**分布式锁的的应用和探讨**

# 1、场景：

假如我们开发12306火车票票务系统，该系统由多个微服务组成的分布式系统，为了避免火车票的超卖或者一票多卖情况的出现，我们应该怎么解决呢？传统的单体应用单机部署情况下，可以使用java并发处理相关的API（synchronized or lock.lock() lock.unlock()）I进行互斥控制，分布式系统由于多线程、多进程分布在不同的机器上，使单机部署部署情况下的并发控制锁策略失效，为了解决跨JVM互斥机制来控制共享资源的访问，所以分布式锁在当前场景中派上用场。

## 2分布式锁的实现

我们选择基于Redisson对redis 分布式锁的实现，

### 2.1高效的分布式锁

（1）在分布式场景下，需要保证同一时刻只能有一个线程获得锁。

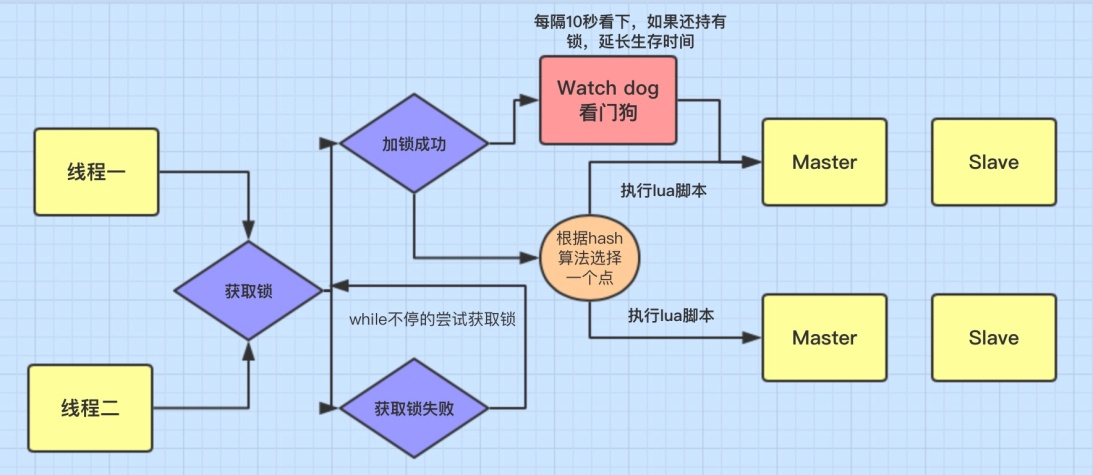
（2）防止死锁：在高并发分布式的条件下，某个线程获得锁，执行任务，没来的及去释放锁，由于系统故障或者其他原因使得他无法执行释放锁的命令，所以非常有必要设置锁的有效时间，确保系统出故障后，在一定时间内可以主动释放锁，避免造成死锁的情况。

（3）性能：锁的颗粒度和范围要小，避免导致大量线程阻塞。

（4）ReentrantLock是可重入锁，那它的特点就是：同一个线程可以重复拿到同一个资源的锁。重入锁非常有利于资源的高效利用

## 2.2 Redisson 原理分析

为了更好的理解分布式锁的原理，我们通过下面这张图来来进行分析。



1. 加锁机制：线程去获取锁，获取成功，执行lua脚本，保存数据到redis数据库中；线程去获取锁失败，一直通过while循环尝试获取锁，获取成功后，执行lua脚本，保存数据到数据库中。
2. Watch dog 自动延期机制：

redissonLock.lock("redisson", 1); //设置一秒的过期时间

redissonLock.release("redisson"); //业务逻辑需要执行大于1秒的时间

**ps：**

**线程1 进来获得锁后，线程一切正常并没有宕机，但它的业务逻辑需要执行2秒，这就会有个问题，在 线程1 执行1秒后，当前锁自动过期，此时线程2进来了，那么就存在线程1和线程2 同时在这段业务逻辑里执行代码，这当然是不合理的。而且如果是这种情况，那么在解锁时系统会抛异常，因为解锁和加锁已经非同一线程。**

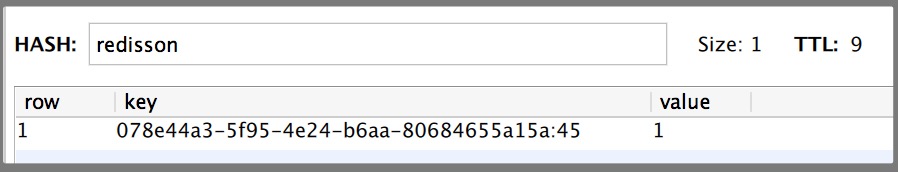
1. 为什么用lua脚本

如果你的业务逻辑复杂的话，通过封装在lua脚本中发送给redis，而且redis是单线程的，这样就保证这段复杂业务逻辑执行的**原子性**。

1. 可重入锁的机制：

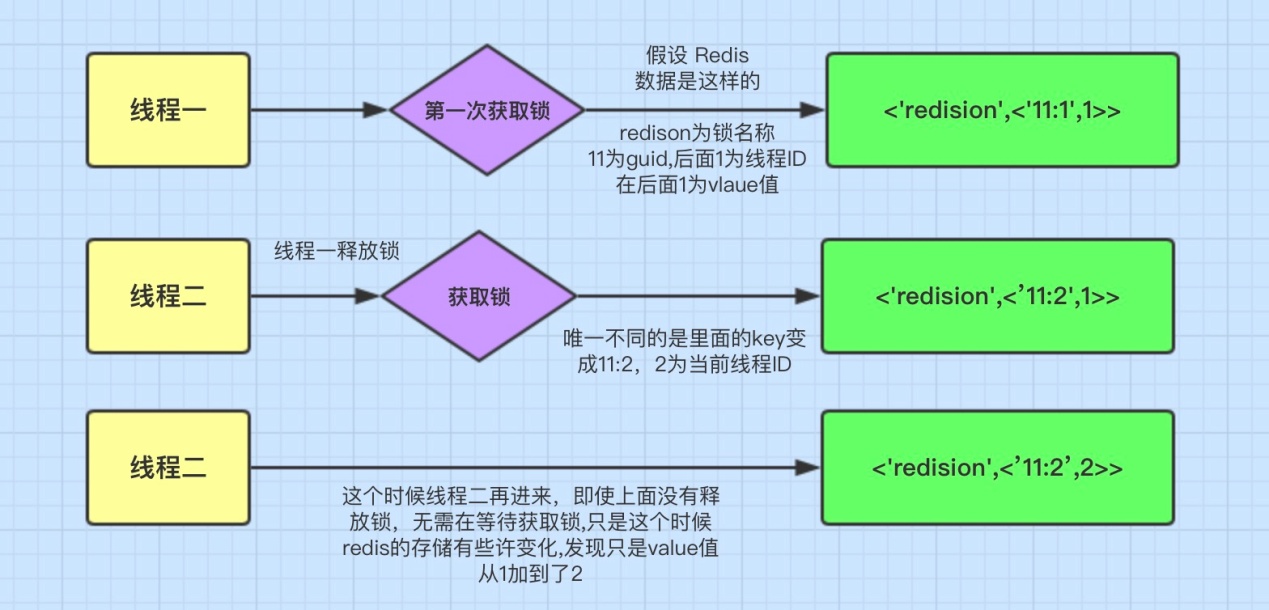
a、Redis存储锁的数据类型是 Hash类型

b、Hash数据类型的key值包含了当前线程信息。



数据类型是Hash类型,Hash类型相当于我们java的 <key,<key1,value>> 类型,这里key是指 'redisson'它的有效期还有9秒，我们再来看里们的key1值为078e44a3-5f95-4e24-b6aa-80684655a15a:45它的组成是:guid + 当前线程的ID。后面的value是就和可重入加锁有关。

如下图：



上面这图的意思就是可重入锁的机制，它最大的优点就是相同线程不需要在等待锁，而是可以直接进行相应操作。

## 2.3分布式锁的落地

### 2.3.1 加锁方式

（1）自定义加锁

@DistributedLock(value="goods", leaseTime=5)

public String lockDecreaseStock(){

//业务逻辑

}

（2）常规加锁

//1、加锁

redissonLock.lock("redisson", 10);

//2、业务逻辑

//3、解锁

redissonLock.unlock("redisson");

### 2.3.2 Redis部署方式

1、单机模式部署

2、集群模式部署

3、主从模式部署

4、哨兵模式部署

### 2.3.3 项目结构

redis-distributed-lock-core # 核心实现

|

---src

|

---com.weimeng.fire.redisson

|# 通过注解方式 实现分布式锁

---annotation

|# 配置类实例化RedissonLock

---config

|# 放置常量信息

---constant

|# 读取application.properties信息后，封装到实体

---entity

|# 支持单机、集群、主从、哨兵 代码实现

---strategy

redis-distributed-lock-web-test # 针对上面实现类的测试类

|

---src

|

---java

|

---com.weimeng.fire.controller

|# 测试 基于注解方式实现分布式锁

---AnnotatinLockController.java

|# 测试 基于常规方式实现分布式锁

---LockController.java

---resources

| # 配置端口号 连接redis信息(如果确定部署类型，那么将连接信息放到core项目中)

---application.properties

### 2.3.4 测试

模拟1秒内100个线程请求接口，来测试结果是否正确。同时测试3种不同的锁:lock锁、trylock锁、注解锁。

1. **lock 锁**

/\*\*

\*

\*

\* 模拟火车票售卖

\*

\* \*/

public static volatile Integer count = 100;

@GetMapping("lock-ticket")

public String lockDecreaseStock() throws InterruptedException {

redissonLock.lock("lock", 10);

if (count > 0) {

}

Thread.sleep(50);

log.info("======售出第===" + count--+”张票”);

}

//如果该线程还持有该锁，那么释放该锁。如果该线程不持有该锁，说明该线程的锁已到过期时间，自动释放锁

if (redissonLock.isHeldByCurrentThread("lock")) {

redissonLock.unlock("lock");

}

return "=================================";

}

没有问题不会出现超卖

**(2) trylock锁**

/\*\*

\* 模拟火车票售卖

\*/

public static volatile Integer count = 10;

@GetMapping("trylock-ticket")

public String trylockDecreaseStock() throws InterruptedException {

if (redissonLock.tryLock("trylock", 10, 5)) {

if (count > 0) {

count--;

}

Thread.sleep(50);

redissonLock.unlock("trylock");

log.info("======售出第===" + count--+”张票”);

} else {

log.info("[ExecutorRedisson]获取锁失败");

}

return "===================================";

}

（3）注解锁

/\*\*

\* 模拟火车票售卖

\*/

public static volatile Integer count= 10;

@GetMapping("annotatin-lock-ticket")

@DistributedLock(value="tickets", leaseTime=5)

public String lockDecreaseStock() throws InterruptedException {

if (count> 0) {

count--;

}

log.info("======售出第===" + count--+”张票”);

return "=================================";

}