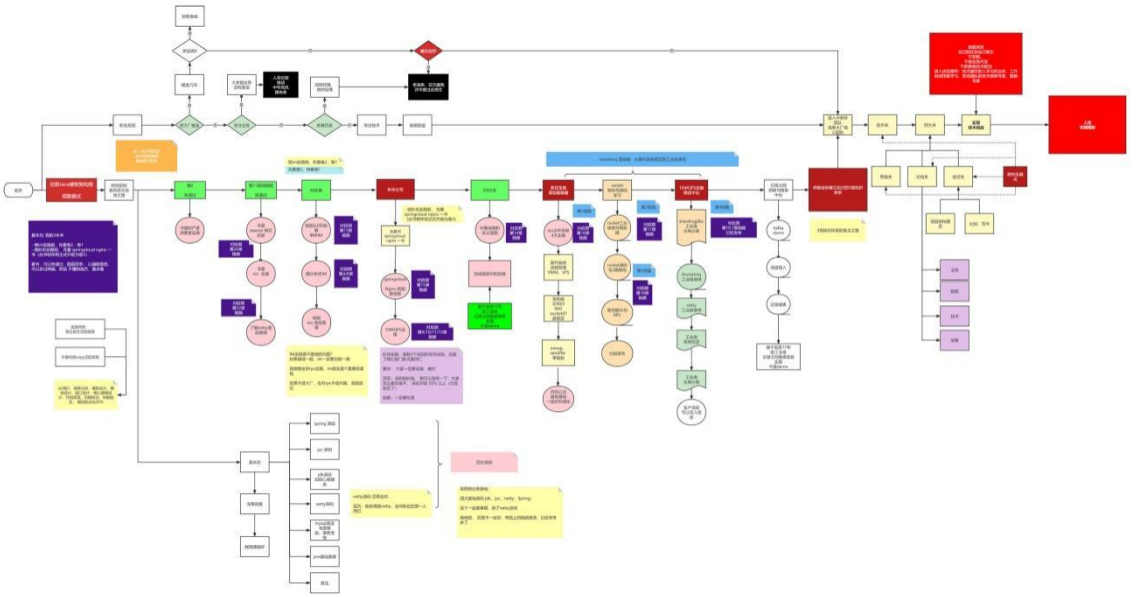
牛逼的职业发展之路

40 岁老架构尼恩用一张图揭秘：Java 工程师的高端职业发展路径，走向食物链顶端的之路

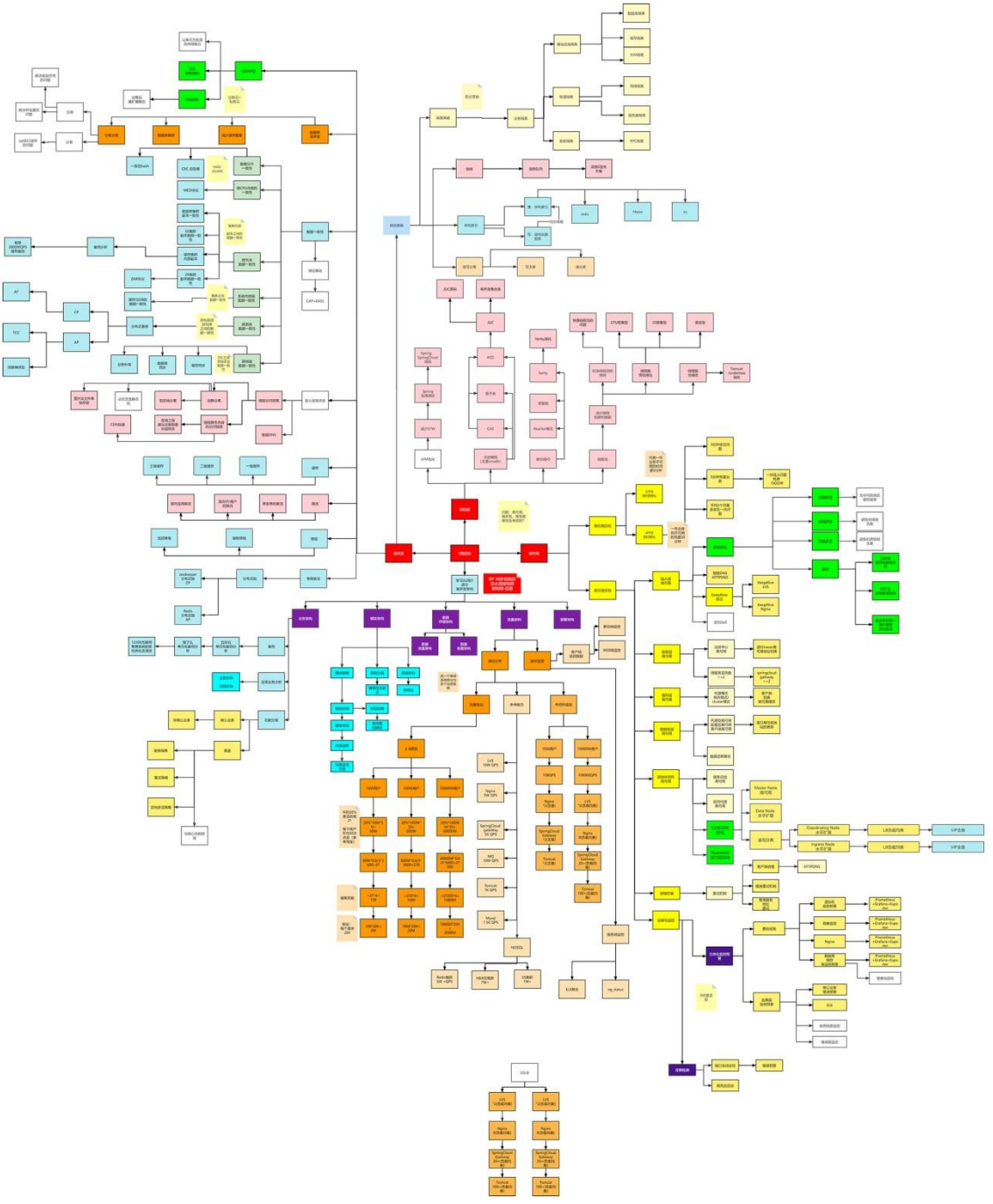
链接：https://www.processon.com/view/link/618a2b62e0b34d73f7eb3cd7



史上最全：价值10W的架构师知识图谱

此图梳理于尼恩的多个 3 高生产项目：多个亿级人民币的大型 SAAS 平台和智慧城市项目

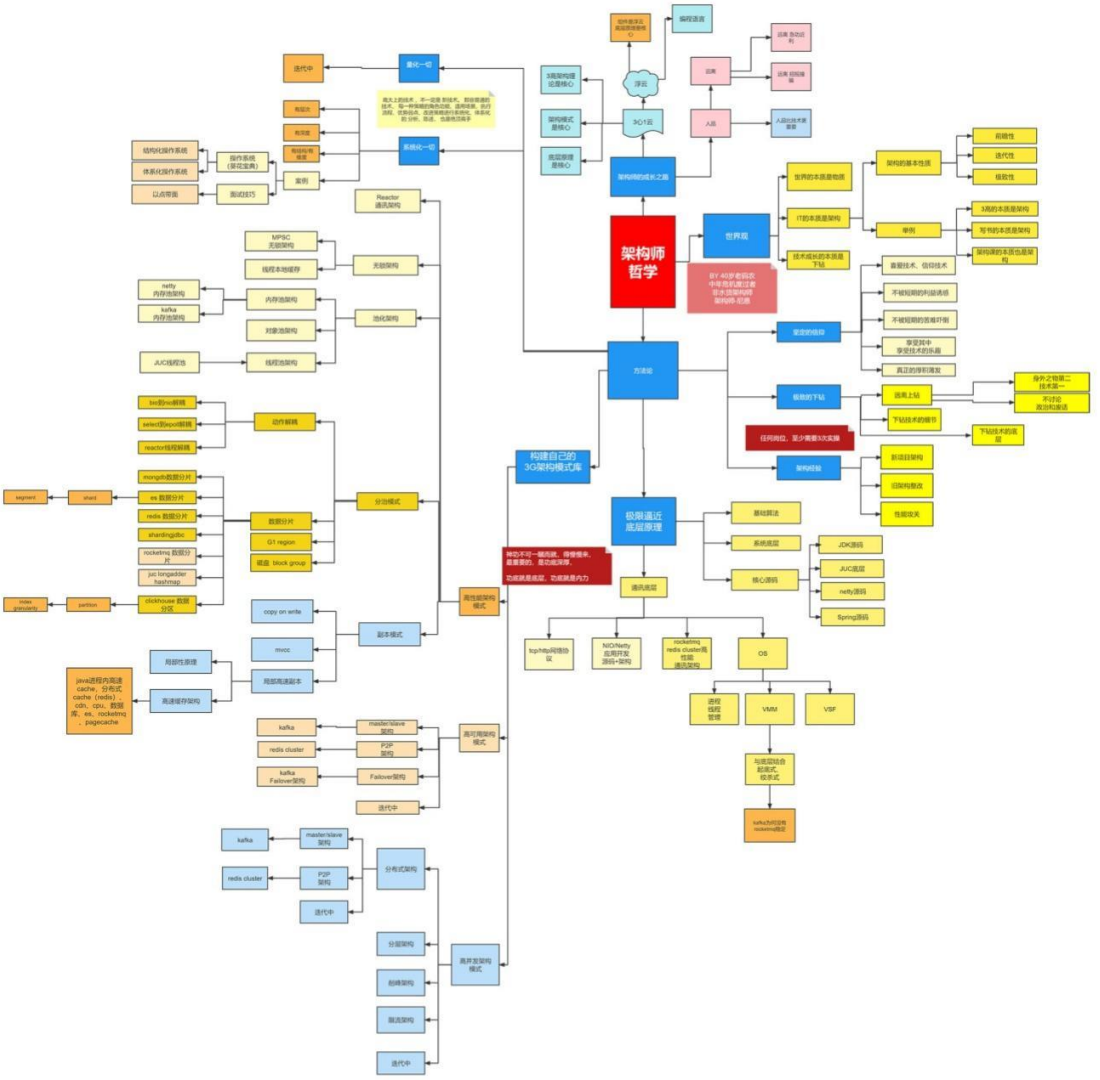
链接：https://www.processon.com/view/link/60fb94216376897l9d246739



牛逼的架构师哲学

40 岁老架构师尼恩对自己的 20 年的开发、架构经验总结

链接：https://www.processon.com/view/link/616f801963768961e9d9aec8



牛逼的3高架构知识宇宙

尼恩 3 高架构知识宇宙，帮助大家穿透 3 高架构，走向技术自由，远离中年危机

链接：https://www.processon.com/view/link/635097d2e0b34d40be778ab4

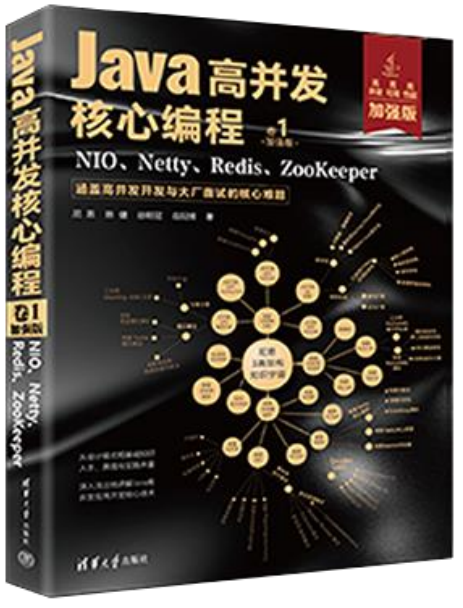
尼恩Java高并发三部曲（卷1加强版）

老版本：《Java 高并发核心编程 卷 1：NIO、Netty、Redis、ZooKeeper》（已经过时，不建议购买）

新版本：《Java 高并发核心编程 卷 1 **加强版**：NIO、Netty、Redis、ZooKeeper》

- 由浅入深地剖析了高并发 IO 的底层原理。
- 图文并茂的介绍了 TCP、HTTP、WebSocket 协议的核心原理。
- 细致深入地揭秘了 Reactor 高性能模式。
- 全面介绍了 Netty 框架，并完成单体 IM、分布式 IM 的实战设计。
- 详尽地介绍了 ZooKeeper、Redis 的使用，以帮助提升高并发、可扩展能力

详情：https://www.cnblogs.com/crazymakercircle/p/16868827.html



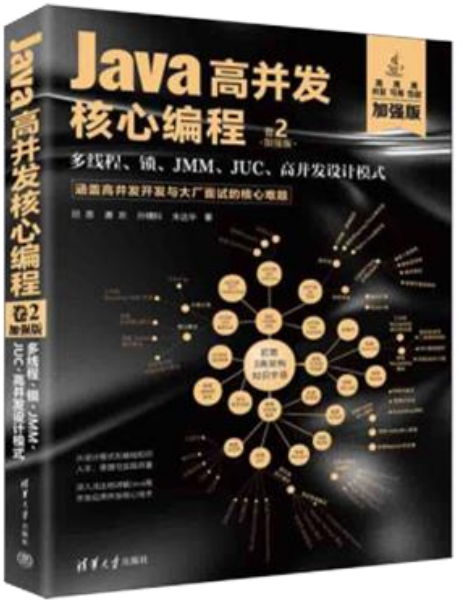
尼恩Java高并发三部曲（卷2加强版）

老版本：《Java 高并发核心编程 卷 2：多线程、锁、JMM、JUC、高并发设计模式》
（已经过时，不建议购买）

新版本：《Java 高并发核心编程 卷 2 **加强版**：多线程、锁、JMM、JUC、高并发设计模式》

- 由浅入深地剖析了 Java 多线程、线程池的底层原理。
- 总结了 IO 密集型、CPU 密集型线程池的线程数预估算法。
- 图文并茂的介绍了 Java 内置锁、JUC 显式锁的核心原理。
- 细致深入地揭秘了 JMM 内存模型。
- 全面介绍了 JUC 框架的设计模式与核心原理，并完成其高核心组件的实战介绍。
- 详尽地介绍了高并发设计模式的使用，以帮助提升高并发、可扩展能力

详情参阅：https://www.cnblogs.com/crazymakercircle/p/16868827.html



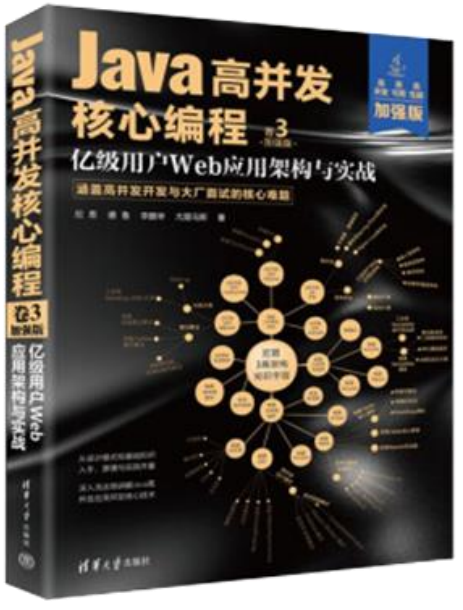
尼恩Java高并发三部曲（卷3加强版）

老版本：《SpringCloud Nginx 高并发核心编程》（已经过时，不建议购买）

新版本：《Java 高并发核心编程 卷 3 **加强版**：亿级用户 Web 应用架构与实战》

- 在当今的面试场景中，3 高知识是大家面试必备的核心知识，本书基于亿级用户 3 高 Web 应用的架构分析理论，为大家对 3 高架构系统做一个系统化和清晰化的介绍。
- 从 Java 静态代理、动态代理模式入手，抽丝剥茧地解读了 Spring Cloud 全家桶中 RPC 核心原理和执行过程，这是高级 Java 工程师面试必备的基础知识。
- 从Reactor 反应器模式入手，抽丝剥茧地解读了Nginx 核心思想和各配置项的底层知识和原理，这是高级 Java 工程师、架构师面试必备的基础知识。
- 从观察者模式入手，抽丝剥茧地解读了 RxJava、Hystrix 的核心思想和使用方法，这也是高级 Java 工程师、架构师面试必备的基础知识。

详情：https://www.cnblogs.com/crazymakercircle/p/16868827.html



专题15：分布式锁（史上最全、定期更新）

本文版本说明：V2

> 此文的格式，由markdown 通过程序转成而来，由于很多表格，没有来的及调整，出现一个格式问题，尼恩在此给大家道歉啦。
>
> 由于社群很多小伙伴，在面试，不断的交流最新的面试难题，所以，《Java面试红宝书》， 后面会不断升级，迭代。
>
> 本专题，作为 《Java面试红宝书》专题之一， 《Java面试红宝书》一共**30个面试专题，后续还会增加**

《Java面试红宝书》升级的规划为：

后续基本上，**每一个月，都会发布一次**，最新版本，可以扫描扫架构师尼恩微信， 发送 “领取电子书” 获取。

尼恩的微信二维码在哪里呢？ 请参见文末

面试问题交流说明：

如果遇到面试难题，或者职业发展问题， 或者中年危机问题，都可以来 疯狂创客圈社群交流，

加入交流群，加尼恩微信即可，

入交流群，加尼恩微信即可， 发送**“入群”**

推荐：入大厂 、做架构、大力提升Java 内功 的 精彩博文

入大厂 、做架构、大力提升Java 内功 必备的精彩博文	2021 秋招涨薪1W + 必备的精彩博文
3: Redis与MySQL双写一致性如何保证？ （面试必备）	4: 面试必备：秒杀超卖 解决方案 （史上最全）
5:面试必备之：Reactor模式	6: 10分钟看懂， Java NIO 底层原理
7:TCP/IP（图解+秒懂+史上最全）	8：Feign原理 （图解）
9:DNS图解（秒懂 + 史上最全 + 高薪必备）	10：CDN图解（秒懂 + 史上最全 + 高薪必备）
11: 分布式事务（ 图解 + 史上最全 + 吐血推荐 ）	12：限流：计数器、漏桶、令牌桶 三大算法的原理与实战（图解+史上最全）
13:架构必看：12306抢票系统亿级流量架构（图解+秒懂+史上最全）	14：seata AT模式实战（图解+秒懂+史上最全）
15：seata 源码解读（图解+秒懂+史上最全）	16：seata TCC模式实战（图解+秒懂+史上最全）

SpringCloud 微服务 精彩博文	
nacos 实战（史上最全）	sentinel （史上最全+入门教程）
SpringCloud gateway （史上最全）	更多专题， 请参见【 疯狂创客圈 高并发 总目录 】

面试必备：分布式锁原理与实战

在单体的应用开发场景中，涉及并发同步的时候，大家往往采用synchronized或者Lock的方式来解决多线程间的同步问题。但在分布式集群工作的开发场景中，那么就需要一种更加高级的锁机制，来处理种跨JVM进程之间的数据同步问题，这就是分布式锁。

公平锁和可重入锁的原理

最经典的分布式锁是可重入的公平锁。什么是可重入的公平锁呢？直接讲解的概念和原理，会比较抽象难懂，还是从具体的实例入手吧！这里用一个简单的故事来类比，估计就简单多了。

故事发生在一个没有自来水的古代，在一个村子有一口井，水质非常的好，村民们都抢着取井里的水。井就那么一口，村里的人很多，村民为争抢取水打架斗殴，甚至头破血流。

问题总是要解决，于是村长绞尽脑汁，最终想出了一个凭号取水的方案。井边安排一个看井人，维护取水的秩序。取水秩序很简单：

（1）取水之前，先取号；

（2）号排在前面的，就可以先取水；

（3）先到的排在前面，那些后到的，一个一个挨着，在井边排成一队。

取水示意图，如图10-3所示。

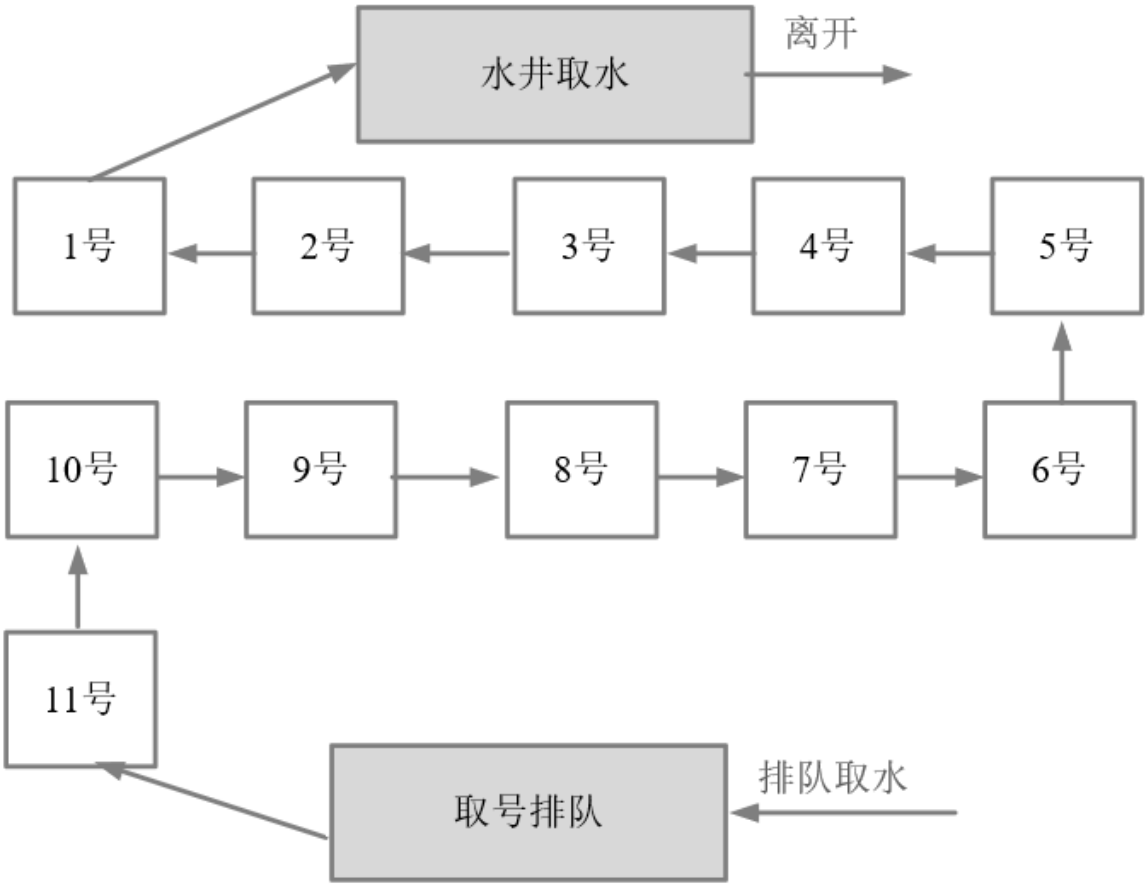


图10-3 排队取水示意图

这种排队取水模型，就是一种锁的模型。排在最前面的号，拥有取水权，就是一种典型的独占锁。另外，先到先得，号排在前面的人先取到水，取水之后就轮到下一个号取水，挺公平的，说明它是一种公平锁。

什么是可重入锁呢?
假定，取水时以家庭为单位，家庭的某人拿到号，其他的家庭成员过来打水，这时候不用再取号，如图10-4所示。

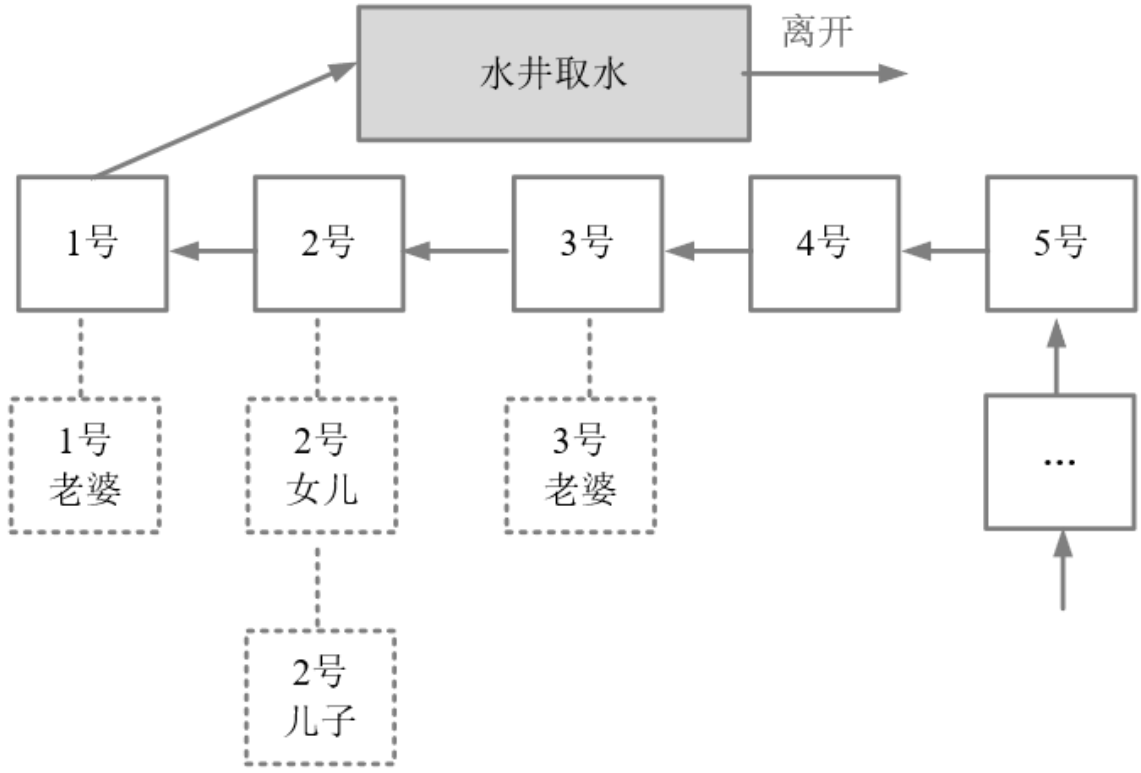


图10-4 同一家庭的人不需要重复排队

图10-4中，排在1号的家庭，老公取号，假设其老婆来了，直接排第一个，正所谓妻凭夫贵。再看上图的2号，父亲正在打水，假设其儿子和女儿也到井边了，直接排第二个，所谓子凭父贵。总之，如果取水时以家庭为单位，则同一个家庭，可以直接复用排号，不用从后面排起重新取号。

以上这个故事模型中，取号一次，可以用来多次取水，其原理为可重入锁的模型。在重入锁模型中，一把独占锁，可以被多次锁定，这就叫做可重入锁。

ZooKeeper分布式锁的原理

理解了经典的公平可重入锁的原理后，再来看在分布式场景下的公平可重入锁的原理。通过前面的分析，基本可以判定：ZooKeeper
的临时顺序节点，天生就有一副实现分布式锁的胚子。为什么呢?

（一） ZooKeeper的每一个节点，都是一个天然的顺序发号器。

在每一个节点下面创建临时顺序节点（EPHEMERAL_SEQUENTIAL）类型，新的子节点后面，会加上一个次序编号，而这个生成的次序编号，是上一个生成的次序编号加一。

例如，有一个用于发号的节点“/test/lock”为父亲节点，可以在这个父节点下面创建相同前缀的临时顺序子节点，假定相同的前缀为“/test/lock/seq-”。第一个创建的子节点基本上应该为/test/lock/seq-0000000000，下一个节点则为/test/lock/seq-0000000001，依次类推，如果10-5所示。

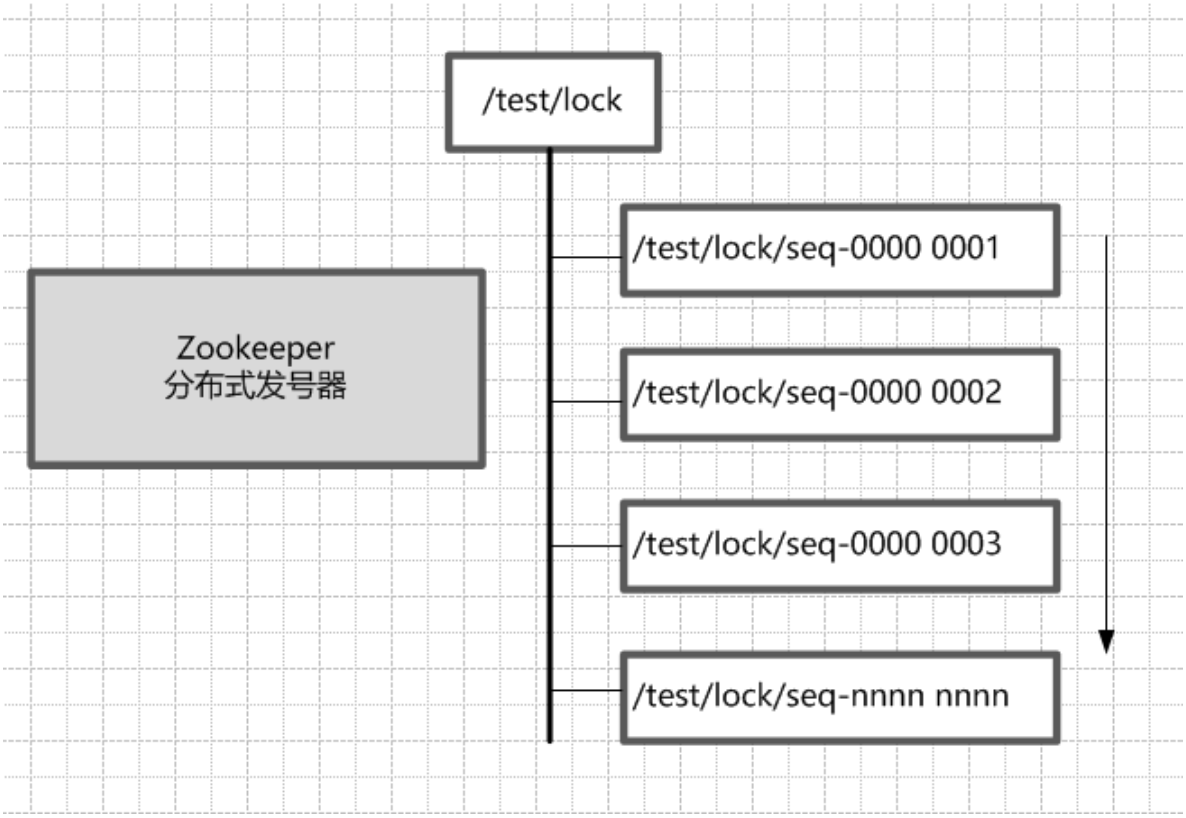


图10-5 Zookeeper临时顺序节点的天然的发号器作用

（二） ZooKeeper节点的递增有序性，可以确保锁的公平

一个ZooKeeper分布式锁，首先需要创建一个父节点，尽量是持久节点（PERSISTENT类型），然后每个要获得锁的线程，都在这个节点下创建个临时顺序节点。由于ZK节点，是按照创建的次序，依次递增的。

为了确保公平，可以简单的规定：编号最小的那个节点，表示获得了锁。所以，每个线程在尝试占用锁之前，首先判断自己是排号是不是当前最小，如果是，则获取锁。

（三）ZooKeeper的节点监听机制，可以保障占有锁的传递有序而且高效

每个线程抢占锁之前，先尝试创建自己的ZNode。同样，释放锁的时候，就需要删除创建的Znode。创建成功后，如果不是排号最小的节点，就处于等待通知的状态。等谁的通知呢？不需要其他人，只需要等前一个Znode
的通知就可以了。前一个Znode删除的时候，会触发Znode事件，当前节点能监听到删除事件，就是轮到了自己占有锁的时候。第一个通知第二个、第二个通知第三个，击鼓传花似的依次向后。

ZooKeeper的节点监听机制，能够非常完美地实现这种击鼓传花似的信息传递。具体的方法是，每一个等通知的Znode节点，只需要监听（linsten）或者监视（watch）排号在自己前面那个，而且紧挨在自己前面的那个节点，就能收到其删除事件了。
只要上一个节点被删除了，就进行再一次判断，看看自己是不是序号最小的那个节点，如果是，自己就获得锁。

另外，ZooKeeper的内部优越的机制，能保证由于网络异常或者其他原因，集群中占用锁的客户端失联时，锁能够被有效释放。一旦占用Znode锁的客户端与ZooKeeper集群服务器失去联系，这个临时Znode也将自动删除。排在它后面的那个节点，也能收到删除事件，从而获得锁。正是由于这个原因，在创建取号节点的时候，尽量创建临时znode
节点，

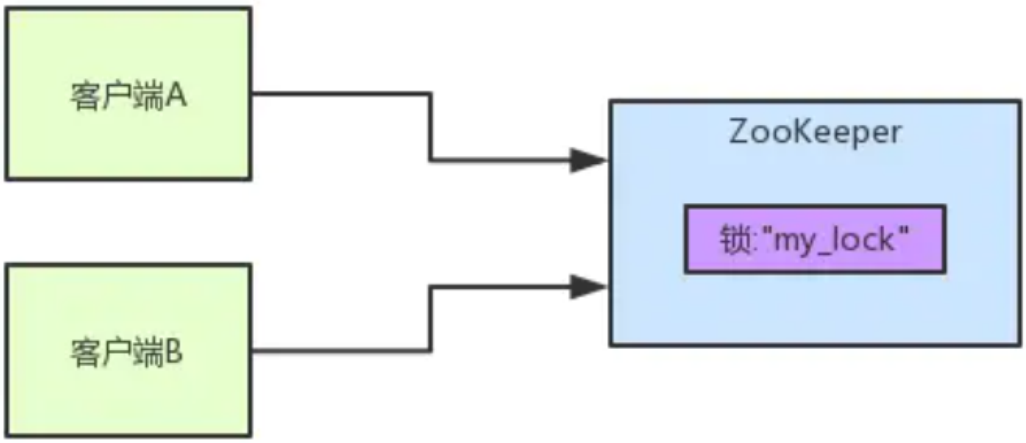
（四）ZooKeeper的节点监听机制，能避免羊群效应

ZooKeeper这种首尾相接，后面监听前面的方式，可以避免羊群效应。所谓羊群效应就是一个节点挂掉，所有节点都去监听，然后做出反应，这样会给服务器带来巨大压力，所以有了临时顺序节点，当一个节点挂掉，只有它后面的那一个节点才做出反应。

图解：分布式锁的抢占过程

接下来我们一起来看看，多客户端获取及释放zk分布式锁的整个流程及背后的原理。

首先大家看看下面的图，如果现在有两个客户端一起要争抢zk上的一把分布式锁，会是个什么场景？



如果大家对zk还不太了解的话，建议先自行百度一下，简单了解点基本概念，比如zk有哪些节点类型等等。

参见上图。zk里有一把锁，这个锁就是zk上的一个节点。然后呢，两个客户端都要来获取这个锁，具体是怎么来获取呢？

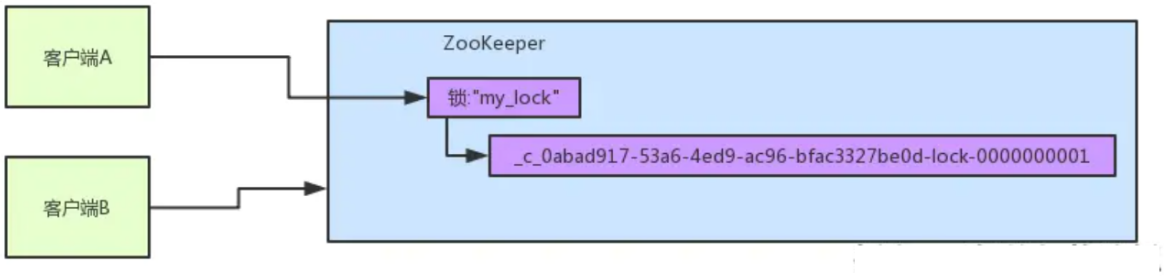
咱们就假设客户端A抢先一步，对zk发起了加分布式锁的请求，这个加锁请求是用到了zk中的一个特殊的概念，叫做“临时顺序节点”。

简单来说，就是直接在"my_lock"这个锁节点下，创建一个顺序节点，这个顺序节点有zk内部自行维护的一个节点序号。

客户端A发起一个加锁请求

比如说，第一个客户端来搞一个顺序节点，zk内部会给起个名字叫做：xxx-000001。然后第二个客户端来搞一个顺序节点，zk可能会起个名字叫做：xxx-000002。大家注意一下，最后一个数字都是依次递增的，从1开始逐次递增。zk会维护这个顺序。

所以这个时候，假如说客户端A先发起请求，就会搞出来一个顺序节点，大家看下面的图，Curator框架大概会弄成如下的样子：



大家看，客户端A发起一个加锁请求，先会在你要加锁的node下搞一个临时顺序节点，这一大坨长长的名字都是Curator框架自己生成出来的。

然后，那个最后一个数字是"1"。大家注意一下，因为客户端A是第一个发起请求的，所以给他搞出来的顺序节点的序号是"1"。

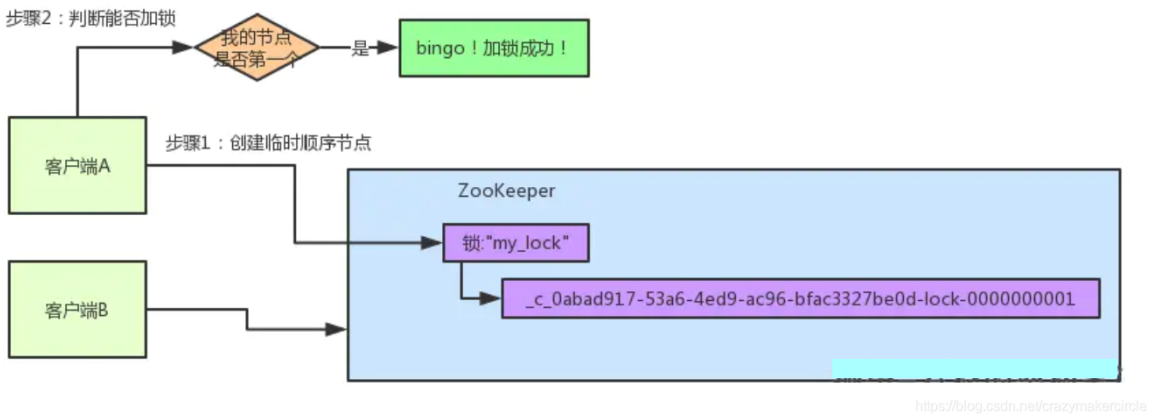
接着客户端A创建完一个顺序节点。还没完，他会查一下"my_lock"这个锁节点下的所有子节点，并且这些子节点是按照序号排序的，这个时候他大概会拿到这么一个集合：

```
[
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001"
]
```

接着客户端A会走一个关键性的判断，就是说：唉！兄弟，这个集合里，我创建的那个顺序节点，是不是排在第一个啊？

如果是的话，那我就可以加锁了啊！因为明明我就是第一个来创建顺序节点的人，所以我就是第一个尝试加分布式锁的人啊！

bingo！加锁成功！大家看下面的图，再来直观的感受一下整个过程。

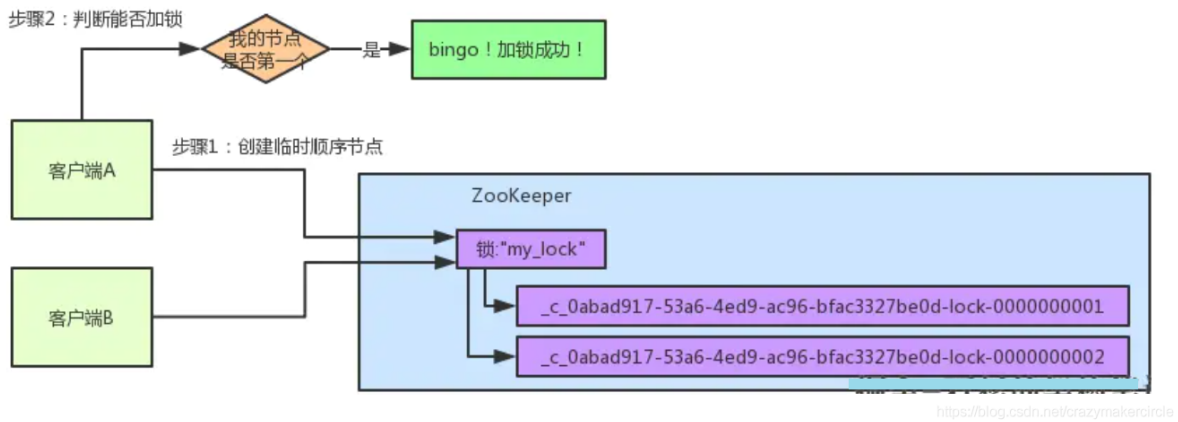


客户端B过来排队

接着假如说，客户端A都加完锁了，客户端B过来想要加锁了，这个时候他会干一样的事儿：先是在"my_lock"这个锁节点下创建一个临时顺序节点，此时名字会变成类似于：

```
_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000002
```

大家看看下面的图：



客户端B因为是第二个来创建顺序节点的，所以zk内部会维护序号为"2"。

接着客户端B会走加锁判断逻辑，查询"my_lock"锁节点下的所有子节点，按序号顺序排列，此时他看到的类似于：

```
[
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001",
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000002"
]
```

同时检查自己创建的顺序节点，是不是集合中的第一个？

明显不是啊，此时第一个是客户端A创建的那个顺序节点，序号为"01"的那个。所以加锁失败！

客户端B开启监听客户端A

加锁失败了以后，客户端B就会通过ZK的API对他的顺序节点的上一个顺序节点加一个监听器。zk天然就可以实现对某个节点的监听。

如果大家还不知道zk的基本用法，可以百度查阅，非常的简单。客户端B的顺序节点是：

```
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000002"
```

他的上一个顺序节点，不就是下面这个吗？

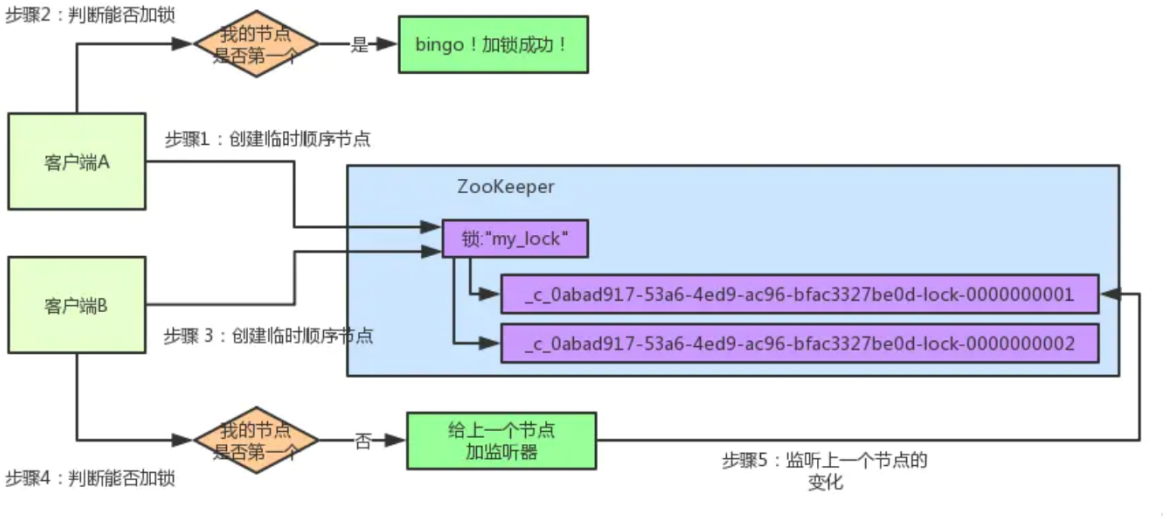
```
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001"
```

即客户端A创建的那个顺序节点！

所以，客户端B会对：

```
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001"
```

这个节点加一个监听器，监听这个节点是否被删除等变化！大家看下面的图。



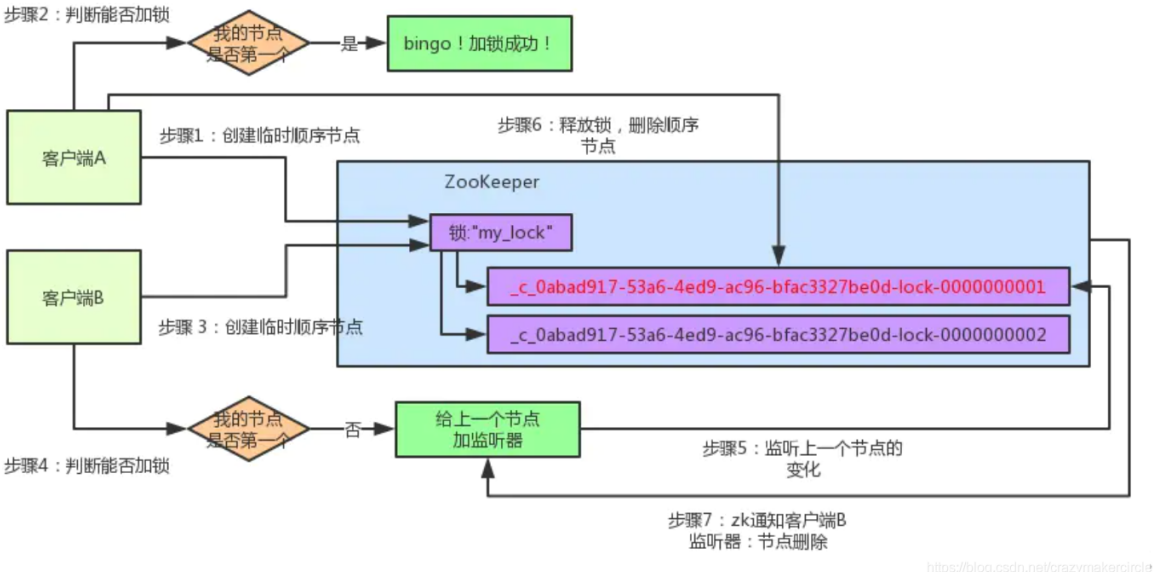
接着，客户端A加锁之后，可能处理了一些代码逻辑，然后就会释放锁。那么，释放锁是个什么过程呢？

其实很简单，就是把自己在zk里创建的那个顺序节点，也就是：

```
"_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001"
```

这个节点给删除。

删除了那个节点之后，zk会负责通知监听这个节点的监听器，也就是客户端B之前加的那个监听器，说：兄弟，你监听的那个节点被删除了，有人释放了锁。



此时客户端B的监听器感知到了上一个顺序节点被删除，也就是排在他之前的某个客户端释放了锁。

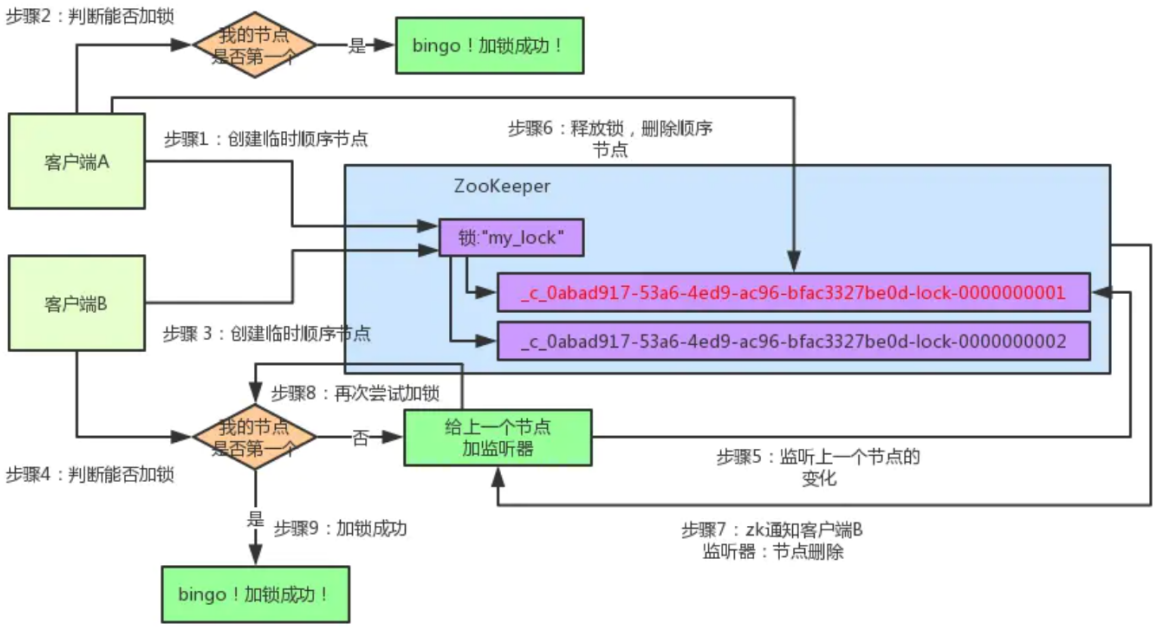
客户端B抢锁成功

此时，就会通知客户端B重新尝试去获取锁，也就是获取"my_lock"节点下的子节点集合，此时为：

```
[
_c_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000002
]
```

集合里此时只有客户端B创建的唯一的一个顺序节点了！

然后呢，客户端B判断自己居然是集合中的第一个顺序节点，bingo！可以加锁了！直接完成加锁，运行后续的业务代码即可，运行完了之后再次释放锁。



分布式锁的基本实现

接下来就是基于ZooKeeper，实现一下分布式锁。首先，定义了一个锁的接口Lock，很简单，仅仅两个抽象方法：一个加锁方法，一个解锁方法。Lock接口的代码如下：

```
package com.crazymakercircle.zk.distributedLock;

/**
 * create by 尼恩 @ 疯狂创客圈
 **/
public interface Lock {
    /**
     * 加锁方法
     *
     * @return 是否成功加锁
     */
    boolean lock() throws Exception;

    /**
     * 解锁方法
     *
     * @return 是否成功解锁
     */
    boolean unlock();
}
```

使用ZooKeeper实现分布式锁的算法，有以下几个要点：

（1）一把分布式锁通常使用一个Znode节点表示；如果锁对应的Znode节点不存在，首先创建Znode节点。这里假设为“/test/lock”，代表了一把需要创建的分布式锁。

（2）抢占锁的所有客户端，使用锁的Znode节点的子节点列表来表示；如果某个客户端需要占用锁，则在“/test/lock”下创建一个临时有序的子节点。

这里，所有临时有序子节点，尽量共用一个有意义的子节点前缀。

比如，如果子节点的前缀为“/test/lock/seq-”，则第一次抢锁对应的子节点为“/test/lock/seq-000000000”，第二次抢锁对应的子节点为“/test/lock/seq-000000001”，以此类推。

再比如，如果子节点前缀为“/test/lock/”，则第一次抢锁对应的子节点为“/test/lock/000000000”，第二次抢锁对应的子节点为“/test/lock/000000001”，以此类推，也非常直观。

（3）如果判定客户端是否占有锁呢?
很简单，客户端创建子节点后，需要进行判断：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，则认为加锁成功；如果不是，则监听前一个Znode子节点变更消息，等待前一个节点释放锁。

（4）一旦队列中的后面的节点，获得前一个子节点变更通知，则开始进行判断，判断自己是否为当前子节点列表中序号最小的子节点，如果是，则认为加锁成功；如果不是，则持续监听，一直到获得锁。

（5）获取锁后，开始处理业务流程。完成业务流程后，删除自己的对应的子节点，完成释放锁的工作，以方面后继节点能捕获到节点变更通知，获得分布式锁。

实战：加锁的实现

Lock接口中加锁的方法是lock（）。lock（）方法的大致流程是：首先尝试着去加锁，如果加锁失败就去等待，然后再重复。

1．lock（）方法的实现代码

lock（）方法加锁的实现代码，大致如下：

```
package com.crazymakercircle.zk.distributedLock;

import com.crazymakercircle.zk.ZKclient;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;

import java.util.Collections;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * create by 尼恩 @ 疯狂创客圈
 **/
@Slf4j
public class ZkLock implements Lock {
    //ZkLock的节点链接
    private static final String ZK_PATH = "/test/lock";
    private static final String LOCK_PREFIX = ZK_PATH + "/";
    private static final long WAIT_TIME = 1000;
```

```
    //Zk客户端
    CuratorFramework client = null;

    private String locked_short_path = null;
    private String locked_path = null;
    private String prior_path = null;
    final AtomicInteger lockCount = new AtomicInteger(0);
    private Thread thread;

    public ZkLock() {
        ZKclient.instance.init();
        synchronized (ZKclient.instance) {
            if (!ZKclient.instance.isNodeExist(ZK_PATH)) {
                ZKclient.instance.createNode(ZK_PATH, null);
            }
        }
        client = ZKclient.instance.getClient();
    }

    @Override
    public boolean lock() {
//可重入，确保同一线程，可以重复加锁

        synchronized (this) {
            if (lockCount.get() == 0) {
                thread = Thread.currentThread();
                lockCount.incrementAndGet();
            } else {
                if (!thread.equals(Thread.currentThread())) {
                    return false;
                }
                lockCount.incrementAndGet();
                return true;
            }
        }

        try {
            boolean locked = false;
//首先尝试着去加锁
            locked = tryLock();

            if (locked) {
                return true;
            }
            //如果加锁失败就去等待
            while (!locked) {

                await();

                //获取等待的子节点列表

                List<String> waiters = getWaiters();
//判断，是否加锁成功
                if (checkLocked(waiters)) {
                    locked = true;
                }
            }
            return true;
```

```
        } catch (Exception e) {
            e.printStackTrace();
            unlock();
        }

        return false;
    }



//...省略其他的方法

}
```

2．tryLock（）尝试加锁

尝试加锁的tryLock方法是关键，做了两件重要的事情：

（1）创建临时顺序节点，并且保存自己的节点路径

（2）判断是否是第一个，如果是第一个，则加锁成功。如果不是，就找到前一个Znode节点，并且保存其路径到prior_path。

尝试加锁的tryLock方法，其实现代码如下：

```
    /**
     * 尝试加锁
     * @return 是否加锁成功
     * @throws Exception 异常
     */
    private boolean tryLock() throws Exception {
        //创建临时Znode
        locked_path = ZKclient.instance
                .createEphemeralSeqNode(LOCK_PREFIX);
        //然后获取所有节点
        List<String> waiters = getWaiters();

        if (null == locked_path) {
            throw new Exception("zk error");
        }
        //取得加锁的排队编号
        locked_short_path = getShortPath(locked_path);

        //获取等待的子节点列表，判断自己是否第一个
        if (checkLocked(waiters)) {
            return true;
        }

        // 判断自己排第几个
        int index = Collections.binarySearch(waiters, locked_short_path);
        if (index < 0) { // 网络抖动，获取到的子节点列表里可能已经没有自己了
            throw new Exception("节点没有找到: " + locked_short_path);
        }

        //如果自己没有获得锁，则要监听前一个节点
        prior_path = ZK_PATH + "/" + waiters.get(index - 1);
```

```
        return false;
    }

    private String getShortPath(String locked_path) {

        int index = locked_path.lastIndexOf(ZK_PATH + "/");
        if (index >= 0) {
            index += ZK_PATH.length() + 1;
            return index <= locked_path.length() ? locked_path.substring(index)
: "";
        }
        return null;
    }
```

创建临时顺序节点后，其完整路径存放在locked_path成员中；另外还截取了一个后缀路径，放在locked_short_path成员中，后缀路径是一个短路径，只有完整路径的最后一层。为什么要单独保存短路径呢?
因为，在获取的远程子节点列表中的其他路径返回结果时，返回的都是短路径，都只有最后一层路径。所以为了方便后续进行比较，也把自己的短路径保存下来。

创建了自己的临时节点后，调用checkLocked方法，判断是否是锁定成功。如果锁定成功，则返回true；如果自己没有获得锁，则要监听前一个节点，此时需要找出前一个节点的路径，并保存在prior_path
成员中，供后面的await（）等待方法去监听使用。在进入await（）等待方法的介绍前，先说下checkLocked
锁定判断方法。

3．checkLocked（）检查是否持有锁

在checkLocked（）方法中，判断是否可以持有锁。判断规则很简单：当前创建的节点，是否在上一步获取到的子节点列表的第一个位置：

（1）如果是，说明可以持有锁，返回true，表示加锁成功；

（2）如果不是，说明有其他线程早已先持有了锁，返回false。

checkLocked（）方法的代码如下：

```
    private boolean checkLocked(List<String> waiters) {

        //节点按照编号，升序排列
        Collections.sort(waiters);

        // 如果是第一个，代表自己已经获得了锁
        if (locked_short_path.equals(waiters.get(0))) {
            log.info("成功的获取分布式锁,节点为{}", locked_short_path);
            return true;
        }
        return false;
    }
```

checkLocked方法比较简单，将参与排队的所有子节点列表，从小到大根据节点名称进行排序。排序主要依靠节点的编号，也就是后Znode路径的10位数字，因为前缀都是一样的。排序之后，做判断，如果自己的locked_short_path编号位置排在第一个，如果是，则代表自己已经获得了锁。如果不是，则会返回false。

如果checkLocked（）为false，外层的调用方法，一般来说会执行await（）等待方法，执行夺锁失败以后的等待逻辑。

4．await（）监听前一个节点释放锁

await（）也很简单，就是监听前一个ZNode节点（prior_path成员）的删除事件，代码如下：

```
    private void await() throws Exception {

        if (null == prior_path) {
            throw new Exception("prior_path error");
        }

        final CountDownLatch latch = new CountDownLatch(1);


        //订阅比自己次小顺序节点的删除事件
        Watcher w = new Watcher() {
            @Override
            public void process(WatchedEvent watchedEvent) {
                System.out.println("监听到的变化 watchedEvent = " + watchedEvent);
                log.info("[WatchedEvent]节点删除");

                latch.countDown();
            }
        };

        client.getData().usingWatcher(w).forPath(prior_path);
/*
        //订阅比自己次小顺序节点的删除事件
        TreeCache treeCache = new TreeCache(client, prior_path);
        TreeCacheListener l = new TreeCacheListener() {
            @Override
            public void childEvent(CuratorFramework client,
                                   TreeCacheEvent event) throws Exception {
                ChildData data = event.getData();
                if (data != null) {
                    switch (event.getType()) {
                        case NODE_REMOVED:
                            log.debug("[TreeCache]节点删除, path={}, data={}",
                                    data.getPath(), data.getData());

                            latch.countDown();
                            break;
                        default:
                            break;
                    }
                }
            }
        };

        treeCache.getListenable().addListener(l);
        treeCache.start();*/
        latch.await(WAIT_TIME, TimeUnit.SECONDS);
    }
```

首先添加一个Watcher监听，而监听的节点，正是前面所保存在prior_path成员的前一个节点的路径。这里，仅仅去监听自己前一个节点的变动，而不是其他节点的变动，提升效率。完成监听之后，调用latch.await（），线程进入等待状态，一直到线程被监听回调代码中的latch.countDown() 所唤醒，或者等待超时。

说 明

> 以上代码用到的CountDownLatch的核心原理和实战知识，《Netty Zookeeper Redis 高并发实战》姊妹篇 《Java高并发核心编程（卷2）》。

上面的代码中，监听前一个节点的删除，可以使用两种监听方式：

（1）Watcher 订阅；

（2）TreeCache 订阅。

两种方式的效果，都差不多。但是这里的删除事件，只需要监听一次即可，不需要反复监听，所以使用的是Watcher
一次性订阅。而TreeCache 订阅的代码在源码工程中已经被注释，仅仅供大家参考。

一旦前一个节点prior_path节点被删除，那么就将线程从等待状态唤醒，重新一轮的锁的争夺，直到获取锁，并且完成业务处理。

至此，分布式Lock加锁的算法，还差一点就介绍完成。这一点，就是实现锁的可重入。

5．可重入的实现代码

什么是可重入呢？只需要保障同一个线程进入加锁的代码，可以重复加锁成功即可。
修改前面的lock方法，在前面加上可重入的判断逻辑。代码如下：

```
@Override

public boolean lock() {

//可重入的判断

synchronized (this) {

if (lockCount.get() == 0) {

thread = Thread.currentThread();

lockCount.incrementAndGet();

} else {

if (!thread.equals(Thread.currentThread())) {

return false;

}

lockCount.incrementAndGet();

return true;

}
```

```
}

//....

}
```

为了变成可重入，在代码中增加了一个加锁的计数器lockCount
，计算重复加锁的次数。如果是同一个线程加锁，只需要增加次数，直接返回，表示加锁成功。

至此，lock（）方法已经介绍完成，接下来，就是去释放锁

实战：释放锁的实现

Lock接口中的unLock（）方法，表示释放锁，释放锁主要有两个工作：

（1）减少重入锁的计数，如果最终的值不是0，直接返回，表示成功的释放了一次；

（2）如果计数器为0，移除Watchers监听器，并且删除创建的Znode临时节点。

unLock（）方法的代码如下：

```
    /**
     * 释放锁
     *
     * @return 是否成功释放锁
     */
    @Override
    public boolean unlock() {
//只有加锁的线程，能够解锁
        if (!thread.equals(Thread.currentThread())) {
            return false;
        }
//减少可重入的计数
        int newLockCount = lockCount.decrementAndGet();
//计数不能小于0
        if (newLockCount < 0) {
            throw new IllegalMonitorStateException("Lock count has gone negative
for lock: " + locked_path);
        }
//如果计数不为0，直接返回
        if (newLockCount != 0) {
            return true;
        }
        //删除临时节点
        try {
            if (ZKclient.instance.isNodeExist(locked_path)) {
                client.delete().forPath(locked_path);
            }
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }

        return true;
    }
```

这里，为了尽量保证线程安全，可重入计数器的类型，使用的不是int类型，而是Java并发包中的原子类型——AtomicInteger。

实战：分布式锁的使用

写一个用例，测试一下ZLock的使用，代码如下：

```
   @Test
    public void testLock() throws InterruptedException {
        for (int i = 0; i < 10; i++) {
            FutureTaskScheduler.add(() -> {
                //创建锁
                ZkLock lock = new ZkLock();
                lock.lock();
//每条线程，执行10次累加
                for (int j = 0; j < 10; j++) {
//公共的资源变量累加
                    count++;
                }
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                log.info("count = " + count);
                //释放锁
                lock.unlock();

            });
        }

        Thread.sleep(Integer.MAX_VALUE);
    }
```

以上代码是10个并发任务，每个任务累加10次，执行以上用例，会发现结果会是预期的和100，如果不使用锁，结果可能就不是100，因为上面的count是一个普通的变量，不是线程安全的。

说 明

有关线程安全的核心原理和实战知识，请参阅本书的下一卷《Java高并发核心编程（卷2）》。

原理上一个Zlock实例代表一把锁，并需要占用一个Znode永久节点，如果需要很多分布式锁，则也需要很多的不同的Znode节点。以上代码，如果要扩展为多个分布式锁的版本，还需要进行简单改造，这种改造留给各位自己去练习和实现吧。

实战：curator的InterProcessMutex 可重入锁

分布式锁Zlock自主实现主要的价值：学习一下分布式锁的原理和基础开发，仅此而已。实际的开发中，如果需要使用到分布式锁，并建议去自己造轮子，建议直接使用Curator客户端中的各种官方实现的分布式锁，比如其中的InterProcessMutex
可重入锁。

这里提供一个简单的InterProcessMutex 可重入锁的使用实例，代码如下：

```
    @Test
    public void testzkMutex() throws InterruptedException {

        CuratorFramework client = ZKclient.instance.getClient();
        final InterProcessMutex zkMutex =
                new InterProcessMutex(client, "/mutex");
        ;
        for (int i = 0; i < 10; i++) {
            FutureTaskScheduler.add(() -> {

                try {
                    //获取互斥锁
                    zkMutex.acquire();

                    for (int j = 0; j < 10; j++) {
//公共的资源变量累加
                        count++;
                    }
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    log.info("count = " + count);
                    //释放互斥锁
                    zkMutex.release();

                } catch (Exception e) {
                    e.printStackTrace();
                }

            });
        }

        Thread.sleep(Integer.MAX_VALUE);
    }
```

ZooKeeper分布式锁的优点和缺点

总结一下ZooKeeper分布式锁：

（1）优点：ZooKeeper分布式锁（如InterProcessMutex），能有效的解决分布式问题，不可重入问题，使用起来也较为简单。

（2）缺点：ZooKeeper实现的分布式锁，性能并不太高。为啥呢？
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。大家知道，ZK中创建和删除节点只能通过Leader服务器来执行，然后Leader服务器还需要将数据同不到所有的Follower机器上，这样频繁的网络通信，性能的短板是非常突出的。

总之，在高性能，高并发的场景下，不建议使用ZooKeeper的分布式锁。而由于ZooKeeper的高可用特性，所以在并发量不是太高的场景，推荐使用ZooKeeper的分布式锁。

在目前分布式锁实现方案中，比较成熟、主流的方案有两种：

（1）基于Redis的分布式锁

（2）基于ZooKeeper的分布式锁

两种锁，分别适用的场景为：

（1）基于ZooKeeper的分布式锁，适用于高可靠（高可用）而并发量不是太大的场景；

（2）基于Redis的分布式锁，适用于并发量很大、性能要求很高的、而可靠性问题可以通过其他方案去弥补的场景。

总之，这里没有谁好谁坏的问题，而是谁更合适的问题。

最后对本章的内容做个总结：在分布式系统中，ZooKeeper是一个重要的协调工具。本章介绍了分布式命名服务、分布式锁的原理以及基于ZooKeeper的参考实现。本章的那些实战案例，建议大家自己去动手掌握，无论是应用实际开始、还是大公司面试，都是非常有用的。另外，主流的分布式协调中间件，也不仅仅只有Zookeeper，还有非常著名的Etcd中间件。但是从学习的层面来说，二者之间的功能设计、核心原理都是差不多的，掌握了Zookeeper，Etcd的上手使用也是很容易的。

redis分布式锁

下面是与redis面试相关的系列博文，建议大家系统化、系统化的学习

Redis 面试题 - 收藏版 （持续更新、吐血推荐）

Redis集群 - 图解 - 秒懂（史上最全）

redis cluster 集群 HA 原理和实操（史上最全、面试必备）

Redis与DB的数据一致性解决方案（史上最全）

Redis 分布式锁 （图解-秒懂-史上最全）

跨JVM的线程安全问题

在单体的应用开发场景中，在多线程的环境下，涉及并发同步的时候，为了保证一个代码块在同一时间只能由一个线程访问，我们一般可以使用synchronized语法和ReetrantLock去保证，这实际上是本地锁的方式。

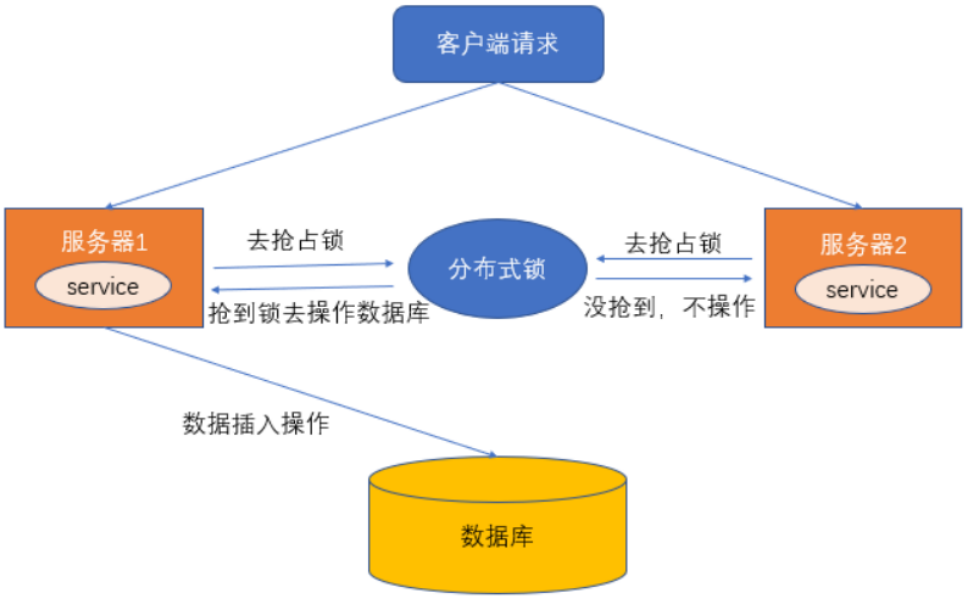
也就是说，在同一个JVM内部，大家往往采用synchronized或者Lock的方式来解决多线程间的安全问题。但在分布式集群工作的开发场景中，在JVM之间，那么就需要一种更加高级的锁机制，来处理种跨JVM进程之间的线程安全问题.

> 解决方案是：使用分布式锁

总之，对于分布式场景，我们可以使用分布式锁，它是控制分布式系统之间**互斥访问共享资源**的一种方式。

比如说在一个分布式系统中，多台机器上部署了多个服务，当客户端一个用户发起一个数据插入请求时，如果没有分布式锁机制保证，那么那多台机器上的多个服务可能进行并发插入操作，导致数据重复插入，对于某些不允许有多余数据的业务来说，这就会造成问题。而分布式锁机制就是为了解决类似这类问题，保证多个服务之间互斥的访问共享资源，如果一个服务抢占了分布式锁，其他服务没获取到锁，就不进行后续操作。

大致意思如下图所示（不一定准确）：



何为分布式锁?

何为分布式锁?

- 当在分布式模型下，数据只有一份（或有限制），此时需要利用锁的技术控制某一时刻修改数据的进程数。
- 用一个状态值表示锁，对锁的占用和释放通过状态值来标识。

分布式锁的条件：

- 互斥性。在任意时刻，只有一个客户端能持有锁。
- 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
- 具有容错性。只要大部分的 Redis 节点正常运行，客户端就可以加锁和解锁。
- 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

分布式锁的实现：

分布式锁的实现由很多种，文件锁、数据库、redis等等，比较多；分布式锁常见的多种实现方式：

1. 数据库悲观锁、
2. 数据库乐观锁；
3. 基于Redis的分布式锁；
4. 基于ZooKeeper的分布式锁。

在实践中，还是redis做分布式锁性能会高一些

数据库悲观锁

所谓悲观锁，悲观锁是对数据被的修改持悲观态度（认为数据在被修改的时候一定会存在并发问题），因此在整个数据处理过程中将数据锁定。

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，即使在应用层中实现了加锁机制，也无法保证外部系统不会修改数据）。

> 数据库的行锁、表锁、排他锁等都是悲观锁，这里以行锁为例，进行介绍。以我们常用的MySQL为例，我们通过使用select...for update语句，执行该语句后，会在表上加持行锁，一直到事务提交，解除行锁。

使用场景举例:

> 在秒杀案例中，生成订单和扣减库存的操作，可以通过商品记录的行锁，进行保护。们通过使用select...for update语句，在查询商品表库存时将该条记录加锁，待下单减库存完成后，再释放锁。

示例的SQL如下：

```
//0.开始事务
begin;

//1.查询出商品信息

select stockCount from seckill_good where id=1 for update;

//2.根据商品信息生成订单

insert into seckill_order (id,good_id) values (null,1);

//3.修改商品stockCount减一

update seckill_good set stockCount=stockCount-1 where id=1;

//4.提交事务

commit;
```

以上，在对id = 1的记录修改前，先通过for update的方式进行加锁，然后再进行修改。这就是比较典型的悲观锁策略。

如果以上修改库存的代码发生并发，同一时间只有一个线程可以开启事务并获得id=1的锁，其它的事务必须等本次事务提交之后才能执行。这样我们可以保证当前的数据不会被其它事务修改。

> 我们使用select_for_update，另外一定要写在事务中.
>
> 注意：要使用悲观锁，我们必须关闭mysql数据库中自动提交的属性，命令set autocommit=0;即可关闭，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。

悲观锁的实现，往往依靠数据库提供的锁机制。在数据库中，悲观锁的流程如下：

- 在对记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。
- 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。具体响应方式由开发者根据实际需要决定。
- 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。
- 其间如果有其他事务对该记录做加锁的操作，都要等待当前事务解锁或直接抛出异常。

数据库乐观锁

> 使用乐观锁就不需要借助数据库的锁机制了。

乐观锁的概念中其实已经阐述了他的具体实现细节：主要就是两个步骤：冲突检测和数据更新。其实现方式有一种比较典型的就是**Compare and Swap(CAS)技术。**

CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，**失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。**

> CAS的实现中，在表中增加一个version字段，操作前先查询version信息，在数据提交时检查version字段是否被修改，如果没有被修改则进行提交，否则认为是过期数据。

比如前面的扣减库存问题，通过乐观锁可以实现如下：

```
//1.查询出商品信息

select stockCount, version from seckill_good where id=1;

//2.根据商品信息生成订单
insert into seckill_order (id,good_id) values (null,1);

//3.修改商品库存
update seckill_good set stockCount=stockCount-1, version = version+1 where id=1,
version=version;
```

以上，我们在更新之前，先查询一下库存表中当前版本（version），然后在做update的时候，以 version 作为一个修改条件。

当我们提交更新的时候，判断数据库表对应记录的当前version与第一次取出来的version进行比对，如果数据库表当前version与第一次取出来的version相等，则予以更新，否则认为是过期数据。

CAS 乐观锁有两个问题：

(1) CAS 存在一个比较重要的问题，即**ABA问题**．解决的办法是version字段顺序递增。

(2) 乐观锁的方式，在高并发时，只有一个线程能执行成功，会造成大量的失败，这给用户的体验显然是很不好的。

Zookeeper分布式锁

除了在数据库层面加分布式锁，通常还可以使用以下更高性能、更高可用的分布式锁：

- 分布式缓存（如redis）锁
- 分布式协调（如zookeeper）锁

> 有关zookeeper分布式锁的原理和实现，具体请参见下面的博客：
> Zookeeper 分布式锁 （图解+秒懂+史上最全）

或者阅读笔者的《Java高并发核心编程（卷1）》

Redis分布式锁

本文重点介绍Redis分布式锁，分为两个维度进行介绍：

（1）基于Jedis手工造轮子分布式锁

（2）介绍Redission 分布式锁的使用和原理。

分布式锁一般有如下的特点：

- 互斥性： 同一时刻只能有一个线程持有锁
- 可重入性： 同一节点上的同一个线程如果获取了锁之后能够再次获取锁
- 锁超时：和J.U.C中的锁一样支持锁超时，防止死锁
- 高性能和高可用： 加锁和解锁需要高效，同时也需要保证高可用，防止分布式锁失效
- 具备阻塞和非阻塞性：能够及时从阻塞状态中被唤醒

手工造轮子：基于Jedis 的API实现分布式锁

我们首先讲解 Jedis 普通分布式锁实现，并且是纯手工的模式，从最为基础的Redis命令开始。

只有充分了解与分布式锁相关的普通Redis命令，才能更好的了解高级的Redis分布式锁的实现，因为高级的分布式锁的实现完全基于普通Redis命令。

Redis几种架构

Redis发展到现在，几种常见的部署架构有：

- 单机模式；
- 主从模式；
- 哨兵模式；
- 集群模式；

从分布式锁的角度来说， 无论是单机模式、主从模式、哨兵模式、集群模式，其原理都是类同的。 只是主从模式、哨兵模式、集群模式的更加的高可用、或者更加高并发。

> 所以，接下来先基于单机模式，基于Jedis手工造轮子实现自己的分布式锁。

首先看两个命令：

> Redis分布式锁机制，主要借助setnx和expire两个命令完成。

setnx命令:

SETNX 是SET if Not eXists的简写。将 key 的值设为 value，当且仅当 key 不存在；若给定的 key 已经存在，则 SETNX 不做任何动作。

下面为客户端使用示例：

```
127.0.0.1:6379> set lock "unlock"
OK
127.0.0.1:6379> setnx lock "unlock"
(integer) 0
127.0.0.1:6379> setnx lock "lock"
(integer) 0
127.0.0.1:6379>
```

expire命令:

expire命令为 key 设置生存时间，当 key 过期时(生存时间为 0)，它会被自动删除. 其格式为：

> EXPIRE key seconds

下面为客户端使用示例：

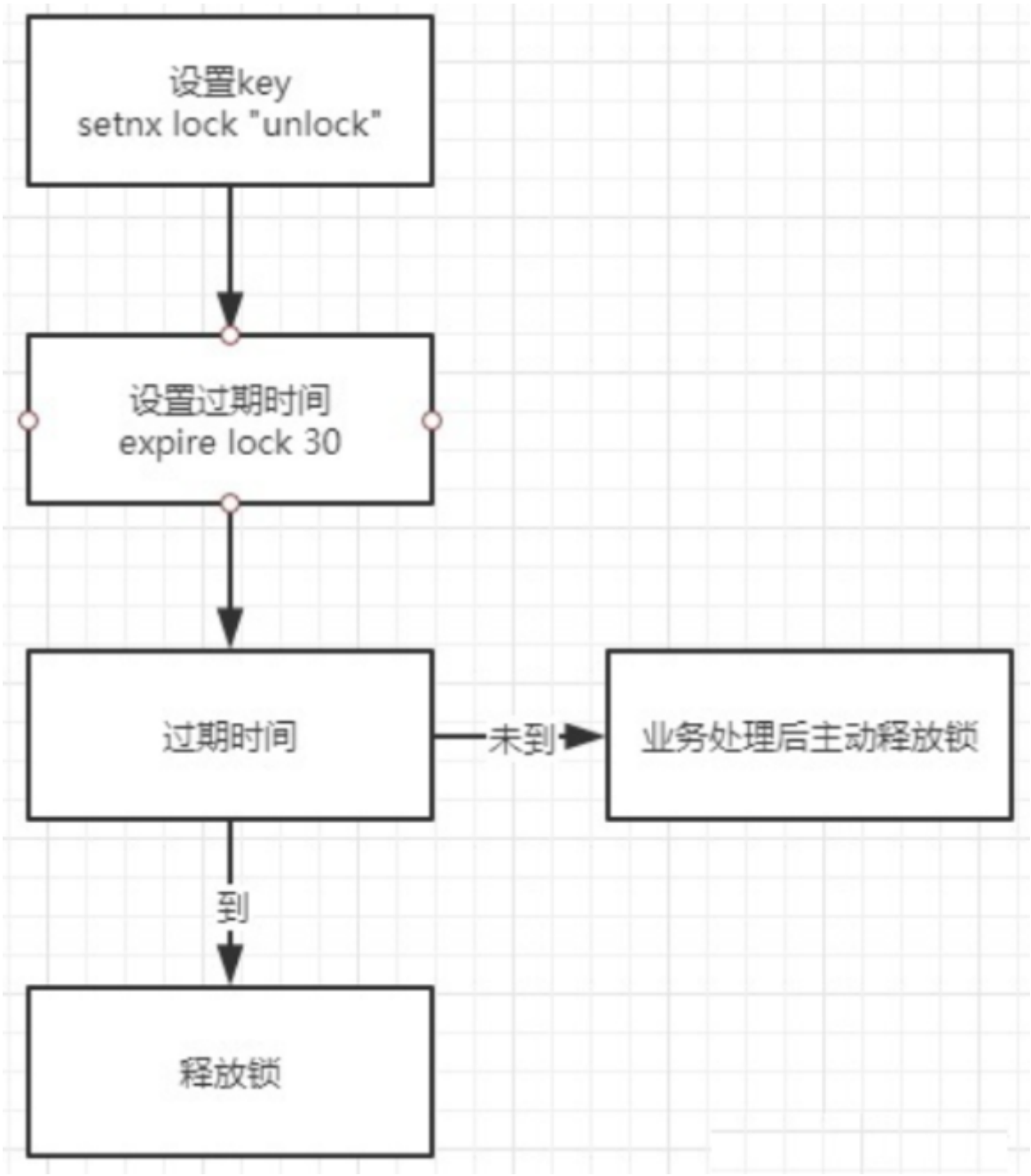
```
127.0.0.1:6379> expire lock 10
(integer) 1
127.0.0.1:6379> ttl lock
8
```

基于Jedis API的分布式锁的总体流程：

通过Redis的setnx、expire命令可以实现简单的锁机制：

- key不存在时创建，并设置value和过期时间，返回值为1；成功获取到锁；
- 如key存在时直接返回0，抢锁失败；
- 持有锁的线程释放锁时，手动删除key； 或者过期时间到，key自动删除，锁释放。

线程调用setnx方法成功返回1认为加锁成功，其他线程要等到当前线程业务操作完成释放锁后，才能再次调用setnx加锁成功。



以上简单redis分布式锁的问题：

如果出现了这么一个问题：如果`setnx`是成功的，但是`expire`设置失败，一旦出现了释放锁失败，或者没有手工释放，那么这个锁永远被占用，其他线程永远也抢不到锁。

> 所以,需要保障setnx和expire两个操作的原子性，要么全部执行，要么全部不执行，二者不能分开。

解决的办法有两种：

- 使用set的命令时，同时设置过期时间，不再单独使用 expire命令
- 使用lua脚本，将加锁的命令放在lua脚本中原子性的执行

简单加锁：使用set的命令时，同时设置过期时间

使用set的命令时，同时设置过期时间的示例如下：

```
127.0.0.1:6379> set unlock "234" EX 100 NX
(nil)
127.0.0.1:6379>
127.0.0.1:6379> set test "111" EX 100 NX
OK
```

这样就完美的解决了分布式锁的原子性； set 命令的完整格式：

> set key value [EX seconds] [PX milliseconds] [NX|XX]

```
EX seconds：设置失效时长，单位秒
PX milliseconds：设置失效时长，单位毫秒
NX：key不存在时设置value，成功返回OK，失败返回(nil)
XX：key存在时设置value，成功返回OK，失败返回(nil)
```

使用set命令实现加锁操作，先展示加锁的简单代码实习，再带大家慢慢解释为什么这样实现。

加锁的简单代码实现

```
package com.crazymaker.springcloud.standard.lock;


@Slf4j
@Data
@AllArgsConstructor
public class JedisCommandLock {

    private  RedisTemplate redisTemplate;

    private static final String LOCK_SUCCESS = "OK";
    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";

    /**
     * 尝试获取分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @param expireTime 超期时间
     * @return 是否获取成功
     */
    public static   boolean tryGetDistributedLock(Jedis jedis, String lockKey,
String requestId, int expireTime) {

        String result = jedis.set(lockKey, requestId, SET_IF_NOT_EXIST,
SET_WITH_EXPIRE_TIME, expireTime);

        if (LOCK_SUCCESS.equals(result)) {
            return true;
```

```
        }
        return false;

    }



}
```

可以看到，我们加锁用到了**Jedis的set Api**：

> `jedis.set(String key, String value, String nxxx, String expx, int time)`

这个set()方法一共有五个形参：

- 第一个为key，我们使用key来当锁，因为key是唯一的。
- 第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。

  > requestId可以使用`UUID.randomUUID().toString()`方法生成。

- 第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
- 第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。
- 第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：

> 1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。
> 2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足前面描述的四个条件中的三个。

- 首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。
- 其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会被永远占用（而发生死锁）。
- 最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。
- 由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

基于Jedis 的API实现简单解锁代码

还是先展示代码，再带大家慢慢解释为什么这样实现。

解锁的简单代码实现：

```
package com.crazymaker.springcloud.standard.lock;
```

```
@Slf4j
@Data
@AllArgsConstructor
public class JedisCommandLock {



    private static final Long RELEASE_SUCCESS = 1L;

    /**
     * 释放分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @return 是否释放成功
     */
    public static boolean releaseDistributedLock(Jedis jedis, String lockKey,
String requestId) {

        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return
redis.call('del', KEYS[1]) else return 0 end";
        Object result = jedis.eval(script, Collections.singletonList(lockKey),
Collections.singletonList(requestId));

        if (RELEASE_SUCCESS.equals(result)) {
            return true;
        }
        return false;

    }

}
```

那么这段Lua代码的功能是什么呢?

其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。

第一行代码，我们写了一个简单的Lua脚本代码。

第二行代码，我们将Lua代码传到`jedis.eval()`方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。

> **那么为什么要使用Lua语言来实现呢?**
>
> 因为要确保上述操作是原子性的。那么为什么执行eval()方法可以确保原子性，源于Redis的特性.
>
> 简单来说，就是在eval命令执行Lua代码的时候，Lua代码将被当成一个命令去执行，并且直到eval命令执行完成，Redis才会执行其他命

错误示例1

最常见的解锁代码就是直接使用 jedis.del() 方法删除锁，这种不先判断锁的拥有者而直接解锁的方式，会导致任何客户端都可以随时进行解锁，即使这把锁不是它的。

```
public static void wrongReleaseLock1(Jedis jedis, String lockKey) {
    jedis.del(lockKey);
}
```

错误示例2

这种解锁代码乍一看也是没问题，甚至我之前也差点这样实现，与正确姿势差不多，唯一区别的是分成两条命令去执行，代码如下：

```
public static void wrongReleaseLock2(Jedis jedis, String lockKey, String
requestId) {

    // 判断加锁与解锁是不是同一个客户端
    if (requestId.equals(jedis.get(lockKey))) {
        // 若在此时，这把锁突然不是这个客户端的，则会误解锁
        jedis.del(lockKey);
    }

}
```

再造轮子：基于Lua脚本实现分布式锁

lua脚本的好处

前面提到，在redis中执行lua脚本，有如下的好处：

> **那么为什么要使用Lua语言来实现呢?**
>
> 因为要确保上述操作是原子性的。那么为什么执行eval()方法可以确保原子性，源于Redis的特性.
>
> 简单来说，就是在eval命令执行Lua代码的时候，Lua代码将被当成一个命令去执行，并且直到eval命令执行完成，Redis才会执行其他命

所以：

> 大部分的开源框架（如 redission）中的分布式锁组件，都是用纯lua脚本实现的。

题外话： lua脚本是**高并发、高性能的必备脚本语言**

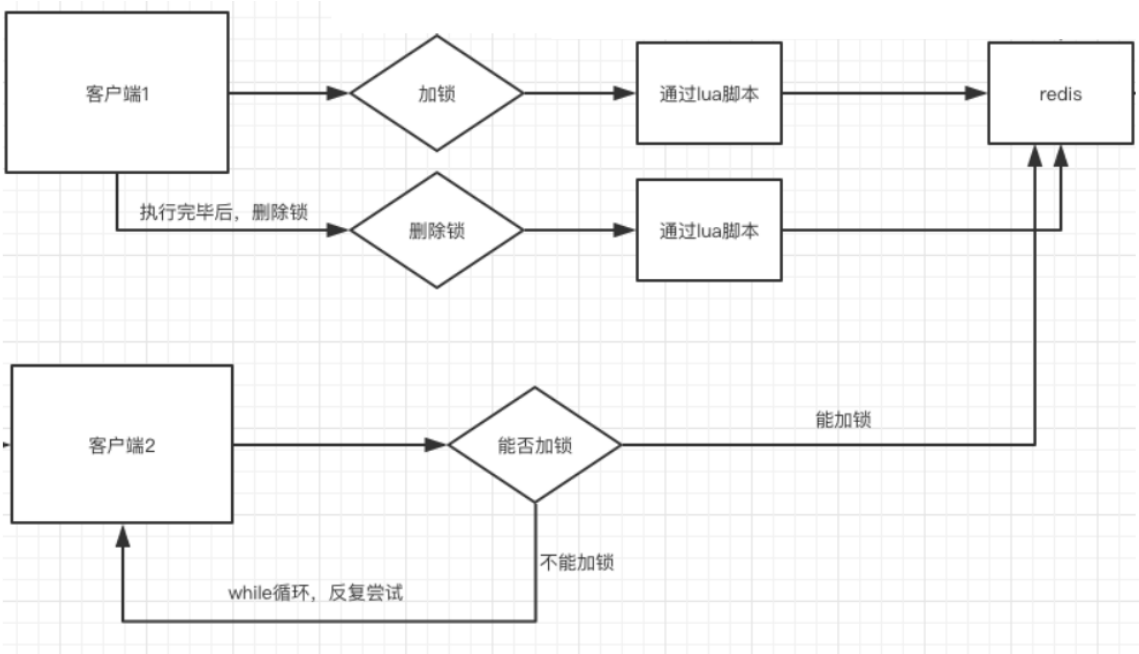
有关lua的详细介绍，请参见以下书籍：

那么，我们也来模拟一下

基于纯Lua脚本的分布式锁的执行流程

加锁和删除锁的操作，使用纯lua进行封装，保障其执行时候的原子性。

基于纯Lua脚本实现分布式锁的执行流程，大致如下：



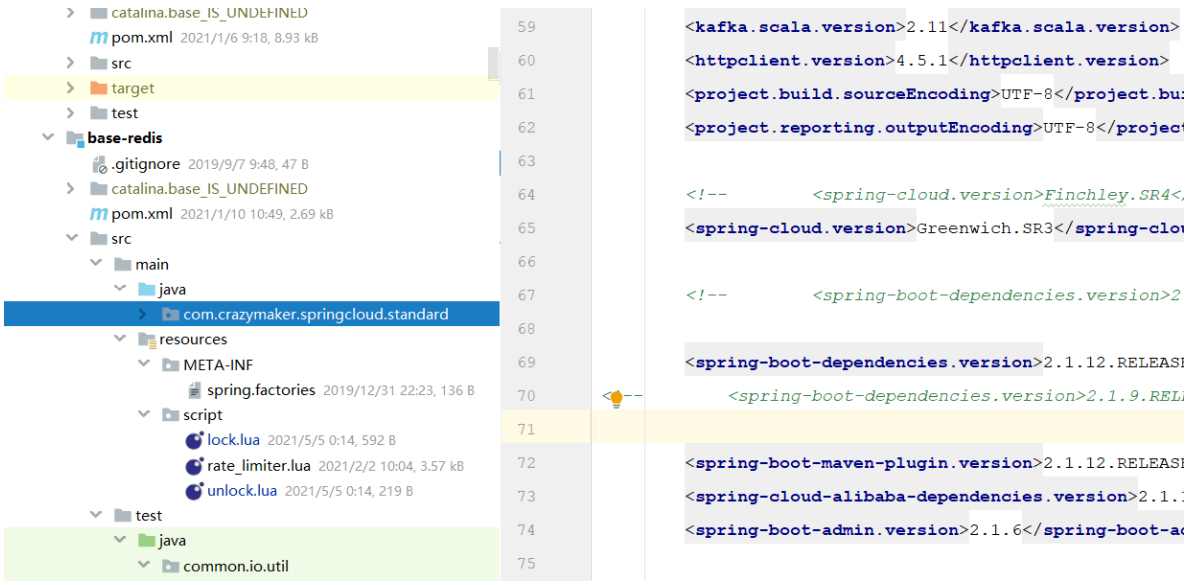
加锁的Lua脚本： lock.lua

```
--- -1 failed
--- 1 success
---
local key = KEYS[1]
local requestId = KEYS[2]
local ttl = tonumber(KEYS[3])
local result = redis.call('setnx', key, requestId)
if result == 1 then
    --PEXPIRE:以毫秒的形式指定过期时间
    redis.call('pexpire', key, ttl)
else
    result = -1;
    -- 如果value相同，则认为是同一个线程的请求，则认为重入锁
    local value = redis.call('get', key)
    if (value == requestId) then
        result = 1;
        redis.call('pexpire', key, ttl)
    end
end
--  如果获取锁成功，则返回 1
return result
```

解锁的Lua脚本： unlock.lua：

```
--- -1 failed
--- 1 success

-- unlock key
local key = KEYS[1]
local requestId = KEYS[2]
local value = redis.call('get', key)
if value == requestId then
    redis.call('del', key);
    return 1;
end
return -1
```

两个文件，放在资源文件夹下备用：



在Java中调用lua脚本，完成加锁操作

```
package com.crazymaker.springcloud.standard.lock;

import com.crazymaker.springcloud.common.exception.BusinessException;
import com.crazymaker.springcloud.common.util.IOUtil;
import com.crazymaker.springcloud.standard.context.SpringContextUtil;
import com.crazymaker.springcloud.standard.lua.ScriptHolder;
import lombok.extern.slf4j.Slf4j;
import org.apache.commons.lang3.StringUtils;
import org.springframework.data.redis.core.RedisCallback;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.data.redis.core.script.RedisScript;

import java.util.ArrayList;
import java.util.List;

@Slf4j
public class InnerLock {

    private RedisTemplate redisTemplate;


    public static final Long LOCKED = Long.valueOf(1);
    public static final Long UNLOCKED = Long.valueOf(1);
    public static final int EXPIRE = 2000;

    String key;
    String requestId;  // lockValue 锁的value ,代表线程的uuid

    /**
     * 默认为2000ms
     */
    long expire = 2000L;


    private volatile boolean isLocked = false;
```

```
    private RedisScript lockScript;
    private RedisScript unLockScript;


    public InnerLock(String lockKey, String requestId) {
        this.key = lockKey;
        this.requestId = requestId;
        lockScript = ScriptHolder.getLockScript();
        unLockScript = ScriptHolder.getUnlockScript();
    }

    /**
     * 抢夺锁
     */
    public void lock() {
        if (null == key) {
            return;
        }
        try {
            List<String> redisKeys = new ArrayList<>();
            redisKeys.add(key);
            redisKeys.add(requestId);
            redisKeys.add(String.valueOf(expire));

            Long res = (Long) getRedisTemplate().execute(lockScript, redisKeys);
            isLocked = false;
        } catch (Exception e) {
            e.printStackTrace();
            throw BusinessException.builder().errMsg("抢锁失败").build();
        }
    }

    /**
     * 有返回值的抢夺锁
     *
     * @param millisToWait
     */
    public boolean lock(Long millisToWait) {
        if (null == key) {
            return false;
        }
        try {
            List<String> redisKeys = new ArrayList<>();
            redisKeys.add(key);
            redisKeys.add(requestId);
            redisKeys.add(String.valueOf(millisToWait));
            Long res = (Long) getRedisTemplate().execute(lockScript, redisKeys);

            return res != null && res.equals(LOCKED);
        } catch (Exception e) {
            e.printStackTrace();
            throw BusinessException.builder().errMsg("抢锁失败").build();
        }

    }

    //释放锁
    public void unlock() {
```

```
        if (key == null || requestId == null) {
            return;
        }
        try {
            List<String> redisKeys = new ArrayList<>();
            redisKeys.add(key);
            redisKeys.add(requestId);
            Long res = (Long) getRedisTemplate().execute(unLockScript,
redisKeys);

//            boolean unlocked = res != null && res.equals(UNLOCKED);


        } catch (Exception e) {
            e.printStackTrace();
            throw BusinessException.builder().errMsg("释放锁失败").build();

        }
    }

    private RedisTemplate getRedisTemplate() {
        if(null==redisTemplate)
        {
            redisTemplate= (RedisTemplate)
SpringContextUtil.getBean("stringRedisTemplate");
        }
        return redisTemplate;
    }
}
```

在Java中调用lua脚本，完成加锁操作

下一步，实现Lock接口，完成JedisLock的分布式锁。

其加锁操作，通过调用 lock.lua脚本完成，代码如下：

```
package com.crazymaker.springcloud.standard.lock;

import com.crazymaker.springcloud.common.exception.BusinessException;
import com.crazymaker.springcloud.common.util.ThreadUtil;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.script.RedisScript;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;

@Slf4j
@Data
```

```
@AllArgsConstructor
public class JedisLock implements Lock {

    private RedisTemplate redisTemplate;

    RedisScript<Long> lockScript = null;
    RedisScript<Long> unLockScript = null;

    public static final int DEFAULT_TIMEOUT = 2000;
    public static final Long LOCKED = Long.valueOf(1);
    public static final Long UNLOCKED = Long.valueOf(1);
    public static final Long WAIT_GAT = Long.valueOf(200);
    public static final int EXPIRE = 2000;


    String key;
    String lockValue;  // lockValue 锁的value ,代表线程的uuid

    /**
     * 默认为2000ms
     */
    long expire = 2000L;

    public JedisLock(String lockKey, String lockValue) {
        this.key = lockKey;
        this.lockValue = lockValue;
    }

    private volatile boolean isLocked = false;

    private Thread thread;

    /**
     * 获取一个分布式锁 , 超时则返回失败
     *
     * @return 获锁成功 - true | 获锁失败 - false
     */
    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException
{

        //本地可重入
        if (isLocked && thread == Thread.currentThread()) {
            return true;
        }
        expire = unit != null ? unit.toMillis(time) : DEFAULT_TIMEOUT;
        long startMillis = System.currentTimeMillis();
        Long millisToWait = expire;

        boolean localLocked = false;

        int turn = 1;
        while (!localLocked) {

            localLocked = this.lockInner(expire);
            if (!localLocked) {
                millisToWait = millisToWait - (System.currentTimeMillis() -
startMillis);
```

```
                startMillis = System.currentTimeMillis();
                if (millisToWait > 0L) {
                    /**
                     * 还没有超时
                     */
                    ThreadUtil.sleepMilliSeconds(WAIT_GAT);
                    log.info("睡眠一下，重新开始，turn:{},剩余时间：{}", turn++,
millisToWait);
                } else {
                    log.info("抢锁超时");
                    return false;
                }
            } else {
                isLocked = true;
                localLocked = true;
            }
        }
        return isLocked;
    }



    /**
     * 有返回值的抢夺锁
     *
     * @param millisToWait
     */
    public boolean lockInner(Long millisToWait) {
        if (null == key) {
            return false;
        }
        try {
            List<String> redisKeys = new ArrayList<>();
            redisKeys.add(key);
            redisKeys.add(lockValue);
            redisKeys.add(String.valueOf(millisToWait));
            Long res = (Long) redisTemplate.execute(lockScript, redisKeys);

            return res != null && res.equals(LOCKED);
        } catch (Exception e) {
            e.printStackTrace();
            throw BusinessException.builder().errMsg("抢锁失败").build();
        }

    }


}
```

实现JUC的Lock显示锁接口，实现一个简单的分布式锁

其解锁操作，通过调用unlock.lua脚本完成，代码如下：

```
package com.crazymaker.springcloud.standard.lock;

import com.crazymaker.springcloud.common.exception.BusinessException;
import com.crazymaker.springcloud.common.util.ThreadUtil;
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.script.RedisScript;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;

@Slf4j
@Data
@AllArgsConstructor
public class JedisLock implements Lock {

    private RedisTemplate redisTemplate;

    RedisScript<Long> lockScript = null;
    RedisScript<Long> unLockScript = null;

    //释放锁
    @Override
    public void unlock() {
        if (key == null || requestId == null) {
            return;
        }
        try {
            List<String> redisKeys = new ArrayList<>();
            redisKeys.add(key);
            redisKeys.add(requestId);
            Long res = (Long) redisTemplate.execute(unLockScript, redisKeys);

        } catch (Exception e) {
            e.printStackTrace();
            throw BusinessException.builder().errMsg("释放锁失败").build();

        }
    }


}
```

编写RedisLockService用于管理JedisLock

编写个分布式锁服务，用于加载lua脚本，创建 分布式锁，代码如下：

```
package com.crazymaker.springcloud.standard.lock;

import com.crazymaker.springcloud.common.util.IOUtil;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.apache.commons.lang3.StringUtils;
import org.springframework.data.redis.core.RedisTemplate;
```

```
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.data.redis.core.script.RedisScript;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;

@Slf4j
@Data
public class RedisLockService
{

    private RedisTemplate redisTemplate;

    static String lockLua = "script/lock.lua";
    static String unLockLua = "script/unlock.lua";
    static RedisScript<Long> lockScript = null;
    static RedisScript<Long> unLockScript = null;
    {
        String script =
IOUtil.loadJarFile(RedisLockService.class.getClassLoader(),lockLua);
//        String script = FileUtil.readString(lockLua, Charset.forName("UTF-8"
));
        if(StringUtils.isEmpty(script))
        {
            log.error("lua load failed:"+lockLua);
        }

        lockScript = new DefaultRedisScript<>(script, Long.class);

//        script = FileUtil.readString(unLockLua, Charset.forName("UTF-8" ));
        script =
IOUtil.loadJarFile(RedisLockService.class.getClassLoader(),unLockLua);
        if(StringUtils.isEmpty(script))
        {
            log.error("lua load failed:"+unLockLua);
        }
        unLockScript = new DefaultRedisScript<>(script, Long.class);

    }

    public RedisLockService(RedisTemplate redisTemplate)
    {
        this.redisTemplate = redisTemplate;
    }


    public Lock getLock(String lockKey, String lockValue) {
        JedisLock lock=new JedisLock(lockKey,lockValue);
        lock.setRedisTemplate(redisTemplate);
        lock.setLockScript(lockScript);
        lock.setUnLockScript(unLockScript);
        return lock;
    }
}
```

测试用例

接下来，终于可以上测试用例了

```
package com.crazymaker.springcloud.lock;

@Slf4j
@RunWith(SpringRunner.class)
@SpringBootTest(classes = {DemoCloudApplication.class})
// 指定启动类
public class RedisLockTest {

    @Resource
    RedisLockService redisLockService;

    private ExecutorService pool = Executors.newFixedThreadPool(10);

    @Test
    public void testLock() {
        int threads = 10;
        final int[] count = {0};
        CountDownLatch countDownLatch = new CountDownLatch(threads);
        long start = System.currentTimeMillis();
        for (int i = 0; i < threads; i++) {
            pool.submit(() ->
            {
                String lockValue = UUID.randomUUID().toString();

                try {
                    Lock lock = redisLockService.getLock("test:lock:1",
lockValue);
                    boolean locked = lock.tryLock(10, TimeUnit.SECONDS);

                    if (locked) {
                        for (int j = 0; j < 1000; j++) {
                            count[0]++;
                        }

                        log.info("count = " + count[0]);
                        lock.unlock();
                    } else {
                        System.out.println("抢锁失败");
                    }


                } catch (Exception e) {
                    e.printStackTrace();
                }
                countDownLatch.countDown();
            });
        }
        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
        System.out.println("10个线程每个累加1000为： = " + count[0]);
        //输出统计结果
        float time = System.currentTimeMillis() - start;

        System.out.println("运行的时长为(ms)：" + time);
        System.out.println("每一次执行的时长为(ms)：" + time / count[0]);

    }

}
```

执行用例，结果如下：

```
2021-05-04 23:02:11.900  INFO 22120 --- [pool-1-thread-7]
c.c.springcloud.lock.RedisLockTest       LN:50 count = 6000
2021-05-04 23:02:11.901  INFO 22120 --- [pool-1-thread-1]
c.c.springcloud.standard.lock.JedisLock  LN:81 睡眠一下，重新开始，turn:3,剩余时间:
9585
2021-05-04 23:02:11.902  INFO 22120 --- [pool-1-thread-1]
c.c.springcloud.lock.RedisLockTest       LN:50 count = 7000
2021-05-04 23:02:12.100  INFO 22120 --- [pool-1-thread-4]
c.c.springcloud.standard.lock.JedisLock  LN:81 睡眠一下，重新开始，turn:3,剩余时间:
9586
2021-05-04 23:02:12.101  INFO 22120 --- [pool-1-thread-5]
c.c.springcloud.standard.lock.JedisLock  LN:81 睡眠一下，重新开始，turn:3,剩余时间:
9585
2021-05-04 23:02:12.101  INFO 22120 --- [pool-1-thread-8]
c.c.springcloud.standard.lock.JedisLock  LN:81 睡眠一下，重新开始，turn:3,剩余时间:
9585
2021-05-04 23:02:12.101  INFO 22120 --- [pool-1-thread-4]
c.c.springcloud.lock.RedisLockTest       LN:50 count = 8000
2021-05-04 23:02:12.102  INFO 22120 --- [pool-1-thread-8]
c.c.springcloud.lock.RedisLockTest       LN:50 count = 9000
2021-05-04 23:02:12.304  INFO 22120 --- [pool-1-thread-5]
c.c.springcloud.standard.lock.JedisLock  LN:81 睡眠一下，重新开始，turn:4,剩余时间:
9383
2021-05-04 23:02:12.307  INFO 22120 --- [pool-1-thread-5]
c.c.springcloud.lock.RedisLockTest       LN:50 count = 10000
10个线程每个累加1000为： = 10000
运行的时长为(ms)：827.0
每一次执行的时长为(ms)：0.0827
```

STW导致的锁过期问题

下面有一个简单的使用锁的例子，在10秒内占着锁：

```
  //写数据到文件
function writeData(filename, data) {
    boolean locked = lock.tryLock(10, TimeUnit.SECONDS);
    if (!locked) {
        throw 'Failed to acquire lock';
```

```
    }

    try {
        //将数据写到文件
        var file = storage.readFile(filename);
        var updated = updateContents(file, data);
        storage.writeFile(filename, updated);
    } finally {
        lock.unlock();
    }
}
```

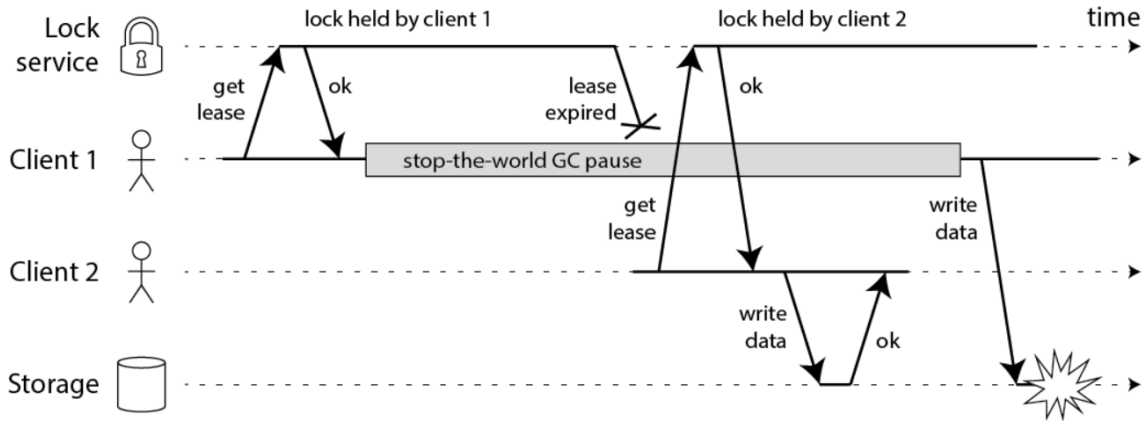
> 问题是：如果在写文件过程中，发生了 fullGC，并且其时间跨度较长， 超过了10秒， 那么，分布式就自动释放了。

在此过程中，client2 抢到锁，写了文件。

client1 的fullGC完成后，也继续写文件，**注意，此时client1 的并没有占用锁**，此时写入会导致文件数据错乱，发生线程安全问题。

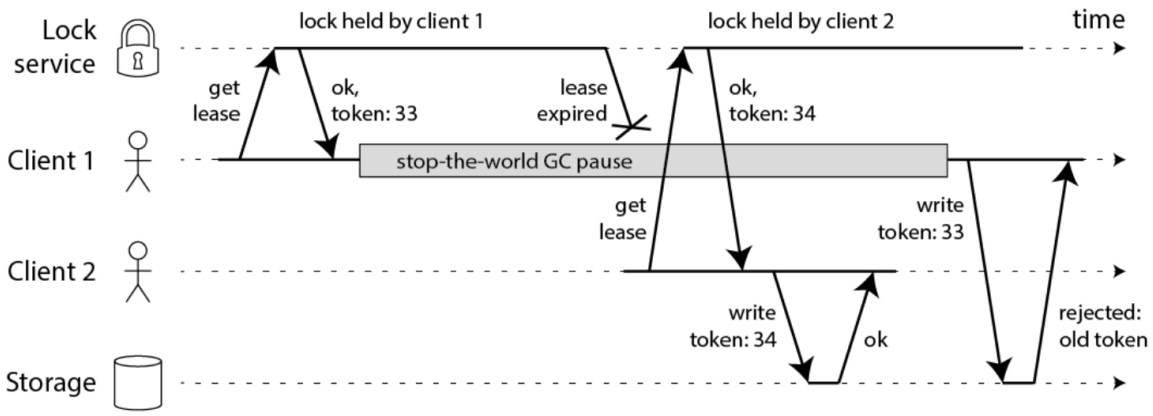
这就是STW导致的锁过期问题。

STW导致的锁过期问题，具体如下图所示：



> STW导致的锁过期问题,大概的解决方案，有：
>
> 1： 模拟CAS乐观锁的方式，增加版本号
>
> 2：**watch dog自动延期机制**

1： 模拟CAS乐观锁的方式，增加版本号（如下图中的token）



此方案如果要实现，需要调整业务逻辑，与之配合，所以会入侵代码。

2：watch dog自动延期机制

客户端1加锁的锁key默认生存时间才30秒，如果超过了30秒，客户端1还想一直持有这把锁，怎么办呢?

简单！只要客户端1一旦加锁成功，就会启动一个watch dog看门狗，**他是一个后台线程，会每隔10秒检查一下**，如果客户端1还持有锁key，那么就会不断的延长锁key的生存时间。

> redission，采用的就是这种方案， 此方案不会入侵业务代码。

注意:

单机版的watch dog 并不能解决 STW的过期问题， 需要分布式版本的 watch dog， 独立的看门狗服务。

锁删除之后， 取消看门狗服务的 对应的key记录， 当然，这就使得系统变得复杂， 还要保证看门狗服务的高并发、高可用、数据一致性的问题。

为啥推荐使用Redission

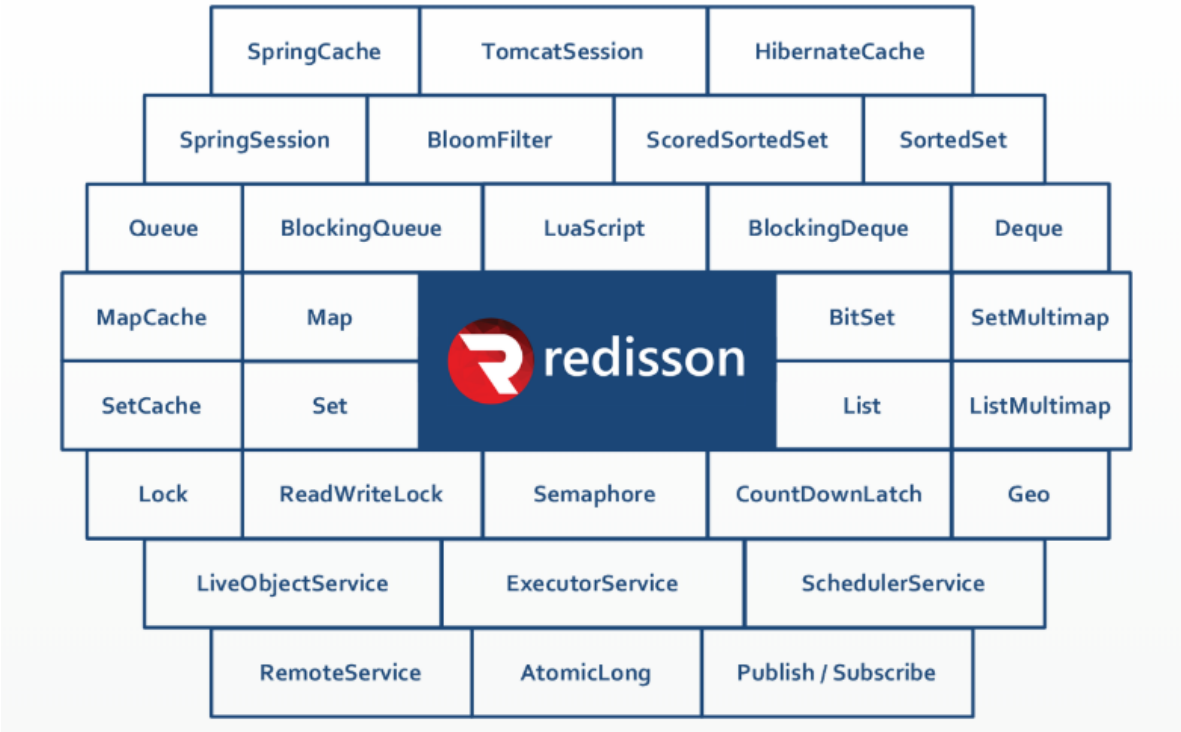
作为 Java 开发人员，我们若想在程序中集成 Redis，必须使用 Redis 的第三方库。目前大家使用的最多的第三方库是jedis。

和SpringCloud gateway一样，Redisson也是基于Netty实现的，是更高性能的第三方库。 所以，这里推荐大家使用Redission替代 jedis。

> 在使用Redission之前，建议大家先掌握Netty的知识。
>
> 推荐大家阅读被很多小伙伴评价为**史上最为易懂的NIO、Netty书籍**：《Java高并发核心编程（卷1）》

Redisson简介

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，还实现了可重入锁（Reentrant Lock）、公平锁（Fair Lock、联锁（MultiLock）、 红锁（RedLock）、 读写锁（ReadWriteLock）等，还提供了许多分布式服务。



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。

Redisson与Jedis的对比

1.概况对比

Jedis是Redis的java实现的客户端，其API提供了比较全面的的Redis命令的支持，Redisson实现了分布式和可扩展的的java数据结构，和Jedis相比，功能较为简单，不支持字符串操作，不支持排序，事物，管道，分区等Redis特性。Redisson的宗旨是促进使用者对Redis的关注分离，从而让使用者能够将精力更集中的放在处理业务逻辑上。

2.可伸缩性

Jedis使用阻塞的I/O，且其方法调用都是同步的，程序流程要等到sockets处理完I/O才能执行，不支持异步，Jedis客户端实例不是线程安全的，所以需要通过连接池来使用Jedis。

Redisson使用非阻塞的I/O和基于Netty框架的事件驱动的通信层，其方法调用时异步的。Redisson的API是线程安全的，所以操作单个Redisson连接来完成各种操作。

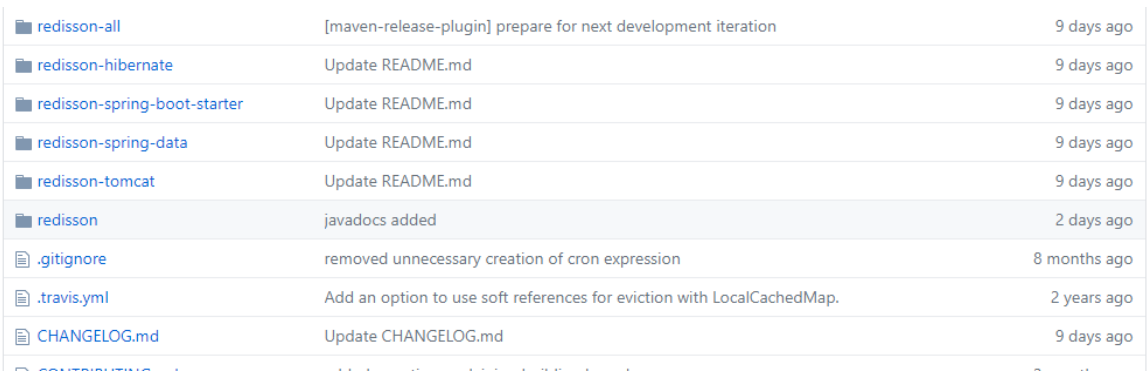
3.第三方框架整合

Redisson在Redis的基础上实现了java缓存标准规范；Redisson还提供了Spring Session回话管理器的实现。

Redission 的源码地址：

官网： https://redisson.org/

github： https://github.com/redisson/redisson#quick-start



特性 & 功能：

- 支持 Redis 单节点（single）模式、哨兵（sentinel）模式、主从（Master/Slave）模式以及集群（Redis Cluster）模式
- 程序接口调用方式采用异步执行和异步流执行两种方式
- 数据序列化，Redisson 的对象编码类是用于将对象进行序列化和反序列化，以实现对该对象在 Redis 里的读取和存储
- 单个集合数据分片，在集群模式下，Redisson 为单个 Redis 集合类型提供了自动分片的功能
- 提供多种分布式对象，如：Object Bucket，Bitset，AtomicLong，Bloom Filter 和 HyperLogLog 等
- 提供丰富的分布式集合，如：Map，Multimap，Set，SortedSet，List，Deque，Queue 等
- 分布式锁和同步器的实现，可重入锁（Reentrant Lock），公平锁（Fair Lock），联锁（MultiLock），红锁（Red Lock），信号量（Semaphonre），可过期性信号锁（PermitExpirableSemaphore）等
- 提供先进的分布式服务，如分布式远程服务（Remote Service），分布式实时对象（Live Object）服务，分布式执行服务（Executor Service），分布式调度任务服务（Schedule Service）和分布式映射归纳服务（MapReduce）

Redisson的使用

如何安装 Redisson

安装 Redisson 最便捷的方法是使用 Maven 或者 Gradle：

•Maven

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.11.4</version>
</dependency>
```

•Gradle

```
compile group: 'org.redisson', name: 'redisson', version: '3.11.4'
```

目前 Redisson 最新版是 3.11.4，当然你也可以通过搜索 Maven 中央仓库 mvnrepository[1] 来找到 Redisson 的各种版本。

获取RedissonClient对象

RedissonClient有多种模式，主要的模式有：

- 单节点模式

- 哨兵模式
- 主从模式
- 集群模式

> 首先介绍单节点模式。

单节点模式的程序化配置方法，大致如下：

```
Config config = new Config();
config.useSingleServer().setAddress("redis://myredisserver:6379");
RedissonClient redisson = Redisson.create(config);xxxxxxxxxx Config config = new
Config();config.useSingleServer().setAddress("redis://myredisserver:6379");Redis
sonClient redisson = Redisson.create(config);// connects to 127.0.0.1:6379 by
defaultRedissonClient redisson = Redisson.create();
```

```
SingleServerConfig singleConfig = config.useSingleServer();
```

SingleServerConfig类的设置参数如下：

> ### address（节点地址）
>
> 可以通过`host:port`的格式来指定节点地址。
>
> ### subscriptionConnectionMinimumIdleSize（发布和订阅连接的最小空闲连接数）
>
> 默认值：`1`
>
> 用于发布和订阅连接的最小保持连接数（长连接）。Redisson内部经常通过发布和订阅来实现许多功能。长期保持一定数量的发布订阅连接是必须的。
>
> ### subscriptionConnectionPoolSize（发布和订阅连接池大小）
>
> 默认值：`50`
>
> 用于发布和订阅连接的连接池最大容量。连接池的连接数量自动弹性伸缩。
>
> ### connectionMinimumIdleSize（最小空闲连接数）
>
> 默认值：`32`
>
> 最小保持连接数（长连接）。长期保持一定数量的连接有利于提高瞬时写入反应速度。
>
> ### connectionPoolSize（连接池大小）
>
> 默认值：`64`
>
> 连接池最大容量。连接池的连接数量自动弹性伸缩。
>
> ### dnsMonitoring（是否启用DNS监测）
>
> 默认值：`false`
>
> 在启用该功能以后，Redisson将会监测DNS的变化情况。
>
> ### dnsMonitoringInterval（DNS监测时间间隔，单位：毫秒）
>
> 默认值：`5000`

监测DNS的变化情况的时间间隔。

idleConnectionTimeout（连接空闲超时，单位：毫秒）

默认值：`10000`

如果当前连接池里的连接数量超过了最小空闲连接数，而同时有连接空闲时间超过了该数值，那么这些连接将会自动被关闭，并从连接池里去掉。时间单位是毫秒。

connectTimeout（连接超时，单位：毫秒）

默认值：`10000`

同节点建立连接时的等待超时。时间单位是毫秒。

timeout（命令等待超时，单位：毫秒）

默认值：`3000`

等待节点回复命令的时间。该时间从命令发送成功时开始计时。

retryAttempts（命令失败重试次数）

默认值：`3`

如果尝试达到 **retryAttempts（命令失败重试次数）** 仍然不能将命令发送至某个指定的节点时，将抛出错误。如果尝试在此限制之内发送成功，则开始启用 **timeout（命令等待超时）** 计时。

retryInterval（命令重试发送时间间隔，单位：毫秒）

默认值：`1500`

在一条命令发送失败以后，等待重试发送的时间间隔。时间单位是毫秒。

reconnectionTimeout（重新连接时间间隔，单位：毫秒）

默认值：`3000`

当与某个节点的连接断开时，等待与其重新建立连接的时间间隔。时间单位是毫秒。

failedAttempts（执行失败最大次数）

默认值：`3`

在某个节点执行相同或不同命令时，**连续** 失败 **failedAttempts（执行失败最大次数）** 时，该节点将被从可用节点列表里清除，直到 **reconnectionTimeout（重新连接时间间隔）** 超时以后再次尝试。

database（数据库编号）

默认值：`0`

尝试连接的数据库编号。

password（密码）

默认值：`null`

用于节点身份验证的密码。

subscriptionsPerConnection（单个连接最大订阅数量）

默认值：`5`

每个连接的最大订阅数量。

clientName（客户端名称）

默认值：`null`

在Redis节点里显示的客户端名称。

sslEnableEndpointIdentification（启用SSL终端识别）

默认值：`true`

开启SSL终端识别能力。

sslProvider（SSL实现方式）

默认值：`JDK`

确定采用哪种方式（JDK或OPENSSL）来实现SSL连接。

sslTruststore（SSL信任证书库路径）

默认值：`null`

指定SSL信任证书库的路径。

sslTruststorePassword（SSL信任证书库密码）

默认值：`null`

指定SSL信任证书库的密码。

sslKeystore（SSL钥匙库路径）

默认值：`null`

指定SSL钥匙库的路径。

sslKeystorePassword（SSL钥匙库密码）

默认值：`null`

指定SSL钥匙库的密码。

SpringBoot整合Redisson

Redisson有多种模式，首先介绍单机模式的整合。

一、导入Maven依赖

```
<!-- redisson-springboot -->
   <dependency>
       <groupId>org.redisson</groupId>
       <artifactId>redisson-spring-boot-starter</artifactId>
       <version>3.11.4</version>
   </dependency>
```

二、核心配置文件

```
spring:
  redis:
    host: 127.0.0.1
    port: 6379
    database: 0
    timeout: 5000
```

三、添加配置类

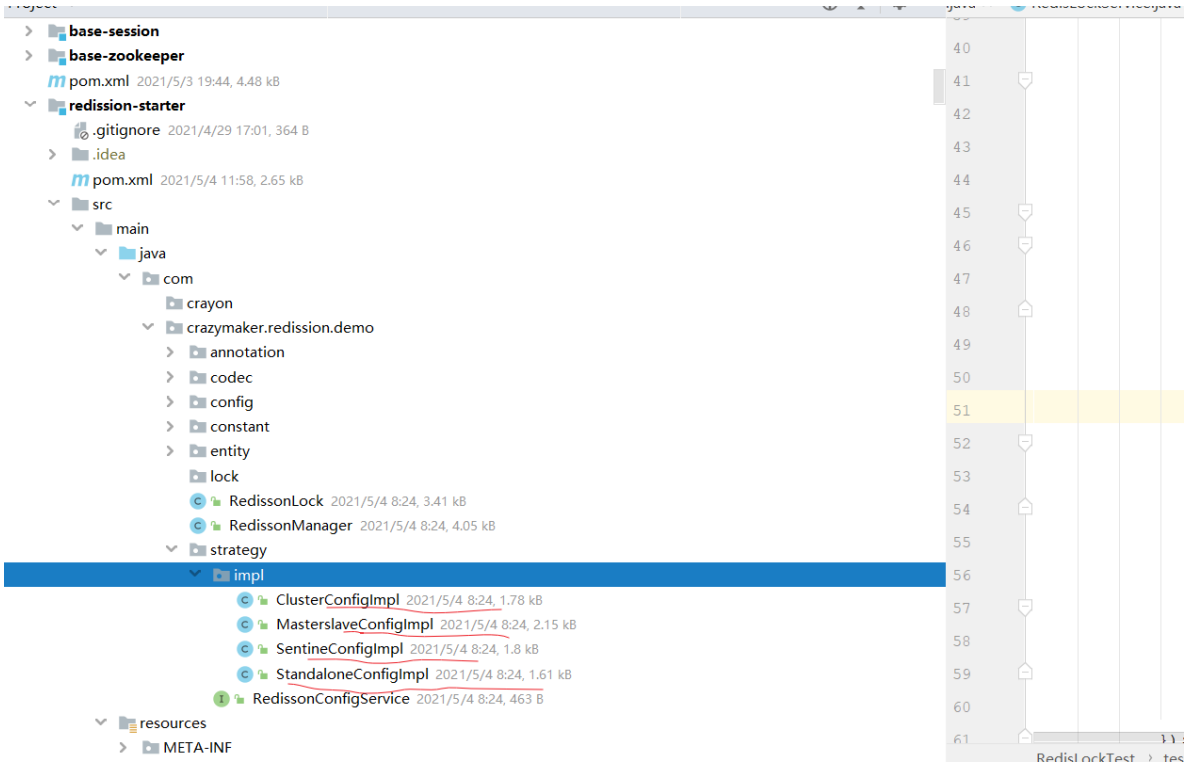
RedissonConfig.java

```
import org.redisson.Redisson;
import org.redisson.api.RedissonClient;
import org.redisson.config.Config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.data.redis.RedisProperties;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;


@Configuration
public class RedissonConfig {

    @Autowired
    private RedisProperties redisProperties;

    @Bean
    public RedissonClient redissonClient() {
        Config config = new Config();
        String redisUrl = String.format("redis://%s:%s",
redisProperties.getHost() + "", redisProperties.getPort() + "");

config.useSingleServer().setAddress(redisUrl).setPassword(redisProperties.getPas
sword());
        config.useSingleServer().setDatabase(3);
        return Redisson.create(config);
    }

}
```

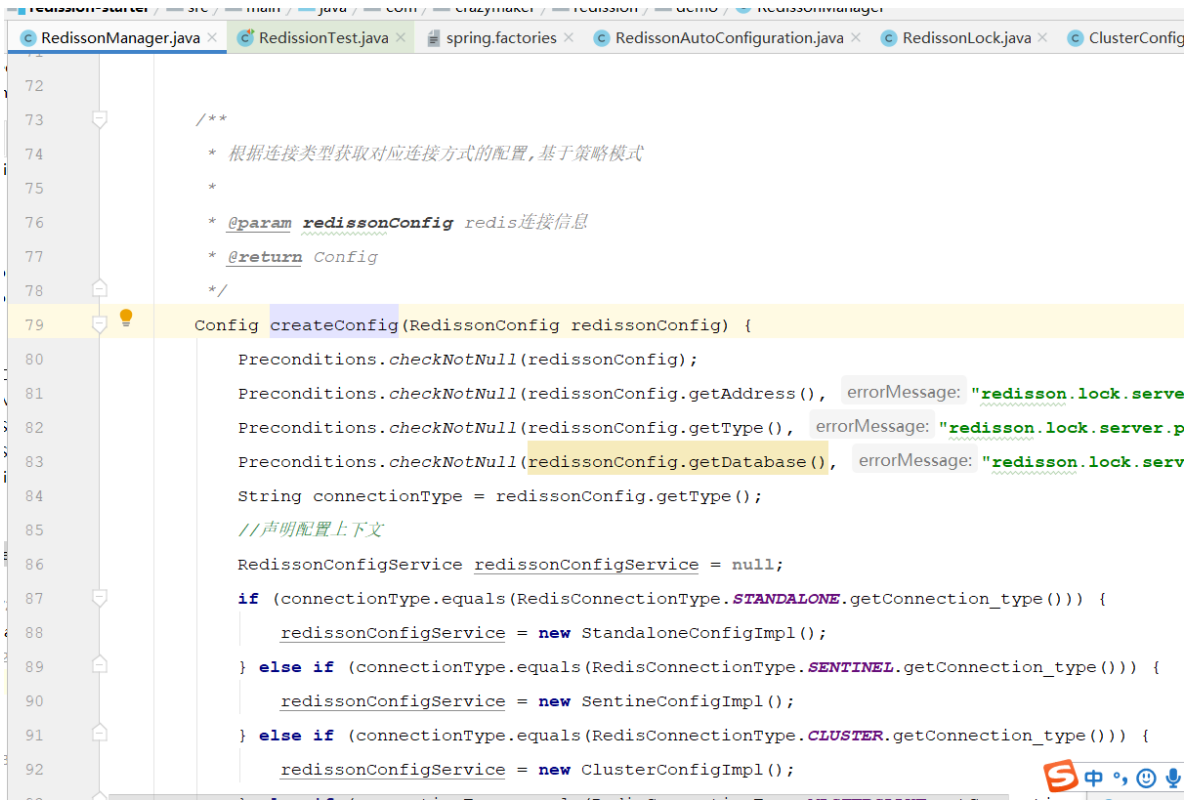
###

自定义starter

由于redission可以有多种模式，处于学习的目的，将多种模式封装成一个start，可以学习一下starter的制作。



封装一个RedissonManager，通过策略模式，根据不同的配置类型，创建 RedissionConfig实例，然后创建RedissionClient对象。



使用RBucket操作分布式对象

Redission模拟了Java的面向对象编程思想，可以简单理解为一切皆为对象。

每一个 Redisson 对象 实现了RObject and RExpirable 两个interfaces.

Usage example:

```
RObject object = redisson.get...()

object.sizeInMemory();

object.delete();

object.rename("newname");

object.isExists();

// catch expired event
object.addListener(new ExpiredObjectListener() {
   ...
});

// catch delete event
object.addListener(new DeletedObjectListener() {
   ...
});
```

每一个Redisson 对象的名字，就是 Redis中的 Key.

```
RMap map = redisson.getMap("mymap");
map.getName(); // = mymap
```

可以通过 RKeys 接口操作Redis中的keys.

Usage example:

```
RKeys keys = redisson.getKeys();

Iterable<String> allKeys = keys.getKeys();

Iterable<String> foundedKeys = keys.getKeysByPattern('key*');

long numOfDeletedKeys = keys.delete("obj1", "obj2", "obj3");

long deletedKeysAmount = keys.deleteByPattern("test?");

String randomKey = keys.randomKey();

long keysAmount = keys.count();

keys.flushall();

keys.flushdb();
```

Redisson通过RBucket接口代表可以访问任何类型的基础对象，或者普通对象。

> RBucket有一系列的工具方法，如compareAndSet()，get()，getAndDelete()，getAndSet()，set()，size()，trySet()等等，用于设值/取值/获取尺寸。

RBucket普通对象的最大大小，为512兆字节。

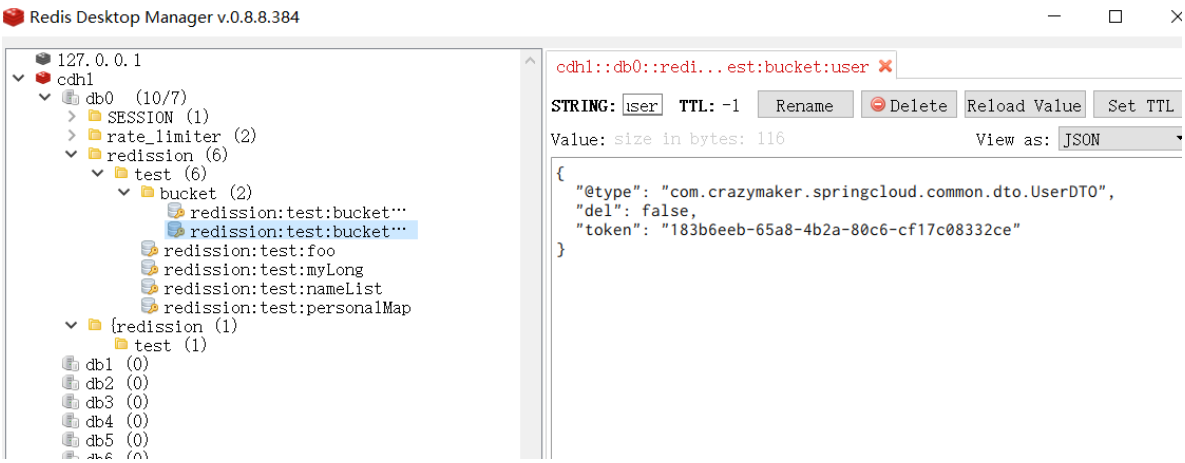
```
RBucket<AnyObject> bucket = redisson.getBucket("anyObject");

bucket.set(new AnyObject(1));
AnyObject obj = bucket.get();

bucket.trySet(new AnyObject(3));
bucket.compareAndSet(new AnyObject(4), new AnyObject(5));
bucket.getAndSet(new AnyObject(6));
```

下面是一个完整的实例：

```
public class RedissionTest {

    @Resource
    RedissonManager redissonManager;

   @Test
    public void testRBucketExamples() {
        // 默认连接上 127.0.0.1:6379
        RedissonClient client = redissonManager.getRedisson();
        // RList 继承了 java.util.List 接口
        RBucket<String> rstring  =
client.getBucket("redission:test:bucket:string");
        rstring.set("this is a string");

        RBucket<UserDTO> ruser  =
client.getBucket("redission:test:bucket:user");
        UserDTO dto = new UserDTO();
        dto.setToken(UUID.randomUUID().toString());
        ruser.set(dto);
        System.out.println("string is: " + rstring.get());
        System.out.println("dto is: " + ruser.get());

        client.shutdown();
    }


}
```

运行上面的代码时，可以获得以下输出：

```
string is: this is a string
dto is: UserDTO(id=null, userId=null, username=null, password=null,
nickname=null, token=183b6eeb-65a8-4b2a-80c6-cf17c08332ce, createTime=null,
updateTime=null, headImgUrl=null, mobile=null, sex=null, enabled=null,
type=null, openId=null, isDel=false)
```



使用 RList 操作 Redis 列表

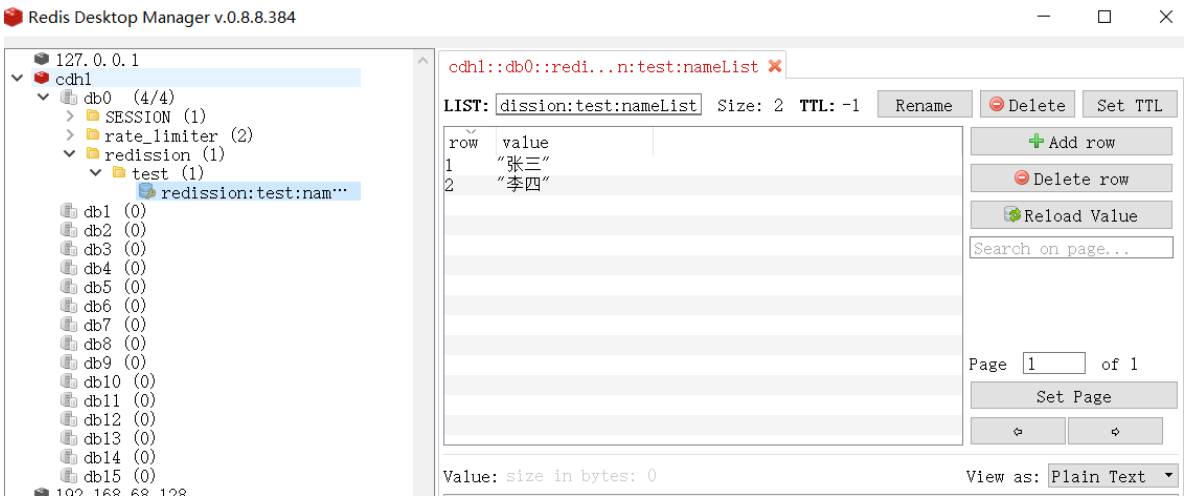
下面的代码简单演示了如何在 Redisson 中使用 `RList` 对象。`RList` 是 Java 的 List 集合的分布式并发实现。

考虑以下代码：

```
public class RedissionTest {

    @Resource
    RedissonManager redissonManager;

    @Test
    public void testListExamples() {
        // 默认连接上 127.0.0.1:6379
        RedissonClient client = redissonManager.getRedisson();
        // RList 继承了 java.util.List 接口
        RList<String> nameList = client.getList("redission:test:nameList");
        nameList.clear();
        nameList.add("张三");
        nameList.add("李四");
        nameList.add("王五");
        nameList.remove(-1);

        System.out.println("List size: " + nameList.size());


        boolean contains = nameList.contains("李四");
        System.out.println("Is list contains name '李四': " + contains);
        nameList.forEach(System.out::println);

        client.shutdown();
    }


}
```

运行上面的代码时，可以获得以下输出：

```
List size: 2
Is list contains name '李四': true
张三
李四
```



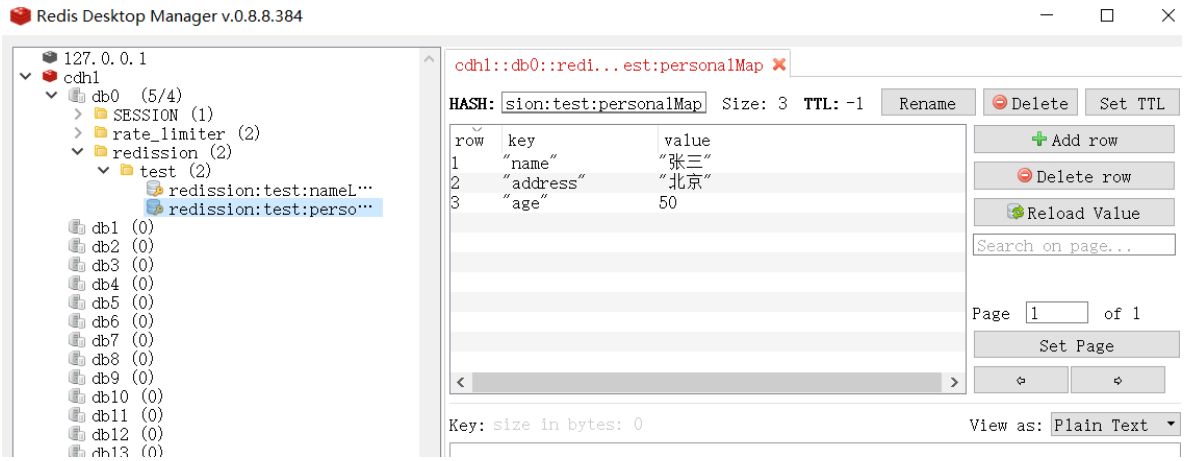
使用 RMap 操作 Redis 哈希

Redisson 还包括 RMap，它是 Java Map 集合的分布式并发实现，考虑以下代码：

```
public class RedissionTest {

    @Resource
    RedissonManager redissonManager;

    @Test
    public void testListExamples() {
         // 默认连接上 127.0.0.1:6379
        RedissonClient client = redissonManager.getRedisson();
        // RMap 继承了 java.util.concurrent.ConcurrentMap 接口
        RMap<String, Object> map = client.getMap("redission:test:personalMap");
        map.put("name", "张三");
        map.put("address", "北京");
        map.put("age", new Integer(50));

        System.out.println("Map size: " + map.size());

        boolean contains = map.containsKey("age");
        System.out.println("Is map contains key 'age': " + contains);
        String value = String.valueOf(map.get("name"));
        System.out.println("Value mapped by key 'name': " + value);

        client.shutdown();
    }


}
```

运行上面的代码时，将会看到以下输出：

```
Map size: 3
Is map contains key 'age': true
Value mapped by key 'name': 张三
```



执行 Lua脚本

Lua是一种开源、简单易学、轻量小巧的脚本语言，用标准C语言编写。

其设计的目的就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。

Redis从2.6版本开始支持Lua脚本，Redis使用Lua可以：

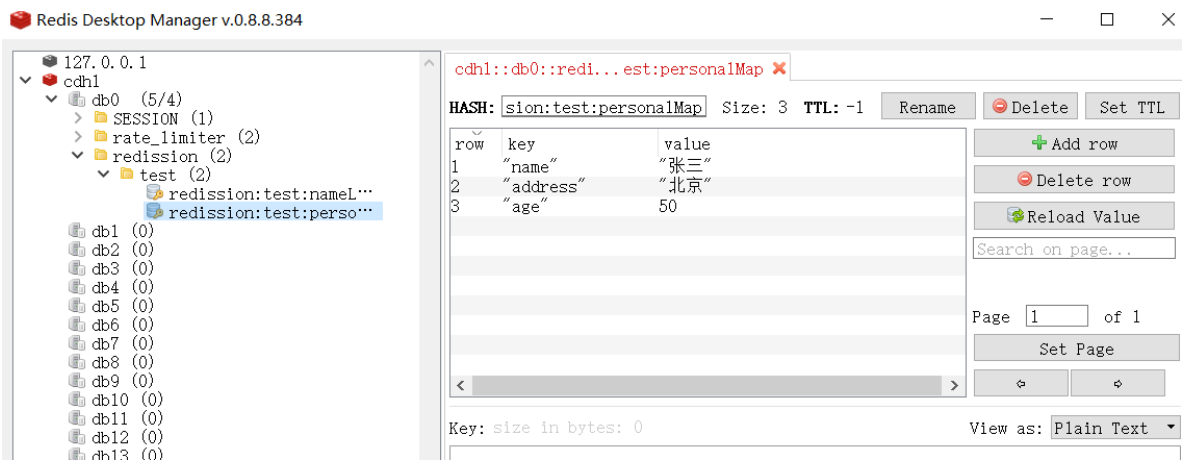
1. 原子操作。Redis会将整个脚本作为一个整体执行，不会被中断。可以用来批量更新、批量插入
2. 减少网络开销。多个Redis操作合并为一个脚本，减少网络时延
3. 代码复用。客户端发送的脚本可以存储在Redis中，其他客户端可以根据脚本的id调用。

```
public class RedissionTest {

    @Resource
    RedissonManager redissonManager;

    @Test
    public void testLuaExamples() {
        // 默认连接上 127.0.0.1:6379
        RedissonClient redisson = redissonManager.getRedisson();

        redisson.getBucket("redission:test:foo").set("bar");
        String r = redisson.getScript().eval(RScript.Mode.READ_ONLY,
                "return redis.call('get', 'redission:test:foo')",
RScript.ReturnType.VALUE);
        System.out.println("foo: " + r);

        // 通过预存的脚本进行同样的操作
        RScript s = redisson.getScript();
        // 首先将脚本加载到Redis
```

```
        String sha1 = s.scriptLoad("return redis.call('get',
'redission:test:foo')");
        // 返回值 res == 282297a0228f48cd3fc6a55de6316f31422f5d17
        System.out.println("sha1: " + sha1);
        // 再通过SHA值调用脚本
        Future<Object> r1 =
redisson.getScript().evalShaAsync(RScript.Mode.READ_ONLY,
                sha1,
                RScript.ReturnType.VALUE,
                Collections.emptyList());
        try {
            System.out.println("res: " + r1.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        client.shutdown();
    }
}
```

运行上面的代码时，将会看到以下输出：

```
foo: bar
sha1: 282297a0228f48cd3fc6a55de6316f31422f5d17
res: bar
```



使用 RLock 实现 Redis 分布式锁

RLock 是 Java 中可重入锁的分布式实现，下面的代码演示了 RLock 的用法：

```
public class RedissionTest {

    @Resource
    RedissonManager redissonManager;

 @Test
    public void testLockExamples() {
        // 默认连接上 127.0.0.1:6379
```

```
        RedissonClient redisson = redissonManager.getRedisson();
        // RLock 继承了 java.util.concurrent.locks.Lock 接口
        RLock lock = redisson.getLock("redission:test:lock:1");

        final int[] count = {0};
        int threads = 10;
        ExecutorService pool = Executors.newFixedThreadPool(10);
        CountDownLatch countDownLatch = new CountDownLatch(threads);
        long start = System.currentTimeMillis();
        for (int i = 0; i < threads; i++) {
            pool.submit(() ->
            {
                for (int j = 0; j < 1000; j++) {
                    lock.lock();

                    count[0]++;
                    lock.unlock();
                }
                countDownLatch.countDown();
            });
        }

        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("10个线程每个累加1000为：  = " + count[0]);
        //输出统计结果
        float time = System.currentTimeMillis() - start;

        System.out.println("运行的时长为：" + time);
        System.out.println("每一次执行的时长为：" + time/count[0]);
    }

}
```

此代码将产生以下输出：

```
10个线程每个累加1000为：  = 10000
运行的时长为：14172.0
每一次执行的时长为：1.4172
```

使用 RAtomicLong 实现 Redis 原子操作

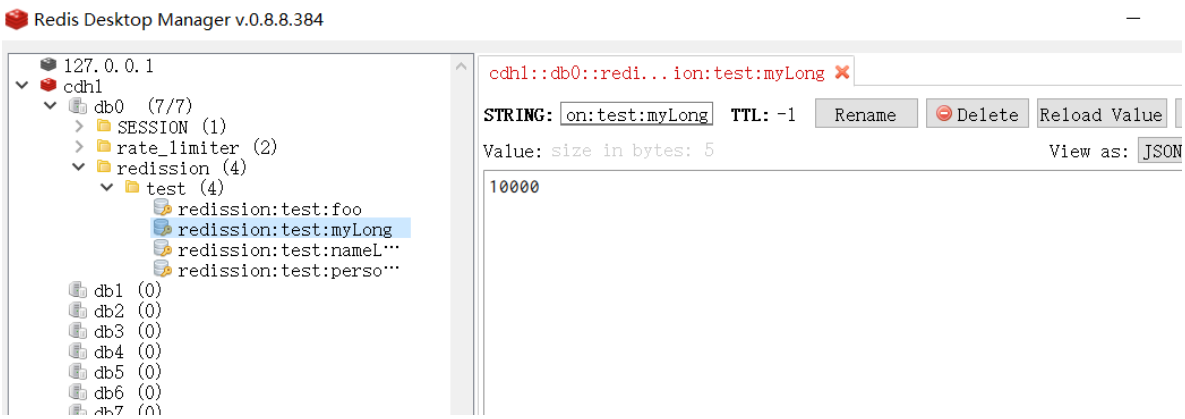
RAtomicLong 是 Java 中 AtomicLong 类的分布式“替代品”，用于在并发环境中保存长值。以下示例代码演示了 RAtomicLong 的用法：

```
public class RedissionTest {

    @Resource
    RedissonManager redissonManager;

    @Test
```

```
    public void testRAtomicLongExamples() {
        // 默认连接上 127.0.0.1:6379
        RedissonClient redisson = redissonManager.getRedisson();
        RAtomicLong atomicLong =
redisson.getAtomicLong("redission:test:myLong");
          // 线程数
        final int threads = 10;
        // 每条线程的执行轮数
        final int turns = 1000;
        ExecutorService pool = Executors.newFixedThreadPool(threads);
        for (int i = 0; i < threads; i++)
        {
            pool.submit(() ->
            {
                try
                {
                    for (int j = 0; j < turns; j++)
                    {
                        atomicLong.incrementAndGet();
                    }

                } catch (Exception e)
                {
                    e.printStackTrace();
                }
            });
        }

        ThreadUtil.sleepSeconds(5);
        System.out.println("atomicLong: " + atomicLong.get());
        redisson.shutdown();
    }

}
```

此代码的输出将是：

```
atomicLong: 10000
```



整长型累加器（LongAdder）

基于Redis的Redisson分布式整长型累加器（LongAdder）采用了与java.util.concurrent.atomic.LongAdder类似的接口。通过利用客户端内置的LongAdder对象，为分布式环境下递增和递减操作提供了很高得性能。据统计其性能最高比分布式`AtomicLong`对象快 **12000** 倍。

完美适用于分布式统计计量场景。下面是RLongAdder的使用案例：

```
RLongAdder atomicLong = redisson.getLongAdder("myLongAdder");
atomicLong.add(12);
atomicLong.increment();
atomicLong.decrement();
atomicLong.sum();
```

以下示例代码演示了 RLongAdder 的用法：

```
public class RedissionTest {

    @Resource
    RedissonManager redissonManager;

    @Test
    public void testRAtomicLongExamples() {
        // 默认连接上 127.0.0.1:6379
        RedissonClient redisson = redissonManager.getRedisson();
        RAtomicLong atomicLong =
redisson.getAtomicLong("redission:test:myLong");
          // 线程数
        final int threads = 10;
        // 每条线程的执行轮数
        final int turns = 1000;
        ExecutorService pool = Executors.newFixedThreadPool(threads);
        for (int i = 0; i < threads; i++)
        {
            pool.submit(() ->
            {
                try
                {
                    for (int j = 0; j < turns; j++)
                    {
                        atomicLong.incrementAndGet();
                    }

                } catch (Exception e)
                {
                    e.printStackTrace();
                }
            });
        }

        ThreadUtil.sleepSeconds(5);
        System.out.println("atomicLong: " + atomicLong.get());
        redisson.shutdown();
    }

}
```

此代码将产生以下输出：

```
longAdder: 10000
运行的时长为：5085.0
每一次执行的时长为：0.5085
```

当不再使用整长型累加器对象的时候应该自行手动销毁，如果Redisson对象被关闭（shutdown）了，则不用手动销毁。

```
RLongAdder atomicLong = ...
atomicLong.destroy();
```

序列化

Redisson的对象编码类是用于将对象进行序列化和反序列化，以实现对该对象在Redis里的读取和存储。Redisson提供了以下几种的对象编码应用，以供大家选择：

编码类名称	说明
`org.redisson.codec.JsonJacksonCodec`	Jackson JSON 编码 **默认编码**
`org.redisson.codec.AvroJacksonCodec`	Avro 一个二进制的JSON编码
`org.redisson.codec.SmileJacksonCodec`	Smile 另一个二进制的JSON编码
`org.redisson.codec.CborJacksonCodec`	CBOR 又一个二进制的JSON编码
`org.redisson.codec.MsgPackJacksonCodec`	MsgPack 再来一个二进制的JSON编码
`org.redisson.codec.IonJacksonCodec`	Amazon Ion 亚马逊的Ion编码，格式与JSON类似
`org.redisson.codec.KryoCodec`	Kryo 二进制对象序列化编码

编码类名称	说明
`org.redisson.codec.SerializationCodec`	JDK序列化编码
`org.redisson.codec.FstCodec`	FST 10倍于JDK序列化性能而且100%兼容的编码
`org.redisson.codec.LZ4Codec`	LZ4 压缩型序列化对象编码
`org.redisson.codec.SnappyCodec`	Snappy 另一个压缩型序列化对象编码
`org.redisson.client.codec.JsonJacksonMapCodec`	基于Jackson的映射类使用的编码。可用于避免序列化类的信息，以及用于解决使用 `byte[]` 遇到的问题。
`org.redisson.client.codec.StringCodec`	纯字符串编码（无转换）
`org.redisson.client.codec.LongCodec`	纯整长型数字编码（无转换）
`org.redisson.client.codec.ByteArrayCodec`	字节数组编码
`org.redisson.codec.CompositeCodec`	用来组合多种不同编码在一起

由Redisson默认的编码器为二进制编码器，为了序列化后的内容可见，需要使用Json文本序列化编码工具类。Redisson提供了编码器 JsonJacksonCodec，作为Json文本序列化编码工具类。

> 问题是：JsonJackson在序列化有双向引用的对象时，会出现无限循环异常。而fastjson在检查出双向引用后会自动用引用符$ref替换，终止循环。

所以，一些特殊场景中：用fastjson能 正常序列化到redis，而JsonJackson则抛出无限循环异常。

为了序列化后的内容可见，所以不用redission其他自带的，自行实现fastjson编码器：

```
package com.crayon.distributedredissionspringbootstarter.codec;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.serializer.SerializerFeature;
import io.netty.buffer.ByteBuf;
import io.netty.buffer.ByteBufAllocator;
import io.netty.buffer.ByteBufInputStream;
import io.netty.buffer.ByteBufOutputStream;
import org.redisson.client.codec.BaseCodec;
import org.redisson.client.protocol.Decoder;
import org.redisson.client.protocol.Encoder;

import java.io.IOException;

public class FastjsonCodec extends BaseCodec {

    private final Encoder encoder = in -> {
        ByteBuf out = ByteBufAllocator.DEFAULT.buffer();
        try {
            ByteBufOutputStream os = new ByteBufOutputStream(out);
            JSON.writeJSONString(os, in, SerializerFeature.WriteClassName);
            return os.buffer();
```

```
        } catch (IOException e) {
            out.release();
            throw e;
        } catch (Exception e) {
            out.release();
            throw new IOException(e);
        }
    };

    private final Decoder<Object> decoder = (buf, state) ->
            JSON.parseObject(new ByteBufInputStream(buf), Object.class);

    @Override
    public Decoder<Object> getValueDecoder() {
        return decoder;
    }

    @Override
    public Encoder getValueEncoder() {
        return encoder;
    }
}
```

替换的方法如下：

```
 */
@Slf4j
public class StandaloneConfigImpl implements RedissonConfigService {

    @Override
    public Config createRedissonConfig(RedissonConfig redissonConfig) {
        Config config = new Config();
        try {
            String address = redissonConfig.getAddress();
            String password = redissonConfig.getPassword();
            int database = redissonConfig.getDatabase();
            String redisAddr =
GlobalConstant.REDIS_CONNECTION_PREFIX.getConstant_value() + address;
            config.useSingleServer().setAddress(redisAddr);
            config.useSingleServer().setDatabase(database);
            //密码可以为空
            if (!StringUtils.isEmpty(password)) {
                config.useSingleServer().setPassword(password);
            }
            log.info("初始化[单机部署]方式Config,redisAddress:" + address);

//            config.setCodec( new FstCodec());
            config.setCodec( new FastjsonCodec());
        } catch (Exception e) {
            log.error("单机部署 Redisson init error", e);
        }
        return config;
    }
}
```

哨兵模式

哨兵模式即sentinel模式，配置Redis哨兵服务的官方文档在这里。

> 哨兵模式实现代码和单机模式几乎一样，唯一的不同就是Config的构造.

程序化配置哨兵模式的方法如下:

```
Config config = new Config();
config.useSentinelServers()
    .setMasterName("mymaster")
    // use "rediss://" for SSL connection
    .addSentinelAddress("redis://127.0.0.1:26389", "redis://127.0.0.1:26379")
    .addSentinelAddress("redis://127.0.0.1:26319");

RedissonClient redisson = Redisson.create(config);
```

Redisson的哨兵模式的使用方法如下:

```
SentinelServersConfig sentinelConfig = config.useSentinelServers();
```

SentinelServersConfig配置参数如下:

> 配置Redis哨兵服务的官方文档在这里。Redisson的哨兵模式的使用方法如下:
> `SentinelServersConfig sentinelConfig = config.useSentinelServers();`
>
> `SentinelServersConfig` 类的设置参数如下:
>
> ### dnsMonitoringInterval（DNS监控间隔，单位：毫秒）
>
> 默认值: `5000`
>
> 用来指定检查节点DNS变化的时间间隔。使用的时候应该确保JVM里的DNS数据的缓存时间保持在足够低的范围才有意义。用`-1`来禁用该功能。
>
> ### masterName（主服务器的名称）
>
> 主服务器的名称是哨兵进程中用来监测主从服务切换情况的。
>
> ### addSentinelAddress（添加哨兵节点地址）
>
> 可以通过`host:port`的格式来指定哨兵节点的地址。多个节点可以一次性批量添加。
>
> ### readMode（读取操作的负载均衡模式）
>
> 默认值: `SLAVE`（只在从服务节点里读取）
>
> 注：在从服务节点里读取的数据说明已经至少有两个节点保存了该数据，确保了数据的高可用性。
>
> 设置读取操作选择节点的模式。可用值为: `SLAVE` - 只在从服务节点里读取。`MASTER` - 只在主服务节点里读取。`MASTER_SLAVE` - 在主从服务节点里都可以读取。
>
> ### subscriptionMode（订阅操作的负载均衡模式）

默认值：`SLAVE`（只在从服务节点里订阅）

设置订阅操作选择节点的模式。可用值为：`SLAVE` - 只在从服务节点里订阅。`MASTER` - 只在主服务节点里订阅。

loadBalancer（负载均衡算法类的选择）

默认值：`org.redisson.connection.balancer.RoundRobinLoadBalancer`

在使用多个Redis服务节点的环境里，可以选用以下几种负载均衡方式选择一个节点：
`org.redisson.connection.balancer.WeightedRoundRobinBalancer` - 权重轮询调度算法
`org.redisson.connection.balancer.RoundRobinLoadBalancer` - 轮询调度算法
`org.redisson.connection.balancer.RandomLoadBalancer` - 随机调度算法

subscriptionConnectionMinimumIdleSize（从节点发布和订阅连接的最小空闲连接数）

默认值：`1`

多从节点的环境里，**每个** 从服务节点里用于发布和订阅连接的最小保持连接数（长连接）。Redisson内部经常通过发布和订阅来实现许多功能。长期保持一定数量的发布订阅连接是必须的。

subscriptionConnectionPoolSize（从节点发布和订阅连接池大小）

默认值：`50`

多从节点的环境里，**每个** 从服务节点里用于发布和订阅连接的连接池最大容量。连接池的连接数量自动弹性伸缩。

slaveConnectionMinimumIdleSize（从节点最小空闲连接数）

默认值：`32`

多从节点的环境里，**每个** 从服务节点里用于普通操作（**非** 发布和订阅）的最小保持连接数（长连接）。长期保持一定数量的连接有利于提高瞬时读取反映速度。

slaveConnectionPoolSize（从节点连接池大小）

默认值：`64`

多从节点的环境里，**每个** 从服务节点里用于普通操作（**非** 发布和订阅）连接的连接池最大容量。连接池的连接数量自动弹性伸缩。

masterConnectionMinimumIdleSize（主节点最小空闲连接数）

默认值：`32`

多从节点的环境里，**每个** 主节点的最小保持连接数（长连接）。长期保持一定数量的连接有利于提高瞬时写入反应速度。

masterConnectionPoolSize（主节点连接池大小）

默认值：`64`

主节点的连接池最大容量。连接池的连接数量自动弹性伸缩。

idleConnectionTimeout（连接空闲超时，单位：毫秒）

默认值：`10000`

如果当前连接池里的连接数量超过了最小空闲连接数，而同时有连接空闲时间超过了该数值，那么这些连接将会自动被关闭，并从连接池里去掉。时间单位是毫秒。

connectTimeout（连接超时，单位：毫秒）

默认值：`10000`

同任何节点建立连接时的等待超时。时间单位是毫秒。

timeout（命令等待超时，单位：毫秒）

默认值：`3000`

等待节点回复命令的时间。该时间从命令发送成功时开始计时。

retryAttempts（命令失败重试次数）

默认值：`3`

如果尝试达到 **retryAttempts（命令失败重试次数）** 仍然不能将命令发送至某个指定的节点时，将抛出错误。如果尝试在此限制之内发送成功，则开始启用 **timeout（命令等待超时）** 计时。

retryInterval（命令重试发送时间间隔，单位：毫秒）

默认值：`1500`

在一条命令发送失败以后，等待重试发送的时间间隔。时间单位是毫秒。

reconnectionTimeout（重新连接时间间隔，单位：毫秒）

默认值：`3000`

当与某个节点的连接断开时，等待与其重新建立连接的时间间隔。时间单位是毫秒。

failedAttempts（执行失败最大次数）

默认值：`3`

在某个节点执行相同或不同命令时，**连续** 失败 **failedAttempts（执行失败最大次数）** 时，该节点将被从可用节点列表里清除，直到 **reconnectionTimeout（重新连接时间间隔）** 超时以后再次尝试。

database（数据库编号）

默认值：`0`

尝试连接的数据库编号。

password（密码）

默认值：`null`

用于节点身份验证的密码。

subscriptionsPerConnection（单个连接最大订阅数量）

默认值：`5`

每个连接的最大订阅数量。

clientName（客户端名称）

默认值：`null`

在Redis节点里显示的客户端名称。

sslEnableEndpointIdentification（启用SSL终端识别）

默认值：`true`

开启SSL终端识别能力。

sslProvider（SSL实现方式）

默认值：`JDK`

确定采用哪种方式（JDK或OPENSSL）来实现SSL连接。

sslTruststore（SSL信任证书库路径）

默认值：`null`

指定SSL信任证书库的路径。

sslTruststorePassword（SSL信任证书库密码）

默认值：`null`

指定SSL信任证书库的密码。

sslKeystore（SSL钥匙库路径）

默认值：`null`

指定SSL钥匙库的路径。

sslKeystorePassword（SSL钥匙库密码）

默认值：`null`

指定SSL钥匙库的密码。

通过属性文件，配置的示例如下：

```
---
sentinelServersConfig:
idleConnectionTimeout: 10000
connectTimeout: 10000
timeout: 3000
retryAttempts: 3
retryInterval: 1500
failedSlaveReconnectionInterval: 3000
failedSlaveCheckInterval: 60000
password: null
subscriptionsPerConnection: 5
clientName: null
loadBalancer: !<org.redisson.connection.balancer.RoundRobinLoadBalancer> {}
subscriptionConnectionMinimumIdleSize: 1
subscriptionConnectionPoolSize: 50
slaveConnectionMinimumIdleSize: 24
slaveConnectionPoolSize: 64
```

```
masterConnectionMinimumIdleSize: 24
masterConnectionPoolSize: 64
readMode: "SLAVE"
subscriptionMode: "SLAVE"
sentinelAddresses:
  - "redis://127.0.0.1:26379"
  - "redis://127.0.0.1:26389"
masterName: "mymaster"
database: 0
threads: 16
nettyThreads: 32
codec: !<org.redisson.codec.MarshallingCodec> {}
transportMode: "NIO"
```

主从模式

介绍配置Redis主从服务组态的文档在这里.

程序化配置主从模式的方法如下:

```
Config config = new Config();
config.useMasterSlaveServers()
    // use "rediss://" for SSL connection
    .setMasterAddress("redis://127.0.0.1:6379")
    .addSlaveAddress("redis://127.0.0.1:6389", "redis://127.0.0.1:6332",
"redis://127.0.0.1:6419")
    .addSlaveAddress("redis://127.0.0.1:6399");

RedissonClient redisson = Redisson.create(config);
```

主从模式使用到MasterSlaveServersConfig：

MasterSlaveServersConfig masterSlaveConfig = config.useMasterSlaveServers();

`MasterSlaveServersConfig` **类的设置参数如下：**

dnsMonitoringInterval（DNS监控间隔，单位：毫秒）

默认值：`5000`

用来指定检查节点DNS变化的时间间隔。使用的时候应该确保JVM里的DNS数据的缓存时间保持在足够低的范围才有意义。用`-1`来禁用该功能。

masterAddress（主节点地址）

可以通过`host:port`的格式来指定主节点地址。

addSlaveAddress（添加从主节点地址）

可以通过`host:port`的格式来指定从节点的地址。多个节点可以一次性批量添加。

readMode（读取操作的负载均衡模式）

默认值：`SLAVE`（只在从服务节点里读取）

注：在从服务节点里读取的数据说明已经至少有两个节点保存了该数据，确保了数据的高可用性。

设置读取操作选择节点的模式。可用值为：`SLAVE` - 只在从服务节点里读取。`MASTER` - 只在主服务节点里读取。`MASTER_SLAVE` - 在主从服务节点里都可以读取。

subscriptionMode（订阅操作的负载均衡模式）

默认值：`SLAVE`（只在从服务节点里订阅）

设置订阅操作选择节点的模式。可用值为：`SLAVE` - 只在从服务节点里订阅。`MASTER` - 只在主服务节点里订阅。

loadBalancer（负载均衡算法类的选择）

默认值：`org.redisson.connection.balancer.RoundRobinLoadBalancer`

在使用多个Redis服务节点的环境里，可以选用以下几种负载均衡方式选择一个节点：
`org.redisson.connection.balancer.WeightedRoundRobinBalancer` - 权重轮询调度算法
`org.redisson.connection.balancer.RoundRobinLoadBalancer` - 轮询调度算法
`org.redisson.connection.balancer.RandomLoadBalancer` - 随机调度算法

subscriptionConnectionMinimumIdleSize（从节点发布和订阅连接的最小空闲连接数）

默认值：`1`

多从节点的环境里，**每个** 从服务节点里用于发布和订阅连接的最小保持连接数（长连接）。Redisson内部经常通过发布和订阅来实现许多功能。长期保持一定数量的发布订阅连接是必须的。

subscriptionConnectionPoolSize（从节点发布和订阅连接池大小）

默认值：`50`

多从节点的环境里，**每个** 从服务节点里用于发布和订阅连接的连接池最大容量。连接池的连接数量自动弹性伸缩。

slaveConnectionMinimumIdleSize（从节点最小空闲连接数）

默认值：`32`

多从节点的环境里，**每个** 从服务节点里用于普通操作（**非** 发布和订阅）的最小保持连接数（长连接）。长期保持一定数量的连接有利于提高瞬时读取反映速度。

slaveConnectionPoolSize（从节点连接池大小）

默认值：`64`

多从节点的环境里，**每个** 从服务节点里用于普通操作（**非** 发布和订阅）连接的连接池最大容量。连接池的连接数量自动弹性伸缩。

masterConnectionMinimumIdleSize（主节点最小空闲连接数）

默认值：`32`

多从节点的环境里，**每个** 主节点的最小保持连接数（长连接）。长期保持一定数量的连接有利于提高瞬时写入反应速度。

masterConnectionPoolSize（主节点连接池大小）

默认值：`64`

主节点的连接池最大容量。连接池的连接数量自动弹性伸缩。

idleConnectionTimeout（连接空闲超时，单位：毫秒）

默认值：`10000`

如果当前连接池里的连接数量超过了最小空闲连接数，而同时有连接空闲时间超过了该数值，那么这些连接将会自动被关闭，并从连接池里去掉。时间单位是毫秒。

connectTimeout（连接超时，单位：毫秒）

默认值：`10000`

同任何节点建立连接时的等待超时。时间单位是毫秒。

timeout（命令等待超时，单位：毫秒）

默认值：`3000`

等待节点回复命令的时间。该时间从命令发送成功时开始计时。

retryAttempts（命令失败重试次数）

默认值：`3`

如果尝试达到 **retryAttempts（命令失败重试次数）** 仍然不能将命令发送至某个指定的节点时，将抛出错误。如果尝试在此限制之内发送成功，则开始启用 **timeout（命令等待超时）** 计时。

retryInterval（命令重试发送时间间隔，单位：毫秒）

默认值：`1500`

在一条命令发送失败以后，等待重试发送的时间间隔。时间单位是毫秒。

reconnectionTimeout（重新连接时间间隔，单位：毫秒）

默认值：`3000`

当与某个节点的连接断开时，等待与其重新建立连接的时间间隔。时间单位是毫秒。

failedAttempts（执行失败最大次数）

默认值：`3`

在某个节点执行相同或不同命令时，**连续** 失败 **failedAttempts（执行失败最大次数）** 时，该节点将被从可用节点列表里清除，直到 **reconnectionTimeout（重新连接时间间隔）** 超时以后再次尝试。

database（数据库编号）

默认值：`0`

尝试连接的数据库编号。

password（密码）

默认值：`null`

用于节点身份验证的密码。

subscriptionsPerConnection（单个连接最大订阅数量）

默认值：`5`

每个连接的最大订阅数量。

clientName（客户端名称）

默认值：`null`

在Redis节点里显示的客户端名称。

sslEnableEndpointIdentification（启用SSL终端识别）

默认值：`true`

开启SSL终端识别能力。

sslProvider（SSL实现方式）

默认值：`JDK`

确定采用哪种方式（JDK或OPENSSL）来实现SSL连接。

sslTruststore（SSL信任证书库路径）

默认值：`null`

指定SSL信任证书库的路径。

sslTruststorePassword（SSL信任证书库密码）

默认值：`null`

指定SSL信任证书库的密码。

sslKeystore（SSL钥匙库路径）

默认值：`null`

指定SSL钥匙库的路径。

sslKeystorePassword（SSL钥匙库密码）

默认值：`null`

指定SSL钥匙库的密码。

集群模式

集群模式除了适用于Redis集群环境，也适用于任何云计算服务商提供的集群模式，例如AWS ElastiCache集群版、Azure Redis Cache和阿里云（Aliyun）的云数据库Redis版。

介绍配置Redis集群组态的文档在这里。 Redis集群组态的最低要求是必须有三个主节点。

集群模式构造Config如下：

```
Config config = new Config();
config.useClusterServers()
    .setScanInterval(2000) // 集群状态扫描间隔时间，单位是毫秒
    //可以用"rediss://"来启用SSL连接
    .addNodeAddress("redis://127.0.0.1:7000", "redis://127.0.0.1:7001")
    .addNodeAddress("redis://127.0.0.1:7002");
RedissonClient redisson = Redisson.create(config);
```

集群模式使用到ClusterServersConfig：

ClusterServersConfig clusterConfig = config.useClusterServers();

ClusterServersConfig 配置参数如下:

> ### nodeAddresses（添加节点地址）
>
> 可以通过`host:port`的格式来添加Redis集群节点的地址。多个节点可以一次性批量添加。
>
> ### scanInterval（集群扫描间隔时间）
>
> 默认值：`1000`
>
> 对Redis集群节点状态扫描的时间间隔。单位是毫秒。
>
> ### slots（分片数量）
>
> 默认值：`231`用于指定数据分片过程中的分片数量。支持数据分片/框架结构有：集（Set）、映射（Map）、BitSet、Bloom filter, Spring Cache和Hibernate Cache等.
>
> ### readMode（读取操作的负载均衡模式）
>
> 默认值：`SLAVE`（只在从服务节点里读取）
>
> 注：在从服务节点里读取的数据说明已经至少有两个节点保存了该数据，确保了数据的高可用性。
>
> 设置读取操作选择节点的模式。可用值为：`SLAVE` - 只在从服务节点里读取。`MASTER` - 只在主服务节点里读取。`MASTER_SLAVE` - 在主从服务节点里都可以读取。
>
> ### subscriptionMode（订阅操作的负载均衡模式）
>
> 默认值：`SLAVE`（只在从服务节点里订阅）
>
> 设置订阅操作选择节点的模式。可用值为：`SLAVE` - 只在从服务节点里订阅。`MASTER` - 只在主服务节点里订阅。
>
> ### loadBalancer（负载均衡算法类的选择）
>
> 默认值：`org.redisson.connection.balancer.RoundRobinLoadBalancer`
>
> 在多Redis服务节点的环境里，可以选用以下几种负载均衡方式选择一个节点:
> `org.redisson.connection.balancer.WeightedRoundRobinBalancer` - 权重轮询调度算法
> `org.redisson.connection.balancer.RoundRobinLoadBalancer` - 轮询调度算法
> `org.redisson.connection.balancer.RandomLoadBalancer` - 随机调度算法
>
> ### subscriptionConnectionMinimumIdleSize（从节点发布和订阅连接的最小空闲连接数）
>
> 默认值：`1`

多从节点的环境里，**每个** 从服务节点里用于发布和订阅连接的最小保持连接数（长连接）。Redisson内部经常通过发布和订阅来实现许多功能。长期保持一定数量的发布订阅连接是必须的。

subscriptionConnectionPoolSize（从节点发布和订阅连接池大小）

默认值：`50`

多从节点的环境里，**每个** 从服务节点里用于发布和订阅连接的连接池最大容量。连接池的连接数量自动弹性伸缩。

slaveConnectionMinimumIdleSize（从节点最小空闲连接数）

默认值：`32`

多从节点的环境里，**每个** 从服务节点里用于普通操作（**非** 发布和订阅）的最小保持连接数（长连接）。长期保持一定数量的连接有利于提高瞬时读取反映速度。

slaveConnectionPoolSize（从节点连接池大小）

默认值：`64`

多从节点的环境里，**每个** 从服务节点里用于普通操作（**非** 发布和订阅）连接的连接池最大容量。连接池的连接数量自动弹性伸缩。

masterConnectionMinimumIdleSize（主节点最小空闲连接数）

默认值：`32`

多节点的环境里，**每个** 主节点的最小保持连接数（长连接）。长期保持一定数量的连接有利于提高瞬时写入反应速度。

masterConnectionPoolSize（主节点连接池大小）

默认值：`64`

多主节点的环境里，**每个** 主节点的连接池最大容量。连接池的连接数量自动弹性伸缩。

idleConnectionTimeout（连接空闲超时，单位：毫秒）

默认值：`10000`

如果当前连接池里的连接数量超过了最小空闲连接数，而同时有连接空闲时间超过了该数值，那么这些连接将会自动被关闭，并从连接池里去掉。时间单位是毫秒。

connectTimeout（连接超时，单位：毫秒）

默认值：`10000`

同任何节点建立连接时的等待超时。时间单位是毫秒。

timeout（命令等待超时，单位：毫秒）

默认值：`3000`

等待节点回复命令的时间。该时间从命令发送成功时开始计时。

retryAttempts（命令失败重试次数）

默认值：`3`

如果尝试达到 **retryAttempts（命令失败重试次数）** 仍然不能将命令发送至某个指定的节点时，将抛出错误。如果尝试在此限制之内发送成功，则开始启用 **timeout（命令等待超时）** 计时。

retryInterval（命令重试发送时间间隔，单位：毫秒）

默认值：`1500`

在一条命令发送失败以后，等待重试发送的时间间隔。时间单位是毫秒。

reconnectionTimeout（重新连接时间间隔，单位：毫秒）

默认值：`3000`

当与某个节点的连接断开时，等待与其重新建立连接的时间间隔。时间单位是毫秒。

failedAttempts（执行失败最大次数）

默认值：`3`

在某个节点执行相同或不同命令时，**连续** 失败 **failedAttempts（执行失败最大次数）** 时，该节点将被从可用节点列表里清除，直到 **reconnectionTimeout（重新连接时间间隔）** 超时以后再次尝试。

password（密码）

默认值：`null`

用于节点身份验证的密码。

subscriptionsPerConnection（单个连接最大订阅数量）

默认值：`5`

每个连接的最大订阅数量。

clientName（客户端名称）

默认值：`null`

在Redis节点里显示的客户端名称。

sslEnableEndpointIdentification（启用SSL终端识别）

默认值：`true`

开启SSL终端识别能力。

sslProvider（SSL实现方式）

默认值：`JDK`

确定采用哪种方式（JDK或OPENSSL）来实现SSL连接。

sslTruststore（SSL信任证书库路径）

默认值：`null`

指定SSL信任证书库的路径。

sslTruststorePassword（SSL信任证书库密码）

> 默认值：`null`
>
> 指定SSL信任证书库的密码。
>
> ### sslKeystore（SSL钥匙库路径）
>
> 默认值：`null`
>
> 指定SSL钥匙库的路径。
>
> ### sslKeystorePassword（SSL钥匙库密码）
>
> 默认值：`null`
>
> 指定SSL钥匙库的密码。

简单Redision锁的原理

Redis发展到现在，几种常见的部署架构有：

1. 单机模式；
2. 哨兵模式；
3. 集群模式；

先介绍，基于单机模式的简单Redision锁的使用。

简单Redision锁的使用

单机模式下，简单Redision锁的使用如下：

```
// 构造redisson实现分布式锁必要的Config
Config config = new Config();
config.useSingleServer().setAddress("redis://172.29.1.180:5379").setPassword("a1
23456").setDatabase(0);
// 构造RedissonClient
RedissonClient redissonClient = Redisson.create(config);
// 设置锁定资源名称
RLock disLock = redissonClient.getLock("DISLOCK");
//尝试获取分布式锁
boolean isLock= disLock.tryLock(500, 15000, TimeUnit.MILLISECONDS);
if (isLock) {
   try {
        //TODO if get lock success, do something;
        Thread.sleep(15000);

   } catch (Exception e) {
   } finally {
    // 无论如何，最后都要解锁
```

```
    disLock.unlock();
   }
}
```

> 通过代码可知，经过Redisson的封装，实现Redis分布式锁非常方便，和显式锁的使用方法是一样的。RLock接口继承了 Lock接口。

我们再看一下Redis中的value是啥，和前文分析一样，hash结构，redis 的key就是资源名称。

hash结构的key就是UUID+threadId，hash结构的value就是重入值，在分布式锁时，这个值为1（Redisson还可以实现重入锁，那么这个值就取决于重入次数了）：

```
172.29.1.180:5379> hgetall DISLOCK
1) "01a6d806-d282-4715-9bec-f51b9aa98110:1"
2) "1"
```

使用客户端工具看到的效果如下：

getLock()方法

```
@Override
public RLock getLock(String name) {
    return new RedissonLock(connectionManager.getCommandExecutor(), name);
}
```

可以看到，调用getLock()方法后实际返回一个RedissonLock对象

tryLock方法

下面来看下tryLock方法，源码如下：

```
    @Override
    public boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws
InterruptedException {
```

```
        long time = unit.toMillis(waitTime);
        long current = System.currentTimeMillis();
        long threadId = Thread.currentThread().getId();
        Long ttl = tryAcquire(leaseTime, unit, threadId);
        // lock acquired
        if (ttl == null) {
            return true;
        }

        time -= System.currentTimeMillis() - current;
        if (time <= 0) {
            acquireFailed(threadId);
            return false;
        }

        current = System.currentTimeMillis();
        RFuture<RedissonLockEntry> subscribeFuture = subscribe(threadId);
        if (!subscribeFuture.await(time, TimeUnit.MILLISECONDS)) {
            if (!subscribeFuture.cancel(false)) {
                subscribeFuture.onComplete((res, e) -> {
                    if (e == null) {
                        unsubscribe(subscribeFuture, threadId);
                    }
                });
            }
            acquireFailed(threadId);
            return false;
        }

        try {
            time -= System.currentTimeMillis() - current;
            if (time <= 0) {
                acquireFailed(threadId);
                return false;
            }

            while (true) {
                long currentTime = System.currentTimeMillis();
                ttl = tryAcquire(leaseTime, unit, threadId);
                // lock acquired
                if (ttl == null) {
                    return true;
                }

                time -= System.currentTimeMillis() - currentTime;
                if (time <= 0) {
                    acquireFailed(threadId);
                    return false;
                }

                // waiting for message
                currentTime = System.currentTimeMillis();
                if (ttl >= 0 && ttl < time) {
                    getEntry(threadId).getLatch().tryAcquire(ttl,
TimeUnit.MILLISECONDS);
                } else {
                    getEntry(threadId).getLatch().tryAcquire(time,
TimeUnit.MILLISECONDS);
```

```
                }

                time -= System.currentTimeMillis() - currentTime;
                if (time <= 0) {
                    acquireFailed(threadId);
                    return false;
                }
            }
        } finally {
            unsubscribe(subscribeFuture, threadId);
        }
//        return get(tryLockAsync(waitTime, leaseTime, unit));
    }
```

以上代码使用了异步回调模式，RFuture 继承了 java.util.concurrent.Future, CompletionStage两大接口，异步回调模式的基础知识，请参见 《Java高并发核心编程 卷2 》

tryAcquire()方法

在RedissonLock对象的lock()方法主要调用tryAcquire()方法

```
@Override
public void lockInterruptibly() throws InterruptedException {
    lockInterruptibly( leaseTime: -1,  unit: null);
}

@Override
public void lockInterruptibly(long leaseTime, TimeUnit unit) throws InterruptedException {
    long threadId = Thread.currentThread().getId();
    Long ttl = tryAcquire(leaseTime, unit, threadId);
    // lock acquired
    if (ttl == null) {
        return;
    }

    RFuture<RedissonLockEntry> future = subscribe(threadId);
    commandExecutor.syncSubscription(future);

    try {
        while (true) {
            ttl = tryAcquire(leaseTime, unit, threadId);
            // lock acquired
            if (ttl == null) {
                break;
            }

            // waiting for message
            if (ttl >= 0) {
                getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
            } else {
                getEntry(threadId).getLatch().acquire();
            }
        }
    } finally {
        unsubscribe(future, threadId);
    }
      get(lockAsync(leaseTime, unit));
}

private Long tryAcquire(long leaseTime, TimeUnit unit, long threadId) {
    return get(tryAcquireAsync(leaseTime, unit, threadId));
}
```

tryLockInnerAsync

```
private <T> RFuture<Long> tryAcquireAsync(long leaseTime, TimeUnit unit, final long threadId) {
    if (leaseTime != -1) {
        return tryLockInnerAsync(leaseTime, unit, threadId, RedisCommands.EVAL_LONG);
    }
    RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(commandExecutor.getConnectionManager()
    ttlRemainingFuture.addListener(new FutureListener<Long>() {
        @Override
        public void operationComplete(Future<Long> future) throws Exception {
            if (!future.isSuccess()) {
                return;
            }

            Long ttlRemaining = future.getNow();
            // lock acquired
            if (ttlRemaining == null) {
                scheduleExpirationRenewal(threadId);
            }
        }
    });
    return ttlRemainingFuture;
}
```

由于leaseTime == -1，于是走tryLockInnerAsync()方法，这个方法才是关键

```
RFuture<T> tryLockInnerAsync(long leaseTime, TimeUnit unit, long threadId, RedisStrictCommand<T> co
internalLockLeaseTime = unit.toMillis(leaseTime);

return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, command,
          script: "if (redis.call('exists', KEYS[1]) == 0) then " +
              "redis.call('hset', KEYS[1], ARGV[2], 1); " +
              "redis.call('pexpire', KEYS[1], ARGV[1]); " +
              "return nil; " +
          "end; " +
          "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
              "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
              "redis.call('pexpire', KEYS[1], ARGV[1]); " +
              "return nil; " +
          "end; " +
          "return redis.call('pttl', KEYS[1]);",
            Collections.<~>singletonList(getName()), internalLockLeaseTime, getLockName(threadId));
```

首先，看一下evalWriteAsync方法的定义

```
<T, R> RFuture<R> evalWriteAsync(String key, Codec codec, RedisCommand<T>
evalCommandType, String script, List<Object> keys, Object ... params);
```

这和前面的jedis调用lua脚本类似，最后两个参数分别是keys和params。

单独将调用的那一段摘出来看，实际调用是这样的：

```
commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, command,
                  "if (redis.call('exists', KEYS[1]) == 0) then " +
                      "redis.call('hset', KEYS[1], ARGV[2], 1); " +
                      "redis.call('pexpire', KEYS[1], ARGV[1]); " +
                      "return nil; " +
                  "end; " +
                  "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
                      "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
                      "redis.call('pexpire', KEYS[1], ARGV[1]); " +
                      "return nil; " +
                  "end; " +
                  "return redis.call('pttl', KEYS[1]);",
                    Collections.<Object>singletonList(getName()),
internalLockLeaseTime, getLockName(threadId));
```

结合上面的参数声明，我们可以知道，这里KEYS[1]就是getName()，ARGV[2]是getLockName(threadId)

假设：

- 前面获取锁时传的name是“DISLOCK”，
- 假设调用的线程ID是1，
- 假设成员变量UUID类型的id是01a6d806-d282-4715-9bec-f51b9aa98110

那么KEYS[1]=DISLOCK，ARGV[2]=01a6d806-d282-4715-9bec-f51b9aa98110:1

因此，这段脚本的意思是

1、判断有没有一个叫“DISLOCK”的key

2、如果没有，则在其下设置一个字段为“01a6d806-d282-4715-9bec-f51b9aa98110:1”，值为“1”的键值对 ，并设置它的过期时间

3、如果存在，则进一步判断“01a6d806-d282-4715-9bec-f51b9aa98110:1”是否存在，若存在，则其值加1，并重新设置过期时间

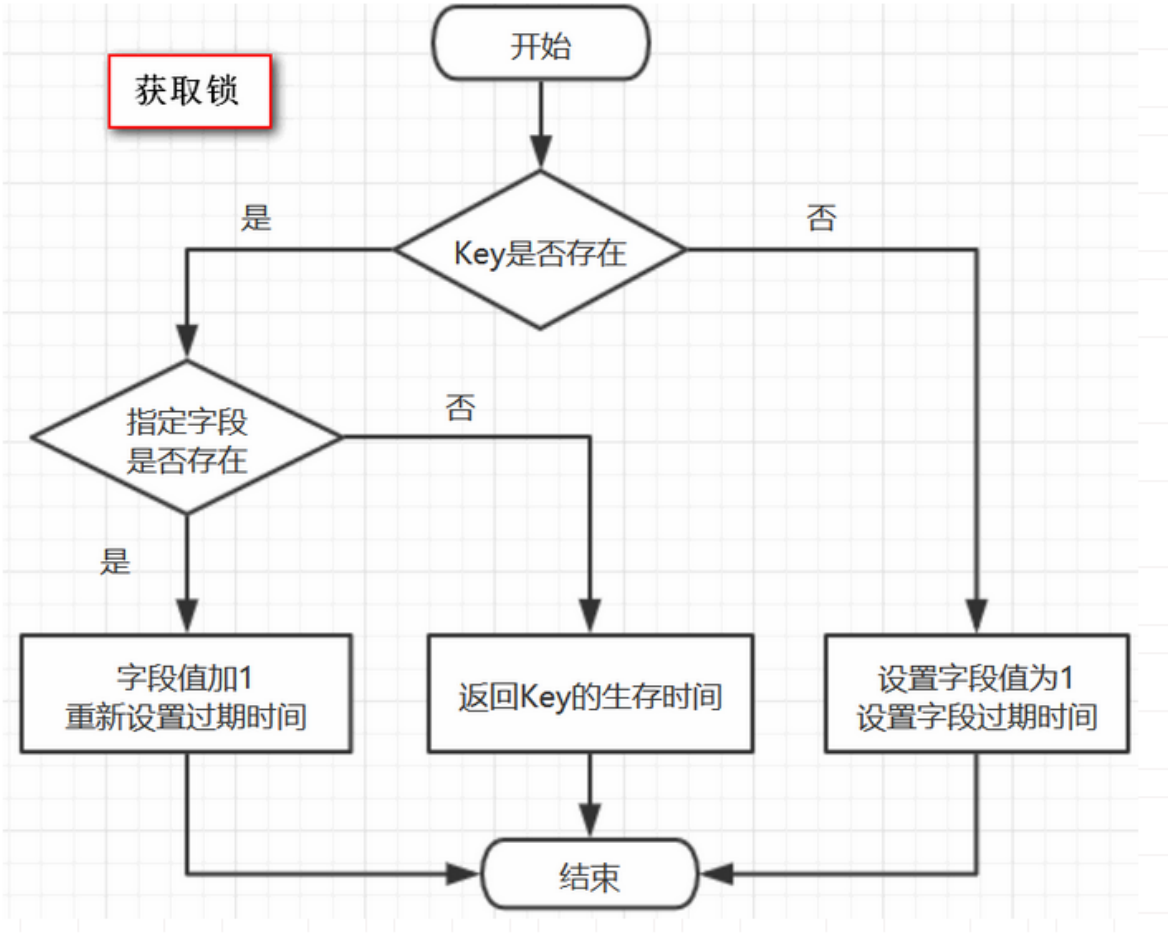
4、返回“DISLOCK”的生存时间（毫秒）

原理：加锁机制

这里用的数据结构是hash，hash的结构是： key 字段1 值1 字段2 值2 。。。

用在锁这个场景下，key就表示锁的名称，也可以理解为临界资源，字段就表示当前获得锁的线程

所有竞争这把锁的线程都要判断在这个key下有没有自己线程的字段，如果没有则不能获得锁，如果有，则相当于重入，字段值加1（次数）



Lua脚本的详解

为何要使用lua语言？

因为一大堆复杂的业务逻辑，可以通过封装在lua脚本中发送给redis，保证这段复杂业务逻辑执行的**原子性**

```
return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, command,
          script: "if (redis.call('exists', KEYS[1]) == 0) then " +
              "redis.call('hset', KEYS[1], ARGV[2], 1); " +
              "redis.call('pexpire', KEYS[1], ARGV[1]); " +
              "return nil; " +
          "end; " +
          "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
              "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
              "redis.call('pexpire', KEYS[1], ARGV[1]); " +
              "return nil; " +
          "end; " +
          "return redis.call('pttl', KEYS[1]);",
            Collections.<Object>singletonList(getName()), internalLockLeaseTime, getLockName(threadId));
```

回顾一下evalWriteAsync方法的定义

```
<T, R> RFuture<R> evalWriteAsync(String key, Codec codec, RedisCommand<T>
evalCommandType, String script, List<Object> keys, Object ... params);
```

注意，其最后两个参数分别是keys和params。

关于 lua脚本的参数解释：

KEYS[1]代表的是你加锁的那个key，比如说：

> RLock lock = redisson.getLock("DISLOCK");

这里你自己设置了加锁的那个锁key就是“DISLOCK”。

ARGV[1]代表的就是锁key的默认生存时间

> 调用的时候，传递的参数为 internalLockLeaseTime ，该值默认30秒。

ARGV[2]代表的是加锁的客户端的ID，类似于下面这样：

> 01a6d806-d282-4715-9bec-f51b9aa98110:1

lua脚本的第一段if判断语句，就是用“exists DISLOCK”命令判断一下，如果你要加锁的那个锁key不存在的话，你就进行加锁。

如何加锁呢？很简单，用下面的redis命令：

> hset DISLOCK 01a6d806-d282-4715-9bec-f51b9aa98110:1 1

通过这个命令设置一个hash数据结构，这行命令执行后，会出现一个类似下面的数据结构：

```
DISLOCK:
    {
        8743c9c0-0795-4907-87fd-6c719a6b4586:1 1
    }
```

接着会执行“pexpire DISLOCK 30000”命令，设置DISLOCK这个锁key的生存时间是30秒(默认)

锁互斥机制

那么在这个时候，如果客户端2来尝试加锁，执行了同样的一段lua脚本，会咋样呢？

很简单，第一个if判断会执行“exists DISLOCK”，发现DISLOCK 这个锁key已经存在了。

接着第二个if判断，判断一下，DISLOCK锁key的hash数据结构中，是否包含客户端2的ID，但是明显不是的，因为那里包含的是客户端1的ID。

所以，客户端2会获取到pttl DISLOCK返回的一个数字，这个数字代表了DISLOCK 这个锁key的**剩余生存时间。**比如还剩15000毫秒的生存时间。

此时客户端2会进入一个while循环，不停的尝试加锁。

可重入加锁机制

如果客户端1都已经持有了这把锁了，结果可重入的加锁会怎么样呢?

```
RLock lock = redisson.getLock("DISLOCK")
lock.lock();
//业务代码
lock.lock();
//业务代码
lock.unlock();
lock.unlock();
```

分析上面那段lua脚本。

第一个if判断肯定不成立，“exists DISLOCK”会显示锁key已经存在了。

第二个if判断会成立，因为DISLOCK的hash数据结构中包含的那个ID，就是客户端1的那个ID，也就是“8743c9c0-0795-4907-87fd-6c719a6b4586:1”

此时就会执行可重入加锁的逻辑，他会用：

incrby DISLOCK

8743c9c0-0795-4907-87fd-6c719a6b4586:1 1

通过这个命令，对客户端1的加锁次数，累加1。

此时DISLOCK数据结构变为下面这样：

```
DISLOCK:
    {
        8743c9c0-0795-4907-87fd-6c719a6b4586:1 2
    }
```

释放锁机制

如果执行lock.unlock()，就可以释放分布式锁，此时的业务逻辑也是非常简单的。

其实说白了，就是每次都对DISLOCK数据结构中的那个加锁次数减1。

如果发现加锁次数是0了，说明这个客户端已经不再持有锁了，此时就会用：

“del DISLOCK”命令，从redis里删除这个key。

然后呢，另外的客户端2就可以尝试完成加锁了。

unlock 源码

```
  @Override
    public void unlock() {
        try {
            get(unlockAsync(Thread.currentThread().getId()));
        } catch (RedisException e) {
            if (e.getCause() instanceof IllegalMonitorStateException) {
                throw (IllegalMonitorStateException) e.getCause();
            } else {
                throw e;
            }
        }

//        Future<Void> future = unlockAsync();
//        future.awaitUninterruptibly();
//        if (future.isSuccess()) {
//            return;
//        }
//        if (future.cause() instanceof IllegalMonitorStateException) {
//            throw (IllegalMonitorStateException)future.cause();
//        }
//        throw commandExecutor.convertException(future);
    }
```

再深入一下，实际调用的是unlockInnerAsync方法

unlockInnerAsync方法

```
protected RFuture<Boolean> unlockInnerAsync(long threadId) {
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands.EVAL_BOOLEAN,
            script: "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " +
                "return nil;" +
            "end; " +
            "local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1); " +
            "if (counter > 0) then " +
                "redis.call('pexpire', KEYS[1], ARGV[2]); " +
                "return 0; " +
            "else " +
                "redis.call('del', KEYS[1]); " +
                "redis.call('publish', KEYS[2], ARGV[1]); " +
                "return 1; "+
            "end; " +
            "return nil;",
            Arrays.<Object>asList(getName(), getChannelName()), LockPubSub.UNLOCK_MESSAGE, internalLockLeaseTime,

}
```

原理：Redision 解锁机制

上图没有截取完整，完整的源码如下：

```
    protected RFuture<Boolean> unlockInnerAsync(long threadId) {
        return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE,
RedisCommands.EVAL_BOOLEAN,
```

```
                "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " +
                    "return nil;" +
                "end; " +
                "local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1); "
+
                "if (counter > 0) then " +
                    "redis.call('pexpire', KEYS[1], ARGV[2]); " +
                    "return 0; " +
                "else " +
                    "redis.call('del', KEYS[1]); " +
                    "redis.call('publish', KEYS[2], ARGV[1]); " +
                    "return 1; "+
                "end; " +
                "return nil;",
                Arrays.<Object>asList(getName(), getChannelName()),
LockPubSub.UNLOCK_MESSAGE, internalLockLeaseTime, getLockName(threadId));

    }
```

我们还是假设name=DISLOCK，假设线程ID是1

同理，我们可以知道

KEYS[1]是getName()，即KEYS[1]=DISLOCK

KEYS[2]是getChannelName()，即KEYS[2]=redisson_lock__channel:{DISLOCK}

ARGV[1]是LockPubSub.unlockMessage，即ARGV[1]=0

ARGV[2]是生存时间

ARGV[3]是getLockName(threadId)，即ARGV[3]=8743c9c0-0795-4907-87fd-6c719a6b4586:1

因此，上面脚本的意思是：

1、判断是否存在一个叫“DISLOCK”的key

2、如果不存在，返回nil

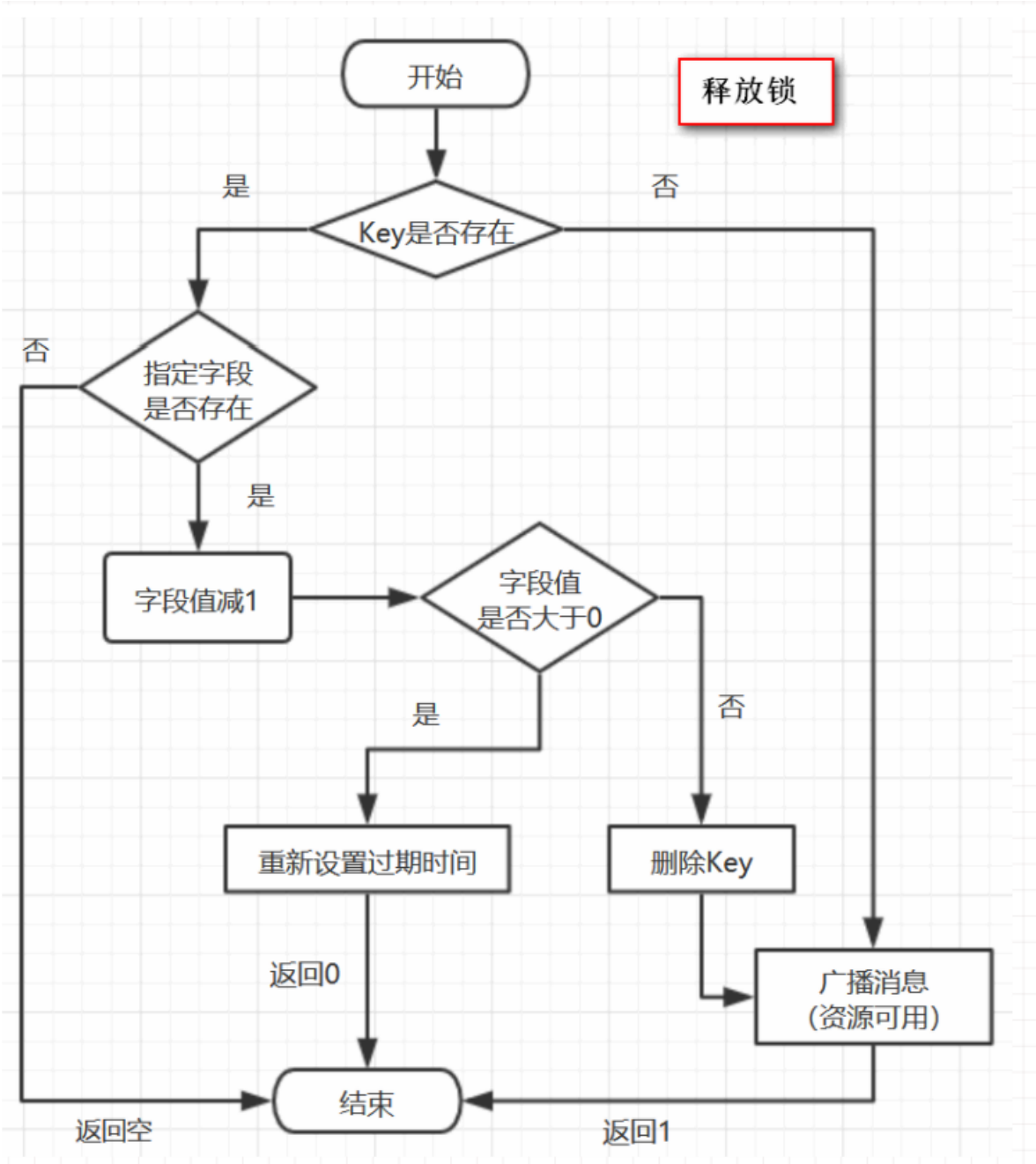
3、如果存在，使用Redis *Hincrby* 命令用于为哈希表中的字段值加上指定增量值 -1 ，代表减去1

4、若counter >，返回空，若字段存在，则字段值减1

5、若减完以后，counter > 0 值仍大于0，则返回0

6、减完后，若字段值小于或等于0，则用 publish 命令广播一条消息，广播内容是0，并返回1；

可以猜测，广播0表示资源可用，即通知那些等待获取锁的线程现在可以获得锁了



通过redis Channel 解锁订阅

以上是正常情况下获取到锁的情况，那么当无法立即获取到锁的时候怎么办呢?

再回到前面获取锁的位置

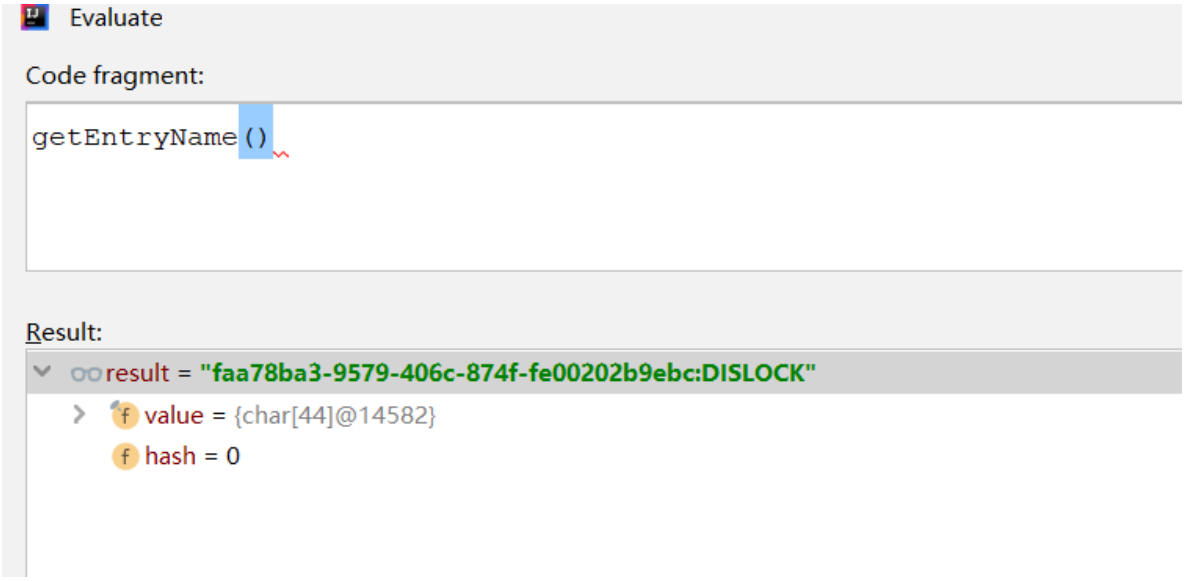
```
@Override
public void lockInterruptibly(long leaseTime, TimeUnit unit) throws
InterruptedException {
    long threadId = Thread.currentThread().getId();
    Long ttl = tryAcquire(leaseTime, unit, threadId);
    // lock acquired
    if (ttl == null) {
        return;
    }

    //     订阅
    RFuture<RedissonLockEntry> future = subscribe(threadId);
```

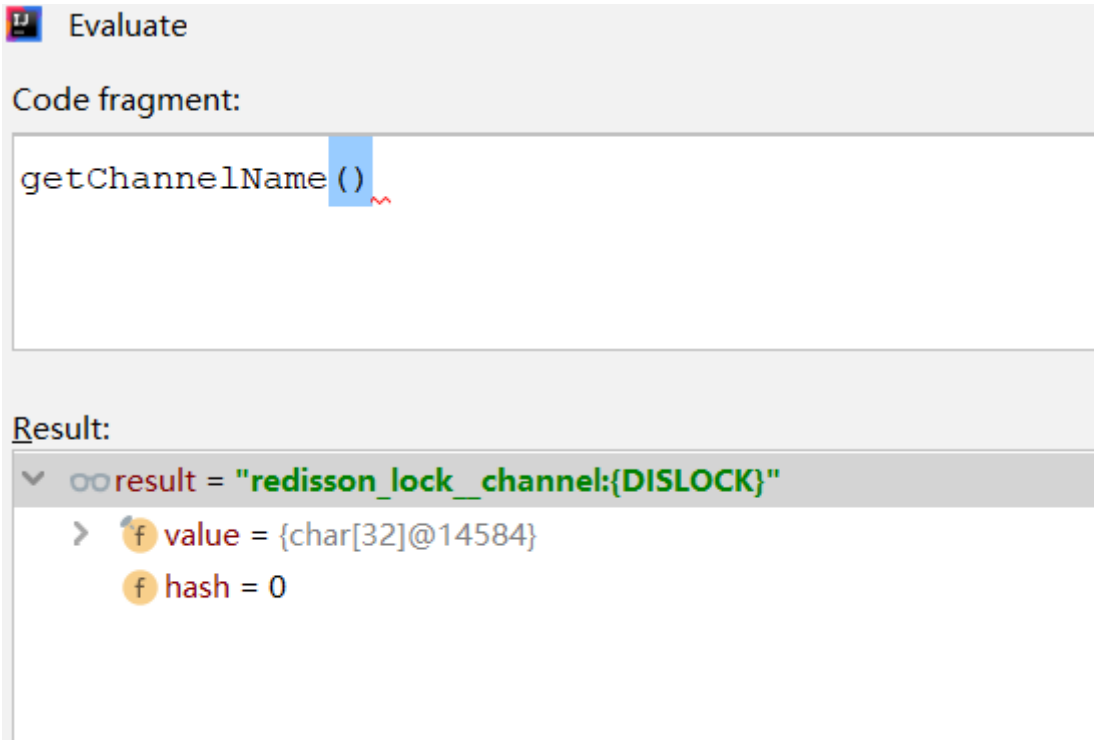
```
    commandExecutor.syncSubscription(future);

    try {
        while (true) {
            ttl = tryAcquire(leaseTime, unit, threadId);
            // lock acquired
            if (ttl == null) {
                break;
            }

            // waiting for message
            if (ttl >= 0) {
                getEntry(threadId).getLatch().tryAcquire(ttl,
TimeUnit.MILLISECONDS);
            } else {
                getEntry(threadId).getLatch().acquire();
            }
        }
    } finally {
        unsubscribe(future, threadId);
    }
//        get(lockAsync(leaseTime, unit));
}


protected static final LockPubSub PUBSUB = new LockPubSub();

protected RFuture<RedissonLockEntry> subscribe(long threadId) {
    return PUBSUB.subscribe(getEntryName(), getChannelName(),
commandExecutor.getConnectionManager().getSubscribeService());
}

protected void unsubscribe(RFuture<RedissonLockEntry> future, long threadId) {
    PUBSUB.unsubscribe(future.getNow(), getEntryName(), getChannelName(),
commandExecutor.getConnectionManager().getSubscribeService());
}
```

这里会订阅Channel，当资源可用时可以及时知道，并抢占，防止无效的轮询而浪费资源

这里的channel为：

> redisson_lock__channel:{DISLOCK}





当资源可用用的时候，循环去尝试获取锁，由于多个线程同时去竞争资源，所以这里用了信号量，对于同一个资源只允许一个线程获得锁，其它的线程阻塞

这点，有点儿类似 Zookeeper分布式锁：

> 有关zookeeper分布式锁的原理和实现，具体请参见下面的博客：
> Zookeeper 分布式锁 （图解+秒懂+史上最全）

watch dog自动延期机制

客户端1加锁的锁key默认生存时间才30秒，如果超过了30秒，客户端1还想一直持有这把锁，怎么办呢?

简单！只要客户端1一旦加锁成功，就会启动一个watch dog看门狗，**他是一个后台线程，会每隔10秒检查一下**，如果客户端1还持有锁key，那么就会不断的延长锁key的生存时间。

使用watchDog机制实现锁的续期

但是聪明的同学肯定会问：

> 有效时间设置多长，假如我的业务操作比有效时间长，我的业务代码还没执行完，就自动给我解锁了，不就完蛋了吗。

这个问题就有点棘手了，在网上也有很多讨论：

第一种解决方法就是靠程序员自己去把握，预估一下业务代码需要执行的时间，然后设置有效期时间比执行时间长一些，保证不会因为自动解锁影响到客户端业务代码的执行。

但是这并不是万全之策，比如网络抖动这种情况是无法预测的，也有可能导致业务代码执行的时间变长，所以并不安全。

第二种方法，使用监事狗watchDog机制实现锁的续期。

> 第二种方法比较靠谱一点，而且无业务入侵。

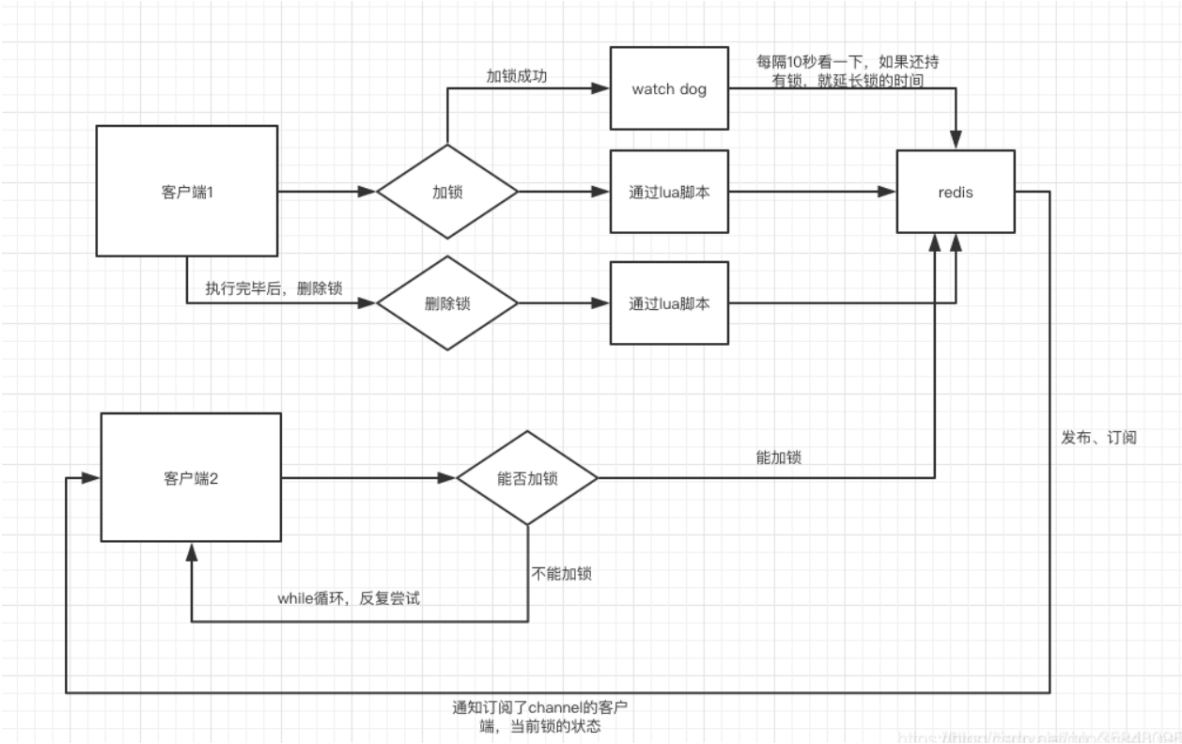
在Redisson框架实现分布式锁的思路，就使用watchDog机制实现锁的续期。

当加锁成功后，同时开启守护线程，默认有效期是30秒，每隔10秒就会给锁续期到30秒，只要持有锁的客户端没有宕机，就能保证一直持有锁，直到业务代码执行完毕由客户端自己解锁，如果宕机了自然就在有效期失效后自动解锁。

> 这里，和前面解决 JVM STW的锁过期问题有点类似，只不过，watchDog自动续期，也没有完全解决JVM STW的锁过期问题。
>
> 如何彻底解决 JVM STW的锁过期问题，可以来疯狂创客圈的社群讨论。

redisson watchdog 使用和原理

实际上，redisson加锁的基本流程图如下：



这里专注于介绍watchdog。

首先watchdog的具体思路是 加锁时，默认加锁 30秒，每10秒钟检查一次，如果存在就重新设置 过期时间为30秒。

然后设置默认加锁时间的参数是 lockWatchdogTimeout（监控锁的看门狗超时，单位：毫秒）

官方文档描述如下

> **lockWatchdogTimeout（监控锁的看门狗超时，单位：毫秒）**
>
> 默认值：`30000`
>
> 监控锁的看门狗超时时间单位为毫秒。该参数只适用于分布式锁的加锁请求中未明确使用`leaseTimeout`参数的情况。如果该看门狗未使用`lockWatchdogTimeout`去重新调整一个分布式锁的`lockWatchdogTimeout`超时，那么这个锁将变为失效状态。这个参数可以用来避免由Redisson客户端节点宕机或其他原因造成死锁的情况。

需要注意的是

1.watchDog 只有在未显示指定加锁时间时才会生效。（这点很重要）

2.lockWatchdogTimeout设定的时间不要太小，比如我之前设置的是 100毫秒，由于网络直接导致加锁完后，watchdog去延期时，这个key在redis中已经被删除了。

tryAcquireAsync原理

在调用lock方法时，会最终调用到tryAcquireAsync。详细解释如下：

```
private <T> RFuture<Long> tryAcquireAsync(long waitTime, long leaseTime,
TimeUnit unit, long threadId) {
    //如果指定了加锁时间，会直接去加锁
        if (leaseTime != -1) {
            return tryLockInnerAsync(waitTime, leaseTime, unit, threadId,
RedisCommands.EVAL_LONG);
```

```
        }
    //没有指定加锁时间 会先进行加锁，并且默认时间就是 LockWatchdogTimeout的时间
    //这个是异步操作 返回RFuture 类似netty中的future
        RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(waitTime,

 commandExecutor.getConnectionManager().getCfg().getLockWatchdogTimeout(),
                                                TimeUnit.MILLISECONDS, threadId,
RedisCommands.EVAL_LONG);
       //这里也是类似netty Future 的addListener，在future内容执行完成后执行
        ttlRemainingFuture.onComplete((ttlRemaining, e) -> {
            if (e != null) {
                return;
            }

            // lock acquired
            if (ttlRemaining == null) {
            //这里是定时执行 当前锁自动延期的动作
                scheduleExpirationRenewal(threadId);
            }
        });
        return ttlRemainingFuture;
    }
```

scheduleExpirationRenewal 中会调用renewExpiration。

renewExpiration执行延期动作

这里我们可以看到是 启用了一个timeout定时，去执行延期动作

```
    private void renewExpiration() {


        Timeout task = commandExecutor.getConnectionManager().newTimeout(new
TimerTask() {
            @Override
            public void run(Timeout timeout) throws Exception {
                ExpirationEntry ent =
EXPIRATION_RENEWAL_MAP.get(getEntryName());
                if (ent == null) {
                    return;
                }
                Long threadId = ent.getFirstThreadId();
                if (threadId == null) {
                    return;
                }

                RFuture<Boolean> future = renewExpirationAsync(threadId);
                future.onComplete((res, e) -> {
                    if (e != null) {
                        log.error("Can't update lock " + getName() + "
expiration", e);
                        return;
                    }

                    if (res) {
                        //如果 没有报错，就再次定时延期
                     // reschedule itself
```

```
                        renewExpiration();
                    }
                });
            }
            // 这里我们可以看到定时任务 是 lockWatchdogTimeout 的1/3时间去执行
renewExpirationAsync
        }, internalLockLeaseTime / 3, TimeUnit.MILLISECONDS);

        ee.setTimeout(task);
    }
```

最终 scheduleExpirationRenewal会调用到 renewExpirationAsync，

renewExpirationAsync

执行下面这段 lua脚本。他主要判断就是 这个锁是否在redis中存在，如果存在就进行 pexpire 延期。

```
    protected RFuture<Boolean> renewExpirationAsync(long threadId) {
        return evalWriteAsync(getName(), LongCodec.INSTANCE,
RedisCommands.EVAL_BOOLEAN,
                "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
                        "redis.call('pexpire', KEYS[1], ARGV[1]); " +
                        "return 1; " +
                        "end; " +
                        "return 0;",
                Collections.singletonList(getName()),
                internalLockLeaseTime, getLockName(threadId));
    }
```

watchLog总结

1.要使 watchLog机制生效， lock时 不要设置 过期时间

2.watchlog的延时时间 可以由 lockWatchdogTimeout指定默认延时时间，但是不要设置太小。如100

3.watchdog 会每 lockWatchdogTimeout/3时间，去延时。

4.watchdog 通过 类似netty的 Future功能来实现异步延时

5.watchdog 最终还是通过 lua脚本来进行延时

Redisson框架的分布式锁

Redisson框架十分强大，除了前面介绍的 getLock方法获取的分布式锁（输入可重入锁的类型），还有很多其他的分布式锁类型。

总体的Redisson框架的分布式锁类型，大致如下：

- 可重入锁
- 公平锁
- 联锁
- 红锁
- 读写锁

- 信号量
- 可过期信号量
- 闭锁（/倒数闩）

1.可重入锁（Reentrant Lock）

Redisson的分布式可重入锁RLock Java对象实现了java.util.concurrent.locks.Lock接口，同时还支持自动过期解锁。

```
public void testReentrantLock(RedissonClient redisson){
    RLock lock = redisson.getLock("anyLock");
    try{
        // 1. 最常见的使用方法
        //lock.lock();
        // 2. 支持过期解锁功能,10秒钟以后自动解锁, 无需调用unlock方法手动解锁
        //lock.lock(10, TimeUnit.SECONDS);
        // 3. 尝试加锁，最多等待3秒，上锁以后10秒自动解锁
        boolean res = lock.tryLock(3, 10, TimeUnit.SECONDS);
        if(res){ //成功
        // do your business
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

Redisson同时还为分布式锁提供了异步执行的相关方法：

```
public void testAsyncReentrantLock(RedissonClient redisson){
    RLock lock = redisson.getLock("anyLock");
    try{
        lock.lockAsync();
        lock.lockAsync(10, TimeUnit.SECONDS);
        Future<Boolean> res = lock.tryLockAsync(3, 10, TimeUnit.SECONDS);
        if(res.get()){
        // do your business
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

2.公平锁（Fair Lock）

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。在提供了自动过期解锁功能的同时，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。

```
public void testFairLock(RedissonClient redisson){
    RLock fairLock = redisson.getFairLock("anyLock");
    try{
        // 最常见的使用方法
        fairLock.lock();
        // 支持过期解锁功能，10秒钟以后自动解锁,无需调用unlock方法手动解锁
        fairLock.lock(10, TimeUnit.SECONDS);
        // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
        boolean res = fairLock.tryLock(100, 10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        fairLock.unlock();
    }
}
```

Redisson同时还为分布式可重入公平锁提供了异步执行的相关方法：

```
RLock fairLock = redisson.getFairLock("anyLock");
fairLock.lockAsync();
fairLock.lockAsync(10, TimeUnit.SECONDS);
Future<Boolean> res = fairLock.tryLockAsync(100, 10, TimeUnit.SECONDS);
```

3.联锁（MultiLock）

Redisson的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。

```
public void testMultiLock(RedissonClient redisson1,RedissonClient redisson2,
RedissonClient redisson3){
    RLock lock1 = redisson1.getLock("lock1");
    RLock lock2 = redisson2.getLock("lock2");
    RLock lock3 = redisson3.getLock("lock3");
    RedissonMultiLock lock = new RedissonMultiLock(lock1, lock2, lock3);
    try {
        // 同时加锁：lock1 lock2 lock3，所有的锁都上锁成功才算成功。
        lock.lock();
        // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
        boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

4.红锁（RedLock）

Redisson的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。

```
public void testRedLock(RedissonClient redisson1,RedissonClient redisson2,
RedissonClient redisson3){
    RLock lock1 = redisson1.getLock("lock1");
    RLock lock2 = redisson2.getLock("lock2");
```

```
    RLock lock3 = redisson3.getLock("lock3");
    RedissonRedLock lock = new RedissonRedLock(lock1, lock2, lock3);
    try {
        // 同时加锁：lock1 lock2 lock3, 红锁在大部分节点上加锁成功就算成功。
        lock.lock();
        // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
        boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

5.读写锁（ReadWriteLock）

Redisson的分布式可重入读写锁RReadWriteLock,Java对象实现了java.util.concurrent.locks.ReadWriteLock接口。同时还支持自动过期解锁。该对象允许同时有多个读取锁，但是最多只能有一个写入锁。

```
RReadWriteLock rwlock = redisson.getLock("anyRWLock");
// 最常见的使用方法
rwlock.readLock().lock();
// 或
rwlock.writeLock().lock();
// 支持过期解锁功能
// 10秒钟以后自动解锁
// 无需调用unlock方法手动解锁
rwlock.readLock().lock(10, TimeUnit.SECONDS);
// 或
rwlock.writeLock().lock(10, TimeUnit.SECONDS);
// 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
boolean res = rwlock.readLock().tryLock(100, 10, TimeUnit.SECONDS);
// 或
boolean res = rwlock.writeLock().tryLock(100, 10, TimeUnit.SECONDS);
...
lock.unlock();
```

6.信号量（Semaphore）

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。

```
RSemaphore semaphore = redisson.getSemaphore("semaphore");
semaphore.acquire();
//或
semaphore.acquireAsync();
semaphore.acquire(23);
semaphore.tryAcquire();
//或
semaphore.tryAcquireAsync();
semaphore.tryAcquire(23, TimeUnit.SECONDS);
//或
semaphore.tryAcquireAsync(23, TimeUnit.SECONDS);
semaphore.release(10);
semaphore.release();
```

```
//或
semaphore.releaseAsync();
```

7.可过期性信号量（PermitExpirableSemaphore）

Redisson的可过期性信号量（PermitExpirableSemaphore）实在RSemaphore对象的基础上，为每个信号增加了一个过期时间。每个信号可以通过独立的ID来辨识，释放时只能通过提交这个ID才能释放。

```
RPermitExpirableSemaphore semaphore =
redisson.getPermitExpirableSemaphore("mySemaphore");
String permitId = semaphore.acquire();
// 获取一个信号，有效期只有2秒钟。
String permitId = semaphore.acquire(2, TimeUnit.SECONDS);
// ...
semaphore.release(permitId);
```

8.闭锁/倒数闩（CountDownLatch）

Redisson的分布式闭锁（CountDownLatch）Java对象RCountDownLatch采用了与java.util.concurrent.CountDownLatch相似的接口和用法。

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

redis分布式锁的高可用

关于Redis分布式锁的高可用问题，大致如下：

在master- slave的集群架构中，就是如果你对某个redis master实例，写入了DISLOCK这种锁key的value，此时会异步复制给对应的master slave实例。

但是，这个过程中一旦发生redis master宕机，主备切换，redis slave变为了redis master。而此时的主从复制没有彻底完成.....

接着就会导致，客户端2来尝试加锁的时候，在新的redis master上完成了加锁，而客户端1也以为自己成功加了锁。

此时就会导致多个客户端对一个分布式锁完成了加锁。

> 这时系统在业务语义上一定会出现问题，导致脏数据的产生。

所以这个是是redis master-slave架构的主从异步复制导致的redis分布式锁的最大缺陷：

> 在redis master实例宕机的时候，可能导致多个客户端同时完成加锁。

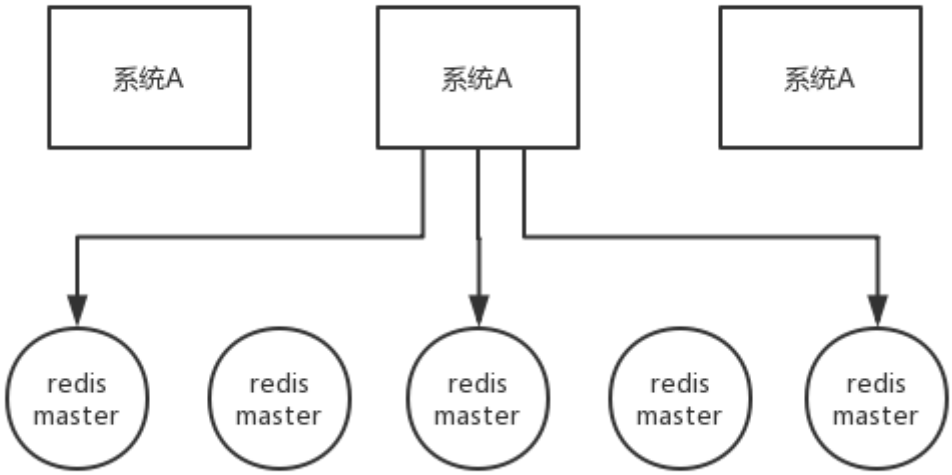
高可用的RedLock（红锁）原理

RedLock算法思想：

> 不能只在一个redis实例上创建锁，应该是在多个redis实例上创建锁，n / 2 + 1，必须在大多数redis节点上都成功创建锁，才能算这个整体的RedLock加锁成功，避免说仅仅在一个redis实例上加锁而带来的问题。

这个场景是假设有一个 redis cluster，有 5 个 redis master 实例。然后执行如下步骤获取一把红锁：

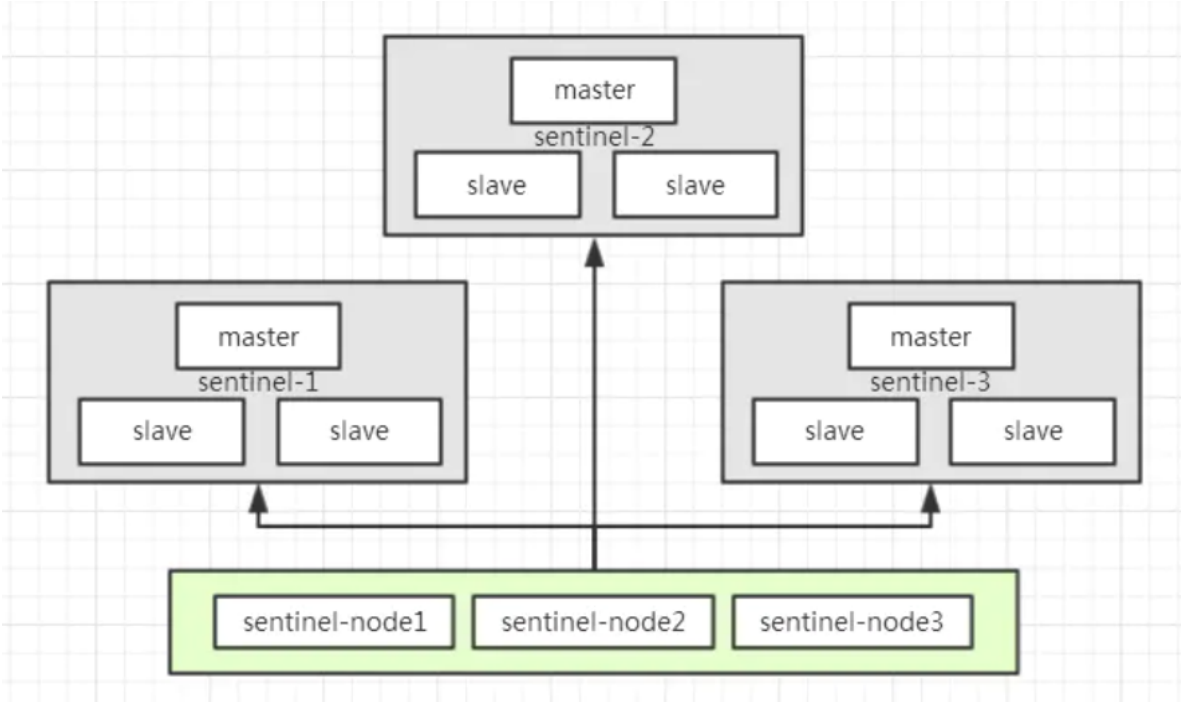
1. 获取当前时间戳，单位是毫秒；
2. 跟上面类似，轮流尝试在每个 master 节点上创建锁，过期时间较短，一般就几十毫秒；
3. 尝试在大多数节点上建立一个锁，比如 5 个节点就要求是 3 个节点 n / 2 + 1；
4. 客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了；
5. 要是锁建立失败了，那么就依次之前建立过的锁删除；
6. 只要别人建立了一把分布式锁，你就得不断轮询去尝试获取锁。



RedLock是基于redis实现的分布式锁，它能够保证以下特性：

- 互斥性：在任何时候，只能有一个客户端能够持有锁；避免死锁：
- 当客户端拿到锁后，即使发生了网络分区或者客户端宕机，也不会发生死锁；（利用key的存活时间）
- 容错性：只要多数节点的redis实例正常运行，就能够对外提供服务，加锁或者释放锁；

以sentinel模式架构为例，如下图所示，有sentinel-1，sentinel-2，sentinel-3总计3个sentinel模式集群，如果要获取分布式锁，那么需要向这3个sentinel集群通过EVAL命令执行LUA脚本，需要3/2+1=2，即至少2个sentinel集群响应成功，才算成功的以Redlock算法获取到分布式锁：



高可用的红锁会导致性能降低

提前说明，使用redis分布式锁，是追求高性能， 在cap理论中，追求的是 ap 而不是cp。

所以，如果追求高可用，建议使用 zookeeper分布式锁。

> redis分布式锁可能导致的数据不一致性，建议使用业务补偿的方式去弥补。所以，不太建议使用红锁，但是从学习的层面来说，大家还是一定要掌握的。

实现原理

Redisson中有一个`MultiLock`的概念，可以将多个锁合并为一个大锁，对一个大锁进行统一的申请加锁以及释放锁

而Redisson中实现RedLock就是基于`MultiLock`去做的，接下来就具体看看对应的实现吧

RedLock使用案例

先看下官方的代码使用：
(https://github.com/redisson/redisson/wiki/8.-distributed-locks-and-synchronizers#84-redlock)

```
RLock lock1 = redisson1.getLock("lock1");
RLock lock2 = redisson2.getLock("lock2");
RLock lock3 = redisson3.getLock("lock3");

RLock redLock = anyRedisson.getRedLock(lock1, lock2, lock3);

// traditional lock method
redLock.lock();

// or acquire lock and automatically unlock it after 10 seconds
redLock.lock(10, TimeUnit.SECONDS);
```

```
// or wait for lock aquisition up to 100 seconds
// and automatically unlock it after 10 seconds
boolean res = redLock.tryLock(100, 10, TimeUnit.SECONDS);
if (res) {
   try {
     ...
   } finally {
       redLock.unlock();
   }
}
```

这里是分别对3个redis实例加锁，然后获取一个最后的加锁结果。

RedissonRedLock实现原理

上面示例中使用redLock.lock()或者tryLock()最终都是执行`RedissonRedLock`中方法。

`RedissonRedLock`继承自`RedissonMultiLock`， 实现了其中的一些方法：

```
public class RedissonRedLock extends RedissonMultiLock {
    public RedissonRedLock(RLock... locks) {
        super(locks);
    }

    /**
     * 锁可以失败的次数，锁的数量-锁成功客户端最小的数量
     */
    @Override
    protected int failedLocksLimit() {
        return locks.size() - minLocksAmount(locks);
    }

    /**
     * 锁的数量 / 2 + 1，例如有3个客户端加锁，那么最少需要2个客户端加锁成功
     */
    protected int minLocksAmount(final List<RLock> locks) {
        return locks.size()/2 + 1;
    }

    /**
     * 计算多个客户端一起加锁的超时时间，每个客户端的等待时间
     * remainTime默认为4.5s
     */
    @Override
    protected long calcLockWaitTime(long remainTime) {
        return Math.max(remainTime / locks.size(), 1);
    }

    @Override
    public void unlock() {
        unlockInner(locks);
    }

}
```

看到`locks.size()/2 + 1`，例如我们有3个客户端实例，那么最少2个实例加锁成功才算分布式锁加锁成功。

接着我们看下`lock()`的具体实现

RedissonMultiLock实现原理

```
public class RedissonMultiLock implements Lock {

    final List<RLock> locks = new ArrayList<RLock>();

    public RedissonMultiLock(RLock... locks) {
        if (locks.length == 0) {
            throw new IllegalArgumentException("Lock objects are not defined");
        }
        this.locks.addAll(Arrays.asList(locks));
    }

    public boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws
InterruptedException {
        long newLeaseTime = -1;
        if (leaseTime != -1) {
            // 如果等待时间设置了，那么将等待时间 * 2
            newLeaseTime = unit.toMillis(waitTime)*2;
        }

        // time为当前时间戳
        long time = System.currentTimeMillis();
        long remainTime = -1;
        if (waitTime != -1) {
            remainTime = unit.toMillis(waitTime);
        }
        // 计算锁的等待时间，RedLock中：如果remainTime=-1，那么lockWaitTime为1
        long lockWaitTime = calcLockWaitTime(remainTime);

        // RedLock中failedLocksLimit即为n/2 + 1
        int failedLocksLimit = failedLocksLimit();
        List<RLock> acquiredLocks = new ArrayList<RLock>(locks.size());
        // 循环每个redis客户端，去获取锁
        for (ListIterator<RLock> iterator = locks.listIterator();
iterator.hasNext();) {
            RLock lock = iterator.next();
            boolean lockAcquired;
            try {
                // 调用tryLock方法去获取锁，如果获取锁成功，则lockAcquired=true
                if (waitTime == -1 && leaseTime == -1) {
                    lockAcquired = lock.tryLock();
                } else {
                    long awaitTime = Math.min(lockWaitTime, remainTime);
                    lockAcquired = lock.tryLock(awaitTime, newLeaseTime,
TimeUnit.MILLISECONDS);
                }
            } catch (Exception e) {
                lockAcquired = false;
            }

            // 如果获取锁成功，将锁加入到list集合中
            if (lockAcquired) {
                acquiredLocks.add(lock);
```

```
            } else {
                // 如果获取锁失败，判断失败次数是否等于失败的限制次数
                // 比如，3个redis客户端，最多只能失败1次
                // 这里locks.size = 3, 3-x=1，说明只要成功了2次就可以直接break掉循环
                if (locks.size() - acquiredLocks.size() == failedLocksLimit()) {
                    break;
                }

                // 如果最大失败次数等于0
                if (failedLocksLimit == 0) {
                    // 释放所有的锁，RedLock加锁失败
                    unlockInner(acquiredLocks);
                    if (waitTime == -1 && leaseTime == -1) {
                        return false;
                    }
                    failedLocksLimit = failedLocksLimit();
                    acquiredLocks.clear();
                    // 重置迭代器 重试再次获取锁
                    while (iterator.hasPrevious()) {
                        iterator.previous();
                    }
                } else {
                    // 失败的限制次数减一
                    // 比如3个redis实例，最大的限制次数是1，如果遍历第一个redis实例，失败
了，那么failedLocksLimit会减成0
                    // 如果failedLocksLimit就会走上面的if逻辑，释放所有的锁，然后返回
false
                    failedLocksLimit--;
                }
            }

            if (remainTime != -1) {
                remainTime -= (System.currentTimeMillis() - time);
                time = System.currentTimeMillis();
                if (remainTime <= 0) {
                    unlockInner(acquiredLocks);
                    return false;
                }
            }
        }

        if (leaseTime != -1) {
            List<RFuture<Boolean>> futures = new ArrayList<RFuture<Boolean>>
(acquiredLocks.size());
            for (RLock rLock : acquiredLocks) {
                RFuture<Boolean> future =
rLock.expireAsync(unit.toMillis(leaseTime), TimeUnit.MILLISECONDS);
                futures.add(future);
            }

            for (RFuture<Boolean> rFuture : futures) {
                rFuture.syncUninterruptibly();
            }
        }

        return true;
    }
}
```

核心代码都已经加了注释，实现原理其实很简单，基于RedLock思想，遍历所有的Redis客户端，然后依次加锁，最后统计成功的次数来判断是否加锁成功。

Redis分段锁

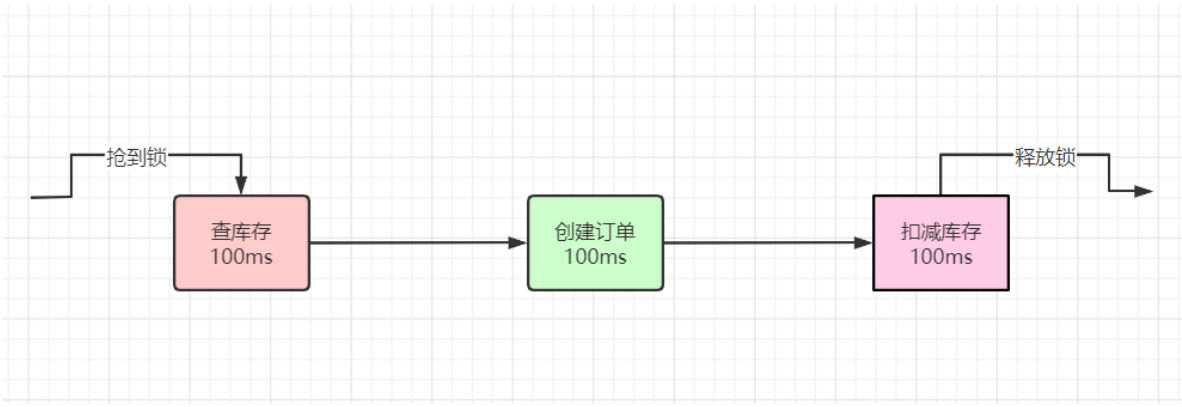
普通Redis分布式锁的性能瓶颈问题

分布式锁一旦加了之后，对同一个商品的下单请求，会导致所有下单操作，都必须对同一个商品key加分布式锁。

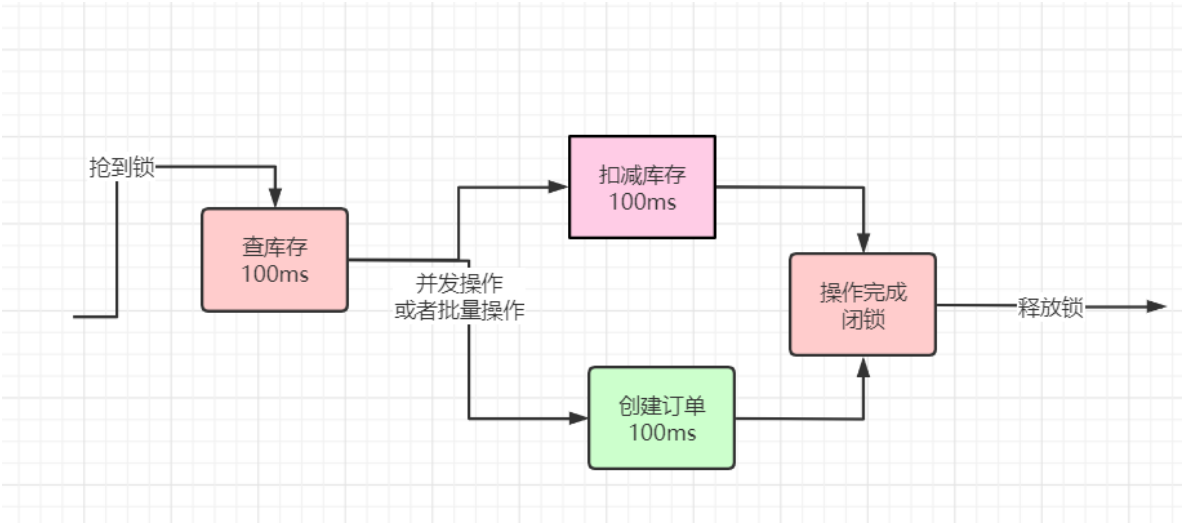
假设某个场景，一个商品1分钟6000订单，每秒的 600个下单操作，

假设加锁之后，释放锁之前，查库存 -> 创建订单 -> 扣减库存，每个IO操作100ms，大概300毫秒。

具体如下图：



可以再进行一下优化，将 创建订单 + 扣减库存 并发执行，将两个100ms 减少为一个100ms，这既是空间换时间的思想，大概200毫秒。



将 创建订单 + 扣减库存 批量执行，减少一次IO，也是大概200毫秒。

> 这个优化方案，有个重要的前提，就是 订单表和库存表在相同的库中，但是，这个前提条件，在数据量大+高并发的场景下，够呛。

```
package com.crazymaker.springcloud;

import com.crazymaker.springcloud.common.util.ThreadUtil;
import org.junit.Test;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;

public class CoCurrentDemo {


    /**
     * 使用CompletableFuture 和  CountDownLatch  进行并发回调
     */

    @Test
    public void testMutiCallBack() {
        CountDownLatch countDownLatch = new CountDownLatch(10);
        //批量异步
        ExecutorService executor = ThreadUtil.getIoIntenseTargetThreadPool();
        long start = System.currentTimeMillis();
        for (int i = 0; i < 10; i++) {
            CompletableFuture<Long> future = CompletableFuture.supplyAsync(() ->
{
                long tid = ThreadUtil.getCurThreadId();
                 try {
                    System.out.println("线程" + tid + "开始了,模拟一下远程调用");
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return tid;
            }, executor);

            future.thenAccept((tid) -> {
                System.out.println("线程" + tid + "结束了");
                countDownLatch.countDown();
            });
        }
        try {
            countDownLatch.await();
            //输出统计结果
            float time = System.currentTimeMillis() - start;

            System.out.println("所有任务已经执行完毕");
            System.out.println("运行的时长为(ms)：" + time);

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

那么，一秒内，只能完成多少个商品的秒杀订单的下单操作呢?

1000毫秒 / 200 =5 个订单

如何达到每秒600个下单呢? 还是要从基础知识里边寻找答案?

分段加锁的思想来源

分段加锁的思想来源与基础知识。

> 我经常在疯狂创客圈社群里边，对小伙伴们强调 基础知识的重要性，反复强调， 《Java 高并发三部曲》 一定要多刷，最好刷三遍。

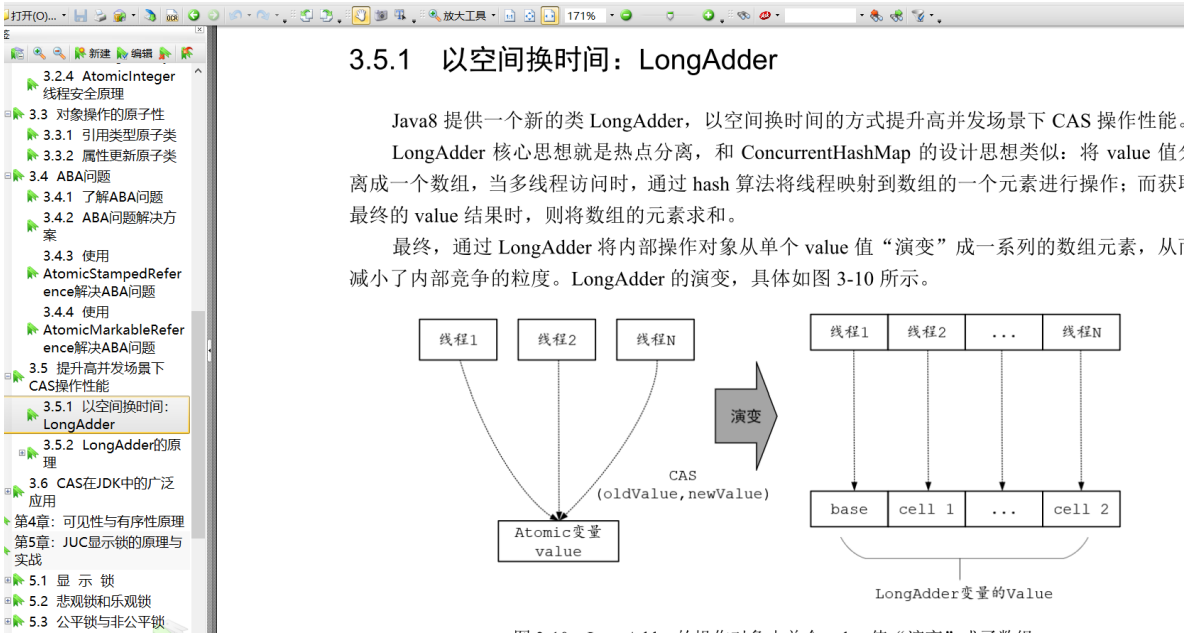
中 《Java 高并发核心编程 卷2》 介绍了 JUC的 LongAdder 和 ConcurrentHashMap的源码和底层原理，他们提升性能的办法是:

> 空间换时间， 分段加锁

尤其是 LongAdder 的实现思想，可以用于 Redis分布式锁 作为性能提升的手段，将 Redis分布式锁 优化为 Redis分段锁。

有关LongAdder 的系统化学习

有关LongAdder 的系统化学习，请参见 《Java 高并发核心编程 卷2》



使用Redis分段锁提升秒杀的并发性能

回到前面的场景:

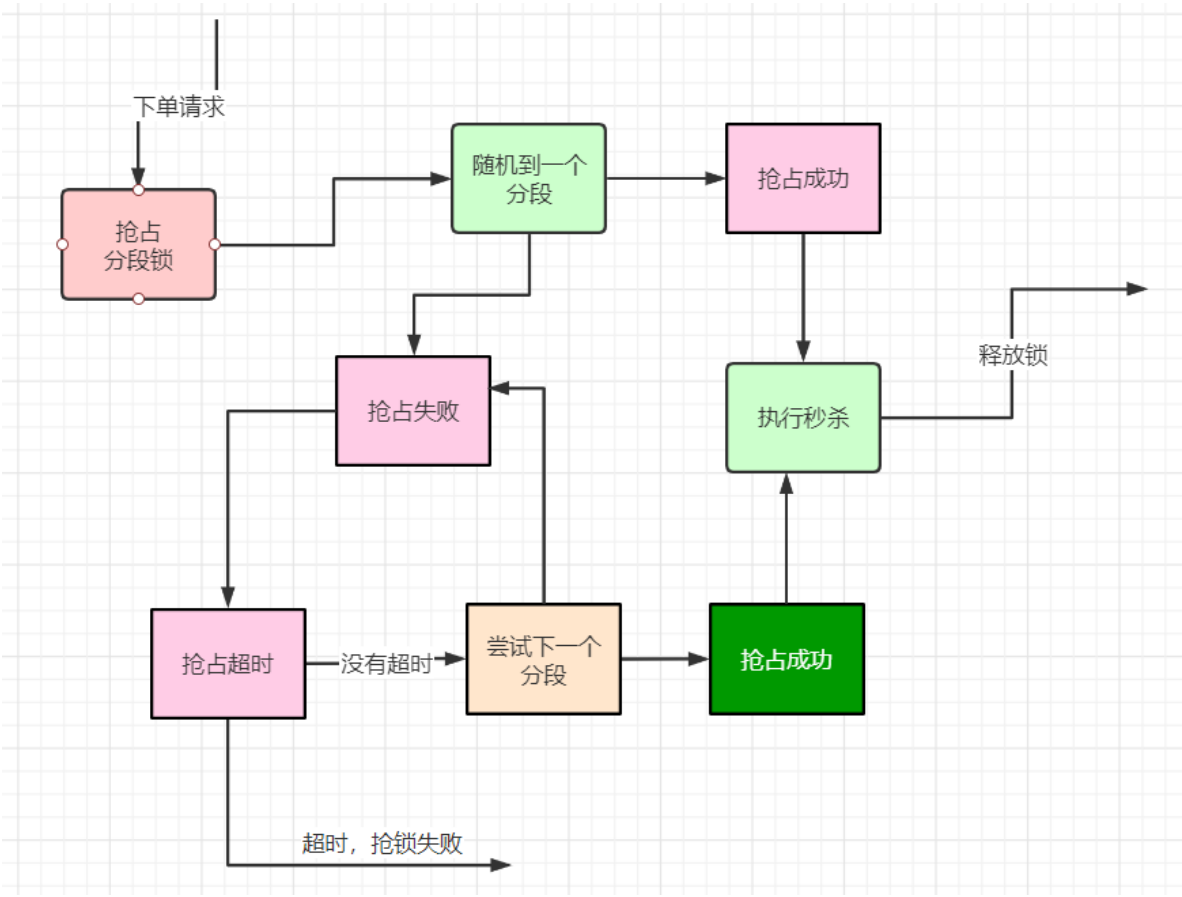
> 假设一个商品1分钟6000订单，每秒的 600个下单操作，
>
> 假设加锁之后，释放锁之前，查库存 -> 创建订单 -> 扣减库存，经过优化，每个IO操作100ms，大概200毫秒，一秒钟5个订单。

如何提高性能呢？ 空间换时间

为了达到每秒600个订单，可以将锁分成 600 /5 =120 个段，反过来， 每个段1秒可以操作5次， 120个段，合起来，及时每秒操作600次。

进行抢夺锁的，如果申请到一个具体的段呢?

每一次使用随机算法，随机到一个分段， 如果不行，就轮询下一个分段，具体的流程，大致如下：



缺点：

> 这个是一个理论的时间预估，没有扣除 尝试下一个分段的 时间， 另外，实际上的性能， 会比理论上差，从咱们实操案例的测试结果，也可以证明这点。

实战： 手写一个Redis分段锁

```
package com.crazymaker.springcloud.standard.lock;
```

```
import com.crazymaker.springcloud.common.util.RandomUtil;
import com.crazymaker.springcloud.common.util.ThreadUtil;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;

@Slf4j
@Data
@AllArgsConstructor
public class JedisMultiSegmentLock implements Lock {

    public static final int NO_SEG = -1;
    //拿到锁的线程
    private Thread thread;

    //拿到锁的状态
    private volatile boolean isLocked = false;

    //段数
    private final int segAmount;

    public static final int DEFAULT_TIMEOUT = 2000;
    public static final Long WAIT_GAT = Long.valueOf(100);

    //内部的锁
    InnerLock[] innerLocks = null;

    //被锁住的分段
    int segmentIndexLocked = NO_SEG;
    /**
     * 默认为2000ms
     */
    long expire = 2000L;
    int segmentIndex = 0;

    public JedisMultiSegmentLock(String lockKey, String requestId, int
segAmount) {
        this.segAmount = segAmount;
        innerLocks = new InnerLock[segAmount];
        for (int i = 0; i < this.segAmount; i++) {
            //每一个分段，加上一个编号
            String innerLockKey = lockKey + ":" + i;
            innerLocks[i] = new InnerLock(innerLockKey, requestId);
        }
        segmentIndex = RandomUtil.randInModLower(this.segAmount);
    }


    /**
     * 获取一个分布式锁 , 超时则返回失败
     *
     * @return 获锁成功 - true | 获锁失败 - false
     */
    @Override
```

```
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException
{

        //本地可重入
        if (isLocked && thread == Thread.currentThread()) {
            return true;
        }
        expire = unit != null ? unit.toMillis(time) : DEFAULT_TIMEOUT;
        long startMillis = System.currentTimeMillis();
        Long millisToWait = expire;

        boolean localLocked = false;

        int turn = 1;

        InnerLock innerLock = innerLocks[segmentIndex];

        while (!localLocked) {

            localLocked = innerLock.lock(expire);
            if (!localLocked) {
                millisToWait = millisToWait - (System.currentTimeMillis() -
startMillis);
                startMillis = System.currentTimeMillis();
                if (millisToWait > 0L) {
                    /**
                     * 还没有超时
                     */
                    ThreadUtil.sleepMilliSeconds(WAIT_GAT);
                    log.info("睡眠一下，重新开始，turn:{},剩余时间：{}", turn++,
millisToWait);

                    segmentIndex++;
                    if (segmentIndex >= this.segAmount) {
                        segmentIndex = 0;
                    }
                    innerLock = innerLocks[segmentIndex];
                } else {
                    log.info("抢锁超时");
                    return false;
                }
            } else {
                segmentIndexLocked = segmentIndex;
                isLocked = true;
                localLocked = true;
                thread = Thread.currentThread();
            }
        }
        return isLocked;
    }


    /**
     * 抢夺锁
     */
    @Override
    public void lock() {
        throw new IllegalStateException(
```

```
                "方法 'lock' 尚未实现!");
    }


    //释放锁
    @Override
    public void unlock() {
        if (segmentIndexLocked == NO_SEG) {
            return;
        }
        this.innerLocks[segmentIndexLocked].unlock();

        segmentIndexLocked = NO_SEG;
        thread = null;
        isLocked = false;
    }

    @Override
    public Condition newCondition() {
        throw new IllegalStateException(
                "方法 'newCondition' 尚未实现!");
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {
        throw new IllegalStateException(
                "方法 'lockInterruptibly' 尚未实现!");

    }


    @Override
    public boolean tryLock() {
        throw new IllegalStateException(
                "方法 'tryLock' 尚未实现!");
    }

}
```

尼恩的忠实建议：

> 强烈参照 LongAdder ， 手写一个Redis分段锁,
>
> 这里， 还是有点复杂，但是很重要，建议大家动手干一票.

- 理论水平的提升，看看视频、看看书，只有两个字，就是需要：多看。
- 实战水平的提升，只有两个字，就是需要：多干。

手写一个Redis分段锁的实操，是高并发实战的重要动手实操之一。

> 有关Redis分段锁的实操的具体材料、源码、问题，欢迎来 疯狂创客圈社群交流。

> 高并发Java发烧友社群 - 疯狂创客圈 总入口 点击了解详情:
>
> 
>

参考文档:

图书：《Netty Zookeeper Redis 高并发实战》 图书简介 - 疯狂创...

Distributed locks with Redis

how-to-do-distributed-locking

redisson watchdog 使用和原理

zookeeper实现分布式锁ava脚本之家

基于Zookeeper 的分布式锁实现 - SegmentFault 思否

分布式锁用 Redis 还是 Zookeeper - 知乎

ZooKeeper分布式锁的实现原理 - 菜鸟奋斗史 - 博客园

https://blog.csdn.net/men_wen/article/details/72853078

硬核推荐：尼恩Java硬核架构班

又名疯狂创客圈社群 VIP

详情：

https://www.cnblogs.com/crazymakercircle/p/9904544.html

架构班（社群 VIP）的起源：

最初的视频，主要是给读者加餐。很多的读者，需要一些高质量的实操、理论视频，所以，我就围绕书，和底层，做了几个实操、理论视频，然后效果还不错，后面就做成迭代模式了。

架构班（社群 VIP）的功能：

提供高质量实操项目整刀真枪的架构指导、快速提升大家的:

- 开发水平
- 设计水平
- 架构水平

弥补业务中 CRUD 开发短板，帮助大家尽早脱离具备 3 高能力，掌握：

- 高性能
- 高并发
- 高可用

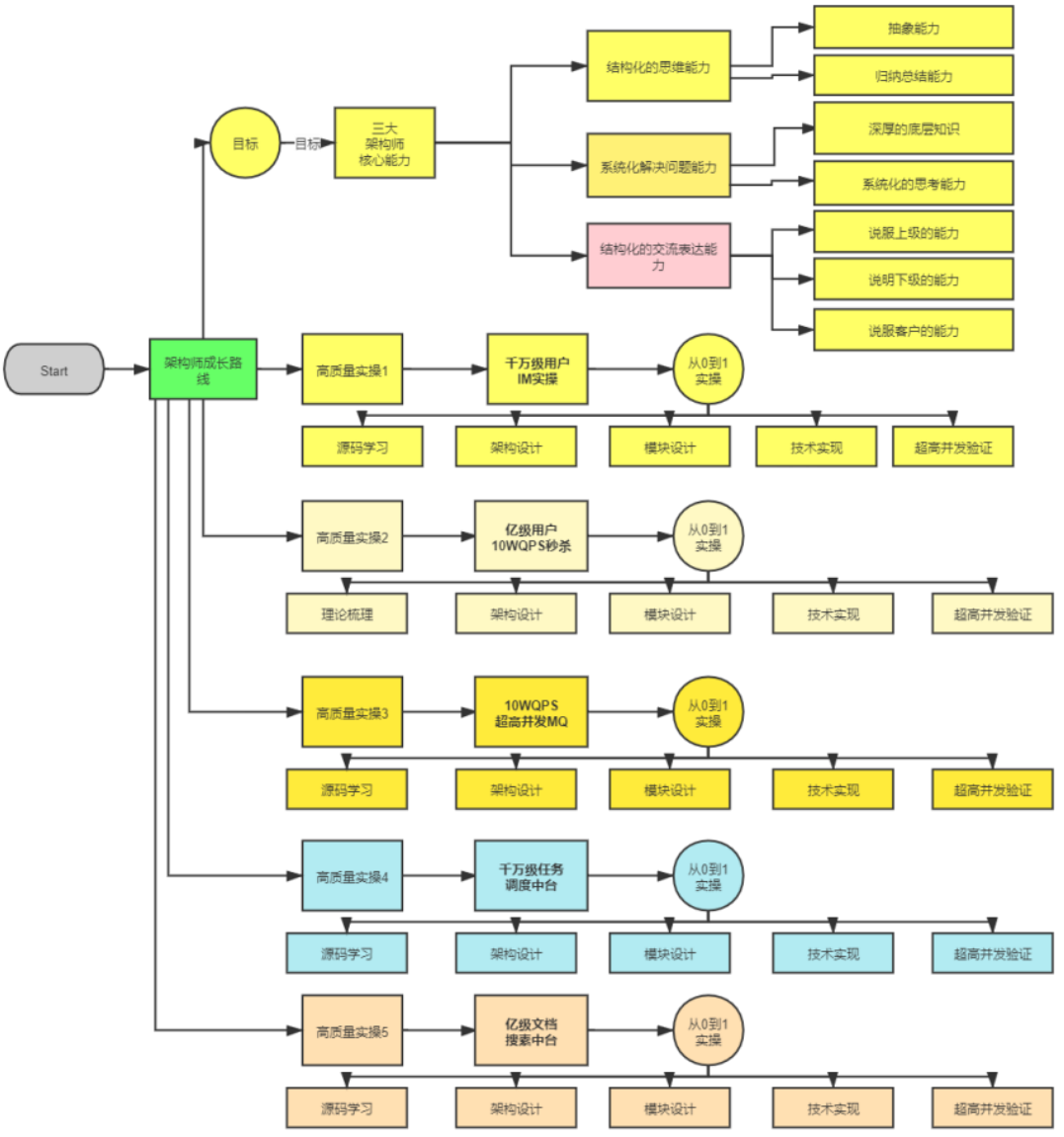
作为一个高质量的架构师成长、人脉社群，把所有的卷王聚焦起来，一起卷：

- 卷高并发实操
- 卷底层原理
- 卷架构理论、架构哲学
- 最终成为顶级架构师，实现人生理想，走向人生巅峰

架构班（社群 VIP）的目的：

- 高质量的实操，大大提升简历的含金量，吸引力，增强面试的召唤率
- 为大家提供九阳真经、葵花宝典，快速提升水平
- 进大厂、拿高薪
- 一路陪伴，提供助学视频和指导，辅导大家成为架构师
- 自学为主，和其他卷王一起，卷高并发实操，卷底层原理、卷大厂面试题，争取狠卷 3 月成高手，狠卷 3 年成为顶级架构师

N 个超高并发实操项目：简历压轴、个顶个精彩



【样章】第 17 章:横扫全网 Rocketmq 视频第 2 部曲: 工业级 rocketmq 高可用(HA)底层原理和实操

工业级 rocketmq 高可用底层原理，包含：消息消费、同步消息、异步消息、单向消息等不同消息的底层原理和源码实现；消息队列非常底层的主从复制、高可用、同步刷盘、异步刷盘等底层原理。

工业级 rocketmq 高可用底层原理和搭建实操，包含：高可用集群的搭建。

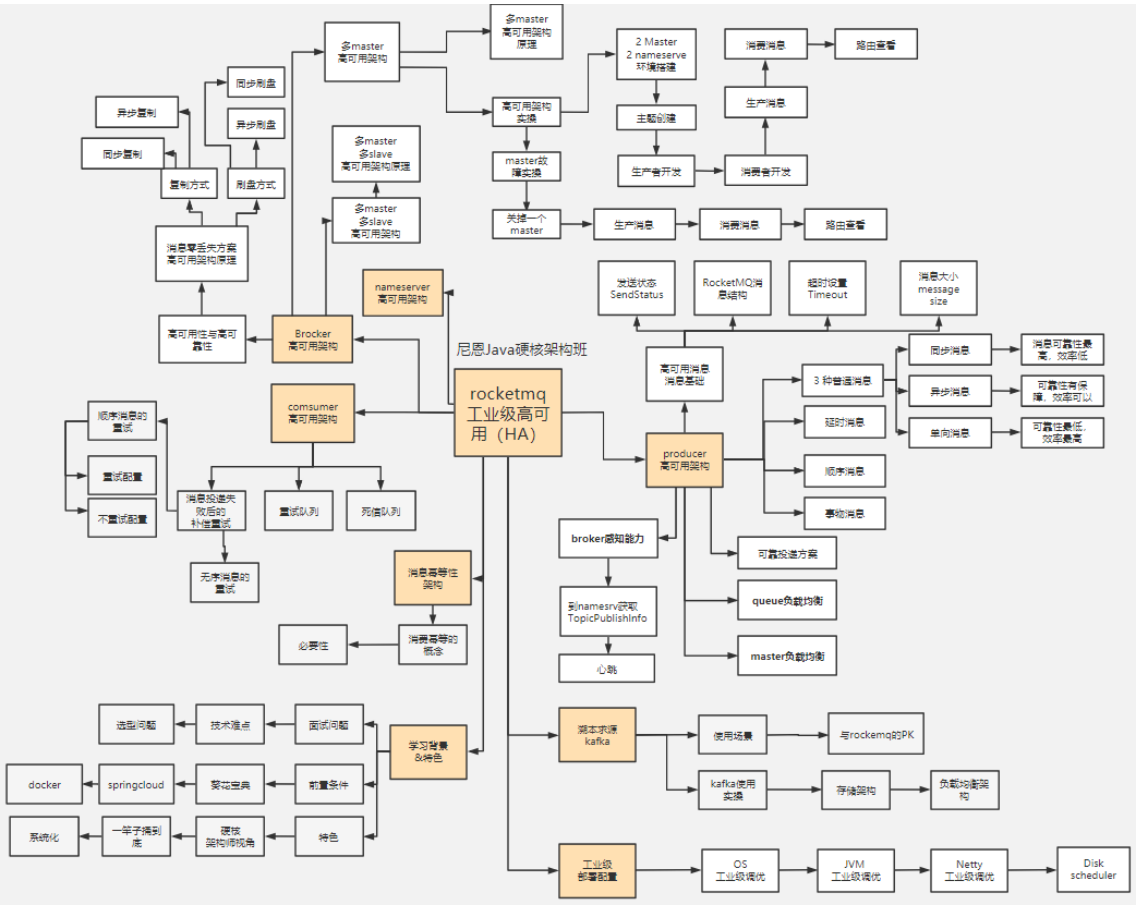
解决以下难题：

1、技术难题：RocketMQ 如何最大限度的保证消息不丢失的呢？RocketMQ 消息如何做到高可靠投递？

2、技术难题：基于消息的分布式事务，核心原理不理解

3、选型难题： kafka or rocketmq ，该娶谁？

下图链接：https://www.processon.com/view/6178e8ae0e3e7416bde9da19



成功案例：2 年翻 3 倍，35 岁卷王成功转型为架构师

详情：http://topcoder.cloud/forum.php?mod=forumdisplay&fid=43&page=1

最新 最后发表 热门 精华

- 成功案例：[1057号卷王] 3年小伙拿到外企offer，薪酬涨了200% — 卷王1号 超级版主 前天 17:41
- 成功案例：[645号卷王] 4年经验卷王逆袭，被毕业后，反涨24W — 卷王1号 超级版主 2022-9-21
- 成功案例：[878号卷王] 小伙8年经验，年薪60W — 卷王1号 超级版主 2022-8-13
- 年薪70W案例：通过尼恩的指导，小伙伴年薪从40W涨到70W — 卷王1号 超级版主 2022-2-11
- 成功案例：[493号卷王] 5年小伙拿满意offer，就业寒冬季逆涨30% — 卷王1号 超级版主 前天 17:43
- 成功案例：[250号卷王] 就业极寒时代，收offer 涨25% — 卷王1号 超级版主 前天 17:38
- 成功案例：[612号卷王] 就业极寒时代，从外包到自研 — 卷王1号 超级版主 前天 17:15
- 成功案例：[913号卷王] 热烈祝贺6年经验卷王，年薪40W — 卷王1号 超级版主 2022-9-21
- 成功案例：[959号卷王] 4年经验卷王，喜获百度、Boss直聘等N个优质offer，最高涨100% — 卷王1号 超级版主 2022-9-21
- 成功案例：[529号卷王] 5年经验卷王喜收2大offer，最高涨5K — 卷王1号 超级版主 2022-9-21
- 成功案例：[811号卷王] 热烈祝贺7年经验卷王，薪酬涨30% — 卷王1号 超级版主 2022-9-21
- 成功案例：[287号卷王] 不惧大寒潮，卷王逆市收4 offer，涨30%，可喜可贺 — 卷王1号 超级版主 2022-5-30
- 成功案例：[1002号卷王] 5月份"被毕业"，改简历后，斩获顶级央企Offer，涨薪7000+ — 卷王1号 超级版主 2022-7-5
- 成功案例：[7号卷王] 热烈祝贺小伙伴涨薪120% — 卷王1号 超级版主 2022-8-13
- 成功案例：[134号卷王] 大三小伙卷1年，斩获顶级央企Offer，成功逆袭 — 卷王1号 超级版主 2022-7-6
- 成功案例：[1008号卷王] 5年经验卷王收42W offer，月涨8000，可喜可贺 — 卷王1号 超级版主 2022-5-30
- 成功案例：[453号卷王] 非全日制 6年卷王喜提3 offer，年薪30W，可喜可贺 — 卷王1号 超级版主 2022-5-21
- 成功案例：[924号卷王] 6年卷王喜提4 offer，最高涨薪9000，可喜可贺 — 卷王1号 超级版主 2022-5-21
- 成功案例：[15号卷王] 4年卷王入职 微软，涨薪50%，可喜可贺 — 卷王1号 超级版主 2022-5-12
- 成功案例：[527号卷王] 4年卷王喜提2 offer，涨薪50%，可喜可贺 — 卷王1号 超级版主 2022-5-13
- 成功案例：[788号卷王] 3年卷王喜提优质Offer，涨薪60% — 卷王1号 超级版主 2022-5-11
- 成功案例：热烈祝贺：非全日制卷王，喜提2个心仪offer，面3家过2家 — 卷王1号 超级版主 2022-4-21

最新 最后发表 热门 精华

- 成功案例：[693号卷王] 二线城市6年卷王喜提4大优质Offer，含央企offer，最高薪酬35W — 卷王1号 超级版主 2022-4-16
- 成功案例：[85号卷王] 双非2本小伙，春招大捷，喜提9个offer，最高薪酬近30万 — 卷王1号 超级版主 2022-4-14
- 成功案例：[741号卷王] 卷王逆袭！6年小伙从很少面试机会到搞定35K*14薪Offer — 卷王1号 超级版主 2022-4-12
- 成功案例：[642号卷王] 热烈祝贺，6年卷王喜提优质国企offer — 卷王1号 超级版主 2022-4-7
- 成功案例：[796号卷王] 热烈祝贺，36岁卷王喜提52万优质offer — 卷王1号 超级版主 2022-3-25
- 成功案例：[15号卷王] 小伙卷1年，涨薪9K+，喜收ebay等多个优质offer — 卷王1号 超级版主 2022-3-24
- 成功案例：[821号卷王] 小伙狠卷3个月，喜提10多个offer — 卷王1号 超级版主 2022-3-21
- 成功案例：[736号卷王] 3年半经验收22k offer，但是小伙志存高远，冲击25k+ — 卷王1号 超级版主 2022-3-20
- 成功案例：热烈祝贺1群小卷王offer拿到手软，甚至拒了阿里offer — 卷王1号 超级版主 2022-3-16
- 简历案例：简历一改，腾讯的邀请就来了！热烈祝贺，小伙收到一大堆面试邀请 — 卷王1号 超级版主 2022-3-10
- 成功案例：祝贺我圈两大超级卷王，一个过了阿里HR面，一个过了阿里2面 — 卷王1号 超级版主 2022-3-10
- 成功案例：小伙伴php转Java，卷1.5年Java，涨薪50%，喜收多个优质offer — 卷王1号 超级版主 2022-3-10
- 成功案例：4年小伙狠卷半年，拿到 移动、京东 两大顶级offer — 尼恩 超级版主 2022-3-5
- 成功案例：[267号卷王] 助力3年经验卷王，拿到蜂巢的17k x 14薪的offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[143号卷王] 二本院校00后卷神，毕业没到一年跳到宇节，年薪45W — 卷王1号 超级版主 2022-2-27
- 成功案例：[494号卷王] 尼恩分布式事务助力卷王拿到 中信银行offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[76号卷王] 2线城市卷王，狠卷1.5年，喜收22K offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[429号卷王] 小伙伴在社群卷5个月，涨8k+ — 卷王1号 超级版主 2022-2-27
- 成功案例：[154号卷王] 双非学校毕业卷王，连拿 京东到家&滴滴 两个大厂Offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[232号卷王] 涨薪10K，继续卷向食物链顶端 — 卷王1号 超级版主 2022-2-27
- 成功案例：狠卷1年技术，喜收 腾讯、阿里、微软三大Offer，最高年薪56W — 卷王1号 超级版主 2022-2-27
- 成功案例：[449号卷王] 应届毕业卷王喜收 滴滴offer，年薪33W — 卷王1号 超级版主 2022-2-27

- 成功案例：[732号卷王] 尼恩助力3年经验卷王收获 京东offer，年薪35W — 卷王1号 超级版主 2022-2-27
- 成功案例：[558号卷王] 2年经验卷王，喜收 网易和阿里子公司两个优质offer — 卷王1号 超级版主 2022-2-27
- 成功案例：[569号卷王] 双非应届生卷王，喜收字节跳动实习offer — 卷王1号 超级版主 2022-2-25
- 成功案例：[420号卷王] 狠卷1年，卷王涨薪80%，涨薪12000元！ — 卷王1号 超级版主 2022-2-25
- 成功案例：[76号卷王] 通过尼恩1年半的指导，专科学历小伙伴从0.8K涨到22K — 卷王1号 超级版主 2022-2-10

- 成功案例：[551号卷王] 小伙伴学完后，成功进入大厂，并且推荐自己的朋友加VIP学习 — 卷王1号 超级版主 2022-2-10
- 成功案例：[214号卷王] 助力2年经验卷王，成功拿到17K月薪 — 卷王1号 超级版主 2022-2-10
- 成功案例：[92号卷王] 课程实操助力社群小伙伴喜收 喜马拉雅Offer — 卷王1号 超级版主 2022-2-10
- 成功案例：社群卷王小伙伴成功过了滴滴三面 获滴滴Offer — 卷王1号 超级版主 2022-2-10
- [612号卷王]滴滴小伙伴，蹲点考察半年，觉得靠谱后加入 疯狂创客圈 — 卷王1号 超级版主 2022-2-10

简历优化后的成功涨薪案例（VIP 含免费简历优化）

修改简历找尼恩（资深简历优化专家）

- 如果面试表达不好，尼恩会提供 简历优化指导
- 如果项目没有亮点，尼恩会提供 项目亮点指导
- 如果面试表达不好，尼恩会提供 面试表达指导

作为 40 岁老架构师，尼恩长期承担技术面试官的角色：

- **从业以来，“阅历”无数，对简历有着点石成金、改头换面、脱胎换骨**的指导能力。
- **尼恩指导过刚刚就业的小白，也指导过 P8 级的老专家，都指导他们上岸。**

如何联系尼恩。尼恩微信，请参考下面的地址：

语雀：https://www.yuque.com/crazymakercircle/gkkw8s/khigna
码云：https://gitee.com/crazymaker/SimpleCrayIM/blob/master/疯狂创客圈总目录.md