# 分布式锁

## 1 分布式锁介绍

### 1.1 什么是分布式

一个大型的系统往往被分为几个子系统来做，一个子系统可以部署在一台机器的多个 JVM(java虚拟机) 上，也可以部署在多台机器上。但是每一个系统不是独立的，不是完全独立的。需要相互通信，共同实现业务功能。

一句话来说：分布式就是通过计算机网络将后端工作分布到多台主机上，多个主机一起协同完成工作。

### 1.2 什么是锁

现实生活中，当我们需要保护一样东西的时候，就会使用锁。例如门锁，车锁等等。很多时候可能许多人会共用这些资源，就会有很多个钥匙。但是有些时候我们希望使用的时候是独自不受打扰的，那么就会在使用的时候从里面反锁，等使用完了再从里面解锁。这样其他人就可以继续使用了。

JAVA程序中，当存在多个线程可以同时改变某个变量（可变共享变量）时，就需要对变量或代码块做同步，使其在修改这种变量时能够线性执行消除并发修改变量，而同步的本质是通过锁来实现的。如 Java 中 synchronize 是在对象头设置标记。

### 1.4 什么是分布式锁

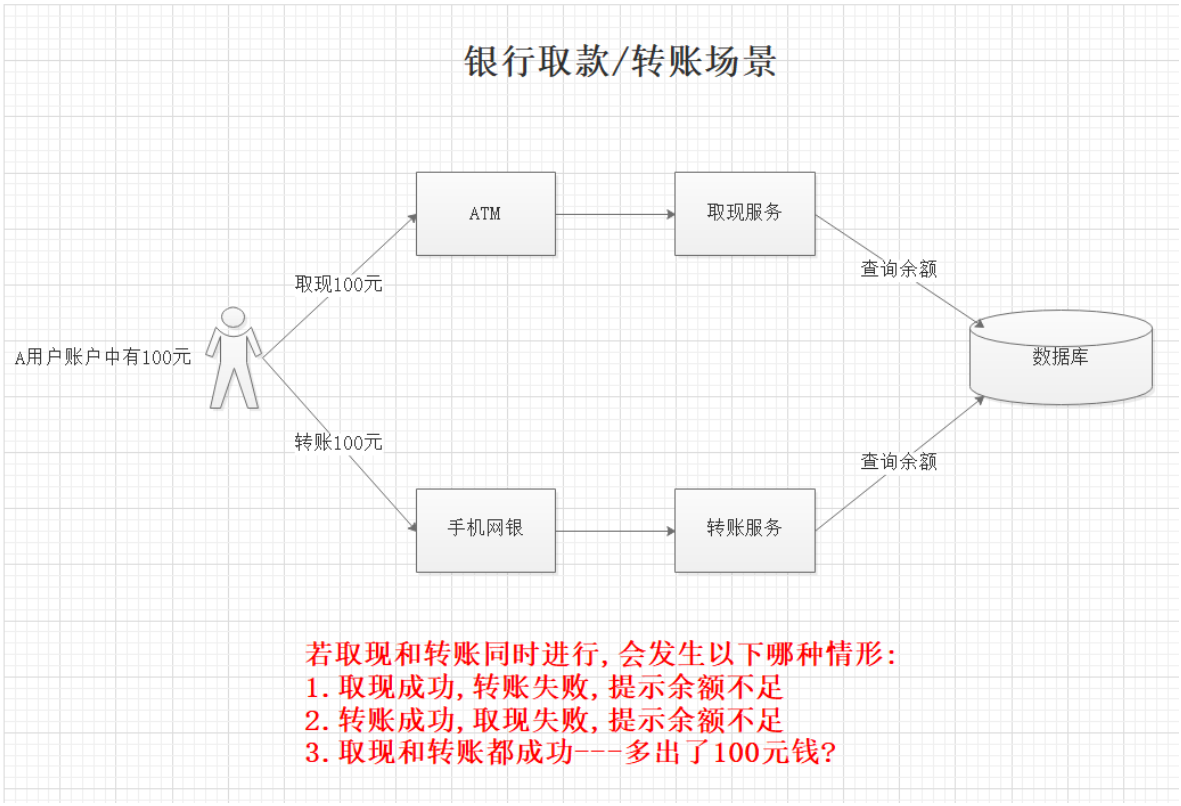
任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。

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。

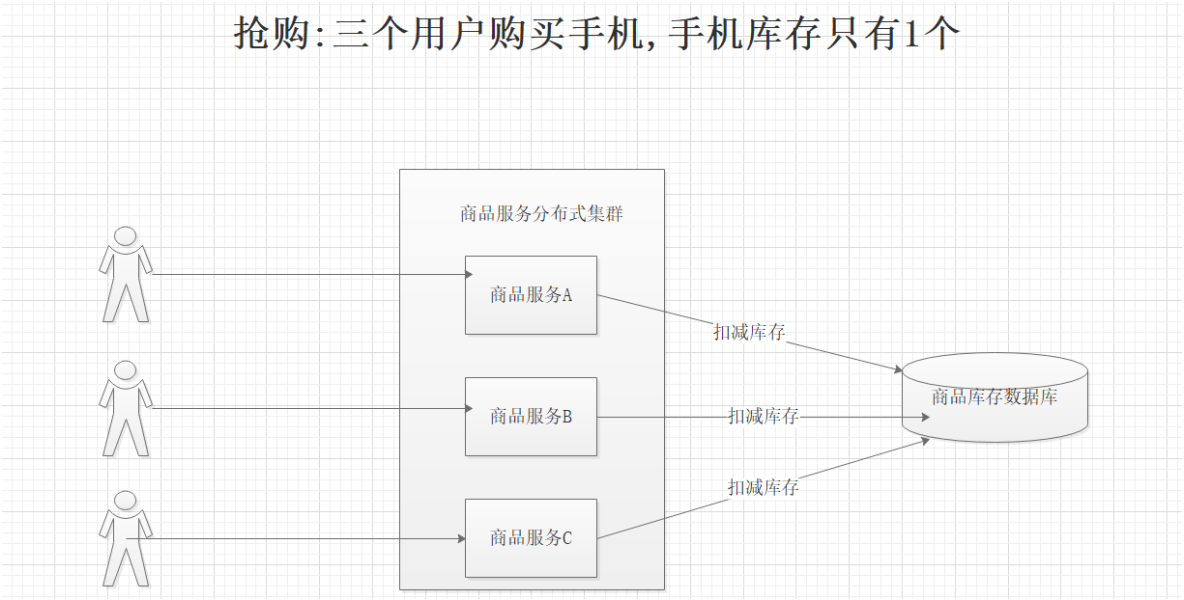
**分布式锁**: 在分布式环境下，多个程序/线程都需要对某一份(或有限制)的数据进行修改时，针对程序进行控制，保证同一时间节点下，只有一个程序/线程对数据进行操作的技术。

### 1.5 分布式锁的真实使用场景

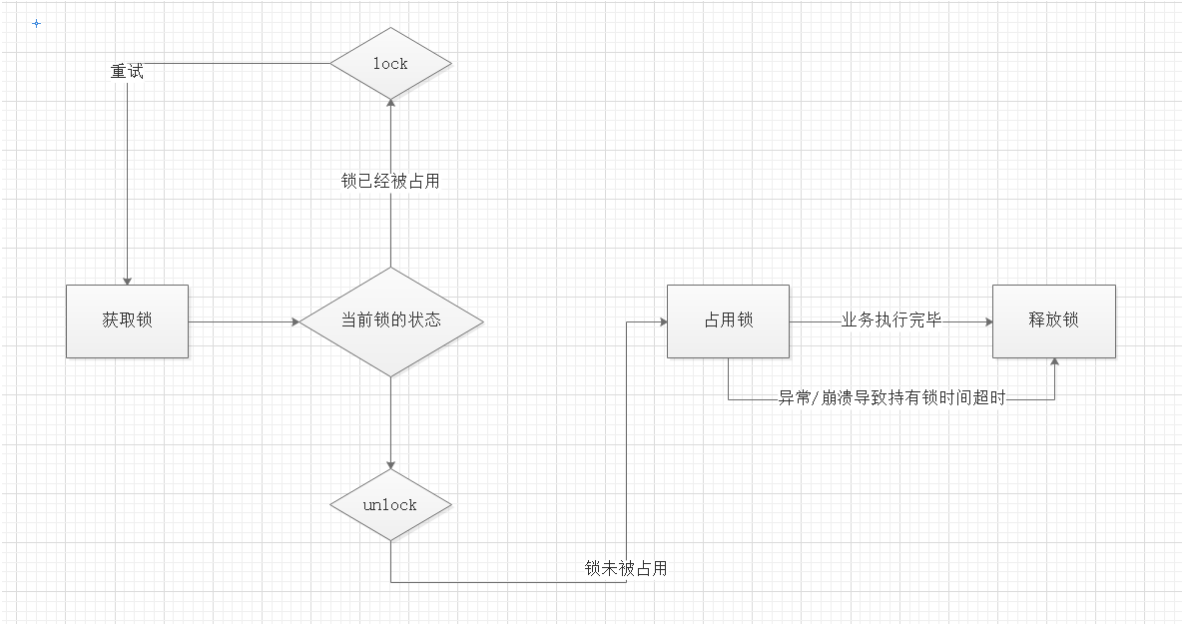
**场景一**:




**场景二：**




## 1.5 分布式锁的执行流程




## 1.6 分布式锁具备的条件

- **互斥性**：同一时刻只能有一个服务(或应用)访问资源。
- **原子性**：一致性要求保证加锁和解锁的行为是原子性的。
- **安全性**：锁只能被持有该锁的服务(或应用)释放。
- **容错性**：在持有锁的服务崩溃时，锁仍能得到释放，避免死锁。
- **高可用**：获取锁和释放锁 要高可用。
- **高性能**：获取锁和释放锁的性能要好。
- **持久性**：锁按业务需要自动续约/自动延期。

# 2.分布式锁的解决方案

## 2.1 数据库实现分布式锁

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>mysqlLockDemo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <!-- Spring boot 版本信息 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.6.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <!-- Environment Settings -->
        <java.version>1.8</java.version>
```

```
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <skipTests>true</skipTests>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.10</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <!-- mybatis-plus begin-->
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-boot-starter</artifactId>
            <version>3.3.1</version>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <!-- mybatis-plus end-->

        <!--web起步依赖-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
</project>
```

application.yml

```
server:
  port: 9001
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://127.0.0.1:3306/mysqlLock?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
    username: root
    password: root
  application:
    name: book
```

MyApplication

```
@SpringBootApplication
@MapperScan(basePackages = {"com.itheima.dao"})
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class,args);
    }
}
```

Book

```
@Data
@TableName(value = "tb_book")
public class Book {

    // 图书ID
    @TableId
    private Integer id;
    // 图书名称
    private String name;

    //数量
    private Integer stock;

    //版本
    private Integer version;
}
```

BookMapper

```
public interface BookMapper extends BaseMapper<Book> {

    @Update("update tb_book set stock=stock-#{saleNum} where id = #{id}")
    void updateNoLock(@Param("id") int id, @Param("saleNum") int saleNum);
}
```

BookService

```
public interface BookService {

    /**
     *
     * @param id
     * @param saleNum 销售数量
     */
    void updateStock(int id, int saleNum);
}
```

BookServiceImpl

```
@Service
public class BookServiceImpl implements BookService {

    @Autowired
```

```
    private BookMapper bookMapper;

    @Override
    @Transactional
    public void updateStock(int id, int saleNum) {

        Book book = bookMapper.selectById(id);
        if (book.getStock()>0){
            bookMapper.updateNoLock(1,1);
        }else {
            System.out.println("没有库存了");
        }
    }

}
```

BookController

```
@RestController
@RequestMapping("/book")
public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping("/updateNoLock/{id}/{saleNum}")
    public void updateNoLock(@PathVariable("id") Integer
id,@PathVariable("saleNum") int saleNum){
        bookService.updateStock(id,saleNum);
    }
}
```

经过jemeter测试可以发现，库存出现了负数。相当于出现了商品超卖。

### 2.1.1 基于数据库表实现

**准备工作**：创建tb_program表，用于记录当前哪个程序正在使用数据

```
CREATE TABLE `tb_program` (
  `program_no` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL
COMMENT '程序的编号'
  PRIMARY KEY (`program_no`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT =
Compact;
```

实现步骤：

1. 程序访问数据时，将程序的编号（insert）存入tb_program表。
2. 当insert成功，代表该程序获得了锁，即可执行逻辑。
3. 当program_no相同的其他程序进行insert时，由于主键冲突会导致insert失败，则代表获取锁失败。
4. 获取锁成功的程序在逻辑执行完以后，删除该数据,代表释放锁。

### 2.1.2 基于条件

对于分布式锁的实现，比较常见的一种就是基于MySQL乐观锁方式来完成，这种方式的思想就是利用MySQL的InnoDB引擎的行锁机制来完成。

对于乐观锁的实现，又分为两种，分别为根据条件和根据版本号。

**实现代码**

BookMapper

```
public interface BookMapper extends BaseMapper<Book> {

    @Update("update tb_book set stock=stock-#{saleNum} where id = #{id} and
stock-#{saleNum}>=0")
    void updateNoLock(@Param("id") int id, @Param("saleNum") int saleNum);
}
```

通过Jemeter进行并发测试，可以发现其已经可以保证库存不会被扣减超卖。

## 2.1.3 基于version版本号

有时我们并没有一些特定的条件让我们去进行判断。此时就会在数据表中新增一个字段版本字段version来进行数据并发控制。

| 名 | 类型 | 长度 | 小数点 | 不是 null | 虚拟 | 键 |
|---|---|---|---|---|---|---|
| id | int | 11 | 0 | ☑ | ☐ | 🔑 1 |
| name | varchar | 50 | 0 | ☐ | ☐ | |
| stock | int | 11 | 0 | ☐ | ☐ | |
| version | int | 10 | 0 | ☐ | ☐ | |

BookMapper

```
@Update("update tb_book set version=version+1,name=#{name} where id = #{id} and
version=#{version}")
int updateByVersion(@Param("id") int id, @Param("name") String name,
@Param("version") int version);
```

BookService

```
void updateByVersion(int id, String name);
```

BookServiceImpl

```
@Override
@Transactional
public void updateByVersion(int id,String name) {

    Book book = bookMapper.selectById(id);
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    int result = bookMapper.updateByVersion(id, name, book.getVersion());
    System.out.println(result==1?"success":"fail");
}
```

VersionTest

```
@Test
public void updateByVersion(){
    new Thread(new Runnable() {
        @Override
        public void run() {
            bookService.updateByVersion(1,"Effective Java");
        }
    }).start();

    bookService.updateByVersion(1,"java核心技术");
}
```

最终运行结果可以发现，只会有一个线程能够修改成功，另外一个修改失败。并且版本号进行了加1。

## 2.2 zookeeper分布式锁

### 2.2.1 实现思想

对于分布式锁的实现，zookeeper天然携带的一些特性能够很完美的实现分布式锁。其内部主要是利用znode节点特性和watch机制完成。

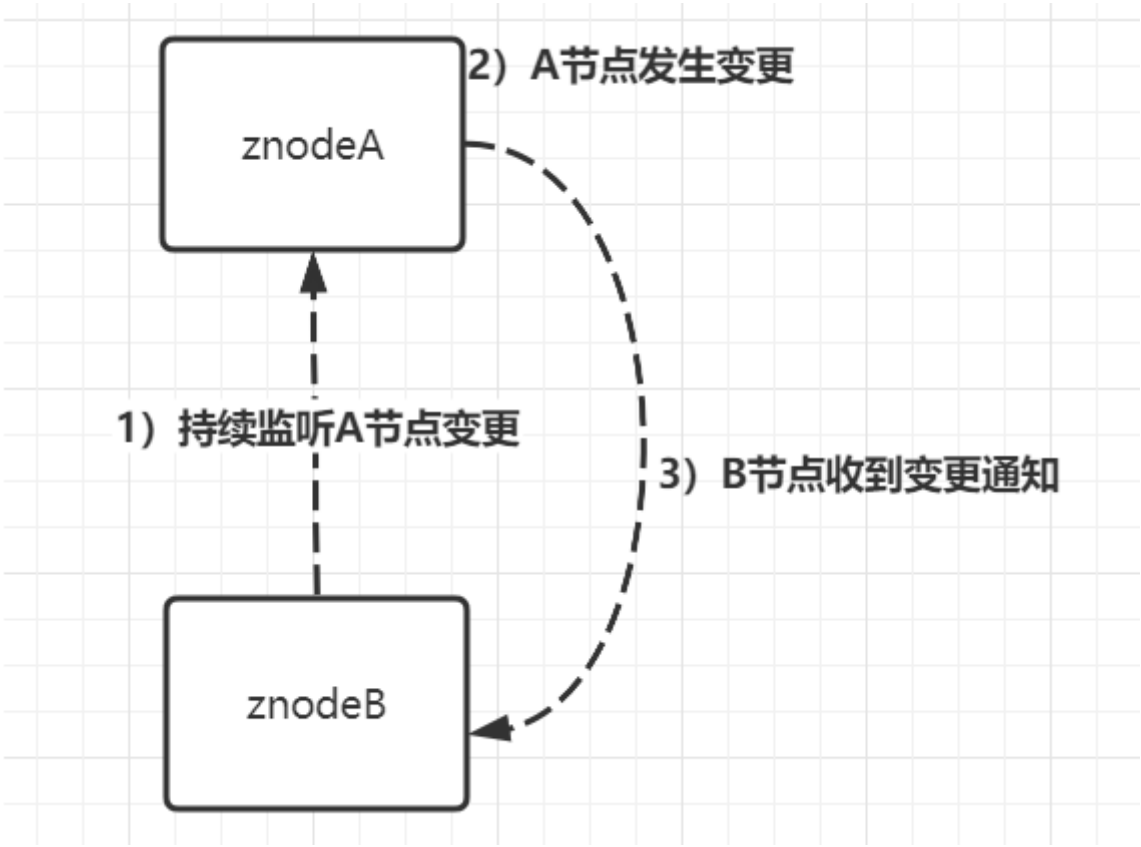
#### 2.2.1.1 znode节点

在zookeeper中节点会分为四类，分别是：

- **持久节点**：一旦创建，则永久存在于zookeeper中，除非手动删除。
- **持久有序节点**：一旦创建，则永久存在于zookeeper中，除非手动删除。同时每个节点都会默认存在节点序号，每个节点的序号都是有序递增的。如demo000001、demo000002.....demo00000N。
- **临时节点**：当节点创建后，一旦服务器重启或宕机，则被自动删除。
- **临时有序节点**：当节点创建后，一旦服务器重启或宕机，则被自动删除。同时每个节点都会默认存在节点序号，每个节点的序号都是有序递增的。如demo000001、demo000002.....demo00000N。

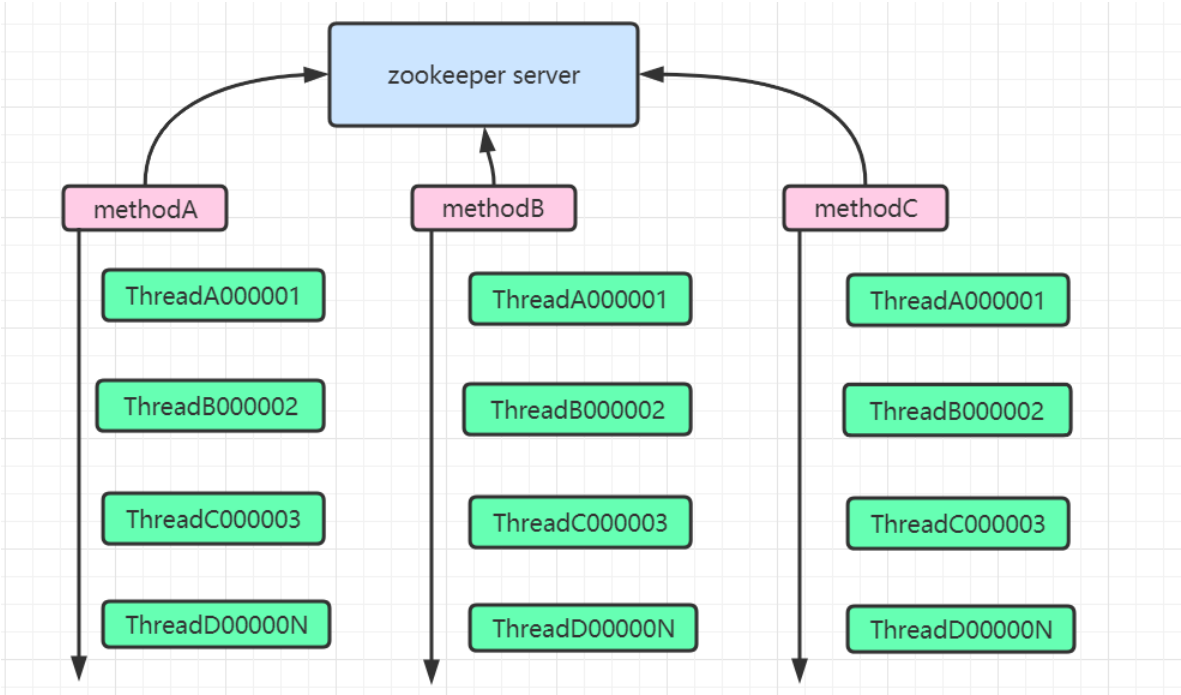
#### 2.2.1.2 watch监听机制

watch监听机制主要用于监听节点状态变更，用于后续事件触发，假设当B节点监听A节点时，一旦A节点发生修改、删除、子节点列表发生变更等事件，B节点则会收到A节点改变的通知，接着完成其他额外事情。

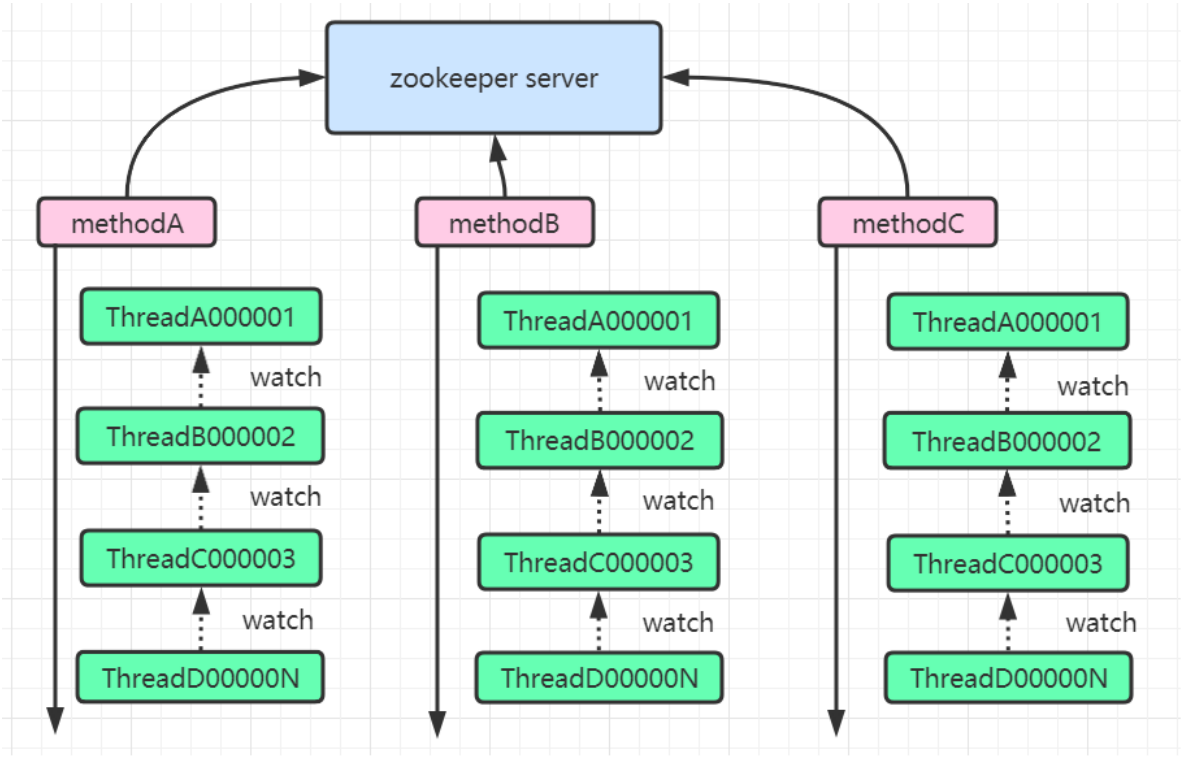



#### 2.2.1.3 实现原理

其实现思想是当某个线程要对方法加锁时，首先会在zookeeper中创建一个与当前方法对应的父节点，接着每个要获取当前方法的锁的线程，都会在父节点下创建一个**临时有序节点**，因为节点序号是递增的，所以后续要获取锁的线程在zookeeper中的序号也是逐次递增的。根据这个特性，当前序号最小的节点一定是首先要获取锁的线程，因此可以规定**序号最小的节点获得锁**。所以，每个线程再要获取锁时，可以判断自己的节点序号是否是最小的，如果是则获取到锁。当释放锁时，只需将自己的临时有序节点删除即可。



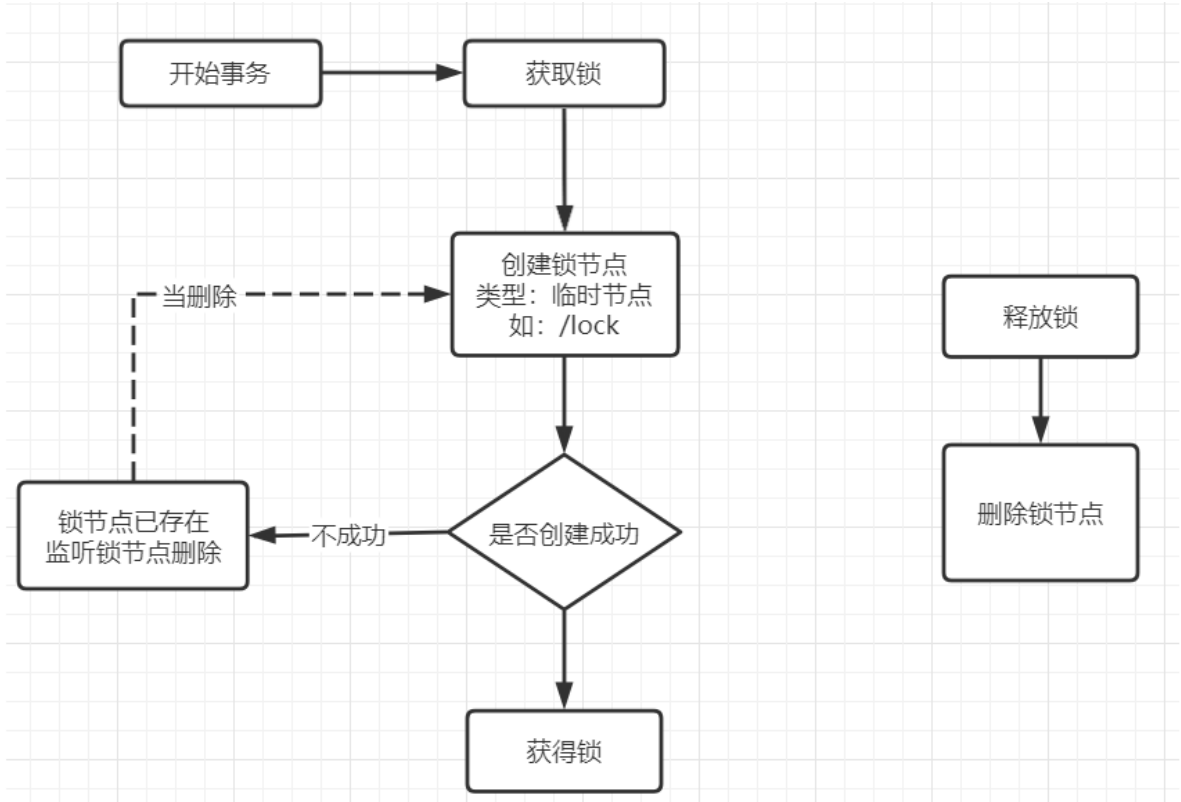

根据上图，在并发下，每个线程都会在对应方法节点下创建属于自己的临时节点，且每个节点都是临时且有序的。那么zookeeper又是如何有序的将锁分配给不同线程呢？ 这里就应用到了watch监听机制。每当添加一个新的临时节点时，其都会基于watcher机制监听着它本身的前一个节点等待前一个节点的通知，当前一个节点删除时，就轮到它来持有锁了。然后依次类推。




## 2.2.2 原理剖析&实现

### 2.2.2.1 低效锁思想&实现

在通过zookeeper实现分布式锁时，有另外一种实现的写法，这种也是非常常见的，但是它的效率并不高，此处可以先对这种实现方式进行探讨。




此种实现方式，只会存在一个锁节点。当创建锁节点时，如果锁节点不存在，则创建成功，代表当前线程获取到锁，如果创建锁节点失败，代表已经有其他线程获取到锁，则该线程会监听锁节点的释放。当锁节点释放后，则继续尝试创建锁节点加锁。

#### 2.2.2.1.1 实现

pom.xml

```
<!--zookeeper-->
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.5.5</version>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
    <exclusions>
        <exclusion>
            <artifactId>slf4j-log4j12</artifactId>
            <groupId>org.slf4j</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

AbstractLock

```
public abstract class AbstractLock {

    //zookeeper服务器地址
    public static final String ZK_SERVER_ADDR="192.168.200.131:2181";

    //zookeeper超时时间
    public static final int CONNECTION_TIME_OUT=30000;
    public static final int SESSION_TIME_OUT=30000;

    //创建zk客户端
    protected ZkClient zkClient = new
ZkClient(ZK_SERVER_ADDR,SESSION_TIME_OUT,CONNECTION_TIME_OUT);

    /**
     * 获取锁
     * @return
     */
    public abstract boolean tryLock();

    /**
     * 等待加锁
     */
    public abstract void waitLock();

    /**
     * 释放锁
     */
    public abstract void releaseLock();
```

```
    public void getLock() {

        String threadName = Thread.currentThread().getName();

        if (tryLock()){
            System.out.println(threadName+":    获取锁成功");
        }else {

            System.out.println(threadName+":    等待获取锁");
            waitLock();
            getLock();
        }
    }
}
```

LowLock

```
public class LowLock extends AbstractLock {

    private static final String LOCK_NODE="/lock_node";

    private CountDownLatch countDownLatch;

    @Override
    public boolean tryLock() {
        if (zkClient == null){
            return false;
        }

        try {
            zkClient.createEphemeral(LOCK_NODE);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    @Override
    public void waitLock() {

        //注册监听
        IZkDataListener listener = new IZkDataListener() {

            //节点数据改变触发
            @Override
            public void handleDataChange(String dataPath, Object data) throws
Exception {

            }

            //节点删除触发
            @Override
            public void handleDataDeleted(String dataPath) throws Exception {
                if (countDownLatch != null){
                    countDownLatch.countDown();
                }
            }
```

```
        };
        zkClient.subscribeDataChanges(LOCK_NODE,listener);

        //如果节点存在，则线程阻塞等待
        if (zkClient.exists(LOCK_NODE)){
            countDownLatch = new CountDownLatch(1);
            try {
                countDownLatch.await();
                System.out.println(Thread.currentThread().getName()+":  等待获取
锁");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        //节点不存在，删除监听
        zkClient.unsubscribeDataChanges(LOCK_NODE,listener);

    }

    @Override
    public void releaseLock() {
        System.out.println(Thread.currentThread().getName()+":    释放锁");
        zkClient.delete(LOCK_NODE);
        zkClient.close();

    }
}
```

LockTest

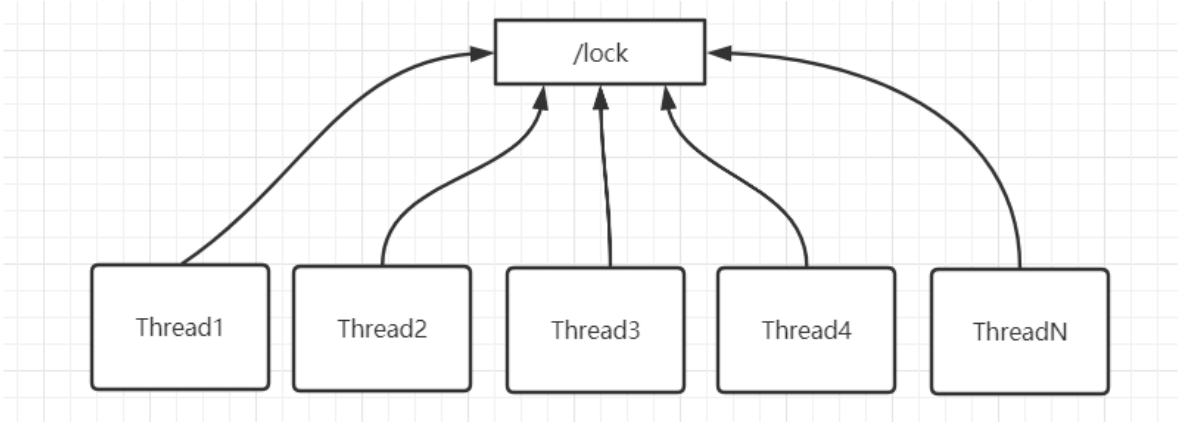
```
public class LockTest {

    public static void main(String[] args) {

        //模拟多个10个客户端
        for (int i=0;i<10;i++) {
            Thread thread = new Thread(new LockRunnable());
            thread.start();
        }
    }

    private static class LockRunnable implements Runnable {
        @Override
        public void run() {

            AbstractLock abstractLock = new LowLock();

            abstractLock.getLock();

            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            abstractLock.releaseLock();
```

```
        }
    }
}
```

测试运行可以发现：当一个线程获取到锁之后，其他线程则一起监听同一个节点，当线程将锁释放后，其他线程再来继续竞争这把锁。

#### 2.2.2.1.2 羊群效应

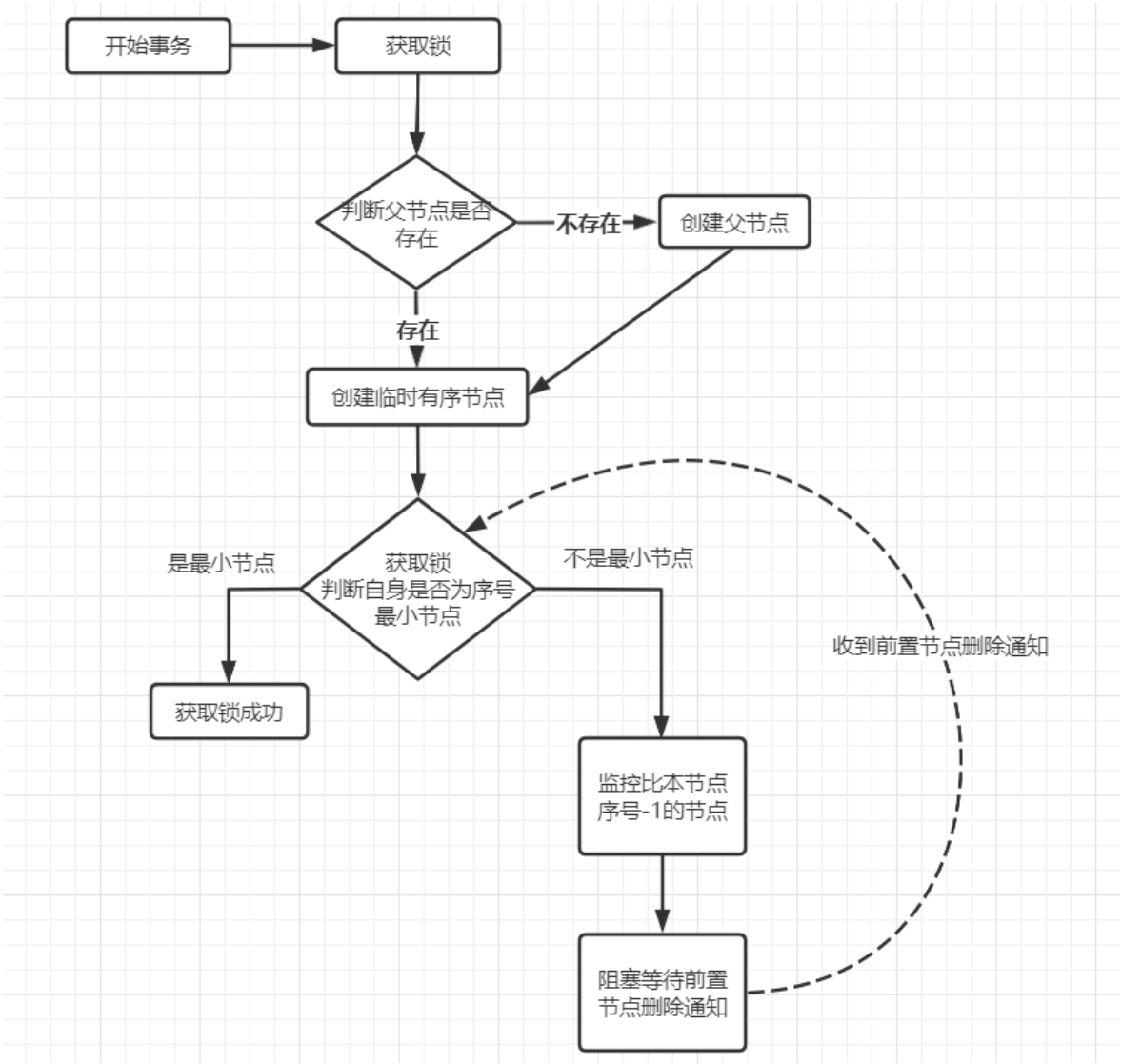
这种方案的低效点就在于，只有一个锁节点，其他线程都会监听同一个锁节点，一旦锁节点释放后，其他线程都会收到通知，然后竞争获取锁节点。这种大量的通知操作会严重降低zookeeper性能，对于这种由于一个被watch的znode节点的变化，而造成大量的通知操作，叫做羊群效应。



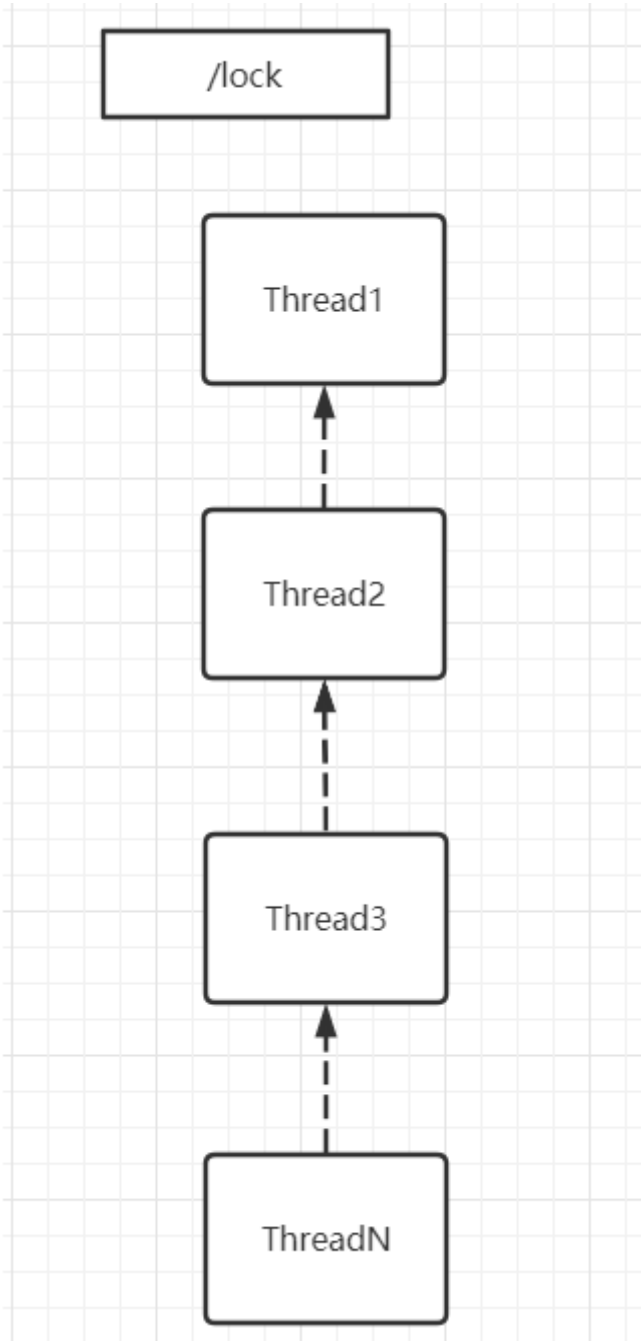

### 2.2.2.2 高效锁思想&实现

#### 2.2.2.2.1 实现

为了避免羊群效应的出现，业界内普遍的解决方案就是，让获取锁的线程产生排队，后一个监听前一个，依次排序。推荐使用这种方式实现分布式锁。




按照上述流程会在根节点下为每一个等待获取锁的线程创建一个对应的临时有序节点，序号最小的节点会持有锁，并且后一个节点只监听其前面的一个节点，从而可以让获取锁的过程有序且高效。




1）定义HighLock类

```
public class HighLock extends AbstractLock {

    private static  String PARENT_NODE_PATH="";

    public HighLock(String parentNodePath){
        PARENT_NODE_PATH = parentNodePath;
    }

    //当前节点路径
    private String currentNodePath;

    //前一个节点的路径
    private String preNodePath;

    private CountDownLatch countDownLatch;

    @Override
```

```
    public boolean tryLock() {

        //判断父节点是否存在
        if (!zkClient.exists(PARENT_NODE_PATH)){
            //父节点不存在，创建持久节点
            try {
                zkClient.createPersistent(PARENT_NODE_PATH);
            } catch (Exception e) {
            }
        }

        //创建第一个临时有序节点
        if (currentNodePath==null || "".equals(currentNodePath)){
            //在父节点下创建临时有序节点
            currentNodePath =
 zkClient.createEphemeralSequential(PARENT_NODE_PATH+"/","lock");
        }

        //不是第一个临时有序节点
        //获取父节点下的所有子节点列表
        List<String> childrenNodeList = zkClient.getChildren(PARENT_NODE_PATH);

        //因为有序号，所以进行升序排序
        Collections.sort(childrenNodeList);

        //判断是否加锁成功，当前节点是否为父节点下序号最小的节点
        if
(currentNodePath.equals(PARENT_NODE_PATH+"/"+childrenNodeList.get(0))){
            //当前节点是序号最小的节点
            return true;
        }else {
            //当前节点不是序号最小的节点，获取其前置节点，并赋值
            int length = PARENT_NODE_PATH.length();
            int currentNodeNumber = Collections.binarySearch(childrenNodeList,
currentNodePath.substring(length + 1));
            preNodePath =
PARENT_NODE_PATH+"/"+childrenNodeList.get(currentNodeNumber-1);
        }

        return false;
    }

    @Override
    public void waitLock() {

        //注册监听
        IZkDataListener listener = new IZkDataListener() {
            @Override
            public void handleDataChange(String dataPath, Object data) throws
Exception {

            }

            @Override
            public void handleDataDeleted(String dataPath) throws Exception {
                if (countDownLatch != null){
                    countDownLatch.countDown();
                }
```

```
            }
        };
        zkClient.subscribeDataChanges(preNodePath,listener);

        //判断前置节点是否存在，存在则阻塞
        if (zkClient.exists(preNodePath)){

            countDownLatch = new CountDownLatch(1);
            try {
                countDownLatch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        //删除监听
        zkClient.unsubscribeDataChanges(preNodePath,listener);

    }


    @Override
    public void releaseLock() {

        zkClient.delete(currentNodePath);
        zkClient.close();
    }
}
```

2）修改测试类并运行，可以发现，线程获取锁的顺序是按照临时节点的序号来进行获取的。

#### 2.2.2.2.2 嵌入业务功能

1）修改BookMapper，添加修改语句

```
@Update("update tb_book set stock=stock-#{saleNum} where id = #{id} and stock-#
{saleNum}>0")
void updateByZk(@Param("id") int id, @Param("saleNum") int saleNum);
```

2）修改BookServiceImpl

```
public void updateByZk(int id, int saleNum) {

    //获取当前方法名
    String methodName = Thread.currentThread().getStackTrace()
[1].getMethodName();
    AbstractLock lock = new HighLock("/"+methodName);

    try {
        //加锁
        lock.getLock();

        bookMapper.updateByZk(1,1);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        //释放锁
```

```
            lock.releaseLock();
        }


    }
```

## 2.3 Redis实现分布式锁

### 2.3.1 单节点Redis实现分布式锁

#### 2.3.1.1 原理&实现

分布式锁的一个很重要的特性就是互斥性，同一时间内多个调用方加锁竞争，只能有一个调用方加锁成功。而redis是基于单线程模型的，可以利用这个特性让调用方的请求排队，对于并发请求，只会有一个请求能获取到锁。

redis实现分布式锁也很简单，基于客户端的几个API就可以完成，主要涉及三个核心API：

- setNx()：向redis中存key-value，只有当key不存在时才会设置成功，否则返回0。用于体现互斥性。
- expire()：设置key的过期时间，用于避免死锁出现。
- delete()：删除key，用于释放锁。

**1）编写工具类实现加锁**

通过jedis.set进行加锁，如果返回值是OK，代表加锁成功

如果加锁失败，则自旋不断尝试获取锁，同时在一定时间内如果仍没有获取到锁，则退出自旋，不再尝试获取锁。

requestId：用于标识当前每个线程自己持有的锁标记

```
public class SingleRedisLock {

    JedisPool jedisPool = new JedisPool("192.168.200.128",6379);

    //锁过期时间
    protected long internalLockLeaseTime = 30000;

    //获取锁的超时时间
    private long timeout = 999999;

    /**
     * 加锁
     * @param lockKey 锁键
     * @param requestId 请求唯一标识
     * @return
     */
    SetParams setParams = SetParams.setParams().nx().px(internalLockLeaseTime);

    public boolean tryLock(String lockKey, String requestId){

        String threadName = Thread.currentThread().getName();

        Jedis jedis = this.jedisPool.getResource();

        Long start = System.currentTimeMillis();
```

```
        try{
            for (;;){
                String lockResult = jedis.set(lockKey, requestId, setParams);
                if ("OK".equals(lockResult)){
                    System.out.println(threadName+":    获取锁成功");
                    return true;
                }
                //否则循环等待，在timeout时间内仍未获取到锁，则获取失败
                System.out.println(threadName+":    获取锁失败，等待中");
                long l = System.currentTimeMillis() - start;
                if (l>=timeout) {
                    return false;
                }
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }finally {
            jedis.close();
        }

    }
}
```

2）编写工具类实现解锁

解锁时，要避免当前线程将别人的锁释放掉。假设线程A加锁成功，当过了一段时间线程A来解锁，但线程A的锁已经过期了，在这个时间节点，线程B也来加锁，因为线程A的锁已经过期，所以线程B时可以加锁成功的。此时，就会出现问题，线程A将线程B的锁给释放了。

对于这个问题，就需要使用到加锁时的requestId。当解锁时要判断当前锁键的value与传入的value是否相同，相同的话，则代表是同一个人，可以解锁。否则不能解锁。

但是对于这个操作，有非常多的人，会先查询做对比，接着相同则删除。虽然思路是对的，但是忽略了一个问题，**原子性**。判断与删除分成两步执行，则无法保证原子性，一样会出现问题。所以解锁时不仅要保证加锁和解锁是同一个人还要保证解锁的原子性。因此结合lua脚本完成查询&删除操作。

```
/**
     * 解锁
     * @param lockKey 锁键
     * @param requestId 请求唯一标识
     * @return
     */
public boolean releaseLock(String lockKey,String requestId){

    String threadName = Thread.currentThread().getName();
    System.out.println(threadName+": 释放锁");
    Jedis jedis = this.jedisPool.getResource();

    String lua =
        "if redis.call('get',KEYS[1]) == ARGV[1] then" +
        "    return redis.call('del',KEYS[1]) " +
        "else" +
        "    return 0 " +
        "end";
```

```
        try {
            Object result = jedis.eval(lua, Collections.singletonList(lockKey),
                                       Collections.singletonList(requestId));
            if("1".equals(result.toString())){
                return true;
            }
            return false;
        }finally {
            jedis.close();
        }

    }
```

**3）编写测试类**

```
public class LocalTest {

    public static void main(String[] args) {

        //模拟多个5个客户端
        for (int i=0;i<5;i++) {
            Thread thread = new Thread(new LockRunnable());
            thread.start();
        }
    }

    private static class LockRunnable implements Runnable {
        @Override
        public void run() {

            SingleRedisLock singleRedisLock = new SingleRedisLock();

            String requestId = UUID.randomUUID().toString();
            boolean lockResult = singleRedisLock.tryLock("lock", requestId);
            if (lockResult){

                try {
                    TimeUnit.SECONDS.sleep(5);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            singleRedisLock.releaseLock("lock",requestId);
        }
    }
}
```

此时可以发现，多线程会竞争同一把锁，且没有获取获取到锁的线程会自旋不断尝试去获取锁。每当一个线程将锁释放后，则会有另外一个线程持有锁。依次类推。

#### 2.3.1.2 单节点问题

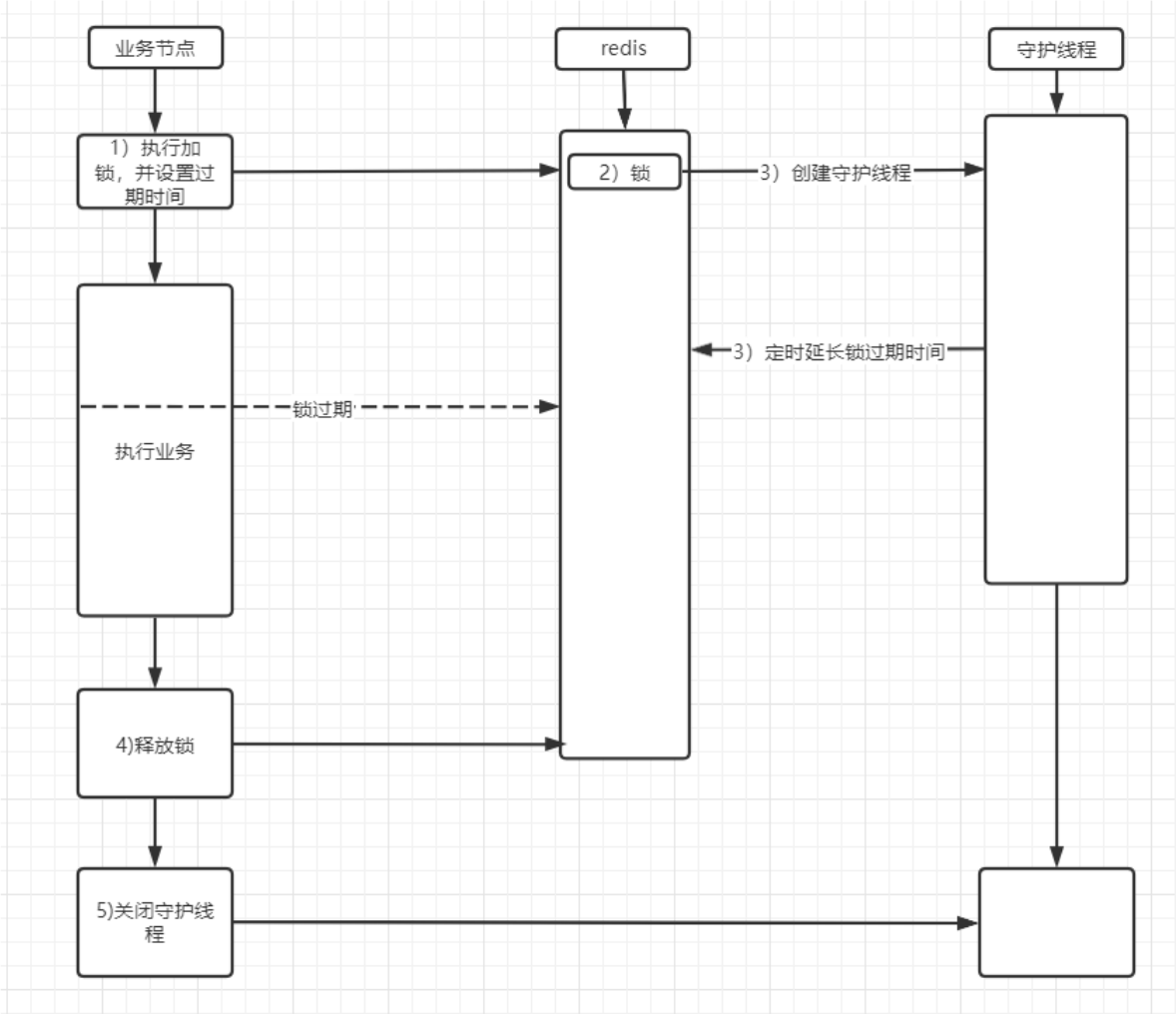
- 锁续期

当对业务进行加锁时，锁的过期时间，绝对不能想当然的设置一个值。假设线程A在执行某个业务时加锁成功并设置锁过期时间。但该业务执行时间过长，业务的执行时间超过了锁过期时间，那么在业务还没执行完时，锁就自动释放了。接着后续线程就可以获取到锁，又来执行该业务。就会造成线程A还没执行完，后续线程又来执行，导致同一个业务逻辑被重复执行。因此对于锁的超时时间，需要结合着业务执行时间来判断，让锁的过期时间大于业务执行时间。

上面的方案是一个基础解决方案，但是仍然是有问题的。

业务执行时间的影响因素太多了，无法确定一个准确值，只能是一个估值。无法百分百保证业务执行期间，锁只能被一个线程占有。

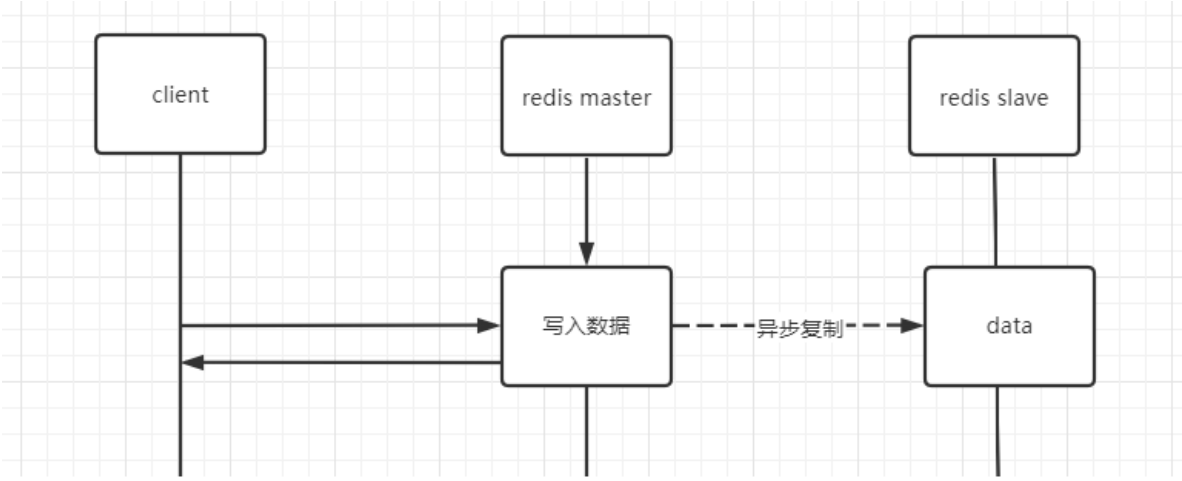
如想保证的话，可以在创建锁的同时创建一个守护线程，同时定义一个定时任务每隔一段时间去为未释放的锁增加过期时间。当业务执行完，释放锁后，再关闭守护线程。 这种实现思想可以用来解决锁续期。




- 服务单点&集群问题

在单点redis虽然可以完成锁操作，可一旦redis服务节点挂掉了，则无法提供锁操作。

在生产环境下，为了保证redis高可用，会采用**异步复制**方法进行主从部署。当主节点写入数据成功，会异步的将数据复制给从节点，并且当主节点宕机，从节点会被提升为主节点继续工作。假设主节点写入数据成功，在没有将数据复制给从节点时，主节点宕机。则会造成提升为主节点的从节点中是没有锁信息的，其他线程则又可以继续加锁，导致互斥失效。




### 2.3.2 Redisson实现分布式锁

redisson是redis官网推荐实现分布式锁的一个第三方类库。其内部完成的功能非常强大，对各种锁都有实现，同时对于使用者来说非常简单，让使用者能够将更多的关注点放在业务逻辑上。此处重点利用Redisson解决单机Redis锁产生的两个问题。

#### 2.3.2.1 单机Redisson实现分布式锁

基于redisson实现分布式锁很简单，直接基于lock()&unlock()方法操作即可。

**1）添加依赖**

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
<!--Redis分布式锁-->
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson-spring-boot-starter</artifactId>
    <version>3.13.1</version>
</dependency>
```

**2）修改配置文件**

```
server:
  redis:
    host: 192.168.200.150
    port: 6379
    database: 0
    jedis:
      pool:
        max-active: 500
        max-idle: 1000
        min-idle: 4
```

**3）修改springboot启动类**

```
@Value("${spring.redis.host}")
private String host;
```

```
@Value("${spring.redis.port}")
private String port;

@Bean
public RedissonClient redissonClient(){
    RedissonClient redissonClient;

    Config config = new Config();
    String url = "redis://" + host + ":" + port;
    config.useSingleServer().setAddress(url);

    try {
        redissonClient = Redisson.create(config);
        return redissonClient;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

#### 4）定义锁工具类

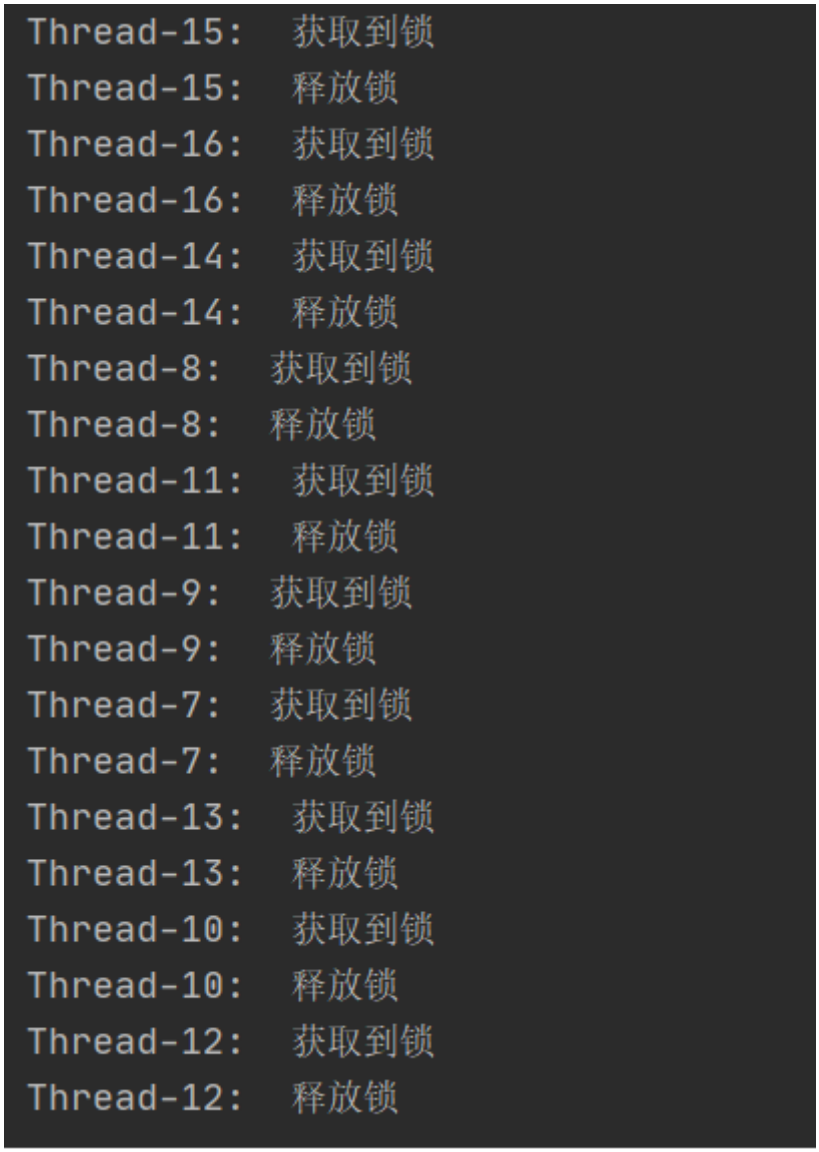
```
@Component
public class RedissonLock {

    @Autowired
    private RedissonClient redissonClient;

    /**
     * 加锁
     * @param lockKey
     * @return
     */
    public boolean addLock(String lockKey){

        try {
            if (redissonClient == null){
                System.out.println("redisson client is null");
                return false;
            }

            RLock lock = redissonClient.getLock(lockKey);

            //设置锁超时时间为5秒，到期自动释放
            lock.lock(10, TimeUnit.SECONDS);

            System.out.println(Thread.currentThread().getName()+":  获取到锁");

            //加锁成功
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    public boolean releaseLock(String lockKey){
```

```
        try{
            if (redissonClient == null){
                System.out.println("redisson client is null");
                return false;
            }

            RLock lock = redissonClient.getLock(lockKey);
            lock.unlock();
            System.out.println(Thread.currentThread().getName()+":  释放锁");
            return true;
        }catch (Exception e){
            e.printStackTrace();
            return false;
        }
    }
}
```

**5）测试**

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class RedissonLockTest {

    @Autowired
    private RedissonLock redissonLock;

    @Test
    public void easyLock(){
        //模拟多个10个客户端
        for (int i=0;i<10;i++) {
            Thread thread = new Thread(new LockRunnable());
            thread.start();
        }

        try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private class LockRunnable implements Runnable {
        @Override
        public void run() {
            redissonLock.addLock("demo");
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            redissonLock.releaseLock("demo");
        }
    }
}
```

**6）执行效果**

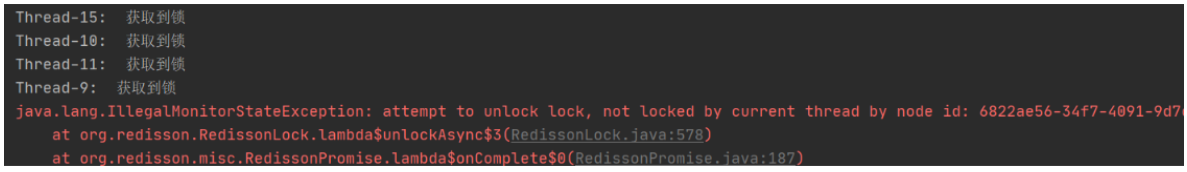



根据执行效果可知，多线程并发获取所时，当一个线程获取到锁，其他线程则获取不到，并且其内部会不断尝试获取锁，当持有锁的线程将锁释放后，其他线程则会继续去竞争锁。

#### 2.3.2.2 看门狗

lock()方法虽然可以设置过期时间，当到期后，锁就会自动释放，因此在业务执行中，通过lock()加锁会存在隐患。Redisson也考虑到了这点，所以提供了看门狗机制。

改造锁示例代码，让锁超时时间为1秒，但是业务执行时，需要耗时3秒，此时执行可以发现，多线程间在上一个锁没有释放的情况下，后续线程又获取到了锁。但是解锁的时候，出现异常，因为加锁时的唯一标识与解锁时的唯一标识发生了改变，造成死锁。




因为业务执行多久无法确定一个准确值，所以在看门狗的实现中，不需要对锁key设置过期时间，当过期时间为-1时，这时会启动一个定时任务，在业务释放锁之前，会一直不停的增加这个锁的有效时间，从而保证在业务执行完毕前，这把锁不会被提前释放掉。

要开启看门狗机制也很简单，只需要将加锁时使用**lock()**改为**tryLock()**即可。

```
//设置锁超时时间为5秒，到期自动释放
//lock.lock(5, TimeUnit.SECONDS);
//获取锁的等待时间
lock.tryLock( time: 100, TimeUnit.SECONDS);
```

并且在lock的源码中，如果没有设置锁超时，默认过期时间为30秒即watchdog每隔30秒来进行一次续期，该值可以修改。

```
config.setLockWatchdogTimeout(3000L);
```

进行测试，当加锁后，线程睡眠10秒钟，然后释放锁，可以看到在这段时间内，当前线程会一直持有锁，直到锁释放。在多线程环境下，也是阻塞等待进行锁的获取。

#### 2.3.2.3 红锁

当在单点redis中实现redis锁时，一旦redis服务器宕机，则无法进行锁操作。因此会考虑将redis配置为主从结构，但在主从结构中，数据复制是异步实现的。假设在主从结构中，master会异步将数据复制到slave中，一旦某个线程持有了锁，在还没有将数据复制到slave时，master宕机。则slave会被提升为master，但被提升为slave的master中并没有之前线程的锁信息，那么其他线程则又可以重新加锁。

##### 2.3.2.3.1 redlock算法

redlock是一种基于多节点redis实现分布式锁的算法，可以有效解决redis单点故障的问题。官方建议搭建**五台redis**服务器对redlock算法进行实现。

在redis官网中，对于redlock算法的实现思想也做了详细的介绍。地址：https://redis.io/topics/distlock。整个实现过程分为五步：

1）记录获取锁前的当前时间

2）使用相同的key，value获取所有redis实例中的锁，并且设置获取锁的时间要远远小于锁自动释放的时间。假设锁自动释放时间是10秒，则获取时间应在5-50毫秒之间。通过这种方式避免客户端长时间等待一个已经关闭的实例，如果一个实例不可用了，则尝试获取下一个实例。

3）客户端通过获取所有实例的锁后的时间减去第一步的时间，得到的差值要小于锁自动释放时间，避免拿到一个已经过期的锁。并且要有超过半数的redis实例成功获取到锁，才算最终获取锁成功。如果不是超过半数，有可能出现多个客户端重复获取到锁，导致锁失效。

4）当已经获取到锁，那么它的真正失效时间应该为：过期时间-第三步的差值。

5）如果客户端获取锁失败，则在所有redis实例中释放掉锁。为了保证更高效的获取锁，还可以设置重试策略，在一定时间后重新尝试获取锁，但不能是无休止的，要设置重试次数。

虽然通过redlock能够更加有效的防止redis单点问题，但是仍然是存在隐患的。假设redis没有开启持久化，clientA获取锁后，所有redis故障重启，则会导致clientA锁记录消失，clientB仍然能够获取到锁。这种情况虽然发生几率极低，但并不能保证肯定不会发生。

保证的方案就是开始AOF持久化，但是要注意同步的策略，使用每秒同步，如果在一秒内重启，仍然数据丢失。使用always又会造成性能急剧下降。

官方推荐使用默认的AOF策略即每秒同步，且在redis停掉后，要在ttl时间后再重启。 缺点就是ttl时间内redis无法对外提供服务。

##### 2.3.2.3.2 实现

redisson对于红锁的实现已经非常完善，通过其内部提供的api既可以完成红锁的操作。

**1）新建配置类**

```
@Configuration
public class RedissonRedLockConfig {

    public RedissonRedLock initRedissonClient(String lockKey){

        Config config1 = new Config();

 config1.useSingleServer().setAddress("redis://192.168.200.150:7000").setDatabas
e(0);
        RedissonClient redissonClient1 = Redisson.create(config1);

        Config config2 = new Config();

 config2.useSingleServer().setAddress("redis://192.168.200.150:7001").setDatabas
e(0);
        RedissonClient redissonClient2 = Redisson.create(config2);

        Config config3 = new Config();

 config3.useSingleServer().setAddress("redis://192.168.200.150:7002").setDatabas
e(0);
        RedissonClient redissonClient3 = Redisson.create(config3);

        Config config4 = new Config();

 config4.useSingleServer().setAddress("redis://192.168.200.150:7003").setDatabas
e(0);
        RedissonClient redissonClient4 = Redisson.create(config4);

        Config config5 = new Config();

 config5.useSingleServer().setAddress("redis://192.168.200.150:7004").setDatabas
e(0);
        RedissonClient redissonClient5 = Redisson.create(config5);

        RLock rLock1 = redissonClient1.getLock(lockKey);
        RLock rLock2 = redissonClient2.getLock(lockKey);
        RLock rLock3 = redissonClient3.getLock(lockKey);
        RLock rLock4 = redissonClient4.getLock(lockKey);
        RLock rLock5 = redissonClient5.getLock(lockKey);

        RedissonRedLock redissonRedLock = new
RedissonRedLock(rLock1,rLock2,rLock3,rLock4,rLock5);

        return redissonRedLock;
    }
}
```

**2）新建测试类,完成加锁与解锁操作**

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class RedLockTest {

    @Autowired
```

```
    private RedissonRedLockConfig redissonRedLockConfig;

    @Test
    public void easyLock(){
        //模拟多个10个客户端
        for (int i=0;i<10;i++) {
            Thread thread = new Thread(new RedLockTest.RedLockRunnable());
            thread.start();
        }

        try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private class RedLockRunnable implements Runnable {
        @Override
        public void run() {
            RedissonRedLock redissonRedLock =
redissonRedLockConfig.initRedissonClient("demo");

            try {
                boolean lockResult = redissonRedLock.tryLock(100, 10,
TimeUnit.SECONDS);

                if (lockResult){
                    System.out.println("获取锁成功");
                    TimeUnit.SECONDS.sleep(3);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                redissonRedLock.unlock();
                System.out.println("释放锁");
            }
        }
    }
}
```

#### 2.3.2.3.3 加锁源码分析

```
public boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws
InterruptedException {
    long newLeaseTime = -1;
    if (leaseTime != -1) {
        newLeaseTime = unit.toMillis(waitTime)*2;
    }

    long time = System.currentTimeMillis();
    long remainTime = -1;
    if (waitTime != -1) {
        remainTime = unit.toMillis(waitTime);
    }
    long lockWaitTime = calcLockWaitTime(remainTime);
    /**
```

```
     * 1. 允许加锁失败节点个数限制（N-(N/2+1)）,当前假设五个节点，则允许失败节点数为2
    */
    int failedLocksLimit = failedLocksLimit();
    /**
     * 2. 遍历所有节点执行lua加锁，用于保证原子性
    */
    List<RLock> acquiredLocks = new ArrayList<>(locks.size());
    for (ListIterator<RLock> iterator = locks.listIterator();
iterator.hasNext();) {
        RLock lock = iterator.next();
        boolean lockAcquired;
        /**
         *  3.对节点尝试加锁
        */
        try {
            if (waitTime == -1 && leaseTime == -1) {
                lockAcquired = lock.tryLock();
            } else {
                long awaitTime = Math.min(lockWaitTime, remainTime);
                lockAcquired = lock.tryLock(awaitTime, newLeaseTime,
TimeUnit.MILLISECONDS);
            }
        } catch (RedisResponseTimeoutException e) {
            // 如果抛出这类异常，为了防止加锁成功，但是响应失败，需要解锁所有节点
            unlockInner(Arrays.asList(lock));
            lockAcquired = false;
        } catch (Exception e) {
            // 抛出异常表示获取锁失败
            lockAcquired = false;
        }

        if (lockAcquired) {
            /**
             *4. 如果获取到锁则添加到已获取锁集合中
            */
            acquiredLocks.add(lock);
        } else {
            /**
             * 5. 计算已经申请锁失败的节点是否已经到达 允许加锁失败节点个数限制 （N-
(N/2+1)）
             * 如果已经到达， 就认定最终申请锁失败，则没有必要继续从后面的节点申请了
             * 因为 Redlock 算法要求至少N/2+1 个节点都加锁成功，才算最终的锁申请成功
             */
            if (locks.size() - acquiredLocks.size() == failedLocksLimit()) {
                break;
            }

            if (failedLocksLimit == 0) {
                unlockInner(acquiredLocks);
                if (waitTime == -1 && leaseTime == -1) {
                    return false;
                }
                failedLocksLimit = failedLocksLimit();
                acquiredLocks.clear();
                // reset iterator
                while (iterator.hasPrevious()) {
                    iterator.previous();
                }
```

```
            } else {
                failedLocksLimit--;
            }
        }

        /**
        * 6.计算从各个节点获取锁已经消耗的总时间，如果已经等于最大等待时间，则申请锁失败，返回
false
        */
        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<>(acquiredLocks.size());
        for (RLock rLock : acquiredLocks) {
            RFuture<Boolean> future = ((RedissonLock)
rLock).expireAsync(unit.toMillis(leaseTime), TimeUnit.MILLISECONDS);
            futures.add(future);
        }

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

    /**
     * 7.如果逻辑正常执行完则认为最终申请锁成功，返回true
    */
    return true;
}
```