Redis基础

Redis是啥？能干啥？

是啥：
Redis是一个开源的，基于内存亦可持久化的日志型、高性能Key-Value数据库，并提供多种语言的API

干啥：
性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。
丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
原子 – Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。
丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

安装Redis

```
#第一步：下载redis安装包 官网：https://redis.io/
wget http://download.redis.io/releases/redis-4.0.6.tar.gz

#第二步：解压压缩包
tar -zxvf redis-4.0.6.tar.gz

#第三步：yum安装gcc依赖
yum install  gcc

#第四步：跳转到redis解压目录下
cd redis-4.0.6

#第五步：编译安装
make MALLOC=libc
#将bin目录安装到当前目录下，当然，你也可以直接使用src下的脚本
make install PREFIX=/usr/local/redis-4.0.6
```

启动服务端：

```
[root@ydt1 bin]# ./redis-server
4645:C 25 Jul 14:40:22.789 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
4645:C 25 Jul 14:40:22.789 # Redis version=4.0.6, bits=64, commit=00000000, modified=0, pid=4645, just started
4645:C 25 Jul 14:40:22.790 # Warning: no config file specified, using the default config. In order to specify a config file use ./redis-server /path/to/redis.conf
4645:M 25 Jul 14:40:22.790 * Increased maximum number of open files to 10032 (it was originally set to 1024).
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 4.0.6 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 4645
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'

4645:M 25 Jul 14:40:22.791 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
4645:M 25 Jul 14:40:22.791 # Server initialized
4645:M 25 Jul 14:40:22.792 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
4645:M 25 Jul 14:40:22.792 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
4645:M 25 Jul 14:40:22.792 * Ready to accept connections
```

启动客户端操作：

```
[root@ydt1 bin]# ./redis-cli
127.0.0.1:6379> set name laohu
OK
127.0.0.1:6379> get name
"laohu"
127.0.0.1:6379>
```

Redis集成Spring Demo

1、引入pom依赖坐标

```
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.0.5.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>

        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.16</version>
        </dependency>
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>2.9.3</version>
        </dependency>
        <!-- spring-redis -->
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-redis</artifactId>
            <version>1.6.4.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-pool2</artifactId>
            <version>2.4.2</version>
        </dependency>
    </dependencies>
```

2、配置spring文件 applicationContext.xml

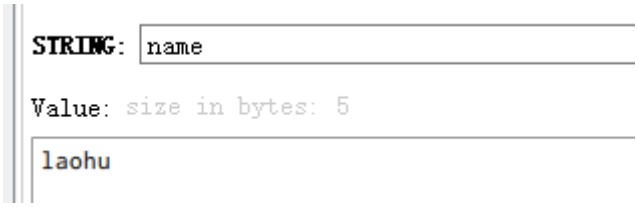
```
    <!--Jedis连接池的相关配置-->
    <bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
        <property name="maxTotal" value="200"/> <!--最大活动对象数-->
        <property name="maxIdle" value="50"/>  <!--最大能够保持Idle（空闲）状态的对象
数-->
```

```
        <property name="testOnBorrow" value="true"/>  <!--当调用jedis实例时，是否进行有效性检查 -->
        <property name="testOnReturn" value="true"/>  <!--当返回jedis实例时，是否进行有效性检查-->
    </bean>
    <bean id="jedisPool" class="redis.clients.jedis.JedisPool">
        <constructor-arg name="poolConfig" ref="jedisPoolConfig" />
        <constructor-arg name="host" value="192.168.223.128" />
        <constructor-arg name="port" value="6379" />
        <constructor-arg name="timeout" value="30000" />
    </bean>

    #最大活动对象数
    redis.pool.maxTotal=1000
    #最大能够保持idel状态的对象数
    redis.pool.maxIdle=100
    #最小能够保持idel状态的对象数
    redis.pool.minIdle=50
    #当池内没有返回对象时，最大等待时间
    redis.pool.maxWaitMillis=10000
    #当调用borrow Object方法时，是否进行有效性检查
    redis.pool.testOnBorrow=true
    #当调用return Object方法时，是否进行有效性检查
    redis.pool.testOnReturn=true
    #“空闲链接”检测线程，检测的周期，毫秒数。如果为负值，表示不运行“检测线程”。默认为-1.
    redis.pool.timeBetweenEvictionRunsMillis=30000
    #向调用者输出“链接”对象时，是否检测它的空闲超时；
    redis.pool.testWhileIdle=true
    # 对于“空闲链接”检测线程而言，每次检测的链接资源的个数。默认为3.
    redis.pool.numTestsPerEvictionRun=50
```

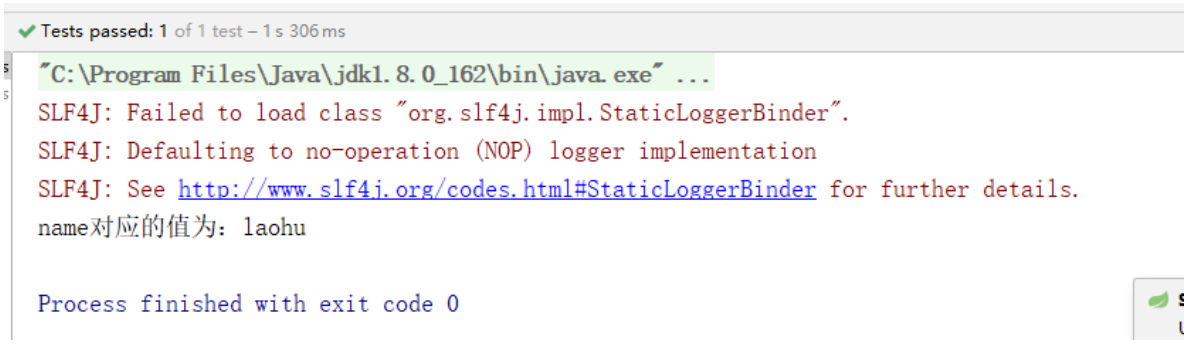
3、应用

```
    private ApplicationContext context;

    @Before
    public void init(){
        context = new ClassPathXmlApplicationContext("applicationContext.xml");
    }

    @Test
    public void testInsert(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        jedisPool.getResource().set("name","laohu");
    }

    @Test
    public void testQuery(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        System.out.println("name对应的值为："+
jedisPool.getResource().get("name"));
    }
```

插入结果：



查询结果：



Redis支持数据类型

- **String(字符串)**

  PS:见demo

- **Hash(哈希)---常用于保存对象**

```
    @Test
    public void testInsertHash(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        jedisPool.getResource().hset("user","name","laohu");
        jedisPool.getResource().hset("user","age","18");

    }

    @Test
    public void testQueryHash(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        System.out.println("user&name对应的值为："+
jedisPool.getResource().hget("user","name"));
        System.out.println("user对应的值为："+
jedisPool.getResource().hgetAll("user"));
    }
```

输出结果：

```
user&name对应的值为：laohu
user对应的值为：{name=laohu, age=18}
```

- **List(列表)---有序**

```
@Test
    public void testInsertList(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        //左压栈
```

```
        jedisPool.getResource().lpush("city1","changsha","wuhan");
        //右压栈
        jedisPool.getResource().rpush("city2","beijing","shenzhen");
    }

    @Test
    public void testQueryList(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        System.out.println("city1对应的值为："+
jedisPool.getResource().lrange("city1",0,-1));
        System.out.println("city2对应的值为："+
jedisPool.getResource().lrange("city2",0,-1));
    }
```

输出结果：

```
city1对应的值为：[wuhan, changsha]
city2对应的值为：[beijing, shenzhen]
```

PS：注意左压栈，右压栈的数据排列区别

- **Set(集合)---无序，差值**

```
@Test
    public void testInsertSet(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        jedisPool.getResource().del("student","teacher");

 jedisPool.getResource().sadd("student","laohu","laowang","laoli","laozhang"
);

 jedisPool.getResource().sadd("teacher","laozhou","laowang","laozhao","laozh
ang");

    }

    @Test
    public void testQuerySet(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        System.out.println("student对应的值
为："+jedisPool.getResource().smembers("student"));
        System.out.println("student&teacher对应的差值
为："+jedisPool.getResource().sdiff("student","teacher"));
    }
```

输出结果：

```
student对应的值为：[laozhang, laowang, laohu, laoli]
student&teacher对应的差值为：[laohu, laoli]
```

- **ZSet(sorted set：有序集合)--- 排行榜**

```
@Test
    public void testInsertZSet(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
```

```
        jedisPool.getResource().del("tuhao");
        jedisPool.getResource().zadd("tuhao",1000,"mayun");
        jedisPool.getResource().zadd("tuhao",500,"mahuateng");
        jedisPool.getResource().zadd("tuhao",100,"wangjianlin");

    }

    @Test
    public void testQueryZSet(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        System.out.println("tuhao升序对应的值
为："+jedisPool.getResource().zrange("tuhao",0,-1));
        System.out.println("tuhao降序对应的值
为："+jedisPool.getResource().zrevrange("tuhao",0,-1));
        System.out.println("降序集合的成
员:"+jedisPool.getResource().zrevrangeByScore("tuhao",500,0));
        System.out.println("升序集合的成
员:"+jedisPool.getResource().zrangeByScore("tuhao",0,500));
    }
```

输出：

```
tuhao升序对应的值为：[wangjianlin, mahuateng, mayun]
tuhao降序对应的值为：[mayun, mahuateng, wangjianlin]
降序集合的成员:[mahuateng, wangjianlin]
升序集合的成员:[wangjianlin, mahuateng]
```

端口开启命令

如果需要外部访问，需要开启访问端口！
开启端口
firewall-cmd --zone=public --add-port=6379/tcp --permanent　　----其他端口照做
重启防火墙
firewall-cmd --reload
查看端口号是否开启
firewall-cmd --query-port=6379/tcp
测试是否可以访问虚拟机端口
telnet 192.168.223.128 6379

分布式锁解决方案-Redis

为什么要学习分布式锁解决方案

为了解决分布式架构带来的数据准确性问题！

我们用synchronized或者 ReentrantLock（瑞恩吹特） 能解决问题吗?

真实生产环境我们采用集群的方式去访问秒杀商品（nginx为我们做了负载均衡）。就会看到数据不一致的现象。如synchronized关键字的作用域其实是一个进程，在这个进程下面的所有线程都能够进行加锁。但是多进程就不行了。对于秒杀商品来说，这个值是固定的。但是每个地区都可能有一台服务器。这样不同地区服务器不一样，地址不一样，进程也不一样。因此synchronized无法保证数据的一致性，这也是为什么要学习分布式解决方案的原因

分布式锁需要满足的几点

1、多任务环境

2、多任务对同一资源进行写操作

3、多任务对资源的写操作是互斥的

4、任务资源访问权限是要能释放的，并且只能释放自己的权限

常见分布式锁的实现分类

1、基于数据库的实现方式

核心思想是：在数据库中创建一个表，表中包含**方法名**等字段，并在**方法名字段上创建唯一索引**，想要执行某个方法，就使用这个方法名向表中插入数据，成功插入则获取锁，执行完成后删除对应的行数据释放锁。

```
#Create Lock Table
DROP TABLE IF EXISTS `method_lock`;
CREATE TABLE `method_lock` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键',
  `method_name` varchar(64) NOT NULL COMMENT '锁定的方法名',
  `desc` varchar(255) NOT NULL COMMENT '备注信息',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY `uidx_method_name` (`method_name`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';

#想要执行某个方法，就使用这个方法名向表中插入数据
INSERT INTO method_lock (method_name, desc) VALUES ('methodName', '测试的
methodName');

#成功插入则获取锁，执行完成后删除对应的行数据释放锁：
delete from method_lock where method_name ='methodName';
```

问题：

1、因为是基于数据库实现的，数据库的可用性和性能将直接影响分布式锁的可用性及性能，所以，数据库需要双机部署、数据同步、主备切换；

2、不具备可重入的特性，因为同一个线程在释放锁之前，行数据一直存在，无法再次成功插入数据，所以，需要在表中新增一列，用于记录当前获取到锁的机器和线程信息，在再次获取锁的时候，先查询表中机器和线程信息是否和当前机器和线程相同，若相同则直接获取锁；

3、没有锁失效机制，因为有可能出现成功插入数据后，服务器宕机了，对应的数据没有被删除，当服务恢复后一直获取不到锁，所以，需要在表中新增一列，用于记录失效时间，并且需要有定时任务清除这些失效的数据；

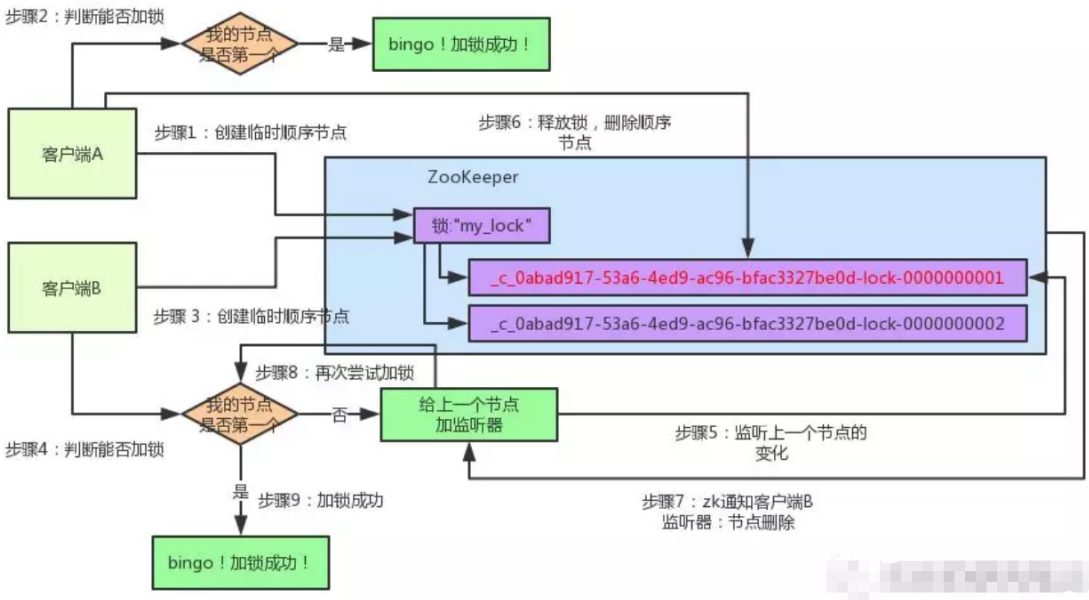
4、不具备阻塞锁特性，获取不到锁直接返回失败，所以需要优化获取逻辑，循环多次去获取。

5、在实施的过程中会遇到各种不同的问题，为了解决这些问题，实现方式将会越来越复杂；依赖数据库需要一定的资源开销，性能问题需要考虑。

2、基于Zookeeper的实现方式

ZooKeeper是一个为分布式应用提供一致性服务的开源组件（Leader机制），它内部是一个分层的文件系统目录树结构，规定同一个目录下只能有一个唯一文件名。基于ZooKeeper实现分布式锁的步骤如下：

（1）创建一个目录mylock；
（2）线程A想获取锁就在mylock目录下创建临时顺序节点；
（3）获取mylock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则说明当前线程顺序号最小，获得锁；
（4）线程B获取所有节点，判断自己不是最小节点，设置监听比自己次小的节点；
（5）线程A处理完，删除自己的节点，线程B监听到变更事件，判断自己是不是最小的节点，如果是则获得锁。



推荐一个Apache的开源库Curator，它是一个ZooKeeper客户端，Curator提供的InterProcessMutex是分布式锁的实现，acquire方法用于获取锁，release方法用于释放锁。

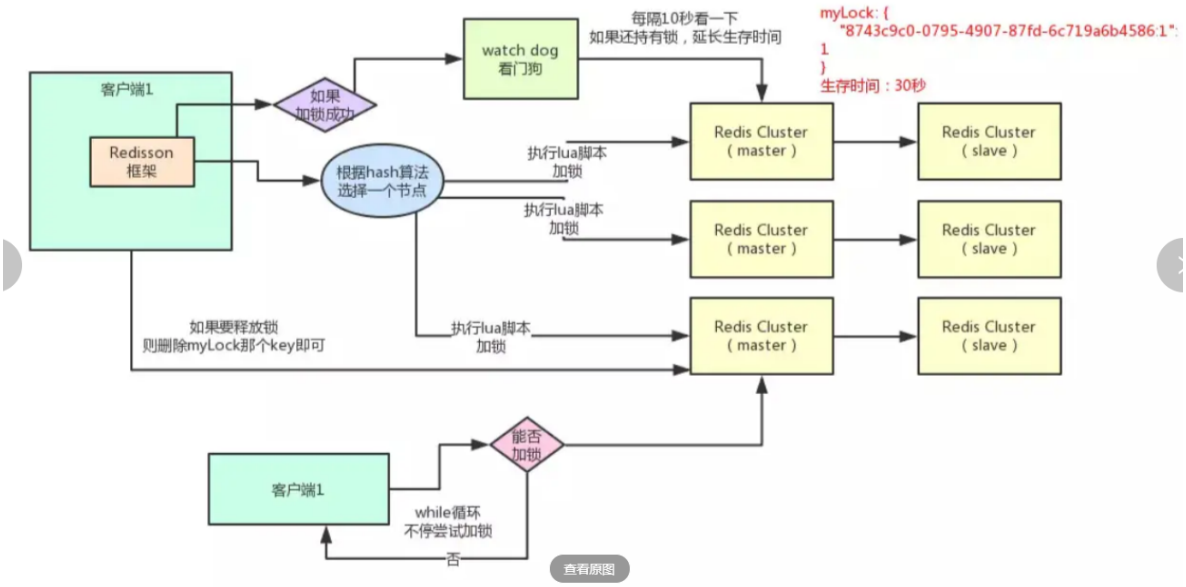
优点：具备高可用、可重入、阻塞锁特性，可解决失效死锁问题。

缺点：因为需要频繁的创建和删除节点，性能上不如Redis方式（为什么？）。

3、基于Redis缓存的实现方式

选用Redis实现分布式锁原因：

（1）Redis有很高的性能；
（2）Redis命令对此支持较好，实现起来比较方便



Redis分布式锁解决方案实战

下面，我们以一个非常经典的案例来说明：秒杀超卖

业务环境搭建

pom.xml

```
<dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
```

```
            <groupId>org.redisson</groupId>
            <artifactId>redisson</artifactId>
            <version>3.7.3</version>
        </dependency>
    </dependencies>
```

application.yml配置

```
spring:
  application:
    name: Redis Distribute Lock

  redis:
    host: 127.0.0.1
    port: 6379
    timeout: 20000
    jedis:
      pool:
        max-active: 8
        min-idle: 0
        max-idle: 8
        max-wait: -1

  datasource:
    url: jdbc:mysql://127.0.0.1:3306/test?
useUnicode=true&characterEncoding=UTF8&zeroDateTimeBehavior=convertToNull&useSSL
=true&allowMultiQueries=true&serverTimezone=Asia/Hong_Kong
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver

  jpa:
    show-sql: true
    hibernate:
      ddl-auto: none

server:
  port: 8888  #该处到时候会要描述多服务并发，以端口来区分
```

库存表：

```
CREATE TABLE `goods_store` (
`code`  varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL ,
`store`  int(11) NULL DEFAULT NULL ,
PRIMARY KEY (`code`)
)
ENGINE=InnoDB
DEFAULT CHARACTER SET=utf8 COLLATE=utf8_general_ci
ROW_FORMAT=DYNAMIC
;
```

库存实体类：

```
package com.ydt.lockredis.domain;
```

```
import java.awt.*;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.swing.*;

@Entity
@Table(name = "goods_store")
public class GoodsStore implements Serializable {
    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id
    private String code;

    @Column(name = "store")
    private int store;


    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public int getStore() {
        return store;
    }

    public void setStore(int store) {
        this.store = store;
    }

    @Override
    public String toString() {
        return "GoodsStore{" +
                "code='" + code + '\'' +
                ", store=" + store +
                '}';
    }
}
```

Controller:

```
package com.ydt.lockredis.controller;

import com.ydt.lockredis.service.GoodsStoreService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.ModelAndView;

import javax.annotation.Resource;

@Controller
@RequestMapping("/storage")
public class StorageController {

    @Resource(name = "goodsStoreServiceImpl3")
    private GoodsStoreService goodsStoreService;

    /**
     * 进入测试页面
     * @param model
     * @return
     */
    @GetMapping("test")
    public ModelAndView stepOne(Model model){
        return new ModelAndView("test", "model", model);
    }

    /**
     * 秒杀提交
     * @param code
     * @param num
     * @return
     */
    @PostMapping("secKill")
    @ResponseBody
    public String secKill(@RequestParam(value="code",required=true) String
code,@RequestParam(value="num",required=true) Integer num){
        String reString = goodsStoreService.updateGoodsStore(code, num);
        return reString;
    }
}
```

Service：

```
package com.ydt.lockredis.service.impl;

import com.ydt.lockredis.dao.GoodsStoreDao;
import com.ydt.lockredis.domain.GoodsStore;
import com.ydt.lockredis.service.GoodsStoreService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Optional;
```

```java
/**
 * 库存管理服务
 */
@Service("goodsStoreServiceImpl0")
public class GoodsStoreServiceImpl0 implements GoodsStoreService {


    @Autowired
    private GoodsStoreDao goodsStoreDao;


    /**
     * 根据产品编号更新库存（啥控制也不加）
     *
     * @param code
     * @return
     */
    @Override
    public String updateGoodsStore(String code, int count) {
        GoodsStore goodsStore = getGoodsStore(code);
        if (goodsStore != null) {
            if (goodsStore.getStore() <= 0) {
                return "对不起，卖完了，库存为：" + goodsStore.getStore();
            }
            if (goodsStore.getStore() < count) {
                return "对不起，库存不足，库存为：" + goodsStore.getStore() + " 您的
购买数量为：" + count;
            }
            System.out.println("剩余库存：" + goodsStore.getStore());
            System.out.println("扣除库存：" + count);
            goodsStore.setStore(goodsStore.getStore() - count);
            goodsStoreDao.save(goodsStore);
            try {
                //为了更好的测试多线程同时进行库存扣减，在进行数据更新之后先等1秒，让多个线程
同时竞争资源
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "恭喜您，购买成功！";
        } else {
            return "获取库存失败。";
        }
    }

    /**
     * 获取库存对象
     *
     * @param code
     * @return
     */
    @Override
    public GoodsStore getGoodsStore(String code) {
        Optional<GoodsStore> optional = goodsStoreDao.findById(code);
        return optional.get();
    }
}
```

测试：

解决方案

解决方案一

添加synchronized铜锁，修改削减库存业务方法如下：

```
/**
     * 根据产品编号更新库存（加入synchronized同步锁）
     *
     * @param code
     * @return
     */
    @Override
    public String updateGoodsStore(String code, int count) {
        synchronized (this){
            GoodsStore goodsStore = getGoodsStore(code);
            if (goodsStore != null) {
                if (goodsStore.getStore() <= 0) {
                    return "对不起，卖完了，库存为：" + goodsStore.getStore();
                }
                if (goodsStore.getStore() < count) {
                    return "对不起，库存不足，库存为：" + goodsStore.getStore() + "
您的购买数量为：" + count;
                }
                System.out.println("剩余库存：" + goodsStore.getStore());
                System.out.println("扣除库存：" + count);
                goodsStore.setStore(goodsStore.getStore() - count);
                goodsStoreDao.save(goodsStore);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return "恭喜您，购买成功！";
            } else {
                return "获取库存失败。";
            }
        }
    }
```

测试：

解决方案二

使用Redis命令完成分布式锁

涉及命令：

setnx key val：当且仅当key不存在时，set一个key为val的字符串，返回1；若key存在，则什么都不做，返回0(加锁)

expire key timeout：为key设置一个超时时间，单位为second，超过这个时间锁会自动释放，避免死锁。

delete key：删除key,释放锁

定义一个RedisNoWaitLock类，实现加锁和释放锁方法

```
package com.ydt.lockredis.util;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;

import java.util.concurrent.TimeUnit;

@Component
public class RedisNoWaitLock {
    @Autowired
    private RedisTemplate redisTemplate;

    /**
     * 加锁
     *
     * @param lockKey   加锁的Key
     * @param timeStamp 时间戳：当前时间+超时时间
     * @return
     */
    public boolean lock(String lockKey, String timeStamp) {
        // 对应setnx命令，可以成功设置,也就是key不存在，获得锁成功
        if (redisTemplate.opsForValue().setIfAbsent(lockKey, timeStamp)) {
            //添加超时时间，防止业务逻辑中锁释放失败，导致死锁
            redisTemplate.expire(lockKey, 5, TimeUnit.SECONDS);
            return true;
        }
        return false;
    }

    /**
     * 释放锁
     *
     * @param lockKey
     */
    public void release(String lockKey) {
        try {
            // 删除锁状态
            redisTemplate.opsForValue().getOperations().delete(lockKey);
        } catch (Exception e) {
            System.out.println("警报！警报！警报！解锁异常");
        }
    }
}
```

修改削减库存的业务方法：

```
/**
```

```
     * 根据产品编号更新库存
     * @param code
     * @return
     */
    @Override
    public String updateGoodsStore(String code,int count) {
        //上锁
        long time = System.currentTimeMillis() + TIMEOUT;
        if(!redisNoWaitLock.lock(code, String.valueOf(time))){
            return "当前排队人数过多，请稍后再试！";
        }
        System.out.println("获得锁的时间戳："+String.valueOf(time));
        try {
            GoodsStore goodsStore = getGoodsStore(code);
            if(goodsStore != null){
                if(goodsStore.getStore() <= 0){
                    return "对不起，卖完了，库存为："+goodsStore.getStore();
                }
                if(goodsStore.getStore() < count){
                    return "对不起，库存不足，库存为："+goodsStore.getStore()+" 您的
购买数量为："+count;
                }
                System.out.println("剩余库存："+goodsStore.getStore());
                System.out.println("扣除库存："+count);
                goodsStore.setStore(goodsStore.getStore() - count);
                goodsStoreDao.save(goodsStore);
                try{
                    //为了更好的测试多线程同时进行库存扣减，在进行数据更新之后先等1秒，让多
个线程同时竞争资源
                    Thread.sleep(1000);
                }catch (InterruptedException e){
                    e.printStackTrace();
                }
                return "恭喜您，购买成功！";
            }else{
                return "获取库存失败。";
            }
        } finally {
            //释放锁
            redisNoWaitLock.release(code);
            System.out.println("释放锁的时间戳："+String.valueOf(time));
        }
    }
```

测试：

解决方案三

使用Redis官方推荐的 Redisson，他本身就提供了分布式锁和相关API服务，支持单机和集群的配置

- Redlock是Redis的作者antirez给出的集群模式的Redis分布式锁，它基于N个完全独立的Redis节点
- 部分节点宕机，依然可以保证锁的可用性

RedissonLock

LUA代码解读：（加锁）

```
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]);"
```

(1) 第一个If判断key是否存在，不存在完成加锁操作

redis.call('hset', KEYS[1], ARGV[2], 1);创建key[1] map中添加key：ARGV[2]，value：1
redis.call('pexpire', KEYS[1], ARGV[1]);设置key[1]过期时间，避免发生死锁。
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才会执行其他命令。可避免第一条命令执行成功第二条命令执行失败导致死锁。

(2)第二个if判断key存在且当前线程已经持有锁, 重入:

redis.call('hexists', KEYS[1], ARGV[2])；判断redis中锁的标记值是否与当前请求的标记值相同，相同代表该线程已经获取锁。
redis.call('hincrby', KEYS[1], ARGV[2], 1);记录同一线程持有锁之后累计加锁次数实现锁重入。
redis.call('pexpire', KEYS[1], ARGV[1]); 重制锁超时时间。
(3)key存在被其他线程获取的锁, 等待:

redis.call('pttl', KEYS[1]);加锁失败返回锁过期时间。

解锁:

```
if (redis.call('exists', KEYS[1]) == 0) then
    redis.call('publish', KEYS[2], ARGV[1]);
    return 1;
end;
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;"
```

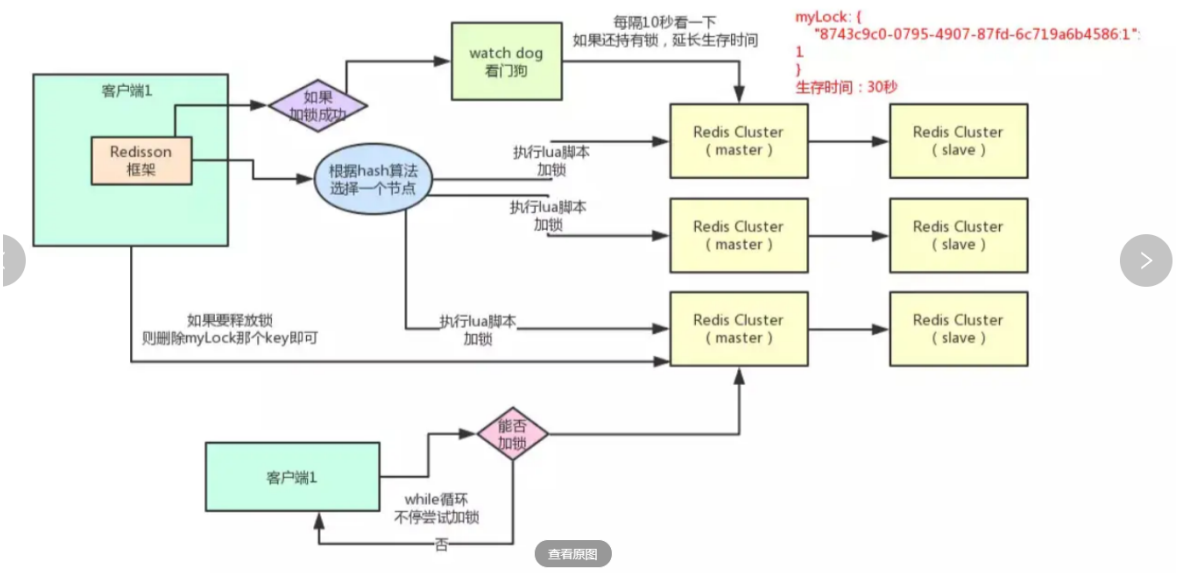
（1）第一个if判断锁对应key是否存在，不存在publish消息，将获取锁被阻塞的线程恢复重新获取锁；

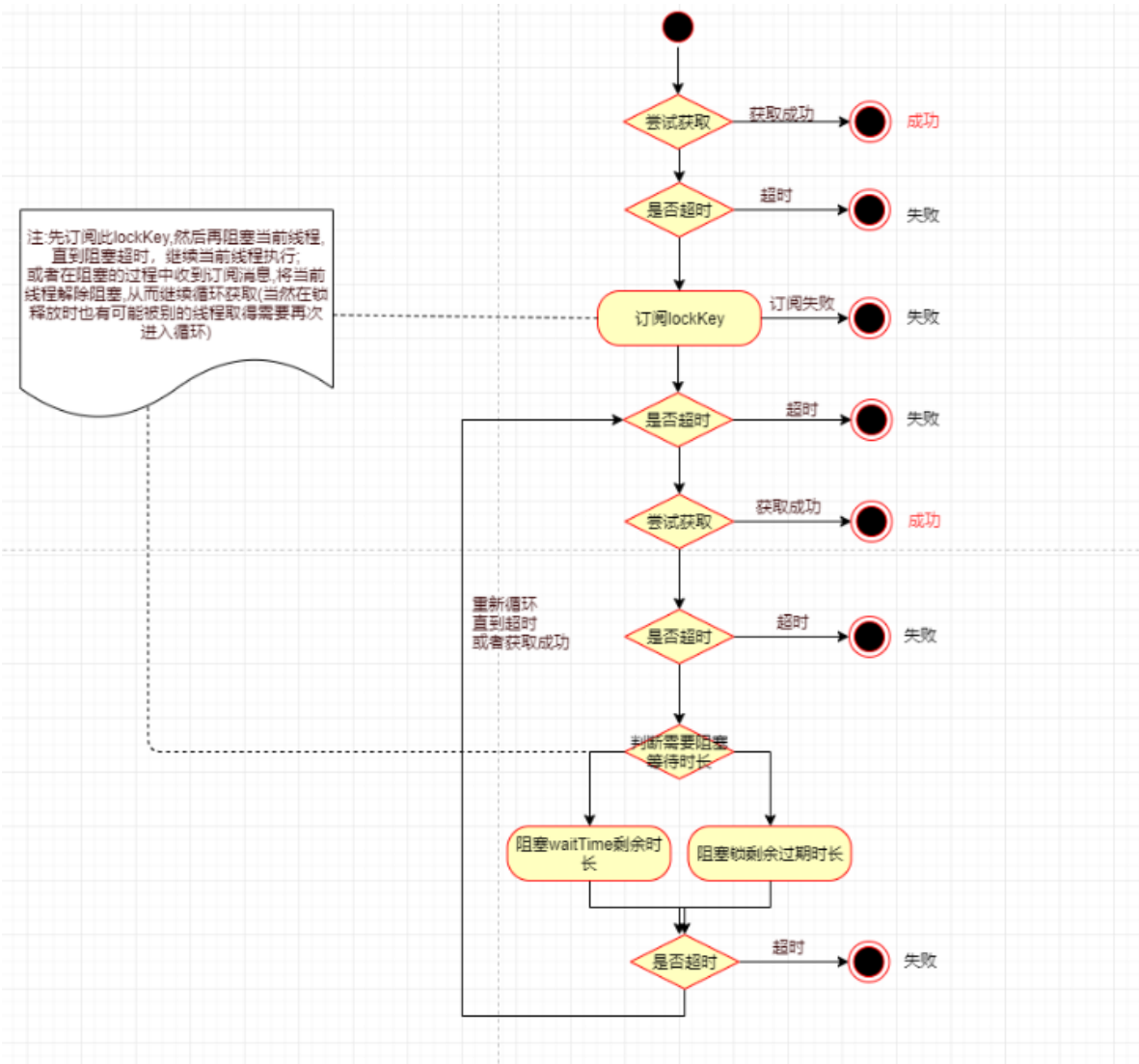
（2）第二个if判断锁对应key存在，value中是否存在当前要释放锁的标示，不存在返回nil，确保锁只能被持有的线程释放。

（3）对应key存在，value中存在当前要释放锁的标示，将锁标示对应值-1，第三个if判断锁标示对应的值是否大于0，大于0，表示有锁重入情况发生，重新设置锁过期时间。

（4）对应key存在，value中存在当前要释放锁的标示，将锁标示对应值-1后等于0，调用del操作释放锁，并publish消息，将获取锁被阻塞的线程恢复重新获取锁；

流程图（基于集群模式）






```
/**********************RedissonLock**********************/
public boolean tryLock(long waitTime /*获取锁等待时间*/, long leaseTime /*锁过期时间
*/, TimeUnit unit) throws InterruptedException {
```

```
    long time = unit.toMillis(waitTime);
    long current = System.currentTimeMillis();
    //当前线程ID 用户拼接key
    final long threadId = Thread.currentThread().getId();
    //尝试获取锁 如果ttl==null则获取到锁 ttl!=null则表示锁已存在,ttl为锁剩余时间(毫秒)
    //tryAcquire最终实现方法tryAcquireAsync在下面代码
    Long ttl = tryAcquire(leaseTime, unit);
    //获取锁成功
    if (ttl == null) {
        return true;
    }

    //time = time-上一次尝试获取锁之后所用时间
    time -= (System.currentTimeMillis() - current);
    //这里有两种情况
    //1 若waitTime为负数(不失效缓存), 则返回false
    //2 若waitTime为正数 经过第一次尝试获取锁之后未获取成功 但已经超过等待时长 返回false
    if (time <= 0) {
        return false;
    }

    current = System.currentTimeMillis();
    //订阅消息
    final RFuture<RedissonLockEntry> subscribeFuture = subscribe(threadId);
    //阻塞等待一段时间(time) 等待订阅结果
    if (!await(subscribeFuture, time, TimeUnit.MILLISECONDS)) {
        if (!subscribeFuture.cancel(false)) {
            subscribeFuture.addListener(new FutureListener<RedissonLockEntry>()
{
                @Override
                public void operationComplete(Future<RedissonLockEntry> future)
throws Exception {
                    if (subscribeFuture.isSuccess()) {
                        unsubscribe(subscribeFuture, threadId);
                    }
                }
            });
        }
        //尝试获取CountDownLatch锁失败时直接返回失败
        return false;
    }

    try {
        time -= (System.currentTimeMillis() - current);
        //超过waittime返回失败
        if (time <= 0) {
            return false;
        }
        //死循环 很简单 就是在等待时长内重复获取锁 直到获取成功或超时
        while (true) {
            long currentTime = System.currentTimeMillis();
            //尝试获取锁 tryAcquire最终实现方法tryAcquireAsync在下面代码
            ttl = tryAcquire(leaseTime, unit);
            // 过期时间为null 则获取成功
            if (ttl == null) {
                return true;
            }
```

```
            time -= (System.currentTimeMillis() - currentTime);
            //依然判断超时
            if (time <= 0) {
                return false;
            }

            currentTime = System.currentTimeMillis();
            //等待时间大于过期时间 那么就等已存在的过期之后再获取
            if (ttl >= 0 && ttl < time) {
                //阻塞时长ttl
                getEntry(threadId).getLatch().tryAcquire(ttl,
TimeUnit.MILLISECONDS);
            }
            //尝试获取锁
            else {
                //阻塞时长time
                getEntry(threadId).getLatch().tryAcquire(time,
TimeUnit.MILLISECONDS);
            }
            //再次计算time>0
            time -= (System.currentTimeMillis() - currentTime);
            //若是小于等于0则为超时 直接返回false 否则进入下一次循环( while (true) )
            if (time <= 0) {
                return false;
            }
        }
    } finally {
        //最终取消订阅 订阅代码为 final RFuture<RedissonLockEntry> subscribeFuture =
subscribe(threadId)
        unsubscribe(subscribeFuture, threadId);
    }
}

//尝试获取锁(同步)
private <T> RFuture<Long> tryAcquireAsync(long leaseTime, TimeUnit unit, final
long threadId) {
    //若过期时间为有限时间(leaseTime==-1为永不过期)
    if (leaseTime != -1) {
        return tryLockInnerAsync(leaseTime, unit, threadId,
RedisCommands.EVAL_LONG);
    }
    //以下为永不过期
    //设置把永不过期时间改为30秒过期
    RFuture<Long> ttlRemainingFuture =
tryLockInnerAsync(LOCK_EXPIRATION_INTERVAL_SECONDS, TimeUnit.SECONDS, threadId,
RedisCommands.EVAL_LONG);
    //注册监听器
    ttlRemainingFuture.addListener(new FutureListener<Long>() {

        //获取锁操作结束触发下面操作
        @Override
        public void operationComplete(Future<Long> future) throws Exception {
            //如果为获取到锁 不做任何事情
            if (!future.isSuccess()) {
                return;
            }
            Long ttlRemaining = future.getNow();
            // 获取到了锁
```

```
            if (ttlRemaining == null) {
                //定期对锁进行延时 达到永不过期目的
                scheduleExpirationRenewal(threadId);
            }
        }
    });
    return ttlRemainingFuture;
}


//尝试获取锁(同步方式)
//看下面代码之前需要先了解Redis的Hash的数据结构，下面脚本使用的就是Hash
//String数据结构为 Map<Key，Value>，通常根据Key取值
//Hash数据结构为Map<Key，Map<Field，Value>>，Field
<T> RFuture<T> tryLockInnerAsync(long leaseTime, TimeUnit unit, long threadId,
RedisStrictCommand<T> command) {
    //时间转换
    internalLockLeaseTime = unit.toMillis(leaseTime);
    //执行调用redis脚本
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE,
command,
              //为了方便理解对下面数组进行解释 KEYS[1]为Hash:设置的lockKey，ARGV[1]过期
时间:leaseTime，ARGV[2]为Hash的Field:当前线程ID和当前redisson拼接
              //如果key不存在 则设置当前key(KEYS[1]) field(ARGV[2]) value(1)，设置
key(KEYS[1])过期时间ARGV[1] 返回空值(nil)
              "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; " +
              //如果根据key和field能查询到值，则value在原来基础上加1
              //这里可能会有误解: 上面已经判断key已经存在，因为这里是独占锁，
              //若根据此锁不属于当前线程(redis.call('hexists', KEYS[1], ARGV[2]) !=
1)，则肯定被其他线程占有,获取锁失败
              "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; " +
              //返回当前key的剩余时间
              "return redis.call('pttl', KEYS[1]);",
                //关键代码
                Collections.<Object>singletonList(getName()) /*KEYS*/,
internalLockLeaseTime /*ARGV[1]*/, getLockName(threadId)/*ARGV[2]*/ );
}
```

定义一个Redisson配置类，获取Redis连接信息

```
package com.ydt.lockredis.config;

import org.redisson.config.Config;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;

@Configuration
```

```
public class RedissonConfig {
    @Value("${spring.redis.host}")
    private String host;
    @Value("${spring.redis.port}")
    private int port;
    @Value("${spring.redis.timeout}")
    private int timeout;

    private static final String REDIS_PROTOCOL = "redis://";

    public String getHost() {
        return host;
    }
    public int getPort() {
        return port;
    }
    public int getTimeout() {
        return timeout;
    }


    /**
     * 获取redisson配置类
     * */
    public Config getConfig() {
        Config config = new Config();
        config.useSingleServer().setAddress(REDIS_PROTOCOL + host + ":" + port)
        .setDatabase(1)
        .setConnectTimeout(timeout);
        System.out.println("================config:" + host + "," + port);
        return config;
    }
}
```

定义一个类：RedissonLock，实现加锁和释放锁方法

```
package com.ydt.lockredis.util;

import com.ydt.lockredis.config.RedissonConfig;
import org.redisson.Redisson;
import org.redisson.api.RLock;
import org.redisson.api.RedissonClient;
import org.redisson.config.Config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import java.util.concurrent.TimeUnit;

@Component
public class RedissonLock {
    private Config config;
    private RedissonClient client;
    @Autowired
    private RedissonConfig redissonConfig;
    @PostConstruct
    public void init() {
```

```
        //初始化RedissonClient客户端
        if(null == client) {
            config = redissonConfig.getConfig();
            client = Redisson.create(config);
            System.out.println("====================redisson created");
        }
    }

    public RedissonClient getClient() {
        config = redissonConfig.getConfig();
        RedissonClient c = Redisson.create(config);
        return c;
    }

    /**
     * 获取锁
     * @param timeout 自动解锁时间(秒)，不需要则传-1
     * */
    public RLock getLock(String key,long timeout) {
        RLock lock = client.getLock(key);
        if(-1 == timeout) {
            lock.lock();
        }else {
            lock.lock(timeout, TimeUnit.SECONDS);
        }
        return lock;
    }

    public boolean release(String code) {
        RLock lock = client.getLock(code);
        if(null != lock) {
            lock.unlock();
            return true;
        }else {
            System.out.println("=============没有找到锁");
            return false;
        }
    }

    public boolean lock(String code){
        RLock lock = client.getLock(code);
        try {
            return lock.tryLock(0,5,TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

修改削减库存业务方法：

```
  public String updateGoodsStore(String code,int count) {
        //上锁
        long time = System.currentTimeMillis() + TIMEOUT;
```

```
        boolean result = redissonLock.lock(code);
        if (result) {
            System.out.println("获得锁的时间戳：" + String.valueOf(time));
            System.out.println("没有线程在操作" + Thread.currentThread().getName()
+ "正常执行");
            GoodsStore goodsStore = getGoodsStore(code);
            if (goodsStore != null) {
                if (goodsStore.getStore() <= 0) {
                    return "对不起，卖完了，库存为：" + goodsStore.getStore();
                }
                if (goodsStore.getStore() < count) {
                    return "对不起，库存不足，库存为：" + goodsStore.getStore() + "
您的购买数量为：" + count;
                }
                System.out.println("剩余库存：" + goodsStore.getStore());
                System.out.println("扣除库存：" + count);
                goodsStore.setStore(goodsStore.getStore() - count);
                goodsStoreDao.save(goodsStore);
                try {
                    //为了更好的测试多线程同时进行库存扣减，在进行数据更新之后先等1秒，让多
个线程同时竞争资源
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                redissonLock.release(code);
                return "恭喜您，购买成功！";
            }
            else {
                redissonLock.release(code);
                return "获取库存失败。";
            }
        }else{
            return "排队人数太多，请稍后再试.";
        }
    }
```

测试：

完善方案

完善用户等待抢单过程，修改加锁方法如下：

```
/**
     * 锁在给定的等待时间内空闲，则获取锁成功 返回true， 否则返回false
     * @return
     */
    public boolean lock(String code,String time) {
        try {
            do {
                // 对应setnx命令，可以成功设置,也就是key不存在，获得锁成功
                if (redisTemplate.opsForValue().setIfAbsent(code, time)) {
                    //添加超时时间，防止业务逻辑中锁释放失败，导致死锁
                    redisTemplate.expire(code, 5, TimeUnit.SECONDS);
                    return true;
```

```
                }

            } while (true);

        } catch (Exception e) {
            LOGGER.error(e.getMessage(), e);
        }
        return Boolean.FALSE;
    }
```

测试：

思考

在集群模式下需要注意的问题

如果A往Master放入了一把锁，然后再数据同步到Slave之前，Master 挂掉，Slave被提拔为Master，这时候新的Master上面就没有锁了，这样其他进程也可以拿到锁，违法了锁的互斥性

所以說当某个节点宕机后，又立即重启了，可能会出现两个客户端同时持有同一把锁，如果节点设置了持久化，出现这种情况的几率会降低

Redis性能调优军规

尽管Redis是一个非常快速的内存数据存储媒介，也并不代表Redis不会产生性能问题

缩短键值对的存储长度

键值对的长度是和性能成反比的， 在 key 不变的情况下，value 值越大操作效率越慢

```
Logger logger = Logger.getLogger(ClusterTest.class);
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        Jedis jedis = jedisPool.getResource();
         Pipeline pipe = jedis.pipelined(); // 先创建一个 pipeline 的链接对象
        long start = System.currentTimeMillis();
        for (int i = 0; i < 100000; i++) {
            pipe.set(String.valueOf(i), String.valueOf(i));
        }
        pipe.sync();
        long end = System.currentTimeMillis();
        logger.info("the general total time is:" + (end - start));
```

```
        String v =
"aaasssssssssssssssssdakldwndonqwdknaknwnddddddooooaddasssssssssssssssssssssss
ssaaasssssssssssssssdakldwndonqwdknaknwnddddddooooaddasssssssssssssssssssssss
sssaaasssssssssssssssdakldwndonqwdknaknwnddddddooooaddasssssssssssssssssssssss
ssssaaasssssssssssssssdakldwndonqwdknaknwnddddddooooaddasssssssssssssssssssssss
sssssaaasssssssssssssssdakldwndonqwdknaknwnddddddooooaddasssssssssssssssssssssss
ssssssaaasssssssssssssssdakldwndonqwdknaknwnddddddooooaddasssssssssssssssssssssss
sssssssaaasssssssssssssssdakldwndonqwdknaknwnddddddooooaddasssssssssssssssssssssss
ssssssssaaasssssssssssssssdakldwndonqwdknaknwnddddddooooaddasssssssssssssssssssssss
sssssssssaaasssssssssssssssdakldwndonqwdknaknwnddddddooooa";
        System.out.println(v.getBytes().length);

        start = System.currentTimeMillis();
        for (int i = 0; i < 100000; i++) {
            pipe.set(String.valueOf(i), v + String.valueOf(i));
        }
        pipe.sync();
        end = System.currentTimeMillis();
        logger.info("the general total time is:" + (end - start));

        #执行结果如下：
        #17:32:17,155  INFO ClusterTest:227 - the general total time is:498
        #699
        #17:32:19,820  INFO ClusterTest:238 - the general total time is:2656
```

使用 lazy free（延迟删除）特性

lazy free 特性是 Redis 4.0 新增的一个非常实用的功能，它可以理解为惰性删除或延迟删除。意思是在删除的时候提供异步延时释放键值的功能，把键值释放操作放在 BIO(Background I/O) 单独的子线程处理中，以减少删除对 Redis 主线程的阻塞，可以有效地避免删除 big key 时带来的性能和可用性问题。

lazy free 对应了 4 种场景，默认都是关闭的：

lazyfree-lazy-eviction no

lazyfree-lazy-expire no

lazyfree-lazy-server-del no

slave-lazy-flush no

它们代表的含义如下：

- lazyfree-lazy-eviction：表示当 Redis 运行内存超过 maxmeory 时，是否开启 lazy free 机制删除；
- lazyfree-lazy-expire：表示设置了过期时间的键值，当过期之后是否开启 lazy free 机制删除；
- lazyfree-lazy-server-del：有些指令在处理已存在的键时，会带有一个隐式的 del 键的操作，比如 rename 命令，当目标键已存在，Redis 会先删除目标键，如果这些目标键是一个 big key，就会造成阻塞删除的问题，此配置表示在这种场景中是否开启 lazy free 机制删除；
- slave-lazy-flush：针对 slave(从节点) 进行全量数据同步，slave 在加载 master 的 RDB 文件前，会运行 flushall 来清理自己的数据，它表示此时是否开启 lazy free 机制删除。

建议开启其中的 lazyfree-lazy-eviction、lazyfree-lazy-expire、lazyfree-lazy-server-del 等配置，这样就可以有效的提高主线程的执行效率。

设置键值的过期时间

我们应该根据实际的业务情况，对键值设置**合理**的过期时间，这样 Redis 会帮你自动清除过期的键值对，以节约对内存的占用，以避免键值过多的堆积，频繁的触发内存淘汰策略。

禁用长耗时的查询命令

以从以下几个方面入手改造：

- 决定禁止使用 keys 命令；
- 尽可能将排序、并集、交集等操作放在客户端执行，以减少 Redis 服务器运行压力；

使用 slowlog 优化耗时命令

我们可以使用 slowlog 功能找出最耗时的 Redis 命令进行相关的优化，以提升 Redis 的运行速度，慢查询有两个重要的配置项：

- slowlog-log-slower-than： 用于设置慢查询的评定时间，也就是说超过此配置项的命令，将会被当成慢操作记录在慢查询日志中，它执行单位是微秒 (1 秒等于 1000000 微秒)；
- slowlog-max-len： 用来配置慢查询日志的最大记录数。

我们可以根据实际的业务情况进行相应的配置，其中慢日志是按照插入的顺序倒序存入慢查询日志中，我们可以使用 slowlog get 来获取相关的慢查询日志，再找到这些慢查询对应的业务进行相关的优化。

```
redis> SLOWLOG GET
1) 1) (integer) 12                          # 唯一性(unique)的日志标识符
   2) (integer) 1324097834                  # 被记录命令的执行时间点，以 UNIX 时间戳格式表示
   3) (integer) 1689                          # 查询执行时间，以微秒为单位
   4) 1) "SET"                              # 执行的命令
      2) "1"
      3) "LAOHU"
      4) "slowlog-log-slower-than"
```

使用 Pipeline 批量操作数据

Pipeline (管道技术) 是客户端提供的一种批处理技术，用于一次处理多个 Redis 命令，从而提高整个交互的性能。

```
    @Test
    public void testGeneralAndPipeline(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        Jedis jedis = jedisPool.getResource();
        Logger logger = Logger.getLogger(ClusterTest.class);
        long start = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            jedis.set(String.valueOf(i), String.valueOf(i));
        }
        long end = System.currentTimeMillis();
        logger.info("the general total time is:" + (end - start));

        Pipeline pipe = jedis.pipelined(); // 先创建一个 pipeline 的链接对象
```

```
        long start_pipe = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            pipe.set(String.valueOf(i), String.valueOf(i));
        }
        pipe.sync(); // 获取所有的 response
        long end_pipe = System.currentTimeMillis();
        logger.info("the pipe total time is:" + (end_pipe - start_pipe));
    }
```

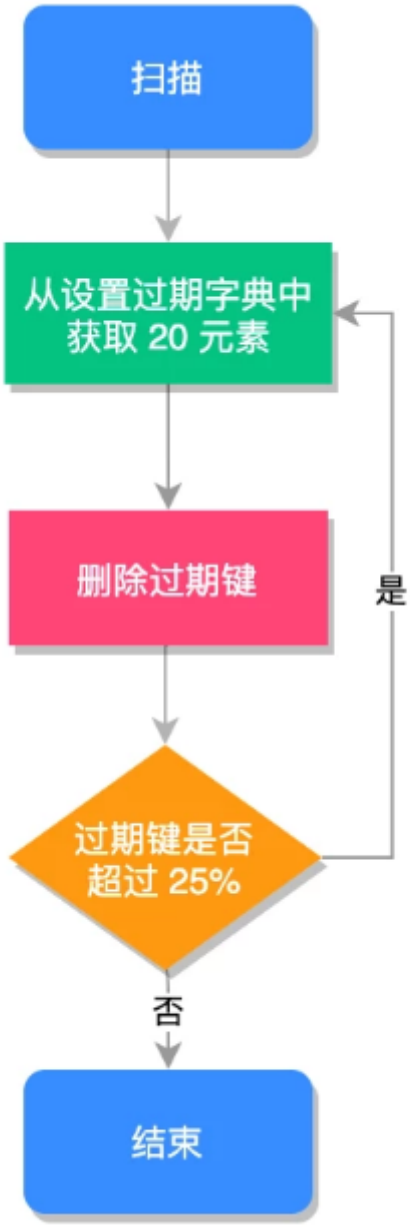
执行结果：

```
16:03:17,592  INFO ClusterTest:114 - the general total time is:2634
16:03:17,692  INFO ClusterTest:123 - the pipe total time is:77
```

避免大量数据同时失效

不仅仅会造成缓存雪崩，数据库服务器承压太大而宕机，对于Redis本身来说，瞬间需要干掉大量的key，也是对cpu性能的一个考验！

hz 10---》每秒执行10次过期扫描



解读：每次从过期字典（所有设置了过期时间键值对的集合）中随机获取20个，如果过期键值对超过四分之一，那么会继续从字典中获取20个继续删除过期键值对，如此循环往复，会造成大量的CPU损耗，并且在整个执行过程会导致 Redis 的读写出现明显的卡顿，卡顿的原因是内存管理器需要频繁回收内存页

客户端使用优化

在客户端的使用上我们除了要尽量使用 Pipeline 的技术外，还需要注意要尽量使用 Redis 连接池，而不是频繁创建销毁 Redis 连接，这样就可以减少网络传输次数和减少了非必要调用指令。

Redis内存管理与数据淘汰机制

最大内存设置

默认情况下，在32位OS中，Redis最大使用3GB的内存，在64位OS中则没有限制。

在使用Redis时，应该对数据占用的最大空间有一个基本准确的预估，并为Redis设定最大使用的内存。

否则在64位OS中Redis会无限制地占用内存（当物理内存被占满后会使用swap空间），容易引发各种各样的问题。

通过如下配置控制Redis使用的最大内存：

```
maxmemory 100mb
```

在内存占用达到了maxmemory后，再向Redis写入数据时，Redis会：

- 根据配置的数据淘汰策略尝试淘汰数据，释放空间
- 如果没有数据可以淘汰，或者没有配置数据淘汰策略，那么Redis会对所有写请求返回错误，但读请求仍然可以正常执行

```
@Test
    public void  testTransaction(){
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        Jedis jedis = jedisPool.getResource();
        Transaction transaction = jedis.multi();
        for (int i = 0; i < 100000; i++) {
            transaction.set("" + i,"value is " + i);
        }
        transaction.exec();
    }
```

```
异常信息：io.lettuce.core.RedisCommandExecutionException: OOM command not
allowed when used memory > 'maxmemory'.
```

在为Redis设置maxmemory时，需要注意：

如果采用了Redis的主从同步，主节点向从节点同步数据时，会占用掉一部分内存空间

如果maxmemory过于接近主机的可用内存，会导致数据同步时内存不足。

所以设置的maxmemory不要过于接近主机可用的内存，留出一部分预留用作主从同步。

数据淘汰机制

Redis提供了5种数据淘汰策略：

- **volatile-lru**：使用LRU算法进行数据淘汰（淘汰上次使用时间最早的，且使用次数最少的key），只淘汰设定了有效期的key
- **allkeys-lru**：使用LRU算法进行数据淘汰，所有的key都可以被淘汰
- **volatile-random**：随机淘汰数据，只淘汰设定了有效期的key
- **allkeys-random**：随机淘汰数据，所有的key都可以被淘汰
- **volatile-ttl**：淘汰剩余有效期最短的key

最好为Redis指定一种有效的数据淘汰策略以配合maxmemory设置，避免在内存使用满后发生写入失败的情况。

一般来说，推荐使用的策略是volatile-lru，并辨识Redis中保存的数据的重要性。

对于那些重要的，绝对不能丢弃的数据（如配置类数据等），应不设置有效期，这样Redis就永远不会淘汰这些数据。

对于那些相对不是那么重要的，并且能够热加载的数据（比如缓存最近登录的用户信息，当在Redis中找不到时，程序会去DB中读取），可以设置上有效期，这样在内存不够时Redis就会淘汰这部分数据。

配置方法：

```
maxmemory-policy volatile-lru   #默认是noeviction，即不进行数据淘汰
```

使用物理机而非虚拟机安装 Redis 服务

在虚拟机中运行 Redis 服务器，因为和物理机共享一个物理网口，并且一台物理机可能有多个虚拟机在运行，因此在内存占用上和网络延迟方面都会有很糟糕的表现，我们可以通过 ./redis-cli --intrinsic-latency 100 命令查看延迟时间，如果对 Redis 的性能有较高要求的话，应尽可能在物理机上直接部署 Redis 服务器。

```
    [root@ydt src]# ./redis-cli --intrinsic-latency 100
Max latency so far: 1 microseconds.
Max latency so far: 7 microseconds.
Max latency so far: 9 microseconds.
Max latency so far: 29 microseconds.
Max latency so far: 399 microseconds.
Max latency so far: 1243 microseconds.
Max latency so far: 2157 microseconds.
Max latency so far: 3847 microseconds.
Max latency so far: 3980 microseconds.
Max latency so far: 5076 microseconds.
Max latency so far: 23236 microseconds.
Max latency so far: 31935 microseconds.
Max latency so far: 46829 microseconds.

1838316588 total runs (avg latency: 0.0544 microseconds / 54.40 nanoseconds per run).
Worst run took 860865x longer than the average latency.

注意：后面的参数100表示100s
 由测试结果可以看出来，redis内部延迟最大为46829微秒(46毫秒)，这会是一个不太好的消息，因为内部延迟超过100微秒性能就不是很好了，最主要的原因是系统有其他应用在共享cpu
```

检查数据持久化策略

Redis 的持久化策略是将内存数据复制到硬盘上，这样才可以进行容灾恢复或者数据迁移，但维护此持久化的功能，需要很大的性能开销。

在 Redis 4.0 之后，Redis 有 3 种持久化的方式：

- RDB（Redis DataBase，快照方式）将某一个时刻的内存数据，以二进制的方式写入磁盘；
- AOF（Append Only File，文件追加方式），记录所有的操作命令，并以文本的形式追加到文件中；
- 混合持久化方式，Redis 4.0 之后新增的方式，混合持久化是结合了 RDB 和 AOF 的优点，在写入的时候，先把当前的数据以 RDB 的形式写入文件的开头，再将后续的操作命令以 AOF 的格式存入文件，这样既能保证 Redis 重启时的速度，又能减低数据丢失的风险。

RDB 和 AOF 持久化各有利弊，RDB 可能会导致一定时间内的数据丢失，而 AOF 由于文件较大则会影响 Redis 的启动速度，为了能同时拥有 RDB 和 AOF 的优点，Redis 4.0 之后新增了混合持久化的方式，因此我们在必须要进行持久化操作时，应该选择混合持久化的方式。

```
appendonly yes  #开启aof持久化，rbd是默认开启的
appendfilename "appendonly.aof" #aof持久化文件

config set aof-use-rdb-preamble yes  #开启aof和rdb混合持久化模式
```

禁用 THP 特性

Linux kernel 在 2.6.38 内核增加了 Transparent Huge Pages (THP) 特性，支持大内存页 2MB 分配，默认开启。

这个特性实际上就是会把这种巨页的使用对用户透明，用户不需要再进行巨页的配置，内存会自动将连续的512个普通页作为一个巨页处理，为了减少维护人员配置巨页的工作

我们Redis在启动的时候也会去检查是否也已经禁用THP特性：

```
30663:M 25 Jul 18:34:48.221 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
```

当开启了 THP 时，fork 的速度会变慢，fork 之后每个内存页从原来 4KB 变为 2MB，会大幅增加重写期间父进程内存消耗。同时每次写命令引起的复制内存页单位放大了 512 倍，会拖慢写操作的执行时间，导致大量写操作慢查询。例如简单的 incr 命令也会出现在慢查询中，因此 Redis 建议将此特性进行禁用，禁用方法命令如下：

echo never > /sys/kernel/mm/transparent_hugepage/enabled

为了使机器重启后 THP 配置依然生效，可以在 /etc/rc.local 中追加

```
`echo never > /sys/kernel/mm/transparent_hugepage/enabled`
```

使用分布式架构来增加读写速度

Redis 分布式架构有三个重要的手段：

- 主从同步
- 哨兵模式
- Redis Cluster 集群

使用主从同步功能我们可以把写入放到主库上执行，把读功能转移到从服务上，因此就可以在单位时间内处理更多的请求，从而提升的 Redis 整体的运行速度。

而哨兵模式是对于主从功能的升级，但当主节点奔溃之后，无需人工干预就能自动恢复 Redis 的正常使用。

Redis Cluster 是 Redis 3.0 正式推出的，Redis 集群是通过将数据库分散存储到多个节点上来平衡各个节点的负载压力。

Redis Cluster 采用虚拟哈希槽分区，所有的键根据哈希函数映射到 0 ~ 16383 整数槽内，计算公式：slot = CRC16(key) & 16383，每一个节点负责维护一部分槽以及槽所映射的键值数据。这样 Redis 就可以把读写压力从一台服务器，分散给多台服务器了，因此性能会有很大的提升。

在这三个功能中，我们只需要使用一个就行了，毫无疑问 Redis Cluster 应该是首选的实现方案，它可以把读写压力自动的分担给更多的服务器，并且拥有自动容灾的能力。