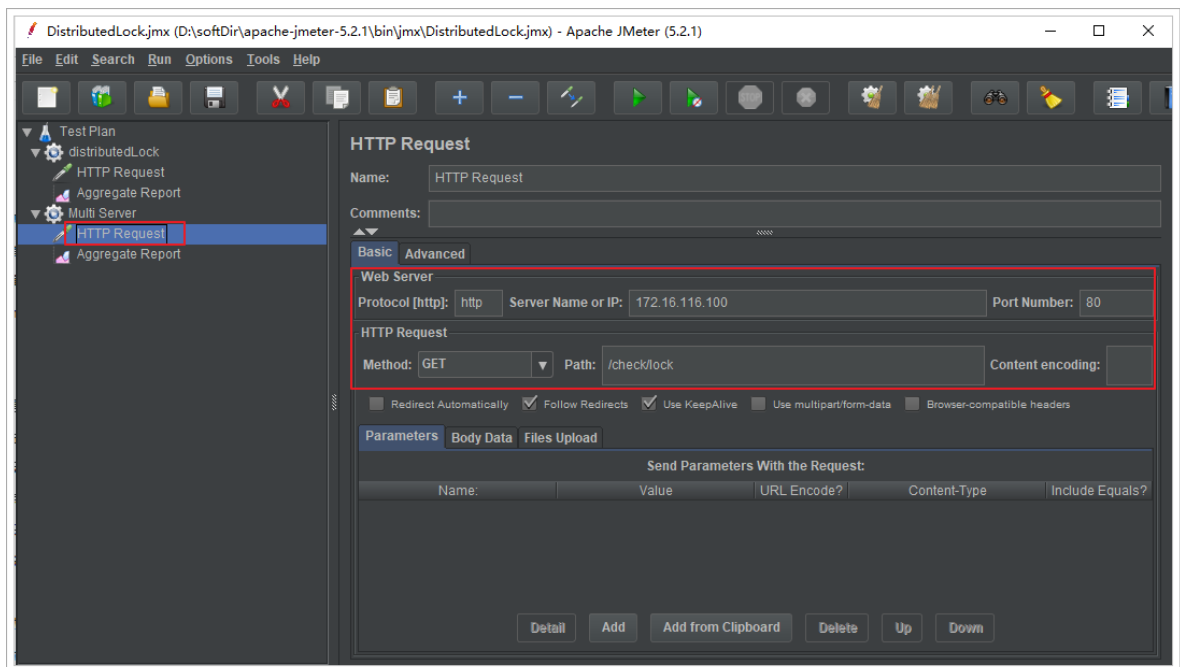
1.5.2. 压力测试

注意：先把数据库库存量还原到5000。

参照之前的测试用例，再创建一个新的测试组：参数给之前一样



配置nginx的地址及 服务的访问路径如下：



测试结果：性能只是略有提升。

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	1117	1082	2028	2175	2579	22	3151	0.00%	85.5/sec	15.86	10.77
TOTAL	5000	1117	1082	2028	2175	2579	22	3151	0.00%	85.5/sec	15.86	10.77

数据库库存剩余量如下：

id	product_code	stock_code	count
1	1001	001	1425

又出现了并发问题，即出现了超卖现象。

1.6. mysql锁演示

除了使用jvm锁之外，还可以使用数据锁：**悲观锁** 或者 **乐观锁**

悲观锁：在读取数据时锁住那几行，其他对这几行的更新需要等到悲观锁结束时才能继续。

乐观锁：读取数据时不锁，更新时检查是否数据已经被更新过，如果是则取消当前更新，一般在悲观锁的等待时间过长而不能接受时我们才会选择乐观锁。

1.6.1. mysql悲观锁

在MySQL的InnoDB中，预设的Tansaction isolation level 为REPEATABLE READ（可重读）

在SELECT 的读取锁定主要分为两种方式：

- SELECT ... LOCK IN SHARE MODE （共享锁）
- SELECT ... FOR UPDATE （悲观锁）

这两种方式在事务(Transaction) 进行当中SELECT 到同一个数据表时，都必须等待其它事务数据被提交(Commit)后才会执行。

而主要的不同在于LOCK IN SHARE MODE 在有一方事务要Update 同一个表单时很容易造成死锁。

简单的说，如果SELECT 后面若要UPDATE 同一个表单，最好使用SELECT ... FOR UPDATE。

代码实现

改造StockService:

```
11 public class StockService {
12
13     @Autowired
14     private StockMapper stockMapper;
15
16     @Transactional 添加事务注解，去掉synchronized关键字
17     public void checkAndLock() {
18
19         // 先查询库存是否充足
20         Stock stock = this.stockMapper.selectStockForUpdate(id: 1L);
21
22         // 再减库存
23         if (stock != null && stock.getCount() > 0) {
24             stock.setCount(stock.getCount() - 1);
25             this.stockMapper.updateById(stock);
26         }
27     }
28 }
```

在StockMapper中定义selectStockForUpdate方法:

```
1 public interface StockMapper extends BaseMapper<Stock> {
2
3     public Stock selectStockForUpdate(Long id);
4 }
```

在StockMapper.xml中定义对应的配置:

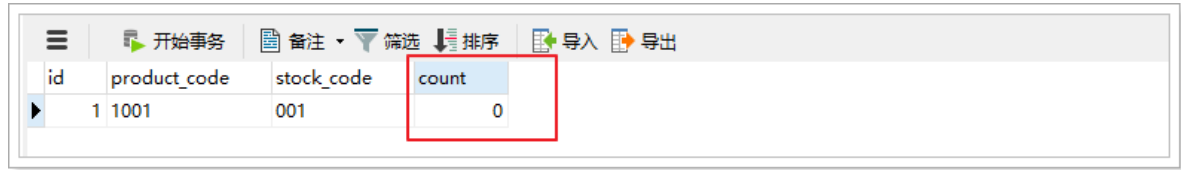
```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4 <mapper namespace="com.atguigu.distributedlock.mapper.StockMapper">
5
6     <select id="selectStockForUpdate"
7         resultType="com.atguigu.distributedlock.pojo.Stock">
8         select * from db_stock where id = #{id} for update
9     </select>
10 </mapper>
```

压力测试

注意：测试之前，需要把库存量改成5000。压测数据如下：比jvm性能高很多，比无锁要低将近1倍

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	425	411	494	547	1196	31	1538	0.00%	231.5/sec	42.95	29.16
TOTAL	5000	425	411	494	547	1196	31	1538	0.00%	231.5/sec	42.95	29.16

mysql数据库存:



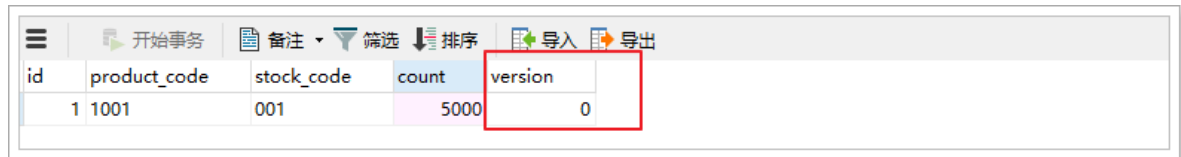
id	product_code	stock_code	count	version
1	1001	001	0	

1.6.2. mysql乐观锁

乐观锁（Optimistic Locking）相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则重试。那么我们如何实现乐观锁呢

使用数据版本（Version）记录机制实现，这是乐观锁最常用的实现方式。一般是通过为数据库表增加一个数字类型的“version”字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新。

给db_stock表添加version字段：



id	product_code	stock_code	count	version
1	1001	001	5000	0

对应也需要给Stock实体类添加version属性。此处略。。。

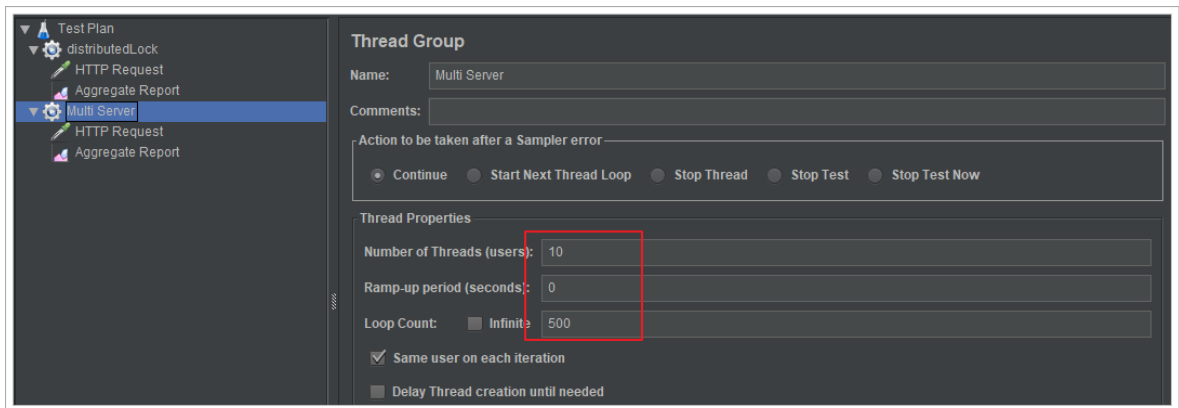
代码实现

```
1 public void checkAndLock() {
2
3     // 先查询库存是否充足
4     Stock stock = this.stockMapper.selectById(1L);
5
6     // 再减库存
7     if (stock != null && stock.getCount() > 0){
8         // 获取版本号
9         Long version = stock.getVersion();
10
11         stock.setCount(stock.getCount() - 1);
12         // 每次更新 版本号 + 1
13         stock.setVersion(stock.getVersion() + 1);
14         // 更新之前先判断是否是之前查询的那个版本，如果不是重试
15         if (this.stockMapper.update(stock, new UpdateWrapper<Stock>
16             ().eq("id", stock.getId()).eq("version", version)) == 0) {
17             checkAndLock();
18         }
19     }
20 }
```

重启后使用jmeter压力测试工具结果如下：

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	1268	42	4283	6646	13223	18	29416	0.00%	70.0/sec	12.98	8.81
TOTAL	5000	1268	42	4283	6646	13223	18	29416	0.00%	70.0/sec	12.98	8.81

修改测试参数如下：



测试结果如下：

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	51	47	76	91	125	19	213	0.00%	190.0/sec	35.25	23.93
TOTAL	5000	51	47	76	91	125	19	213	0.00%	190.0/sec	35.25	23.93

说明乐观锁在并发量越大的情况下，性能越低（因为需要大量的重试）；并发量越小，性能越高。

1.6.3. mysql锁缺陷

在数据库集群情况下会导致数据库锁失效，并且很多数据库集群的中间件压根就不支持悲观锁。例如：mycat

在读写分离的场景下可能会导致乐观锁不可靠。

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. 基于mysql实现分布式锁

不管是jvm锁还是mysql锁，为了保证线程的并发安全，都提供了悲观独占排他锁。所以**独占排他**也是分布式锁的基本要求。

可以利用唯一键索引不能重复插入的特点实现。设计表如下：

```

1 CREATE TABLE `db_lock` (
2   `id` bigint(20) NOT NULL AUTO_INCREMENT,
3   `lock_name` varchar(50) NOT NULL COMMENT '锁名',
4   `class_name` varchar(100) DEFAULT NULL COMMENT '类名',
5   `method_name` varchar(50) DEFAULT NULL COMMENT '方法名',
6   `server_name` varchar(50) DEFAULT NULL COMMENT '服务器ip',
7   `thread_name` varchar(50) DEFAULT NULL COMMENT '线程名',
8   `create_time` timestamp NULL DEFAULT NULL ON UPDATE CURRENT_TIMESTAMP
  COMMENT '获取锁时间',
9   `desc` varchar(100) DEFAULT NULL COMMENT '描述',
10  PRIMARY KEY (`id`),
11  UNIQUE KEY `idx_unique` (`lock_name`)
12 ) ENGINE=InnoDB AUTO_INCREMENT=1332899824461455363 DEFAULT CHARSET=utf8;

```

Lock实体类:

```

1 @Data
2 @AllArgsConstructor
3 @NoArgsConstructor
4 @TableName("db_lock")
5 public class Lock {
6
7     private Long id;
8     private String lockName;
9     private String className;
10    private String methodName;
11    private String serverName;
12    private String threadName;
13    private Date createTime;
14    private String desc;
15 }

```

LockMapper接口:

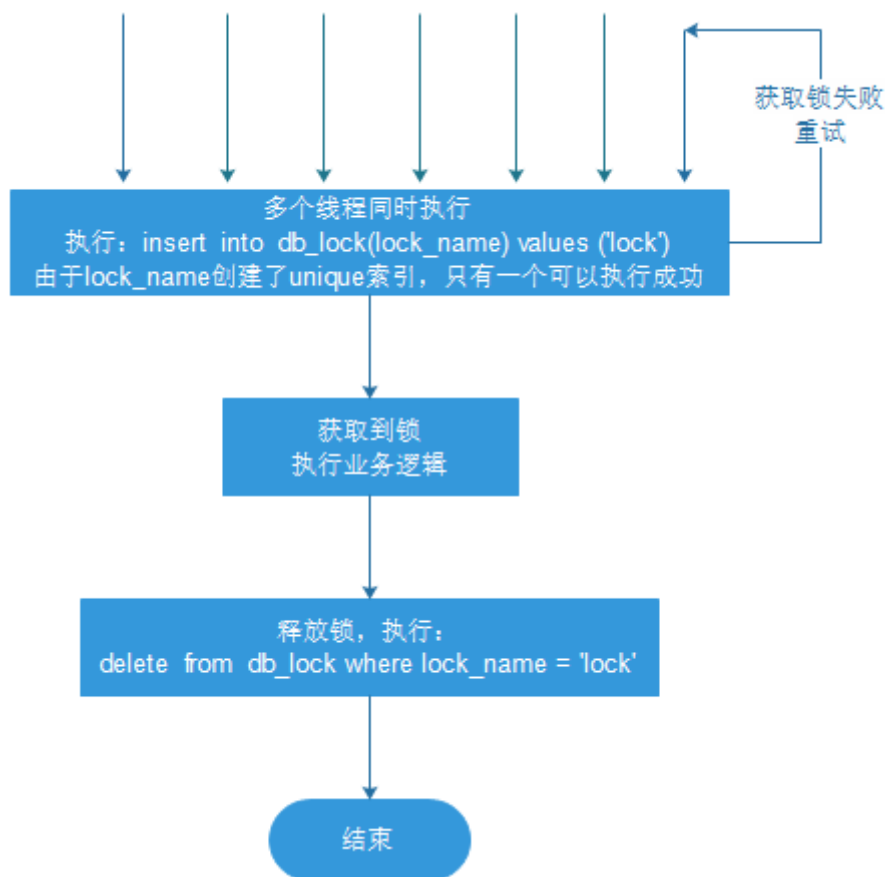
```

1 public interface LockMapper extends BaseMapper<Lock> {
2 }

```

2.1. 基本思路

synchronized关键字和ReentrantLock锁都是独占排他锁，即多个线程争抢一个资源时，同一时刻只有一个线程可以抢占该资源，其他线程只能阻塞等待，直到占有资源的线程释放该资源。



1. 线程同时获取锁 (insert)
2. 获取成功, 执行业务逻辑, 执行完成释放锁 (delete)
3. 其他线程等待重试

2.2. 代码实现

改造StockService:

```
1  @Service
2  public class StockService {
3
4      @Autowired
5      private StockMapper stockMapper;
6
7      @Autowired
8      private LockMapper lockMapper;
9
10     /**
11      * 数据库分布式锁
12      */
13     public void checkAndLock() {
14
15         // 加锁
16         Lock lock = new Lock(null, "lock", this.getClass().getName(), new
Date(), null);
17         try {
18             this.lockMapper.insert(lock);
19         } catch (Exception ex) {
20             // 获取锁失败, 则重试
21             try {
```

```

22         Thread.sleep(50);
23         this.checkAndLock();
24     } catch (InterruptedException e) {
25         e.printStackTrace();
26     }
27 }
28
29 // 先查询库存是否充足
30 Stock stock = this.stockMapper.selectById(1L);
31
32 // 再减库存
33 if (stock != null && stock.getCount() > 0){
34
35     stock.setCount(stock.getCount() - 1);
36     this.stockMapper.updateById(stock);
37 }
38
39 // 释放锁
40 this.lockMapper.deleteById(lock.getId());
41 }
42 }

```

加锁:

```

1 // 加锁
2 Lock lock = new Lock(null, "lock", this.getClass().getName(), new Date(),
3 null);
4 try {
5     this.lockMapper.insert(lock);
6 } catch (Exception ex) {
7     // 获取锁失败，则重试
8     try {
9         Thread.sleep(50);
10        this.checkAndLock();
11    } catch (InterruptedException e) {
12        e.printStackTrace();
13    }
14 }

```

解锁:

```

1 // 释放锁
2 this.lockMapper.deleteById(lock.getId());

```

使用Jmeter压力测试结果:

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	476	152	1345	1872	3119	37	6957	0.00%	20.1/sec	3.74	2.54
TOTAL	5000	476	152	1345	1872	3119	37	6957	0.00%	20.1/sec	3.74	2.54

可以看到性能感人。mysql数据库库存余量为0，可以保证线程安全。

2.3. 缺陷及解决方案

缺点：

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。

解决方案：给 锁数据库 搭建主备

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。

解决方案：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。

3. 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。

解决方案：记录获取锁的主机信息和线程信息，如果相同线程要获取锁，直接重入。

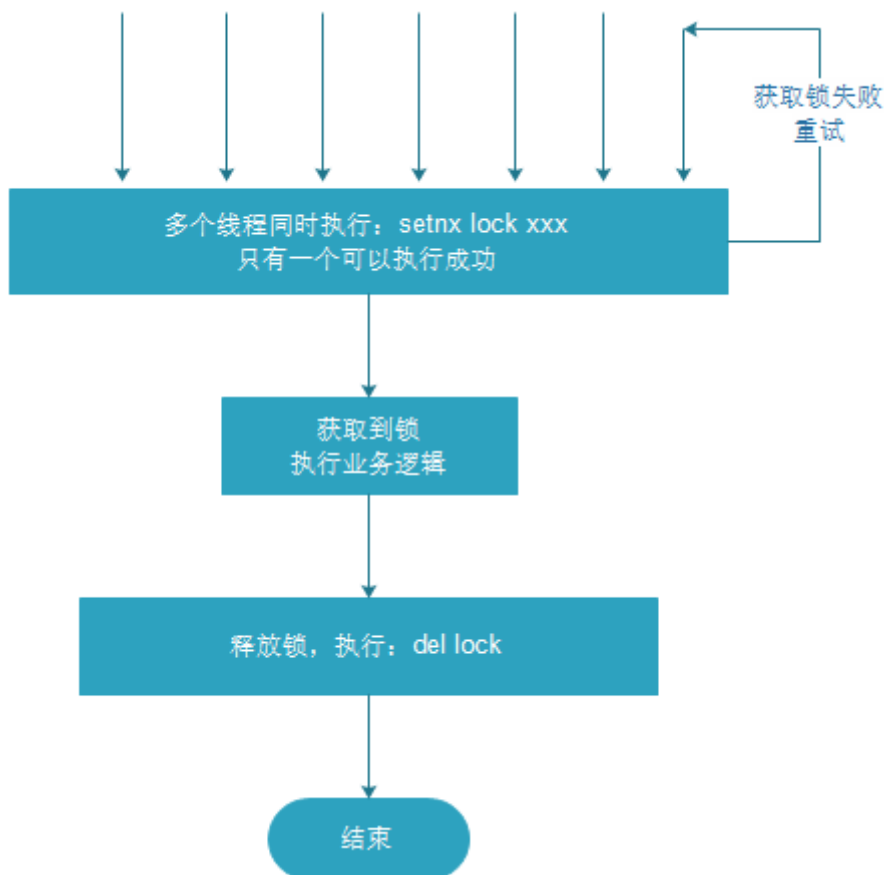
4. 受制于数据库性能，并发能力有限。

解决方案：无法解决。

3. 基于redis实现分布式锁

3.1. 基本实现

借助于redis中的命令setnx(key, value)，key不存在就新增，存在就什么都不做。同时有多个客户端发送setnx命令，只有一个客户端可以成功，返回1（true）；其他的客户端返回0（false）。



1. 多个客户端同时获取锁（setnx）

2. 获取成功，执行业务逻辑，执行完成释放锁 (del)
3. 其他客户端等待重试

改造StockService方法:

```
1  @Service
2  public class StockService {
3
4      @Autowired
5      private StockMapper stockMapper;
6
7      @Autowired
8      private LockMapper lockMapper;
9
10     @Autowired
11     private StringRedisTemplate redisTemplate;
12
13     public void checkAndLock() {
14
15         // 加锁，获取锁失败重试
16         while (!this.redisTemplate.opsForValue().setIfAbsent("lock",
17 "xxx")){
18             try {
19                 Thread.sleep(100);
20             } catch (InterruptedException e) {
21                 e.printStackTrace();
22             }
23
24             // 先查询库存是否充足
25             Stock stock = this.stockMapper.selectById(1L);
26
27             // 再减库存
28             if (stock != null && stock.getCount() > 0){
29
30                 stock.setCount(stock.getCount() - 1);
31                 this.stockMapper.updateById(stock);
32             }
33
34             // 释放锁
35             this.redisTemplate.delete("lock");
36         }
37     }
```

其中，加锁:

```
1  // 加锁，获取锁失败重试
2  while (!this.redisTemplate.opsForValue().setIfAbsent("lock", "xxx")){
3      try {
4          Thread.sleep(100);
5      } catch (InterruptedException e) {
6          e.printStackTrace();
7      }
8  }
```

解锁:

```
1 // 释放锁
2 this.redisTemplate.delete("lock");
```

使用Jmeter压力测试如下:

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	1752	25	5933	10329	20459	11	53720	0.00%	47.8/sec	8.86	6.02
TOTAL	5000	1752	25	5933	10329	20459	11	53720	0.00%	47.8/sec	8.86	6.02

查看mysql数据库:

id	product_code	stock_code	count	version
1	1001	001	0	25000

3.2. 防死锁

```
28 public void checkAndLock() {
29     // 加锁, 获取锁失败重试
30     while (!this.redisTemplate.opsForValue().setIfAbsent("lock", "xxx")) {
31         try {
32             Thread.sleep( millis: 100);
33         } catch (InterruptedException e) {
34             e.printStackTrace();
35         }
36     }
37
38     // 先查询库存是否充足
39     Stock stock = this.stockMapper.selectById(1L);
40     // 再减库存
41     if (stock != null && stock.getCount() > 0) {
42         stock.setCount(stock.getCount() - 1);
43         this.stockMapper.updateById(stock);
44     }
45
46     // 释放锁
47     this.redisTemplate.delete( key: "lock");
48 }
```

解决: 给锁设置过期时间, 自动释放锁。

设置过期时间两种方式:

1. 通过expire设置过期时间 (缺乏原子性: 如果在setnx和expire之间出现异常, 锁也无法释放)
2. 使用set指令设置过期时间: set key value ex 3 nx (既达到setnx的效果, 又设置了过期时间)


```

26      @Autowired
27      private StringRedisTemplate redisTemplate;
28
29      public void checkAndLock() {
30          // 加锁，获取锁失败重试
31          while (!this.redisTemplate.opsForValue().setIfAbsent(key: "lock", value: "xxx", timeout: 3, TimeUnit.SECONDS)) {
32              try {
33                  Thread.sleep(millis: 50);
34              } catch (InterruptedException e) {
35                  e.printStackTrace();
36              }
37          }
38      }

```

压力测试肯定也没有问题。

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	1800	25	5973	10254	21034	9	45239	0.00%	47.7/sec	8.86	6.02
TOTAL	5000	1800	25	5973	10254	21034	9	45239	0.00%	47.7/sec	8.86	6.02

问题：可能会释放其他服务器的锁。

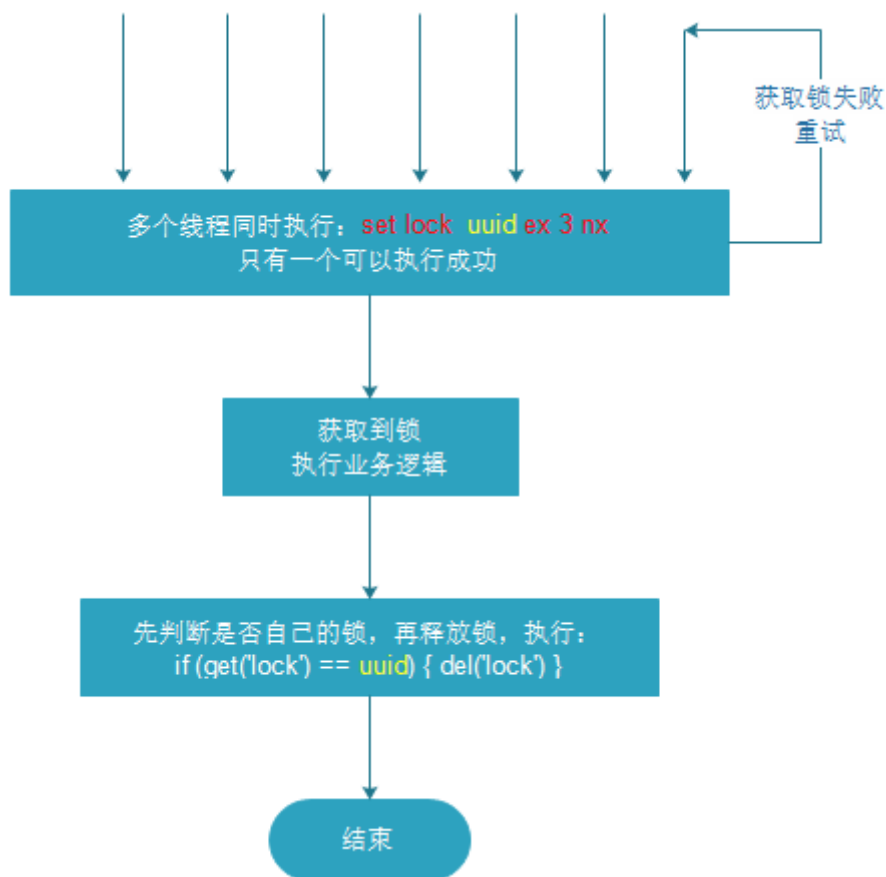
场景：如果业务逻辑的执行时间是7s。执行流程如下

1. index1业务逻辑没执行完，3秒后锁被自动释放。
2. index2获取到锁，执行业务逻辑，3秒后锁被自动释放。
3. index3获取到锁，执行业务逻辑
4. index1业务逻辑执行完成，开始调用del释放锁，这时释放的是index3的锁，导致index3的业务只执行1s就被别人释放。

最终等于没锁的情况。

解决：setnx获取锁时，设置一个指定的唯一值（例如：uuid）；释放前获取这个值，判断是否自己的锁

3.3. 防误删



实现如下：

```
31 public void checkAndLock() {
32     // 加锁，获取锁失败重试
33     String uuid = UUID.randomUUID().toString();
34     while (!this.redisTemplate.opsForValue().setIfAbsent(key: "lock", uuid, timeout 3, TimeUnit.SECONDS)) {
35         try {...} catch (InterruptedException e) {...}
36     }
37
38     // 先查询库存是否充足
39     Stock stock = this.stockMapper.selectById(1L);
40     // 再减库存
41     if (stock != null && stock.getCount() > 0) {
42         stock.setCount(stock.getCount() - 1);
43         this.stockMapper.updateById(stock);
44     }
45
46     // 释放锁
47     if (StringUtils.equals(uuid, this.redisTemplate.opsForValue().get("lock"))) {
48         this.redisTemplate.delete(key: "lock");
49     }
50 }
```

问题：删除操作缺乏原子性。

场景：

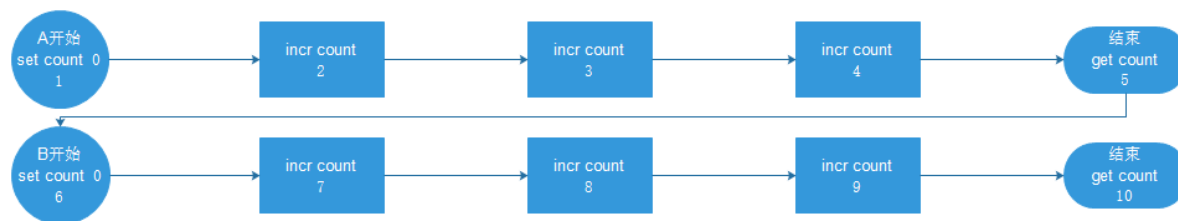
1. index1执行删除时，查询到的lock值确实和uuid相等
2. index1执行删除前，lock刚好过期时间已到，被redis自动释放
3. index2获取了lock
4. index1执行删除，此时会把index2的lock删除

解决方案：没有一个命令可以同时做到判断 + 删除，所有只能通过其他方式实现（LUA脚本）

3.4. redis中的lua脚本

3.4.1. 现实问题

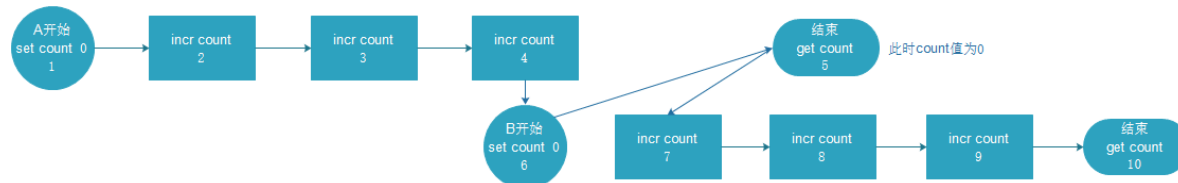
redis采用单线程架构，可以保证单个命令的原子性，但是无法保证一组命令在高并发场景下的原子性。
例如：



在串行场景下：A和B的值肯定都是3

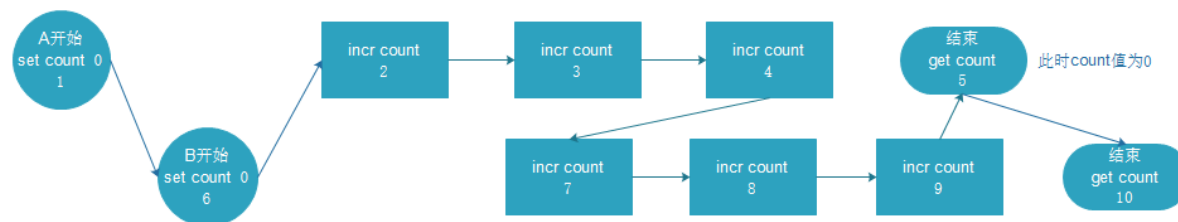
在并发场景下：A和B的值可能在0-6之间。

极限情况下1：



则A的结果是0，B的结果是3

极限情况下2：



则A和B的结果都是6

如果redis客户端通过lua脚本把3个命令一次性发送给redis服务器，那么这三个指令就不会被其他客户端指令打断。Redis 也保证脚本会以原子性(atomic)的方式执行：当某个脚本正在运行的时候，不会有其他脚本或 Redis 命令被执行。这和使用 MULTI/ EXEC 包围的事务很类似。

但是MULTI/ EXEC方法来使用事务功能，将一组命令打包执行，无法进行业务逻辑的操作。这期间有某一条命令执行报错（例如给字符串自增），其他的命令还是会执行，并不会回滚。

3.4.2. lua介绍

Lua 是一种轻量小巧的脚本语言，用标准C语言编写并以源代码形式开放，其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

设计目的

其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Lua 特性

- **轻量级**：它用标准C语言编写并以源代码形式开放，编译后仅仅一百余K，可以很方便的嵌入别的程序里。
- **可扩展**：Lua提供了非常易于使用的扩展接口和机制：由宿主语言(通常是C或C++)提供这些功能，Lua可以使用它们，就像是本来就内置的功能一样。
- 其它特性：

- 支持面向过程(procedure-oriented)编程和函数式编程(functional programming);
- 自动内存管理; 只提供了一种通用类型的表 (table) , 用它可以实现数组, 哈希表, 集合, 对象;
- 语言内置模式匹配; 闭包(closure); 函数也可以看做一个值; 提供多线程 (协同进程, 并非操作系统所支持的线程) 支持;
- 通过闭包和table可以很方便地支持面向对象编程所需要的一些关键机制, 比如数据抽象, 虚函数, 继承和重载等。

3.4.3. lua基本语法

对lua脚本感兴趣的同学, 请移步到官方教程或者《菜鸟教程》。这里仅以redis中可能会用到的部分语法作介绍。

```
1 a = 5          -- 全局变量
2 local b = 5    -- 局部变量, redis只支持局部变量
3 a, b = 10, 2*x -- 等价于      a=10; b=2*x
```

流程控制:

```
1 if( 布尔表达式 1)
2 then
3     --[ 在布尔表达式 1 为 true 时执行该语句块 --]
4 elseif( 布尔表达式 2)
5 then
6     --[ 在布尔表达式 2 为 true 时执行该语句块 --]
7 else
8     --[ 如果以上布尔表达式都不为 true 则执行该语句块 --]
9 end
```

3.4.4. redis执行lua脚本 - EVAL指令

在redis中需要通过eval命令执行lua脚本。

格式:

```
1 EVAL script numkeys key [key ...] arg [arg ...]
2 script: lua脚本字符串, 这段Lua脚本不需要 (也不应该) 定义函数。
3 numkeys: lua脚本中KEYS数组的大小
4 key [key ...]: KEYS数组中的元素
5 arg [arg ...]: ARGV数组中的元素
```

案例1: 基本案例

```
1 EVAL "return 10" 0
```

输出: (integer) 1

案例2: 动态传参

```

1 EVAL "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 5 10 20 30 40 50 60 70 80 90
2 # 输出: 10 20 60 70
3
4 EVAL "if KEYS[1] > ARGV[1] then return 1 else return 0 end" 1 10 20
5 # 输出: 0
6
7 EVAL "if KEYS[1] > ARGV[1] then return 1 else return 0 end" 1 20 10
8 # 输出: 1

```

传入了两个参数10和20，KEYS的长度是1，所以KEYS中有一个元素10，剩余的一个20就是ARGV数组的元素。

redis.call()中的redis是redis中提供的lua脚本类库，仅在redis环境中可以使用该类库。

案例3：执行redis类库方法

```

1 set aaa 10 -- 设置一个aaa值为10
2 EVAL "return redis.call('get', 'aaa')" 0
3 ## 通过return把call方法返回给redis客户端，打印: "10"

```

注意：脚本里使用的所有键都应该由KEYS数组来传递。但并不是强制性的，代价是这样写出的脚本不能被Redis集群所兼容。

案例4：给redis类库方法动态传参

```

1 EVAL "return redis.call('set', KEYS[1], ARGV[1])" 1 bbb 20

```

```

127.0.0.1:6379> EVAL "return redis.call('set', KEYS[1], ARGV[1])" 1 bbb 20
OK
127.0.0.1:6379> get bbb
"20"
127.0.0.1:6379>

```

学到这里基本可以应付redis分布式锁所需要的脚本知识了。

案例5：pcall函数的使用（了解）

```

1 -- 当call() 在执行命令的过程中发生错误时，脚本会停止执行，并返回一个脚本错误，输出错误信息
2 EVAL "return redis.call('sets', KEYS[1], ARGV[1]), redis.call('set', KEYS[2], ARGV[2])" 2 bbb ccc 20 30
3 -- pcall函数不影响后续指令的执行
4 EVAL "return redis.pcall('sets', KEYS[1], ARGV[1]), redis.pcall('set', KEYS[2], ARGV[2])" 2 bbb ccc 20 30

```

注意：set方法写成了sets，肯定会报错。

```

127.0.0.1:6379> keys *
1) "bbb"
127.0.0.1:6379> EVAL "return redis.call('sets', KEYS[1], ARGV[1]), redis.call('set', KEYS[2], ARGV[2])" 2 bbb ccc 20 30
(error) ERR Error running script (call to f_18b8c6febe98e7623c138beee90821d8060ad528): @user_script:1: @user_script: 1: Unknown Redis command called from Lua script
127.0.0.1:6379> keys *
1) "bbb"
127.0.0.1:6379> EVAL "return redis.pcall('sets', KEYS[1], ARGV[1]), redis.pcall('set', KEYS[2], ARGV[2])" 2 bbb ccc 20 30
(error) @user_script: 1: Unknown Redis command called from Lua script
127.0.0.1:6379> keys *
1) "bbb"
2) "ccc"
127.0.0.1:6379>

```

3.4.5. 性能优化 - EVALSHA指令

EVAL 命令要求你在每次执行脚本的时候都发送一次脚本主体(script body)。Redis 有一个内部的缓存机制，因此它不会每次都重新编译脚本，不过在很多场合，付出无谓的带宽来传送脚本主体并不是最佳选择。

为了减少带宽的消耗，Redis 实现了 EVALSHA命令，它的作用和 **EVAL** 一样，都用于执行lua脚本，但它接受的第一个参数不是脚本，而是脚本的 SHA1 编码。

EVALSHA 命令的表现如下：

如果服务器存在SHA1编码对应的的脚本，那么就会执行这个脚本；如果服务器不存在SHA1编码对应的脚本，那么会返回一个特殊的错误，提醒用户使用 EVAL 代替 EVALSHA

```

1  -- script load会对redis脚本进行sha1加密生成加密字符串，无论脚本多长，密文长度固定，会以
   密文为key缓存lua基本
2  SCRIPT LOAD "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}"
3  -- 通过密文方式执行缓存的lua脚本
4  EVALSHA a42059b356c875f0717db19a51f6aaca9ae659ea 2 aa bb cc dd
5  -- 判断缓存中是否存在某个lua脚本，有返回1，无返回0
6  SCRIPT EXISTS a42059b356c875f0717db19a51f6aaca9ae659ea
7  -- 删除缓存中的lua脚本
8  SCRIPT FLUSH

```

测试：

```

127.0.0.1:6379> SCRIPT LOAD "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}"
"a42059b356c875f0717db19a51f6aaca9ae659ea"
127.0.0.1:6379> EVALSHA a42059b356c875f0717db19a51f6aaca9ae659ea 2 aa bb cc dd
1) "aa"
2) "bb"
3) "cc"
4) "dd"
127.0.0.1:6379> SCRIPT FLUSH
OK
127.0.0.1:6379> EVALSHA a42059b356c875f0717db19a51f6aaca9ae659ea 2 aa bb cc dd
(error) NOSCRIPT No matching script. Please use EVAL.
127.0.0.1:6379>

```

3.5. 使用lua保证删除原子性

删除LUA脚本：

```

1  if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del',
   KEYS[1]) else return 0 end

```

代码实现：

```
1 public void checkAndLock() {
2     // 加锁，获取锁失败重试
3     String uuid = UUID.randomUUID().toString();
4     while (!this.redisTemplate.opsForValue().setIfAbsent("lock", uuid, 3,
5         TimeUnit.SECONDS)){
6         try {
7             Thread.sleep(50);
8         } catch (InterruptedException e) {
9             e.printStackTrace();
10        }
11
12        // 先查询库存是否充足
13        Stock stock = this.stockMapper.selectById(1L);
14        // 再减库存
15        if (stock != null && stock.getCount() > 0){
16            stock.setCount(stock.getCount() - 1);
17            this.stockMapper.updateById(stock);
18        }
19
20        // 释放锁
21        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return
22        redis.call('del', KEYS[1]) else return 0 end";
23        this.redisTemplate.execute(new DefaultRedisScript<>(script,
24            Long.class), Arrays.asList("lock"), uuid);
25    }
```

压力测试：

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	1694	29	5565	9141	18444	15	53846	0.00%	49.7/sec	9.23	6.27
TOTAL	5000	1694	29	5565	9141	18444	15	53846	0.00%	49.7/sec	9.23	6.27

库存量也没有问题，截图略过。。。

3.6. 可重入锁

由于上述加锁命令使用了 SETNX，一旦键存在就无法再设置成功，这就导致后续同一线程内继续加锁，将会加锁失败。当一个线程执行一段代码成功获取锁之后，继续执行时，又遇到加锁的子任务代码，可重入性就保证线程能继续执行，而不可重入就是需要等待锁释放之后，再次获取锁成功，才能继续往下执行。

用一段 Java 代码解释可重入：

```

1 public synchronized void a() {
2     b();
3 }
4
5 public synchronized void b() {
6     // pass
7 }

```

假设 X 线程在 a 方法获取锁之后，继续执行 b 方法，如果此时**不可重入**，线程就必须等待锁释放，再次争抢锁。

锁明明是被 X 线程拥有，却还需要等待自己释放锁，然后再去抢锁，这看起来就很奇怪，我释放我自己~

可重入性就可以解决这个问题，当线程拥有锁之后，往后再遇到加锁方法，直接将加锁次数加 1，然后再执行方法逻辑。退出加锁方法之后，加锁次数再减 1，当加锁次数为 0 时，锁才被真正的释放。

可以看到可重入锁最大特性就是计数，计算加锁的次数。所以当可重入锁需要在分布式环境实现时，我们也就需要统计加锁次数。

解决方案：redis + Hash

3.6.1. 加锁脚本

Redis 提供了 Hash（哈希表）这种可以存储键值对数据结构。所以我们可以使用 Redis Hash 存储的锁的重入次数，然后利用 lua 脚本判断逻辑。

```

1 if (redis.call('exists', KEYS[1]) == 0 or redis.call('hexists', KEYS[1],
2 ARGV[1]) == 1)
3 then
4     redis.call('hincrby', KEYS[1], ARGV[1], 1);
5     redis.call('expire', KEYS[1], ARGV[2]);
6     return 1;
7 else
8     return 0;
9 end

```

假设值为：KEYS:[lock], ARGV[uuid, expire]

如果锁不存在或者这是自己的锁，就通过hincrby（不存在就新增并加1，存在就加1）获取锁或者锁次数加1。

3.6.2. 解锁脚本


```

1  -- 判断 hash set 可重入 key 的值是否等于 0
2  -- 如果为 nil 代表 自己的锁已不存在，在尝试解其他线程的锁，解锁失败
3  -- 如果为 0 代表 可重入次数被减 1
4  -- 如果为 1 代表 该可重入 key 解锁成功
5  if(redis.call('hexists', KEYS[1], ARGV[1]) == 0) then
6      return nil;
7  elseif(redis.call('hincrby', KEYS[1], ARGV[1], -1) > 0) then
8      return 0;
9  else
10     redis.call('del', KEYS[1]);
11     return 1;
12 end;

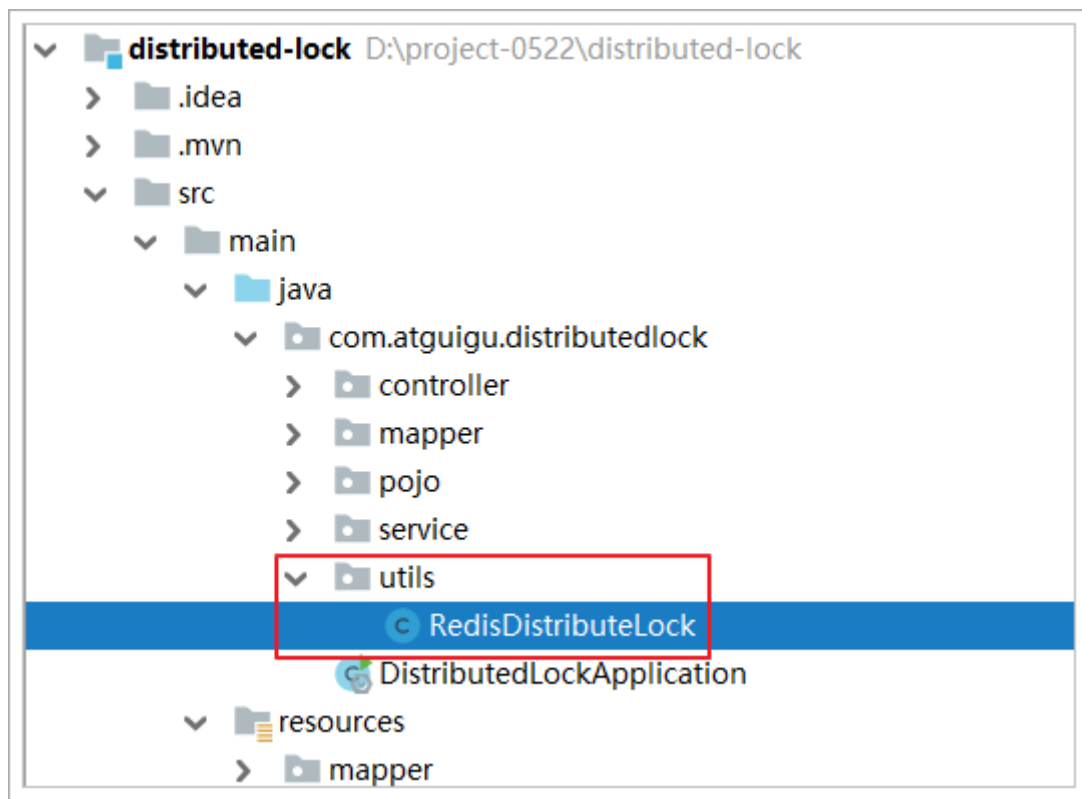
```

这里之所以没有跟加锁一样使用 Boolean ,这是因为解锁 lua 脚本中，三个返回值含义如下：

- 1 代表解锁成功，锁被释放
- 0 代表可重入次数被减 1
- null 代表其他线程尝试解锁，解锁失败

3.6.3. 代码实现

由于加解锁代码量相对较多，这里可以封装成一个工具类：



具体实现：

```

1  public class RedisDistributeLock {
2
3      private StringRedisTemplate redisTemplate;
4
5      // 线程局部变量，可以在线程内共享参数
6      private String lockName;
7      private String uuid;

```

```

8     private Integer expire = 30;
9     private static final ThreadLocal<String> THREAD_LOCAL = new
ThreadLocal<>();
10
11     public RedisDistributeLock(StringRedisTemplate redisTemplate, String
lockName) {
12         this.redisTemplate = redisTemplate;
13         this.lockName = lockName;
14         this.uuid = THREAD_LOCAL.get();
15         if (StringUtils.isBlank(uuid)) {
16             this.uuid = UUID.randomUUID().toString();
17             THREAD_LOCAL.set(uuid);
18         }
19     }
20
21     public void lock(){
22         this.lock(expire);
23     }
24
25     public void lock(Integer expire){
26         this.expire = expire;
27         String script = "if (redis.call('exists', KEYS[1]) == 0 or
redis.call('hexists', KEYS[1], ARGV[1]) == 1) " +
28             "then" +
29             "    redis.call('hincrby', KEYS[1], ARGV[1], 1);" +
30             "    redis.call('expire', KEYS[1], ARGV[2]);" +
31             "    return 1;" +
32             "else" +
33             "    return 0;" +
34             "end";
35         if (!this.redisTemplate.execute(new DefaultRedisScript<>(script,
Boolean.class), Arrays.asList(lockName), uuid, expire.toString())){
36             try {
37                 // 没有获取到锁，重试
38                 Thread.sleep(60);
39                 lock(expire);
40             } catch (InterruptedException e) {
41                 e.printStackTrace();
42             }
43         }
44     }
45
46     public void unlock(){
47         String script = "if(redis.call('hexists', KEYS[1], ARGV[1]) == 0)
then " +
48             "    return nil; " +
49             "elseif(redis.call('hincrby', KEYS[1], ARGV[1], -1) > 0)
then " +
50             "    return 0; " +
51             "else " +
52             "    redis.call('del', KEYS[1]); " +
53             "    return 1; " +
54             "end";
55         // 如果返回值没有使用Boolean, Spring-data-redis 进行类型转换时将会把 null
转为 false, 这就会影响我们逻辑判断
56         // 所以返回类型只好使用 Long: null-解锁失败; 0-重入次数减1; 1-解锁成功。
57         Long result = this.redisTemplate.execute(new DefaultRedisScript<>
(script, Long.class), Arrays.asList(lockName), uuid);

```

```

58         // 如果未返回值，代表尝试解锁其他线程的锁
59         if (result == null) {
60             throw new IllegalMonitorStateException("attempt to unlock lock,
not locked by lockName: "
61                 + lockName + " with request: " + uuid);
62         } else if (result == 1) {
63             THREAD_LOCAL.remove();
64         }
65     }
66 }

```

3.6.4. 使用及测试

在业务代码中使用：

```

1  public void checkAndLock() {
2      // 加锁，获取锁失败重试
3      RedisDistributeLock lock = new RedisDistributeLock(this.redisTemplate,
"lock");
4      lock.lock();
5
6      // 先查询库存是否充足
7      Stock stock = this.stockMapper.selectById(1L);
8      // 再减库存
9      if (stock != null && stock.getCount() > 0){
10         stock.setCount(stock.getCount() - 1);
11         this.stockMapper.updateById(stock);
12     }
13     // this.testSubLock();
14
15     // 释放锁
16     lock.unlock();
17 }

```

测试：

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	1781	32	5921	9290	16464	10	32488	0.00%	47.7/sec	8.86	6.01
TOTAL	5000	1781	32	5921	9290	16464	10	32488	0.00%	47.7/sec	8.86	6.01

测试可重入性：

```

59     public void checkAndLock() {
60         // 加锁, 获取锁失败重试
61         RedisDistributeLock lock = new RedisDistributeLock(this.redisTemplate, lockName: "lock");
62         lock.lock();
63
64         // 先查询库存是否充足
65         Stock stock = this.stockMapper.selectById(1L);
66         // 再减库存
67         if (stock != null && stock.getCount() > 0) {...}
71         this.testSubLock();
72
73         // 释放锁
74         lock.unlock();
75     }
76
77     public void testSubLock() {
78         RedisDistributeLock lock = new RedisDistributeLock(this.redisTemplate, lockName: "lock");
79         lock.lock();
80         System.out.println("测试可重入。 . . ");
81         lock.unlock();
82     }

```

3.7. 自动续期

lua脚本:

```

1  if(redis.call('hexists', KEYS[1], ARGV[1]) == 1) then
2      redis.call('expire', KEYS[1], ARGV[2]);
3      return 1;
4  else
5      return 0;
6  end

```

在RedisDistributeLock中添加renewExpire方法:

```

1  private static final Timer TIMER = new Timer();
2
3  /**
4   * 开启定时器, 自动续期
5   */
6  private void renewExpire(){
7      String script = "if(redis.call('hexists', KEYS[1], ARGV[1]) == 1) then
8      redis.call('expire', KEYS[1], ARGV[2]); return 1; else return 0; end";
9      TIMER.schedule(new TimerTask() {
10         @Override
11         public void run() {
12             // 如果uuid为空, 则终止定时任务
13             if (StringUtils.isNotBlank(uuid)) {
14                 redisTemplate.execute(new DefaultRedisScript<>(script,
15                 Boolean.class), Arrays.asList(lockName), RedisDistributeLock.this.uuid,
16                 expire.toString());
17                 renewExpire();
18             }
19         }
20     }, expire * 1000 / 3);
21 }

```

在lock方法中使用：

```
37 @ public void lock(Integer expire){
38     this.expire = expire;
39     String script = "if (redis.call('exists', KEYS[1]) == 0 or re
40         "then" +
41         "    redis.call('hincrby', KEYS[1], ARGV[1], 1);" +
42         "    redis.call('expire', KEYS[1], ARGV[2]);" +
43         "    return 1;" +
44         "else" +
45         "    return 0;" +
46         "end";
47     if (!this.redisTemplate.execute(new DefaultRedisScript<>(scri
56     // 获取锁成功后，自动续期
57     renewExpire();
58 }
```

在unlock方法中添加红框中的代码：

```
60 public void unlock(){
61     String script = "if(redis.call('hexists', KEYS[1], ARGV[1]) =
62         "    return nil; " +
63         "elseif(redis.call('hincrby', KEYS[1], ARGV[1], -1) >
64         "    return 0; " +
65         "else " +
66         "    redis.call('del', KEYS[1]); " +
67         "    return 1; " +
68         "end";
69     //...
71     Long result = this.redisTemplate.execute(new DefaultRedisScri
72     // 如果未返回值，代表尝试解其他线程的锁
73     if (result == null) {...} else if (result == 1) {...}
79     // 释放锁之后，把uuid置为空，停止定时任务
80     this.uuid = null;
81 }
```

3.8. 红锁算法

redis集群状态下的问题：

1. 客户端A从master获取到锁
2. 在master将锁同步到slave之前，master宕掉了。
3. slave节点被晋级为master节点
4. 客户端B取得了同一个资源被客户端A已经获取到的另外一个锁。

安全失效！

解决集群下锁失效，参照redis官方网站针对redlock文档：<https://redis.io/topics/distlock>

在算法的分布式版本中，我们假设有N个Redis服务器。这些节点是完全独立的，因此我们不使用复制或任何其他隐式协调系统。前几节已经描述了如何在单个实例中安全地获取和释放锁，在分布式锁算法中，将使用相同的方法在单个实例中获取和释放锁。将N设置为5是一个合理的值，因此需要在不同的计算机或虚拟机上运行5个Redis主服务器，确保它们以独立的方式发生故障。

为了获取锁，客户端执行以下操作：

1. 客户端以毫秒为单位获取当前时间的时间戳，作为**起始时间**。
2. 客户端尝试在所有N个实例中顺序使用相同的键名、相同的随机值来获取锁定。每个实例尝试获取锁都需要时间，客户端应该设置一个远小于总锁定时间的超时时间。例如，如果自动释放时间为10秒，则**尝试获取锁的超时时间**可能在5到50毫秒之间。这样可以防止客户端长时间与处于故障状态的Redis节点进行通信：如果某个实例不可用，尽快尝试与下一个实例进行通信。
3. 客户端获取当前时间 减去在步骤1中获得的**起始时间**，来计算**获取锁所花费的时间**。当且仅当客户端能够在大多数实例（至少3个）中获取锁时，并且获取锁所花费的总时间小于锁有效时间，则认为已获取锁。
4. 如果获取了锁，则将锁有效时间减去 **获取锁所花费的时间**，如步骤3中所计算。
5. 如果客户端由于某种原因（无法锁定 $N / 2 + 1$ 个实例或有效时间为负）而未能获得该锁，它将尝试解锁所有实例（即使没有锁定成功的实例）。

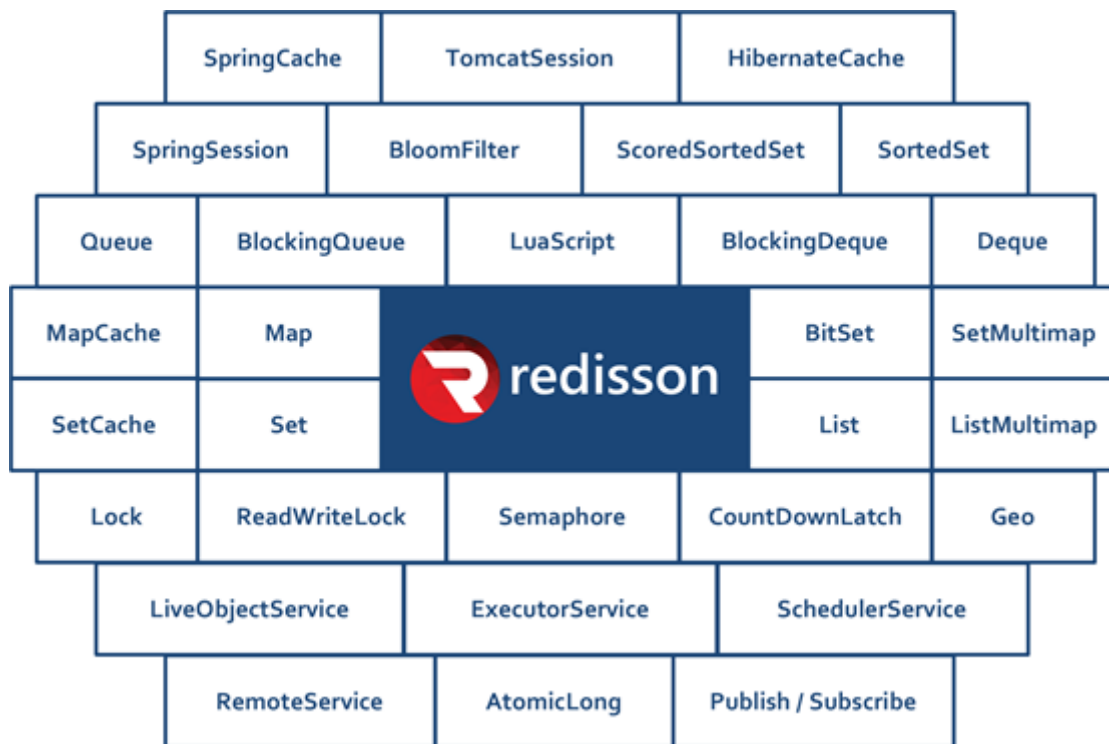
每台计算机都有一个本地时钟，我们通常可以依靠不同的计算机来产生很小的时钟漂移。只有在拥有锁的客户端将在锁有效时间内（如步骤3中获得的）减去一段时间（仅几毫秒）的情况下终止工作，才能保证这一点。以补偿进程之间的时钟漂移

当客户端无法获取锁时，它应该在随机延迟后重试，以避免同时获取同一资源的多个客户端之间不同步（这可能会导致脑裂的情况：没人胜）。同样，客户端在大多数Redis实例中尝试获取锁的速度越快，出现裂脑情况（以及需要重试）的窗口就越小，因此理想情况下，客户端应尝试将SET命令发送到N个实例同时使用多路复用。

值得强调的是，对于未能获得大多数锁的客户端，尽快释放（部分）获得的锁有多么重要，这样就不必等待锁定期满才能再次获得锁（但是，如果发生了网络分区，并且客户端不再能够与Redis实例进行通信，则在等待密钥到期时需要付出可用性损失）。

3.9. redisson中的分布式锁

Redisson是一个在Redis的基础上实现的Java驻内存数据网格（In-Memory Data Grid）。它不仅提供了一系列的分布式的Java常用对象，还提供了许多分布式服务。其中包括(BitSet, Set, Multimap, SortedSet, Map, List, Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock, AtomicLong, CountDownLatch, Publish / Subscribe, Bloom filter, Remote service, Spring cache, Executor service, Live Object service, Scheduler service) Redisson提供了使用Redis的最简单和最便捷的方法。Redisson的宗旨是促进使用者对Redis的关注分离（Separation of Concern），从而让使用者能够将精力更集中地放在处理业务逻辑上。



官方文档地址: <https://github.com/redisson/redisson/wiki>

3.9.1. 可重入锁 (Reentrant Lock)

基于Redis的Redisson分布式可重入锁 `RLock` Java对象实现了 `java.util.concurrent.locks.Lock` 接口。

大家都知道，如果负责储存这个分布式锁的Redisson节点宕机以后，而且这个锁正好处于锁住的状态时，这个锁会出现锁死的状态。为了避免这种情况的发生，Redisson内部提供了一个监控锁的看门狗，它的作用是在Redisson实例被关闭前，不断的延长锁的有效期。默认情况下，看门狗检查锁的超时时间是30秒钟，也可以通过修改 `Config.lockWatchdogTimeout` 来另行指定。

`RLock` 对象完全符合Java的Lock规范。也就是说只有拥有锁的进程才能解锁，其他进程解锁则会抛出 `IllegalMonitorStateException` 错误。

另外Redisson还通过加锁的方法提供了 `leaseTime` 的参数来指定加锁的时间。超过这个时间后锁便自动解开了。

```

1  RLock lock = redisson.getLock("anyLock");
2  // 最常见的使用方法
3  lock.lock();
4
5  // 加锁以后10秒钟自动解锁
6  // 无需调用unlock方法手动解锁
7  lock.lock(10, TimeUnit.SECONDS);
8
9  // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
10 boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
11 if (res) {
12     try {
13         ...
14     } finally {
15         lock.unlock();

```

```
16     }
17 }
```

1. 引入依赖

```
1 <dependency>
2     <groupId>org.redisson</groupId>
3     <artifactId>redisson</artifactId>
4     <version>3.11.2</version>
5 </dependency>
```

2. 添加配置

```
1 @Configuration
2 public class RedissonConfig {
3
4     @Bean
5     public RedissonClient redissonClient(){
6         Config config = new Config();
7         // 可以用"rediss://"来启用SSL连接
8         config.useSingleServer().setAddress("redis://172.16.116.100:6379");
9         return Redisson.create(config);
10    }
11 }
```

3. 代码中使用

```
1 @Autowired
2 private RedissonClient redissonClient;
3
4 public void checkAndLock() {
5     // 加锁，获取锁失败重试
6     RLock lock = this.redissonClient.getLock("lock");
7     lock.lock();
8
9     // 先查询库存是否充足
10    Stock stock = this.stockMapper.selectById(1L);
11    // 再减库存
12    if (stock != null && stock.getCount() > 0){
13        stock.setCount(stock.getCount() - 1);
14        this.stockMapper.updateById(stock);
15    }
16
17    // 释放锁
18    lock.unlock();
19 }
```

4. 压力测试

性能跟我们手写的区别不大。

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	1779	264	5433	8620	15268	17	39804	0.00%	49.2/sec	9.13	6.20
TOTAL	5000	1779	264	5433	8620	15268	17	39804	0.00%	49.2/sec	9.13	6.20

数据库也没有问题

3.9.2. 公平锁 (Fair Lock)

基于Redis的Redisson分布式可重入公平锁也是实现了 `java.util.concurrent.locks.Lock` 接口的一种 `RLock` 对象。同时还提供了 [异步 \(Async\)](#)、[反射式 \(Reactive\)](#) 和 [RxJava2标准](#) 的接口。它保证了当多个Redisson客户端线程同时请求加锁时，优先分配给先发出请求的线程。所有请求线程会在一个队列中排队，当某个线程出现宕机时，Redisson会等待5秒后继续下一个线程，也就是说如果前面有5个线程都处于等待状态，那么后面的线程会等待至少25秒。

```
1 RLock fairLock = redisson.getFairLock("anyLock");
2 // 最常见的使用方法
3 fairLock.lock();
4
5 // 10秒钟以后自动解锁
6 // 无需调用unlock方法手动解锁
7 fairLock.lock(10, TimeUnit.SECONDS);
8
9 // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
10 boolean res = fairLock.tryLock(100, 10, TimeUnit.SECONDS);
11 fairLock.unlock();
```

3.9.3. 联锁 (MultiLock)

基于Redis的Redisson分布式联锁 [RedissonMultiLock](#) 对象可以将多个 `RLock` 对象关联为一个联锁，每个 `RLock` 对象实例可以来自于不同的Redisson实例。

```
1 RLock lock1 = redissonInstance1.getLock("lock1");
2 RLock lock2 = redissonInstance2.getLock("lock2");
3 RLock lock3 = redissonInstance3.getLock("lock3");
4
5 RedissonMultiLock lock = new RedissonMultiLock(lock1, lock2, lock3);
6 // 同时加锁: lock1 lock2 lock3
7 // 所有的锁都上锁成功才算成功。
8 lock.lock();
9 ...
10 lock.unlock();
```

3.9.4. 红锁 (RedLock)

基于Redis的Redisson红锁 [RedissonRedLock](#) 对象实现了[Redlock](#)介绍的加锁算法。该对象也可以用来将多个 `RLock` 对象关联为一个红锁，每个 `RLock` 对象实例可以来自于不同的Redisson实例。

```

1 RLock lock1 = redissonInstance1.getLock("lock1");
2 RLock lock2 = redissonInstance2.getLock("lock2");
3 RLock lock3 = redissonInstance3.getLock("lock3");
4
5 RedissonRedLock lock = new RedissonRedLock(lock1, lock2, lock3);
6 // 同时加锁: lock1 lock2 lock3
7 // 红锁在大部分节点上加锁成功就算成功。
8 lock.lock();
9 ...
10 lock.unlock();

```

3.9.5. 读写锁 (ReadWriteLock)

基于Redis的Redisson分布式可重入读写锁 [RReadWriteLock](#) Java对象实现了

`java.util.concurrent.locks.ReadWriteLock` 接口。其中读锁和写锁都继承了[RLock](#)接口。

分布式可重入读写锁允许同时有多个读锁和一个写锁处于加锁状态。

```

1 RReadWriteLock rwlock = redisson.getReadWriteLock("anyRWLock");
2 // 最常见的使用方法
3 rwlock.readLock().lock();
4 // 或
5 rwlock.writeLock().lock();
6
7 // 10秒钟以后自动解锁
8 // 无需调用unlock方法手动解锁
9 rwlock.readLock().lock(10, TimeUnit.SECONDS);
10 // 或
11 rwlock.writeLock().lock(10, TimeUnit.SECONDS);
12
13 // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
14 boolean res = rwlock.readLock().tryLock(100, 10, TimeUnit.SECONDS);
15 // 或
16 boolean res = rwlock.writeLock().tryLock(100, 10, TimeUnit.SECONDS);
17 ...
18 lock.unlock();

```

添加StockController方法:

```

1 @GetMapping("test/read")
2 public String testRead(){
3     String msg = stockService.testRead();
4
5     return "测试读";
6 }
7
8 @GetMapping("test/write")
9 public String testWrite(){
10    String msg = stockService.testWrite();
11
12    return "测试写";
13 }

```

添加StockService方法：

```
1 public String testRead() {
2     RReadWriteLock rwLock = this.redissonClient.getReadWriteLock("rwLock");
3     rwLock.readLock().lock(10, TimeUnit.SECONDS);
4
5     System.out.println("测试读锁。。。");
6     // rwLock.readLock().unlock();
7
8     return null;
9 }
10
11 public String testWrite() {
12     RReadWriteLock rwLock = this.redissonClient.getReadWriteLock("rwLock");
13     rwLock.writeLock().lock(10, TimeUnit.SECONDS);
14
15     System.out.println("测试写锁。。。");
16     // rwLock.writeLock().unlock();
17
18     return null;
19 }
```

打开两个浏览器窗口测试：

- 同时访问写：一个写完之后，等待一会儿（约10s），另一个写开始
- 同时访问读：不用等待
- 先写后读：读要等待（约10s）写完成
- 先读后写：写要等待（约10s）读完成

3.9.6. 信号量 (Semaphore)

基于Redis的Redisson的分布式信号量 ([Semaphore](#)) Java对象 RSemaphore 采用了与 `java.util.concurrent.Semaphore` 相似的接口和用法。同时还提供了[异步 \(Async\)](#)、[反射式 \(Reactive\)](#) 和[RxJava2标准](#)的接口。

```
1 RSemaphore semaphore = redisson.getSemaphore("semaphore");
2 semaphore.acquire();
3 semaphore.release();
```

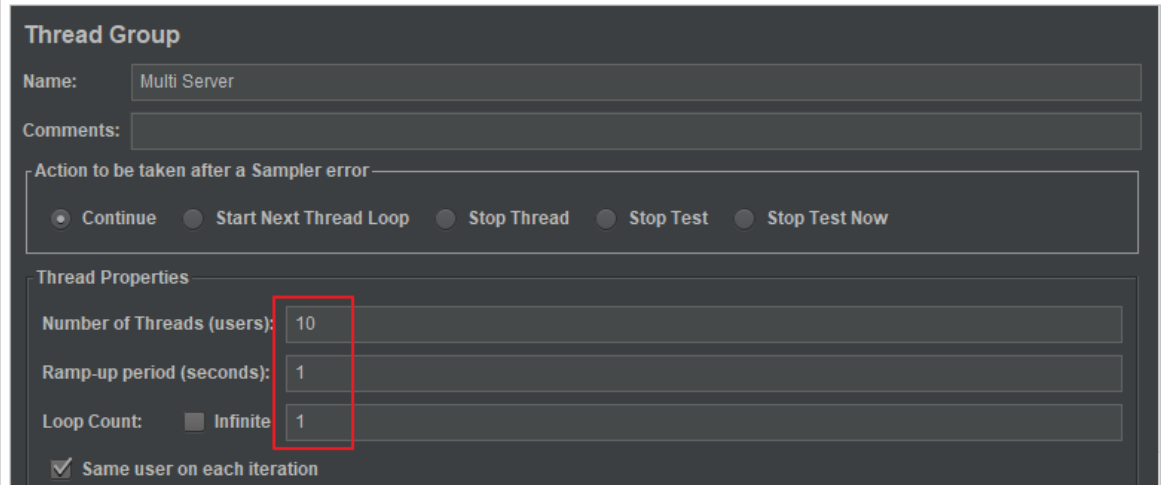
在StockController添加方法：

```
1 @GetMapping("test/semaphore")
2 public String testSemaphore(){
3     this.stockService.testSemaphore();
4
5     return "测试信号量";
6 }
```

在StockService添加方法：

```
1 public void testSemaphore() {
2     RSemaphore semaphore = this.redissonClient.getSemaphore("semaphore");
3     semaphore.trySetPermits(3);
4     try {
5         semaphore.acquire();
6
7         TimeUnit.SECONDS.sleep(5);
8         System.out.println(System.currentTimeMillis());
9
10        semaphore.release();
11    } catch (InterruptedException e) {
12        e.printStackTrace();
13    }
14 }
```

添加测试用例：并发10次，循环一次



Thread Group

Name: Multi Server

Comments:

Action to be taken after a Sampler error

☒ Continue ☐ Start Next Thread Loop ☐ Stop Thread ☐ Stop Test ☐ Stop Test Now

Thread Properties

Number of Threads (users): 10

Ramp-up period (seconds): 1

Loop Count: ☐ Infinite 1

☒ Same user on each iteration

控制台效果：

```
1 控制台1:
2 1606960790234
3 1606960800337
4 1606960800443
5 1606960805248
6
7 控制台2:
8 1606960790328
9 1606960795332
10 1606960800245
11
12 控制台3:
13 1606960790433
14 1606960795238
15 1606960795437
```

由此可知：

1606960790秒有3次请求进来：每个控制台各1次

1606960795秒有3次请求进来：控制台2有1次，控制台3有2次

1606960800秒有3次请求进来：控制台1有2次，控制台2有1次

1606960805秒有1次请求进来：控制台1有1次

3.9.7. 闭锁 (CountDownLatch)

基于Redisson的Redisson分布式闭锁 ([CountDownLatch](#)) Java对象 `RCountDownLatch` 采用了与 `java.util.concurrent.CountDownLatch` 相似的接口和用法。

```
1 RCountDownLatch latch = redisson.getCountDownLatch("anyCountDownLatch");
2 latch.trySetCount(1);
3 latch.await();
4
5 // 在其他线程或其他JVM里
6 RCountDownLatch latch = redisson.getCountDownLatch("anyCountDownLatch");
7 latch.countDown();
```

需要两个方法：一个等待，一个计数countDown

给StockController添加测试方法：

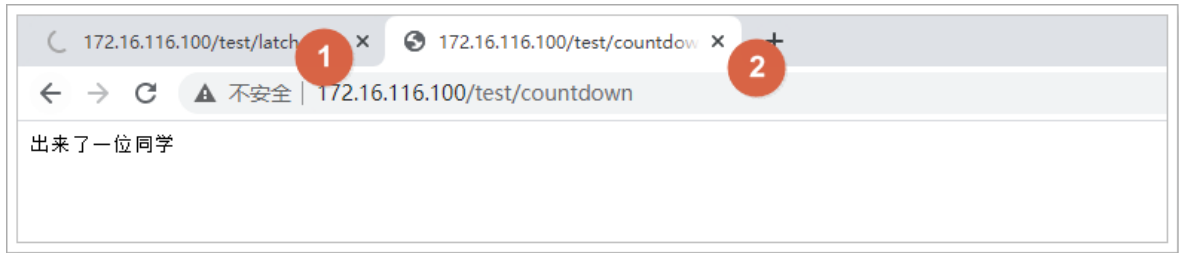
```
1 @GetMapping("test/latch")
2 public String testLatch(){
3     this.stockService.testLatch();
4
5     return "班长锁门。。。";
6 }
7
8 @GetMapping("test/countdown")
9 public String testCountDown(){
10     this.stockService.testCountDown();
11
12     return "出来了一位同学";
13 }
```

给StockService添加测试方法：

```
1 public void testLatch() {
2     RCountDownLatch latch = this.redissonClient.getCountDownLatch("latch");
3     latch.trySetCount(6);
4
5     try {
6         latch.await();
7     } catch (InterruptedException e) {
8         e.printStackTrace();
9     }
```

```
10 }
11
12 public void testCountDown() {
13     RCountDownLatch latch = this.redissonClient.getCountDownLatch("latch");
14     latch.trySetCount(6);
15
16     latch.countDown();
17 }
```

重启测试，打开两个页面：当第二个请求执行6次之后，第一个请求才会执行。



4. 基于zookeeper实现分布式锁

实现分布式锁目前有三种流行方案，分别为基于数据库、Redis、Zookeeper的方案。这里主要介绍基于zk怎么实现分布式锁。在实现分布式锁之前，先回顾zookeeper的知识点

4.1. 知识点回顾

Zookeeper（业界简称zk）是一种提供配置管理、分布式协同以及命名的中心化服务，这些提供的功能都是分布式系统中非常底层且必不可少的基本功能，但是如果自己实现这些功能而且要达到高吞吐、低延迟同时还要保持一致性和可用性，实际上非常困难。因此zookeeper提供了这些功能，开发者在zookeeper之上构建自己的各种分布式系统。

4.1.1. 相关概念

Zookeeper提供一个多层级的节点命名空间（节点称为znode），每个节点都用一个以斜杠（/）分隔的路径表示，而且每个节点都有父节点（根节点除外），非常类似于文件系统。并且每个节点都是唯一的。

znode节点有四种类型：

- **PERSISTENT**：永久节点。客户端与zookeeper断开连接后，该节点依旧存在
- **EPHEMERAL**：临时节点。客户端与zookeeper断开连接后，该节点被删除
- **PERSISTENT_SEQUENTIAL**：永久节点、序列化。客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号
- **EPHEMERAL_SEQUENTIAL**：临时节点、序列化。客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

创建这四种节点：

```

[zookeeper]
[zk: localhost:2181(CONNECTED) 32] create /aa "test" 创建持久节点
Created /aa
[zk: localhost:2181(CONNECTED) 33] create -s /bb "test" 持久序列化节点
Created /bb000000000006
[zk: localhost:2181(CONNECTED) 34] create -e /cc "test" 临时节点
Created /cc
[zk: localhost:2181(CONNECTED) 35] create -e -s /dd "test" 临时序列化节点
Created /dd000000000008
[zk: localhost:2181(CONNECTED) 36] stat /
cZxid = 0x0
ctime = Thu Jan 01 08:00:00 CST 1970
mZxid = 0x0
mtime = Thu Jan 01 08:00:00 CST 1970
pZxid = 0x1d
cversion = 13
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 0
numChildren = 5
[zk: localhost:2181(CONNECTED) 37]

```

事件监听：在读取数据时，我们可以同时对节点设置事件监听，当节点数据或结构变化时，zookeeper会通知客户端。当前zookeeper有如下四种事件：

1. 节点创建
2. 节点删除
3. 节点数据修改
4. 子节点变更

4.1.2. java客户端

1. 引入依赖

```

1 <dependency>
2   <groupId>org.apache.zookeeper</groupId>
3   <artifactId>zookeeper</artifactId>
4   <version>3.4.14</version>
5 </dependency>

```

2. 常用api及其方法

```

1 // 初始化zookeeper客户端类，负责建立与zkserver的会话
2 new ZooKeeper(connectString, 30000, new Watcher() {
3   @Override
4   public void process(WatchedEvent event) {
5     System.out.println("获取链接成功!!");
6   }
7 });
8
9 // 创建一个节点，1-节点路径 2-节点内容 3-访问控制控制 4-节点类型
10 String fullPath = zooKeeper.create(path, null, ZooDefs.Ids.OPEN_ACL_UNSAFE,
    CreateMode.PERSISTENT);
11
12 // 判断一个节点是否存在
13 Stat stat = zooKeeper.exists(rootPath, false);
14 if(stat != null){...}
15
16 // 查询一个节点的内容

```

```

17 Stat stat = new Stat();
18 byte[] data = zooKeeper.getData(path, false, stat);
19
20 // 更新一个节点
21 zooKeeper.setData(rootPath, new byte[] {}, stat.getVersion() + 1);
22
23 // 删除一个节点
24 zooKeeper.delete(path, stat.getVersion());
25
26 // 查询一个节点的子节点列表
27 List<String> children = zooKeeper.getChildren(rootPath, false);
28
29 // 关闭链接
30 if (zooKeeper != null){ zooKeeper.close(); }

```

4.2. 思路分析

分布式锁的步骤：

1. 获取锁：create一个节点
2. 删除锁：delete一个节点
3. 重试：没有获取到锁的请求重试

参照redis分布式锁的特点：

1. 互斥 排他
2. 防死锁：
 1. 可自动释放锁（临时节点）：获得锁之后客户端所在机器宕机了，客户端没有主动删除子节点；如果创建的是永久的节点，那么这个锁永远不会释放，导致死锁；由于创建的是临时节点，客户端宕机后，过了一定时间zookeeper没有收到客户端的心跳包判断会话失效，将临时节点删除从而释放锁。
 2. 可重入锁：借助于ThreadLocal
3. 防误删：宕机自动释放临时节点，不需要设置过期时间，也就不存在误删问题。
4. 加锁/解锁要具备原子性
5. 单点问题：使用Zookeeper可以有效的解决单点问题，ZK一般是集群部署的。
6. 集群问题：zookeeper集群是强一致性的，只要集群中有半数以上的机器存活，就可以对外提供服务。

4.3. 基本实现

实现思路：

1. 多个请求同时添加一个相同的临时节点，只有一个可以添加成功。添加成功的获取到锁
2. 执行业务逻辑
3. 完成业务流程后，删除节点释放锁。

由于zookeeper获取链接是一个耗时过程，这里可以在项目启动时，初始化链接，并且只初始化一次。借助于spring特性，代码实现如下：


```

1  @Component
2  public class zkClient {
3
4      private static final String connectString = "172.16.116.100:2181";
5
6      private static final String ROOT_PATH = "/distributed";
7
8      private ZooKeeper zooKeeper;
9
10     @PostConstruct
11     public void init(){
12         try {
13             // 连接zookeeper服务器
14             this.zooKeeper = new ZooKeeper(connectString, 30000, new
15             Watcher() {
16                 @Override
17                 public void process(WatchedEvent event) {
18                     System.out.println("获取链接成功!!");
19                 }
20             });
21
22             // 创建分布式锁根节点
23             if (this.zooKeeper.exists(ROOT_PATH, false) == null){
24                 this.zooKeeper.create(ROOT_PATH, null,
25                 ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
26             }
27             } catch (Exception e) {
28                 system.out.println("获取链接失败!");
29                 e.printStackTrace();
30             }
31         }
32
33     @PreDestroy
34     public void destroy(){
35         try {
36             if (zooKeeper != null){
37                 zooKeeper.close();
38             }
39             } catch (InterruptedException e) {
40                 e.printStackTrace();
41             }
42         }
43
44     /**
45      * 初始化zk分布式锁对象方法
46      * @param lockName
47      * @return
48      */
49     public ZkDistributedLock getZkDistributedLock(String lockName){
50         return new ZkDistributedLock(zooKeeper, lockName);
51     }
52 }

```

zk分布式锁具体实现：

```

1  public class ZkDistributedLock {
2

```

```

3     private static final String ROOT_PATH = "/distributed";
4
5     private String path;
6
7     private ZooKeeper zooKeeper;
8
9     public ZkDistributedLock(ZooKeeper zooKeeper, String lockName){
10         this.zooKeeper = zooKeeper;
11         this.path = ROOT_PATH + "/" + lockName;
12     }
13
14     public void lock(){
15         try {
16             zooKeeper.create(path, null, ZooDefs.Ids.OPEN_ACL_UNSAFE,
17 CreateMode.EPHEMERAL);
18         } catch (Exception e) {
19             // 重试
20             try {
21                 Thread.sleep(200);
22                 lock();
23             } catch (InterruptedException ex) {
24                 ex.printStackTrace();
25             }
26         }
27
28     public void unlock(){
29         try {
30             this.zooKeeper.delete(path, 0);
31         } catch (InterruptedException e) {
32             e.printStackTrace();
33         } catch (KeeperException e) {
34             e.printStackTrace();
35         }
36     }
37 }

```

改造StockService的checkAndLock方法:

```

1     @Autowired
2     private ZkClient client;
3
4     public void checkAndLock() {
5         // 加锁，获取锁失败重试
6         ZkDistributedLock lock = this.client.getZkDistributedLock("lock");
7         lock.lock();
8
9         // 先查询库存是否充足
10        Stock stock = this.stockMapper.selectById(1L);
11        // 再减库存
12        if (stock != null && stock.getCount() > 0){
13            stock.setCount(stock.getCount() - 1);
14            this.stockMapper.updateById(stock);
15        }
16    }

```

```

17 // 释放锁
18 lock.unlock();
19 }

```

Jmeter压力测试：

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	3304	49	11210	18764	35742	14	63383	0.00%	25.7/sec	4.76	3.23
TOTAL	5000	3304	49	11210	18764	35742	14	63383	0.00%	25.7/sec	4.76	3.23

性能一般，mysql数据库的库存余量为0（注意：所有测试之前都要先修改库存量为5000）

基本实现存在的问题：

1. 性能一般（比mysql略好）
2. 不可重入

接下来首先来提高性能

4.4. 优化：性能优化

基本实现中由于无限自旋影响性能：

```

16 public void lock(){
17     try {
18         zooKeeper.create(path, data: null, ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL);
19     } catch (Exception e) {
20         // 重试
21         try {
22             Thread.sleep( millis: 200);
23             lock(); 只要没有获取到锁无限自旋，争抢资源影响性能
24         } catch (InterruptedException ex) {
25             ex.printStackTrace();
26         }
27     }
28 }

```

试想：每个请求要想正常的执行完成，最终都是要创建节点，如果能够避免争抢必然可以提高性能。

这里借助于zk的临时序列化节点，实现分布式锁：

```

[zk: localhost:2181(CONNECTED) 1] create /test "xx"
Created /test
[zk: localhost:2181(CONNECTED) 2] create -e -s /test/lock- "xx"
Created /test/lock-0000000000
[zk: localhost:2181(CONNECTED) 3] create -e -s /test/lock- "xx"
Created /test/lock-0000000001
[zk: localhost:2181(CONNECTED) 4] create -e -s /test/lock- "xx"
Created /test/lock-0000000002
[zk: localhost:2181(CONNECTED) 5] create -e -s /test/lock- "xx"
Created /test/lock-0000000003
[zk: localhost:2181(CONNECTED) 6] create -e -s /test/lock- "xx"
Created /test/lock-0000000004
[zk: localhost:2181(CONNECTED) 7] ls /test 每个请求直接创建临时序列化节点，序号最小的获取锁
[lock-0000000001, lock-0000000000, lock-0000000004, lock-0000000003, lock-0000000002]
[zk: localhost:2181(CONNECTED) 8]

```

4.4.1. 实现阻塞锁

代码实现:

```
1 public class ZkDistributedLock {
2
3     private static final String ROOT_PATH = "/distributed";
4
5     private String path;
6
7     private ZooKeeper zooKeeper;
8
9     public ZkDistributedLock(ZooKeeper zooKeeper, String lockName){
10         try {
11             this.zooKeeper = zooKeeper;
12             this.path = zooKeeper.create(ROOT_PATH + "/" + lockName + "-",
13 null, ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);
14         } catch (KeeperException e) {
15             e.printStackTrace();
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19     }
20
21     public void lock(){
22         String preNode = getPreNode(path);
23         // 如果该节点没有前一个节点,说明该节点时最小节点,放行执行业务逻辑
24         if (StringUtils.isEmpty(preNode)){
25             return ;
26         }
27         // 重新检查。是否获取到锁
28         try {
29             Thread.sleep(20);
30         } catch (InterruptedException ex) {
31             ex.printStackTrace();
32         }
33         lock();
34     }
35
36     public void unlock(){
37         try {
38             this.zooKeeper.delete(path, 0);
39         } catch (InterruptedException e) {
40             e.printStackTrace();
41         } catch (KeeperException e) {
42             e.printStackTrace();
43         }
44     }
45
46     /**
47      * 获取指定节点的前节点
48      * @param path
49      * @return
50      */
51     private String getPreNode(String path){
52         try {
53             // 获取当前节点的序列化号
```

```

54         Long curSerial =
Long.valueOf(StringUtils.substringAfterLast(path, "-"));
55         // 获取根路径下的所有序列化子节点
56         List<String> nodes = this.zooKeeper.getChildren(ROOT_PATH,
false);
57
58         // 判空
59         if (CollectionUtils.isEmpty(nodes)){
60             return null;
61         }
62
63         // 获取前一个节点
64         Long flag = 0L;
65         String preNode = null;
66         for (String node : nodes) {
67             // 获取每个节点的序列化号
68             Long serial =
Long.valueOf(StringUtils.substringAfterLast(node, "-"));
69             if (serial < curSerial && serial > flag){
70                 flag = serial;
71                 preNode = node;
72             }
73         }
74
75         return preNode;
76     } catch (KeeperException e) {
77         e.printStackTrace();
78     } catch (InterruptedException e) {
79         e.printStackTrace();
80     }
81     return null;
82 }
83 }

```

主要修改了构造方法和lock方法：

```

17 @ public ZkDistributedLock(ZooKeeper zooKeeper, String lockName){
18     try {
19         this.zooKeeper = zooKeeper;    创建临时序列化节点
20         this.path = zooKeeper.create( path: ROOT_PATH + "/" + lockName);
21     } catch (KeeperException e) {
22         e.printStackTrace();
23     } catch (InterruptedException e) {
24         e.printStackTrace();
25     }
26 }
27
28 public void lock(){
29     String preNode = getPreNode(path);
30     // 如果该节点没有前一个节点，说明该节点时最小节点，放行执行业务逻辑
31     if (StringUtils.isEmpty(preNode)) {
32         return ;
33     }
34     // 重新检查。是否获取到锁
35     lock();
36 }

```

并添加了getPreNode获取前置节点的方法。

测试结果如下：

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	5156	5140	5617	5792	6519	374	6796	0.00%	19.2/sec	3.57	2.42
TOTAL	5000	5156	5140	5617	5792	6519	374	6796	0.00%	19.2/sec	3.57	2.42

性能反而更弱了。

原因：虽然不用反复争抢创建节点了，但是会自选判断自己是最小的节点，这个判断逻辑反而更复杂更耗时。

解决方案：监听。

4.4.2. 监听实现阻塞锁

对于这个算法有个极大的优化点：假如当前有1000个节点在等待锁，如果获得锁的客户端释放锁时，这1000个客户端都会被唤醒，这种情况称为“羊群效应”；在这种羊群效应中，zookeeper需要通知1000个客户端，这会阻塞其他的操作，最好的情况应该只唤醒新的最小节点对应的客户端。应该怎么做呢？在设置事件监听时，每个客户端应该对刚好在它之前的子节点设置事件监听，例如子节点列表为/lock/lock-0000000000、/lock/lock-0000000001、/lock/lock-0000000002，序号为1的客户端监听序号为0的子节点删除消息，序号为2的监听序号为1的子节点删除消息。

所以调整后的分布式锁算法流程如下：

- 客户端连接zookeeper，并在/lock下创建临时的且有序的子节点，第一个客户端对应的子节点为/lock/lock-0000000000，第二个为/lock/lock-0000000001，以此类推；
- 客户端获取/lock下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听刚好在自己之前一位的子节点删除消息，获得子节点变更通知后重复此步骤直至获得锁；
- 执行业务代码；
- 完成业务流程后，删除对应的子节点释放锁。

改造ZkDistributedLock的lock方法：

```
1 public void lock(){
2     try {
3         String preNode = getPreNode(path);
4         // 如果该节点没有前一个节点，说明该节点时最小节点，放行执行业务逻辑
5         if (StringUtils.isEmpty(preNode)){
6             return ;
7         } else {
8             CountdownLatch countDownLatch = new CountdownLatch(1);
9             if (this.zookeeper.exists(ROOT_PATH + "/" + preNode, new
10 watcher()){
11                 @Override
12                 public void process(WatchedEvent event) {
13                     countDownLatch.countDown();
14                 }
15             } == null) {
16                 return;
17             }
18             // 阻塞。。。
19             countDownLatch.await();
20             return;
21         }
22     } catch (Exception e) {
23         e.printStackTrace();
24         // 重新检查。是否获取到锁
25         try {
26             Thread.sleep(200);
27         } catch (InterruptedException ex) {
28             ex.printStackTrace();
29         }
30         lock();
31     }
```

压力测试效果如下：

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	2475	2344	3050	3514	3750	268	3942	0.00%	39.8/sec	7.38	5.01
TOTAL	5000	2475	2344	3050	3514	3750	268	3942	0.00%	39.8/sec	7.38	5.01

由此可见性能提高不少仅次于redis的分布式锁

4.5. 优化：可重入锁

引入ThreadLocal线程局部变量保证zk分布式锁的可重入性。

```
1 public class ZkDistributedLock {
2
3     private static final String ROOT_PATH = "/distributed";
```

```

4     private static final ThreadLocal<Integer> THREAD_LOCAL = new
      ThreadLocal<>();
5
6     private String path;
7
8     private ZooKeeper zooKeeper;
9
10    public ZkDistributedLock(ZooKeeper zooKeeper, String lockName){
11        try {
12            this.zooKeeper = zooKeeper;
13            if (THREAD_LOCAL.get() == null || THREAD_LOCAL.get() == 0){
14                this.path = zooKeeper.create(ROOT_PATH + "/" + lockName +
15                "-", null, ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);
16            }
17        } catch (KeeperException e) {
18            e.printStackTrace();
19        } catch (InterruptedException e) {
20            e.printStackTrace();
21        }
22    }
23
24    public void lock(){
25        Integer flag = THREAD_LOCAL.get();
26        if (flag != null && flag > 0) {
27            THREAD_LOCAL.set(flag + 1);
28            return;
29        }
30        try {
31            String preNode = getPreNode(path);
32            // 如果该节点没有前一个节点，说明该节点时最小节点，放行执行业务逻辑
33            if (StringUtils.isEmpty(preNode)){
34                THREAD_LOCAL.set(1);
35                return ;
36            } else {
37                CountDownLatch countDownLatch = new CountDownLatch(1);
38                if (this.zooKeeper.exists(ROOT_PATH + "/" + preNode, new
39                watcher()){
40
41                    @Override
42                    public void process(WatchedEvent event) {
43                        countDownLatch.countDown();
44                    }
45                }) == null) {
46                    THREAD_LOCAL.set(1);
47                    return;
48                }
49                // 阻塞。。。
50                countDownLatch.await();
51                THREAD_LOCAL.set(1);
52                return;
53            }
54        } catch (Exception e) {
55            e.printStackTrace();
56            // 重新检查。是否获取到锁
57            try {
58                Thread.sleep(200);
59            } catch (InterruptedException ex) {
60                ex.printStackTrace();
61            }
62        }
63    }

```



```

59         lock();
60     }
61 }
62
63 public void unlock(){
64     try {
65         THREAD_LOCAL.set(THREAD_LOCAL.get() - 1);
66         if (THREAD_LOCAL.get() == 0) {
67             this.zookeeper.delete(path, 0);
68             THREAD_LOCAL.remove();
69         }
70     } catch (InterruptedException e) {
71         e.printStackTrace();
72     } catch (KeeperException e) {
73         e.printStackTrace();
74     }
75 }
76
77 /**
78  * 获取指定节点的前节点
79  * @param path
80  * @return
81  */
82 private String getPreNode(String path){
83
84     try {
85         // 获取当前节点的序列化号
86         Long curSerial =
87 Long.valueOf(StringUtils.substringAfterLast(path, "-"));
88         // 获取根路径下的所有序列化子节点
89         List<String> nodes = this.zookeeper.getChildren(ROOT_PATH,
90 false);
91
92         // 判空
93         if (CollectionUtils.isEmpty(nodes)){
94             return null;
95         }
96
97         // 获取前一个节点
98         Long flag = 0L;
99         String preNode = null;
100         for (String node : nodes) {
101             // 获取每个节点的序列化号
102             Long serial =
103 Long.valueOf(StringUtils.substringAfterLast(node, "-"));
104             if (serial < curSerial && serial > flag){
105                 flag = serial;
106                 preNode = node;
107             }
108         }
109
110         return preNode;
111     } catch (KeeperException e) {
112         e.printStackTrace();
113     } catch (InterruptedException e) {
114         e.printStackTrace();
115     }
116     return null;
117 }

```

```
114     }  
115 }
```

4.6. zk分布式锁小结

参照redis分布式锁的特点：

1. 互斥 排他：zk节点的不可重复性，以及序列化节点的有序性
2. 防死锁：
 1. 可自动释放锁：临时节点
 2. 可重入锁：借助于ThreadLocal
3. 防误删：临时节点
4. 加锁/解锁要具备原子性
5. 单点问题：使用Zookeeper可以有效的解决单点问题，ZK一般是集群部署的。
6. 集群问题：zookeeper集群是强一致性的，只要集群中有半数以上的机器存活，就可以对外提供服务。
7. 公平锁：有序性节点

4.6. Curator中的分布式锁

Curator是netflix公司开源的一套zookeeper客户端，目前是Apache的顶级项目。与Zookeeper提供的原生客户端相比，Curator的抽象层次更高，简化了Zookeeper客户端的开发量。Curator解决了很多zookeeper客户端非常底层的细节开发工作，包括连接重连、反复注册wathcer和NodeExistsException 异常等。

通过查看官方文档，可以发现Curator主要解决了三类问题：

- 封装ZooKeeper client与ZooKeeper server之间的连接处理
- 提供了一套Fluent风格的操作API
- 提供ZooKeeper各种应用场景(recipe，比如：分布式锁服务、集群领导选举、共享计数器、缓存机制、分布式队列等)的抽象封装，这些实现都遵循了zk的最佳实践，并考虑了各种极端情况

Curator由一系列的模块构成，对于一般开发者而言，常用的是curator-framework和curator-recipes：

- curator-framework：提供了常见的zk相关的底层操作
- curator-recipes：提供了一些zk的典型使用场景的参考。本节重点关注的分布式锁就是该包提供的

引入依赖：

最新版本的curator 4.3.0支持zookeeper 3.4.x和3.5，但是需要注意curator传递进来的依赖，需要和实际服务器端使用的版本相符，以我们目前使用的zookeeper 3.4.14为例。

```
1 <dependency>  
2     <groupId>org.apache.curator</groupId>  
3     <artifactId>curator-framework</artifactId>  
4     <version>4.3.0</version>
```

```

5      <exclusions>
6          <exclusion>
7              <groupId>org.apache.zookeeper</groupId>
8              <artifactId>zookeeper</artifactId>
9          </exclusion>
10     </exclusions>
11 </dependency>
12 <dependency>
13     <groupId>org.apache.curator</groupId>
14     <artifactId>curator-recipes</artifactId>
15     <version>4.3.0</version>
16     <exclusions>
17         <exclusion>
18             <groupId>org.apache.zookeeper</groupId>
19             <artifactId>zookeeper</artifactId>
20         </exclusion>
21     </exclusions>
22 </dependency>
23 <dependency>
24     <groupId>org.apache.zookeeper</groupId>
25     <artifactId>zookeeper</artifactId>
26     <version>3.4.14</version>
27 </dependency>

```

4.6.1. 可重入锁InterProcessMutex

Reentrant和JDK的ReentrantLock类似，意味着同一个客户端在拥有锁的同时，可以多次获取，不会被阻塞。它是由类InterProcessMutex来实现。

```

1  // 常用构造方法
2  public InterProcessMutex(CuratorFramework client, String path)
3  // 获取锁
4  public void acquire();
5  // 带超时时间的可重入锁
6  public boolean acquire(long time, TimeUnit unit);
7  // 释放锁
8  public void release();

```

添加curator客户端配置：

```

1  @Configuration
2  public class ZkCuratorConfig {
3
4      @Bean
5      public CuratorFramework curatorFramework(){
6          // 后台重试，每个1000ms重试一次，重试3次
7          RetryPolicy retry = new ExponentialBackoffRetry(1000, 3);
8          // 初始化CuratorFramework客户端，如果有多个zk地址，以逗号分割。
9          CuratorFramework client =
10 CuratorFrameworkFactory.newClient("172.16.116.100:2181", retry);
11      client.start();
12      return client;
13  }

```

改造service测试方法：

```

1  @Autowired
2  private CuratorFramework curatorFramework;
3
4  public void checkAndLock() {
5      try {
6          // 加锁，获取锁失败重试
7          InterProcessMutex mutex = new InterProcessMutex(curatorFramework,
8              "/curator/lock");
9          mutex.acquire();
10
11          // 先查询库存是否充足
12          Stock stock = this.stockMapper.selectById(1L);
13          // 再减库存
14          if (stock != null && stock.getCount() > 0){
15              stock.setCount(stock.getCount() - 1);
16              this.stockMapper.updateById(stock);
17          }
18
19          // 释放锁
20          mutex.release();
21      } catch (Exception e) {
22          e.printStackTrace();
23      }

```

注意：如想重入，则需要使用同一个InterProcessMutex对象。

压力测试结果：

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Received...	Sent KB/...
HTTP R...	5000	2693	2570	3189	3494	4197	386	4601	0.00%	36.6/sec	6.79	4.61
TOTAL	5000	2693	2570	3189	3494	4197	386	4601	0.00%	36.6/sec	6.79	4.61

4.6.2. 不可重入锁InterProcessSemaphoreMutex

具体实现：InterProcessSemaphoreMutex。与InterProcessMutex调用方法类似，区别在于该锁是不可重入的，在同一个线程中不可重入。

```
1 public InterProcessSemaphoreMutex(CuratorFramework client, String path);
2 public void acquire();
3 public boolean acquire(long time, TimeUnit unit);
4 public void release();
```

4.6.3. 可重入读写锁InterProcessReadWriteLock

类似JDK的ReentrantReadWriteLock。一个拥有写锁的线程可重入读锁，但是读锁却不能进入写锁。这也意味着写锁可以降级成读锁。从读锁升级成写锁是不成的。主要实现类InterProcessReadWriteLock：

```
1 // 构造方法
2 public InterProcessReadWriteLock(CuratorFramework client, String basePath);
3 // 获取读锁对象
4 InterProcessMutex readLock();
5 // 获取写锁对象
6 InterProcessMutex writeLock();
```

4.6.4. 联锁InterProcessMultiLock

Multi Shared Lock是一个锁的容器。当调用acquire，所有的锁都会被acquire，如果请求失败，所有的锁都会被release。同样调用release时所有的锁都被release(失败被忽略)。基本上，它就是组锁的代表，在它上面的请求释放操作都会传递给它包含的所有的锁。实现类InterProcessMultiLock：

```
1 // 构造函数需要包含的锁的集合，或者一组Zookeeper的path
2 public InterProcessMultiLock(List<InterProcessLock> locks);
3 public InterProcessMultiLock(CuratorFramework client, List<String> paths);
4
5 // 获取锁
6 public void acquire();
7 public boolean acquire(long time, TimeUnit unit);
8
9 // 释放锁
10 public synchronized void release();
```

4.6.5. 信号量InterProcessSemaphoreV2

一个计数的信号量类似JDK的Semaphore。JDK中Semaphore维护的一组许可(permits)，而Curator中称之为租约(Lease)。注意，所有的实例必须使用相同的numberOfLeases值。调用acquire会返回一个租约对象。客户端必须在finally中close这些租约对象，否则这些租约会丢失掉。但是，如果客户端session由于某种原因比如crash丢掉，那么这些客户端持有的租约会自动close，这样其它客户端可以继续使用这些租约。主要实现类InterProcessSemaphoreV2：

```

1 // 构造方法
2 public InterProcessSemaphoreV2(CuratorFramework client, String path, int
  maxLeases);
3
4 // 注意一次你可以请求多个租约，如果Semaphore当前的租约不够，则请求线程会被阻塞。
5 // 同时还提供了超时的重载方法
6 public Lease acquire();
7 public Collection<Lease> acquire(int qty);
8 public Lease acquire(long time, TimeUnit unit);
9 public Collection<Lease> acquire(int qty, long time, TimeUnit unit)
10
11 // 租约还可以通过下面的方式返还
12 public void returnAll(Collection<Lease> leases);
13 public void returnLease(Lease lease);

```

4.6.6. 栅栏barrier

1. **DistributedBarrier**构造函数中barrierPath参数用来确定一个栅栏，只要barrierPath参数相同(路径相同)就是同一个栅栏。通常情况下栅栏的使用如下：

1. 主client设置一个栅栏
2. 其他客户端就会调用waitOnBarrier()等待栅栏移除，程序处理线程阻塞
3. 主client移除栅栏，其他客户端的处理程序就会同时继续运行。

DistributedBarrier类的主要方法如下：

```

1 setBarrier() - 设置栅栏
2 waitOnBarrier() - 等待栅栏移除
3 removeBarrier() - 移除栅栏

```

2. **DistributedDoubleBarrier**双栅栏，允许客户端在计算的开始和结束时同步。当足够的进程加入到双栅栏时，进程开始计算，当计算完成时，离开栅栏。DistributedDoubleBarrier实现了双栅栏的功能。构造函数如下：

```

1 // client - the client
2 // barrierPath - path to use
3 // memberQty - the number of members in the barrier
4 public DistributedDoubleBarrier(CuratorFramework client, String
  barrierPath, int memberQty);
5
6 enter()、enter(long maxWait, TimeUnit unit) - 等待同时进入栅栏
7 leave()、leave(long maxWait, TimeUnit unit) - 等待同时离开栅栏

```

memberQty是成员数量，当enter方法被调用时，成员被阻塞，直到所有的成员都调用了enter。当leave方法被调用时，它也阻塞调用线程，直到所有的成员都调用了leave。

注意：参数memberQty的值只是一个阈值，而不是一个限制值。当等待栅栏的数量大于或等于这个值栅栏就会打开！

与栅栏(DistributedBarrier)一样,双栅栏的barrierPath参数也是用来确定是否是同一个栅栏的，双栅栏的使用情况如下：

1. 从多个客户端在同一个路径上创建双栅栏(DistributedDoubleBarrier),然后调用enter()方法, 等待栅栏数量达到memberQty时就可以进入栅栏。
2. 栅栏数量达到memberQty, 多个客户端同时停止阻塞继续运行, 直到执行leave()方法, 等待memberQty个数量的栅栏同时阻塞到leave()方法中。
3. memberQty个数量的栅栏同时阻塞到leave()方法中, 多个客户端的leave()方法停止阻塞, 继续运行。

4.6.7. 倒计时器

利用ZooKeeper可以实现一个集群共享的计数器。只要使用相同的path就可以得到最新的计数器值, 这是由ZooKeeper的一致性保证的。Curator有两个计数器, 一个是用int来计数, 一个用long来计数。

SharedCount

这个类使用int类型来计数。主要涉及三个类。

```
1 * SharedCount
2 * SharedCountReader
3 * SharedCountListener
```

SharedCount代表计数器, 可以为它增加一个SharedCountListener, 当计数器改变时此Listener可以监听到改变的事件, 而SharedCountReader可以读取到最新的值, 包括字面值和带版本信息的值VersionedValue。

DistributedAtomicLong

除了计数的范围比SharedCount大了之外, 它首先尝试使用乐观锁的方式设置计数器, 如果不成功(比如期间计数器已经被其它client更新了), 它使用InterProcessMutex方式来更新计数值。此计数器有一系列的操作:

- get(): 获取当前值
- increment(): 加一
- decrement(): 减一
- add(): 增加特定的值
- subtract(): 减去特定的值
- trySet(): 尝试设置计数值
- forceSet(): 强制设置计数值

你必须检查返回结果的succeeded(), 它代表此操作是否成功。如果操作成功, preValue()代表操作前的值, postValue()代表操作后的值。

5. 总结

实现的复杂性或者难度角度: Zookeeper > 缓存 > 数据库

实际性能角度: 缓存 > Zookeeper > 数据库

可靠性角度: Zookeeper > 缓存 > 数据库