

企业实战

本章学习目标：

- 理解缓存设计要素
- 掌握缓存预热
- 能够进行缓存问题分析和提供解决方案
- 能够整合mybatis使用缓存
- 理解分布式锁原理并掌握使用
- 理解乐观锁并掌握秒杀的实现
- 理解Redisson的原理
- 了解阿里Redis使用手册

架构设计

组件选择/多级

缓存的设计要分多个层次，在不同的层次上选择不同的缓存，包括JVM缓存、文件缓存和Redis缓存

JVM缓存

JVM缓存就是本地缓存，设计在应用服务器中（tomcat）。

通常可以采用Ehcache和Guava Cache，在互联网应用中，由于要处理高并发，通常选择Guava Cache。

适用本地（JVM）缓存的场景：

- 1、对性能有非常高的要求。
- 2、不经常变化
- 3、占用内存不大
- 4、有访问整个集合的需求
- 5、数据允许不时时一致

文件缓存

这里的文件缓存是基于http协议的文件缓存，一般放在nginx中。

因为静态文件（比如css，js，图片）中，很多都是不经常更新的。nginx使用proxy_cache将用户的请求缓存到本地一个目录。下一个相同请求可以直接调取缓存文件，就不用去请求服务器了。

```
server {  
    listen      80 default_server;  
    server_name localhost;  
    root /mnt/blog/;  
  
    location / {  
  
    }
```

#要缓存文件的后缀，可以在以下设置。

```

        location ~ .*\. (gif|jpg|png|css|js) (.* ) {
            proxy_pass http://ip地址:90;
            proxy_redirect off;
            proxy_set_header Host $host;
            proxy_cache cache_one;
            proxy_cache_valid 200 302 24h;
            proxy_cache_valid 301 30d;
            proxy_cache_valid any 5m;
            expires 90d;
            add_header wall "hello lagou.";
        }
    }
}

```

Redis缓存

分布式缓存，采用主从+哨兵或RedisCluster的方式缓存数据库的数据。

在实际开发中

作为数据库使用，数据要完整

作为缓存使用

作为Mybatis的二级缓存使用

缓存大小

GuavaCache的缓存设置方式:

```
CacheBuilder.newBuilder().maximumSize(num) // 超过num会按照LRU算法来移除缓存
```

Nginx的缓存设置方式:

```

http {
    ...
    proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_cache:10m max_size=10g
    inactive=60m use_temp_path=off;

    server {
        proxy_cache mycache;
        location / {
            proxy_pass http://localhost:8000;
        }
    }
}

```

Redis缓存设置:

```

maxmemory=num # 最大缓存量 一般为内存的3/4
maxmemory-policy allkeys lru #

```

缓存淘汰策略的选择

- allkeys-lru : 在不确定时一般采用策略。
- volatile-lru : 比allkeys-lru性能差 存:过期时间

- allkeys-random：希望请求符合平均分布(每个元素以相同的概率被访问)
- 自己控制：volatile-ttl 缓存穿透
- 禁止驱逐 用作DB 不设置maxmemory

key数量

官方说Redis单例能处理key：2.5亿个

一个key或是value大小最大是512M

读写峰值

Redis采用的是基于内存的采用的是**单进程单线程**模型的 **KV 数据库**，由**C语言编写**，官方提供的数据是可以达到110000+的QPS（每秒内查询次数）。80000的写

命中率

命中：可以直接通过缓存获取到需要的数据。

不命中：无法直接通过缓存获取到想要的的数据，需要再次查询数据库或者执行其它的操作。原因可能是由于缓存中根本不存在，或者缓存已经过期。

通常来讲，缓存的命中率越高则表示使用缓存的收益越高，应用的性能越好（响应时间越短、吞吐量越高），抗并发的能力越强。

由此可见，在高并发的互联网系统中，缓存的命中率是至关重要的指标。

通过info命令可以监控服务器状态

```
127.0.0.1:6379> info
# Server
redis_version:5.0.5
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:e188a39ce7a16352
redis_mode:standalone
os:Linux 3.10.0-229.el7.x86_64 x86_64
arch_bits:64
#缓存命中
keyspace_hits:1000
#缓存未命中
keyspace_misses:20
used_memory:433264648
expired_keys:1333536
evicted_keys:1547380
```

命中率=1000/1000+20=83%

一个缓存失效机制，和过期时间设计良好的系统，命中率可以做到95%以上。

影响缓存命中率的因素：

- 1、缓存的数量越少命中率越高，比如缓存单个对象的命中率要高于缓存集合
- 2、过期时间越长命中率越高
- 3、缓存越大缓存的对象越多，则命中的越多

过期策略

Redis的过期策略是定时删除+惰性删除，这个前面已经讲了。

性能监控指标

利用info命令就可以了解Redis的状态了，主要监控指标有：

```
connected_clients:68 #连接的客户端数量
used_memory_rss_human:847.62M #系统给redis分配的内存
used_memory_peak_human:794.42M #内存使用的峰值大小
total_connections_received:619104 #服务器已接受的连接请求数量
instantaneous_ops_per_sec:1159 #服务器每秒钟执行的命令数量 qps
instantaneous_input_kbps:55.85 #redis网络入口kps
instantaneous_output_kbps:3553.89 #redis网络出口kps
rejected_connections:0 #因为最大客户端数量限制而被拒绝的连接请求数量
expired_keys:0 #因为过期而被自动删除的数据库键数量
evicted_keys:0 #因为最大内存容量限制而被驱逐（evict）的键数量
keyspace_hits:0 #查找数据库键成功的次数
keyspace_misses:0 #查找数据库键失败的次数
```

Redis监控平台：

grafana、prometheus以及redis_exporter。

缓存预热

缓存预热就是系统启动前,提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候,先查询数据库,然后再将数据缓存的问题!用户直接查询实现被预热的缓存数据。

加载缓存思路：

- 数据量不大，可以在项目启动的时候自动进行加载
- 利用定时任务刷新缓存，将数据库的数据刷新到缓存中

缓存问题

缓存穿透

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。

缓存穿透是指在高并发下查询key不存在的数据，会穿过缓存查询数据库。导致数据库压力过大而宕机

解决方案：

- 对查询结果为空的情况也进行缓存，缓存时间（ttl）设置短一点，或者该key对应的数据insert了之后清理缓存。

问题：缓存太多空值占用了更多的空间

- 使用布隆过滤器。在缓存之前在加一层布隆过滤器，在查询的时候先去布隆过滤器查询 key 是否存在，如果不存在就直接返回，存在再查缓存和DB。

把字符串----->位 省空间 (1或0)

不用循环----->比较位置 省时间

缓存雪崩

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。

突然间大量的key失效了或redis重启，大量访问数据库，数据库崩溃

解决方案:

- 1、key的失效期分散开 不同的key设置不同的有效期
- 2、设置二级缓存（数据不一定一致）
- 3、高可用（脏读）

缓存击穿

对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。

缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

解决方案:

- 1、用分布式锁控制访问的线程

使用redis的setnx互斥锁先进行判断，这样其他线程就处于等待状态，保证不会有大并发操作去操作数据库。

- 2、不设超时时间，volatile-lru 但会造成写一致问题

当数据库数据发生更新时，缓存中的数据不会及时更新，这样会造成数据库中的数据与缓存中的数据的不一致，应用会从缓存中读取到脏数据。可采用延时双删策略处理，这个我们后面会详细讲到。

数据不一致

缓存和DB的数据不一致的根源：数据源不一样

如何解决

强一致性很难，追求最终一致性（时间）

互联网业务数据处理的特点

高吞吐量

低延迟

数据敏感性低于金融业

时序控制是否可行？

先更新数据库再更新缓存或者先更新缓存再更新数据库

本质上不是一个原子操作，所以时序控制不可行

高并发情况下会产生不一致

保证数据的最终一致性(延时双删)

- 1、先更新数据库同时删除缓存项(key)，等读的时候再填充缓存
- 2、2秒后再删除一次缓存项(key)
- 3、设置缓存过期时间 Expired Time 比如 10秒 或1小时
- 4、将缓存删除失败记录到日志中，利用脚本提取失败记录再次删除（缓存失效期过长 7*24）

升级方案

通过数据库的binlog来异步淘汰key，利用工具(canal)将binlog日志采集发送到MQ中，然后通过ACK机制确认处理删除缓存。

数据并发竞争

这里的并发指的是多个redis的client同时set 同一个key引起的并发问题。

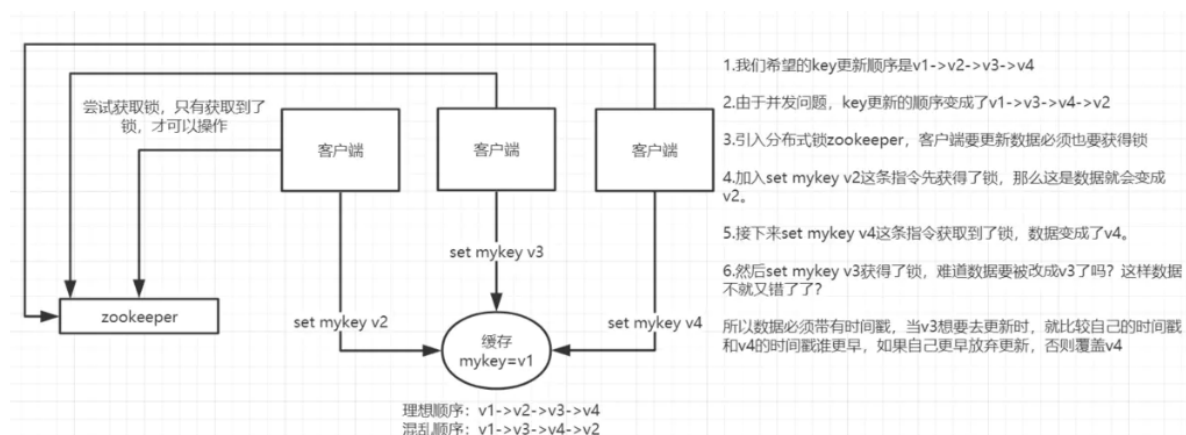
多客户端 (Jedis) 同时并发写一个key，一个key的值是1，本来按顺序修改为2,3,4，最后是4，但是顺序变成了4,3,2，最后变成了2。

第一种方案：分布式锁+时间戳

1.整体技术方案

这种情况，主要是准备一个分布式锁，大家去抢锁，抢到锁就做set操作。

加锁的目的实际上就是把并行读写改成串行读写的方式，从而来避免资源竞争。



2.Redis分布式锁的实现

主要用到的redis函数是setnx()

用SETNX实现分布式锁

时间戳

由于上面举的例子，要求key的操作需要顺序执行，所以需要保存一个时间戳判断set顺序。

系统A key 1 {ValueA 7:00}

系统B key 1 { ValueB 7:05}

假设系统B先抢到锁，将key1设置为{ValueB 7:05}。接下来系统A抢到锁，发现自己的key1的时间戳早于缓存中的时间戳（7:00<7:05），那就不做set操作了。

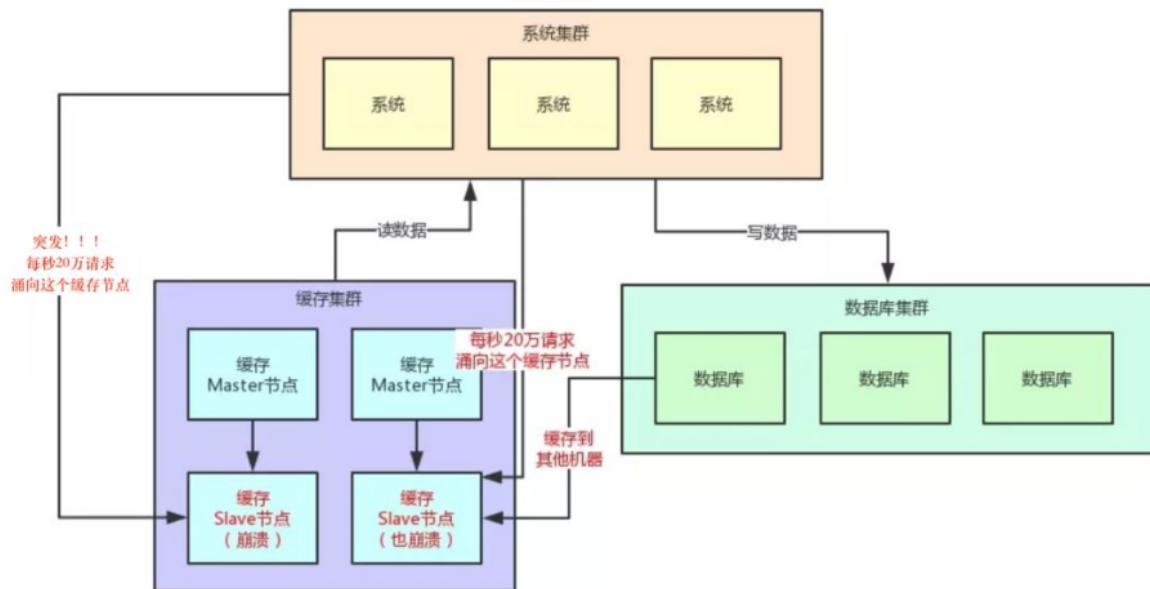
第二种方案：利用消息队列

在并发量过大的情况下,可以通过消息中间件进行处理,把并行读写进行串行化。

把Redis的set操作放在队列中使其串行化,必须的一个一个执行。

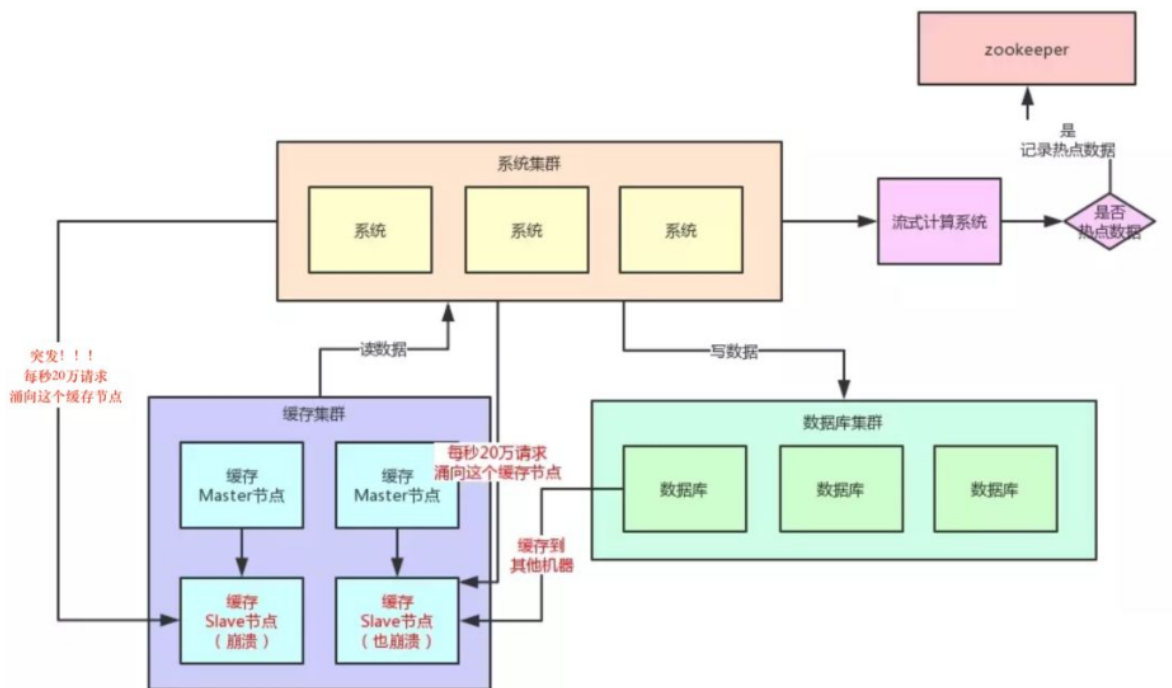
Hot Key

当有大量的请求(几十万)访问某个Redis某个key时，由于流量集中达到网络上限，从而导致这个redis的服务器宕机。造成缓存击穿，接下来对这个key的访问将直接访问数据库造成数据库崩溃，或者访问数据库回填Redis再访问Redis，继续崩溃。



如何发现热key

- 1、预估热key，比如秒杀的商品、火爆的新闻等
- 2、在客户端进行统计，实现简单，加一行代码即可
- 3、如果是Proxy，比如Codis，可以在Proxy端收集
- 4、利用Redis自带的命令，monitor、hotkeys。但是执行缓慢（不要用）
- 5、利用基于大数据领域的流式计算技术来进行实时数据访问次数的统计，比如 Storm、Spark Streaming、Flink，这些技术都是可以的。发现热点数据后可以写到zookeeper中



如何处理热Key：

1、变分布式缓存为本地缓存

发现热key后，把缓存数据取出后，直接加载到本地缓存中。可以采用Ehcache、Guava Cache都可以，这样系统在访问热key数据时就可以直接访问自己的缓存了。（数据不要求时时一致）

2、在每个Redis主节点上备份热key数据，这样在读取时可以采用随机读取的方式，将访问压力负载到每个Redis上。

3、利用对热点数据访问的限流熔断保护措施

每个系统实例每秒最多请求缓存集群读操作不超过 400 次，一超过就可以熔断掉，不让请求缓存集群，直接返回一个空白信息，然后用户稍后会自行再次重新刷新页面之类的。（首页不行，系统友好性差）

通过系统层自己直接加限流熔断保护措施，可以很好的保护后面的缓存集群。

Big Key

大key指的是存储的值（Value）非常大，常见场景：

- 热门话题下的讨论
- 大V的粉丝列表
- 序列化后的图片
- 没有及时处理的垃圾数据

.....

大key的影响：

- 大key会大量占用内存，在集群中无法均衡
- Redis的性能下降，主从复制异常
- 在主动删除或过期删除时会操作时间过长而引起服务阻塞

如何发现大key：

1、redis-cli --bigkeys命令。可以找到某个实例5种数据类型(String、hash、list、set、zset)的最大key。

但如果Redis的key比较多，执行该命令会比较慢

2、获取生产Redis的rdb文件，通过rdbtools分析rdb生成csv文件，再导入MySQL或其他数据库中进行分析统计，根据size_in_bytes统计bigkey

大key的处理：

优化big key的原则就是string减少字符串长度，list、hash、set、zset等减少成员数。

1、string类型的big key，尽量不要存入Redis中，可以使用文档型数据库MongoDB或缓存到CDN上。

如果必须用Redis存储，最好单独存储，不要和其他的key一起存储。采用一主一从或多从。

2、单个简单的key存储的value很大，可以尝试将对象分拆成几个key-value，使用mget获取值，这样分拆的意义在于分拆单次操作的压力，将操作压力平摊到多次操作中，降低对redis的IO影响。

2、hash，set，zset，list中存储过多的元素，可以将这些元素分拆。（常见）

以hash类型举例来说，对于field过多的场景，可以根据field进行hash取模，生成一个新的key，例如原来的

hash_key:{filed1:value, filed2:value, filed3:value ...}，可以hash取模后形成如下key:value形式

hash_key:1:{filed1:value}

hash_key:2:{filed2:value}

hash_key:3:{filed3:value}

...

取模后，将原先单个key分成多个key，每个key field个数为原先的1/N

3、删除大key时不要使用del,因为del是阻塞命令，删除时会影响性能。

4、使用 lazy delete (unlink命令)

删除指定的key(s),若key不存在则该key被跳过。但是，相比DEL会产生阻塞，该命令会在另一个线程中回收内存，因此它是非阻塞的。这也是该命令名字的由来：仅将keys从key空间中删除，真正的数据删除会在后续异步操作。

```
redis> SET key1 "Hello"
"OK"
redis> SET key2 "world"
"OK"
redis> UNLINK key1 key2 key3
(integer) 2
```

缓存与数据库一致性

缓存更新策略

- 利用Redis的缓存淘汰策略被动更新 LRU、LFU
- 利用TTL被动更新
- 在更新数据库时主动更新（先更数据库再删缓存----延时双删）
- 异步更新 定时任务 数据不保证时时一致 不穿DB

不同策略之间的优缺点

策略	一致性	维护成本
利用Redis的缓存淘汰策略被动更新	最差	最低
利用TTL被动更新	较差	较低
在更新数据库时主动更新	较强	最高

与Mybatis整合

可以使用Redis做Mybatis的二级缓存，在分布式环境下可以使用。

框架采用springboot+Mybatis+Redis。框架的搭建就不赘述了。

1、在pom.xml中添加Redis依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2、在application.yml中添加Redis配置

```
#开发配置
spring:
  #数据源配置
  datasource:
    url: jdbc:mysql://192.168.127.128:3306/test?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource

  redis:
    host: 192.168.127.128
    port: 6379
    jedis:
      pool:
        min-idle: 0
        max-idle: 8
        max-active: 8
        max-wait: -1ms

#公共配置与profiles选择无关
mybatis:
  typeAliasesPackage: com.lagou.rcache.entity
  mapperLocations: classpath:mapper/*.xml
```

3、缓存实现

ApplicationContextHolder 用于注入RedisTemplate

```
package com.lagou.rcache.utils;

import org.springframework.beans.BeansException;
```

```

import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;

@Component
public class ApplicationContextHolder implements ApplicationContextAware {
    private static ApplicationContext ctx;

    @Override
    //向工具类注入applicationContext
    public void setApplicationContext(ApplicationContext applicationContext)
throws BeansException {
        ctx = applicationContext;    //ctx就是注入的applicationContext
    }

    //外部调用ctx
    public static ApplicationContext getCtx() {
        return ctx;
    }

    public static <T> T getBean(Class<T> tClass) {
        return ctx.getBean(tClass);
    }

    @SuppressWarnings("unchecked")
    public static <T> T getBean(String name) {
        return (T) ctx.getBean(name);
    }
}

```

RedisCache 使用redis实现mybatis二级缓存

```

package com.lagou.rcache.utils;

import org.apache.ibatis.cache.Cache;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.redis.core.RedisCallback;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.ValueOperations;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * 使用redis实现mybatis二级缓存
 */
public class RedisCache implements Cache {

    //缓存对象唯一标识
    private final String id; //orm的框架都是按对象的方式缓存，而每个对象都需要一个唯一标识。

    //用于事务性缓存操作的读写锁
    private static ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    //处理事务性缓存中做的
    //操作数据缓存的--跟着线程走的

```

```

private RedisTemplate redisTemplate; //Redis的模板负责将缓存对象写到redis服务器
里面去
//缓存对象的是失效时间，30分钟
private static final long EXPIRE_TIME_IN_MINUT = 30;

//构造方法---把对象唯一标识传进来
public RedisCache(String id) {
    if (id == null) {
        throw new IllegalArgumentException("缓存对象id是不能为空的");
    }
    this.id = id;
}

@Override
public String getId() {
    return this.id;
}

//给模板对象RedisTemplate赋值，并传出去
private RedisTemplate getRedisTemplate() {
    if (redisTemplate == null) { //每个连接池的连接都要获得RedisTemplate
        redisTemplate = ApplicationContextHolder.getBean("redisTemplate");
    }
    return redisTemplate;
}

/*
    保存缓存对象的方法
*/
@Override
public void putObject(Object key, Object value) {
    try {
        RedisTemplate redisTemplate = getRedisTemplate();
        //使用redisTemplate得到值操作对象
        ValueOperations operation = redisTemplate.opsForValue();
        //使用值操作对象operation设置缓存对象
        operation.set(key, value, EXPIRE_TIME_IN_MINUT, TimeUnit.MINUTES);
        //TimeUnit.MINUTES系统当前时间的分钟数
        System.out.println("缓存对象保存成功");
    } catch (Throwable t) {
        System.out.println("缓存对象保存失败" + t);
    }
}

/*
    获取缓存对象的方法
*/
@Override
public Object getObject(Object key) {
    try {
        RedisTemplate redisTemplate = getRedisTemplate();
        ValueOperations operations = redisTemplate.opsForValue();
        Object result = operations.get(key);
        System.out.println("获取缓存对象");
        return result;
    }
}

```

```

    } catch (Throwable t) {
        System.out.println("缓存对象获取失败" + t);
        return null;
    }
}

/*
    删除缓存对象
*/
@Override
public Object (Object key) {
    try {
        RedisTemplate redisTemplate = getRedisTemplate();
        redisTemplate.delete(key);
        System.out.println("删除缓存对象成功! ");
    } catch (Throwable t) {
        System.out.println("删除缓存对象失败! " + t);
    }
    return null;
}

/*
    清空缓存对象
    当缓存的对象更新了的化，就执行此方法
*/
@Override
public void clear() {
    RedisTemplate redisTemplate = getRedisTemplate();
    //回调函数
    redisTemplate.execute((RedisCallback) collection -> {
        collection.flushDb();
        return null;
    });
    System.out.println("清空缓存对象成功! ");
}

//可选实现的方法
@Override
public int getSize() {
    return 0;
}

@Override
public ReadWriteLock getReadWriteLock() {
    return readWriteLock;
}
}

```

4、在mapper中增加二级缓存开启（默认不开启）

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.lagou.rcache.dao.UserDao" >
    <cache type="com.lagou.rcache.utils.RedisCache" />
    <resultMap id="BaseResultMap" type="com.lagou.rcache.entity.TUser" >

```

```

        <id column="id" property="id" jdbcType="INTEGER" />
        <result column="name" property="name" jdbcType="VARCHAR" />
        <result column="address" property="address" jdbcType="VARCHAR" />
    </resultMap>
    <sql id="Base_Column_List" >
        id, name, address
    </sql>
    <select id="selectUser" resultMap="BaseResultMap">
        select
            <include refid="Base_Column_List" />
        from tuser
    </select>
</mapper>

```

5、在启动时允许缓存

```

package com.lagou.rcache;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@SpringBootApplication
@MapperScan("com.lagou.rcache.dao")
@EnableCaching
public class RcacheApplication {

    public static void main(String[] args) {

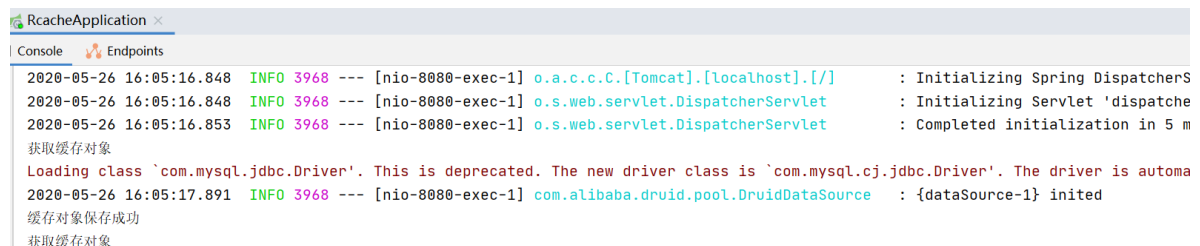
        SpringApplication.run(RcacheApplication.class, args);
    }

}

```

运行结果：

控制台：



```

RcacheApplication x
Console Endpoints
2020-05-26 16:05:16.848 INFO 3968 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherS
2020-05-26 16:05:16.848 INFO 3968 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatche
2020-05-26 16:05:16.853 INFO 3968 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 5 m
获取缓存对象
Loading class `com.mysql.jdbc.Driver'. This is deprecated. The new driver class is `com.mysql.cj.jdbc.Driver'. The driver is automa
2020-05-26 16:05:17.891 INFO 3968 --- [nio-8080-exec-1] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} inited
缓存对象保存成功
获取缓存对象

```

Redis客户端：

```
127.0.0.1:6379> keys *
1) "\xac\xed\x00\x05sr\x00
org.apache.ibatis.cache.CacheKey\x0f\xe9\xd5\xb4\xcd3\xa8\x82\x02\x00\x05j\x00\b
checksumI\x00\x05countI\x00\bcryptcodeI\x00\nmultiplierL\x00\nupdateListt\x00\x10
Ljava/util/List;xp\x00\x00\x00\x00\x87\xd8u%\x00\x00\x00\x05\xb0\xf6RU\x00\x00\x
00%sr\x00\x13java.util.ArrayListx\x81\xd2\x1d\x99\xc7a\x9d\x03\x00\x01I\x00\x04s
izexp\x00\x00\x00\x05w\x04\x00\x00\x00\x05t\x00'com.lagou.rcache.dao.UserDao.sel
ectUsersr\x00\x11java.lang.Integer\x12\xe2\xa0\xa4\xf7\x81\x878\x02\x00\x01I\x00
\x05valuexr\x00\x10java.lang.Number\x86\xac\x95\x1d\x0b\x94\xe0\x8b\x02\x00\x00x
p\x00\x00\x00\x00sq\x00~\x00\x06\x7f\xff\xff\xff\x00cselect\n          \n
id, name, address\n          \n          from tuser\x00\x15SqlSessionFactoryBeanx"
```

分布式锁

watch

利用Watch实现Redis乐观锁

乐观锁基于CAS（Compare And Swap）思想（比较并替换），是不具有互斥性，不会产生锁等待而消耗资源，但是需要反复的重试，但也是因为重试的机制，能比较快的响应。因此我们可以利用redis来实现乐观锁。具体思路如下：

- 1、利用redis的watch功能，监控这个redisKey的状态值
- 2、获取redisKey的值
- 3、创建redis事务
- 4、给这个key的值+1
- 5、然后去执行这个事务，如果key的值被修改过则回滚，key不加1

Redis乐观锁实现秒杀

```
public class Second {
    public static void main(String[] arg) {
        String redisKey = "lock";

        ExecutorService executorService = Executors.newFixedThreadPool(20);
        try {
            Jedis jedis = new Jedis("127.0.0.1", 6378);
            // 初始值
            jedis.set(redisKey, "0");
            jedis.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

        for (int i = 0; i < 1000; i++) {

            executorService.execute(() -> {

                Jedis jedis1 = new Jedis("127.0.0.1", 6378);
                try {
                    jedis1.watch(redisKey);
                    String redisValue = jedis1.get(redisKey);
                    int valInteger = Integer.valueOf(redisValue);
                    String userInfo = UUID.randomUUID().toString();
```



```

        // 没有秒完
        if (valInteger < 20) {
            Transaction tx = jedis1.multi();
            tx.incr(rediskey);
            List list = tx.exec();
            // 秒成功    失败返回空list而不是空
            if (list != null && list.size() > 0) {

                System.out.println("用户: " + userInfo + ", 秒杀成功!");
                当前成功人数: " + (valInteger + 1));

            }
            // 版本变化, 被别人抢了。
            else {
                System.out.println("用户: " + userInfo + ", 秒杀失败");
            }
        }
        // 秒完了
        else {
            System.out.println("已经有20人秒杀成功, 秒杀结束");
        }

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        jedis1.close();
    }

    });
}
executorService.shutdown();

}

}

```

setnx

实现原理

共享资源互斥

共享资源串行化

单应用中使用锁: (单进程多线程)

synchronized、ReentrantLock

分布式应用中使用锁: (多进程多线程)

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。

利用Redis的单线程特性对共享资源进行串行化处理

实现方式

获取锁

方式1 (使用set命令实现) --推荐

```

/**
 * 使用redis的set命令实现获取分布式锁
 * @param lockKey      可以就是锁
 * @param requestId    请求ID，保证同一性      uuid+threadID
 * @param expireTime   过期时间，避免死锁
 * @return
 */
public boolean getLock(String lockKey,String requestId,int expireTime) {
    //NX:保证互斥性
    // hset  原子性操作 只要lockKey有效 则说明有进程在使用分布式锁
    String result = jedis.set(lockKey, requestId, "NX", "EX", expireTime);
    if("OK".equals(result)) {
        return true;
    }

    return false;
}

```

方式2（使用setnx命令实现） -- 并发会产生问题

```

public boolean getLock(String lockKey,String requestId,int expireTime) {
    Long result = jedis.setnx(lockKey, requestId);
    if(result == 1) {
        //成功设置 进程down 永久有效 别的进程就无法获得锁
        jedis.expire(lockKey, expireTime);
        return true;
    }

    return false;
}

```

释放锁

方式1（del命令实现） -- 并发

```

/**
 * 释放分布式锁
 * @param lockKey
 * @param requestId
 */
public static void releaseLock(String lockKey,String requestId) {

    if (requestId.equals(jedis.get(lockKey))) {
        jedis.del(lockKey);
    }
}

```

问题在于如果调用jedis.del()方法的时候，这把锁已经不属于当前客户端的时候会解除他人加的锁。那么是否真的有这种场景？答案是肯定的，比如客户端A加锁，一段时间之后客户端A解锁，在执行jedis.del()之前，锁突然过期了，此时客户端B尝试加锁成功，然后客户端A再执行del()方法，则将客户端B的锁给解除了。

方式2 (redis+lua脚本实现) --推荐

```
public static boolean releaseLock(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 (result.equals(1L)) {  
        return true;  
    }  
    return false;  
}
```

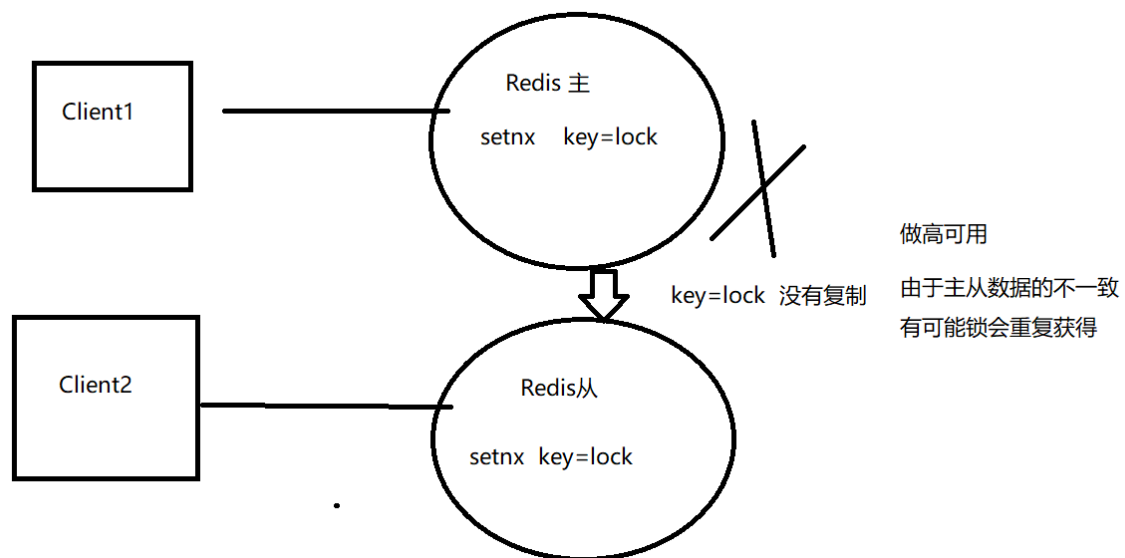
存在问题

单机

无法保证高可用

主--从

无法保证数据的强一致性，在主机宕机时会造成锁的重复获得。



无法续租

超过expireTime后，不能继续使用

本质分析

CAP模型分析

在分布式环境下不可能满足三者共存，只能满足其中的两者共存，在分布式下P不能舍弃(舍弃P就是单机了)。

所以只能是CP（强一致性模型）和AP(高可用模型)。

分布式锁是CP模型，Redis集群是AP模型。(base)

Redis集群不能保证数据的随时一致性，只能保证数据的最终一致性。

为什么还可以用Redis实现分布式锁？

与业务有关

当业务不需要数据强一致性时，比如：社交场景，就可以使用Redis实现分布式锁

当业务必须要数据的强一致性，即不允许重复获得锁，比如金融场景（重复下单，重复转账）就不要使用

可以使用CP模型实现，比如：zookeeper和etcd。

Redisson分布式锁的使用

Redisson是架设在Redis基础上的一个Java驻内存数据网格（In-Memory Data Grid）。

Redisson在基于NIO的Netty框架上，生产环境使用分布式锁。

加入jar包的依赖

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>2.7.0</version>
</dependency>
```

配置Redisson

```
public class RedissonManager {
    private static Config config = new Config();
    //声明redisso对象
    private static Redisson redisson = null;
    //实例化redisson
    static{
        config.useClusterServers()

        // 集群状态扫描间隔时间，单位是毫秒

        .setScanInterval(2000)

        //cluster方式至少6个节点(3主3从，3主做sharding，3从用来保证主宕机后可以高可用)

        .addNodeAddress("redis://127.0.0.1:6379" )

        .addNodeAddress("redis://127.0.0.1:6380")

        .addNodeAddress("redis://127.0.0.1:6381")

        .addNodeAddress("redis://127.0.0.1:6382")

        .addNodeAddress("redis://127.0.0.1:6383")

        .addNodeAddress("redis://127.0.0.1:6384");

        //得到redisson对象
        redisson = (Redisson) Redisson.create(config);
    }

    //获取redisson对象的方法
    public static Redisson getRedisson(){
```

```

        return redisson;
    }
}

```

锁的获取和释放

```

public class DistributedRedisLock {
    //从配置类中获取redisson对象
    private static Redisson redisson = RedissonManager.getRedisson();
    private static final String LOCK_TITLE = "redisLock_";
    //加锁
    public static boolean acquire(String lockName){
        //声明key对象
        String key = LOCK_TITLE + lockName;
        //获取锁对象
        RLock mylock = redisson.getLock(key);
        //加锁，并且设置锁过期时间3秒，防止死锁的产生    uuid+threadId
        mylock.lock(2,3,TimeUtil.SECOND);
        //加锁成功
        return true;
    }
    //锁的释放
    public static void release(String lockName){
        //必须是和加锁时的同一个key
        String key = LOCK_TITLE + lockName;
        //获取锁对象
        RLock mylock = redisson.getLock(key);
        //释放锁（解锁）
        mylock.unlock();
    }
}

```

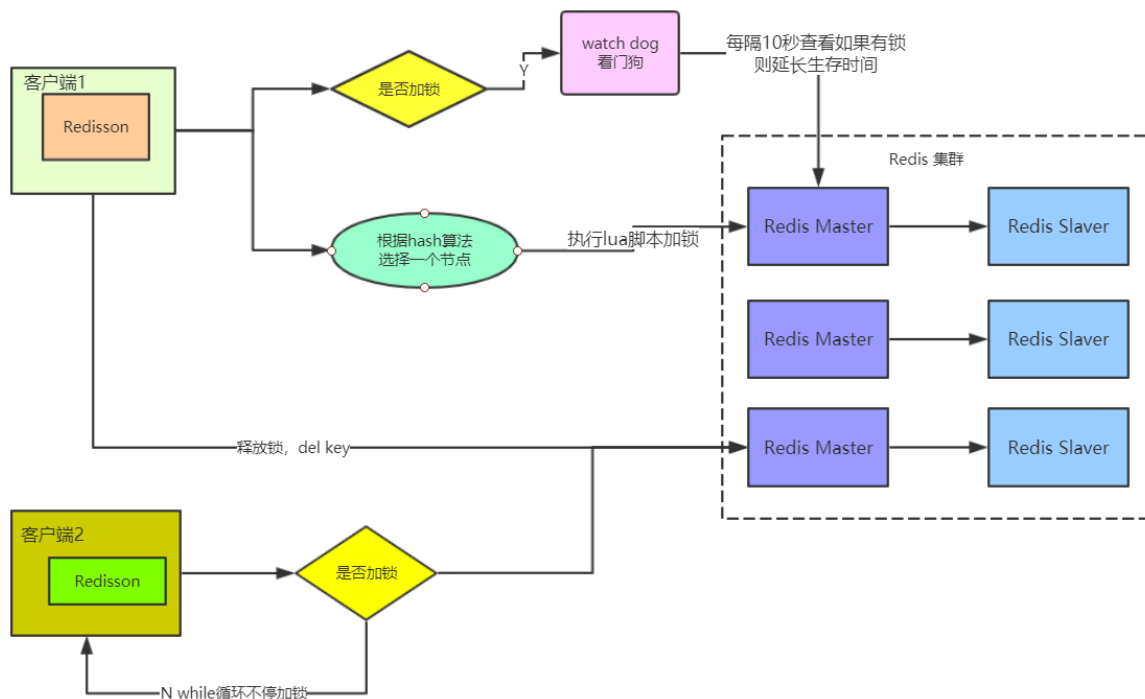
业务逻辑中使用分布式锁

```

public String discount() throws IOException{
    String key = "lock001";
    //加锁
    DistributedRedisLock.acquire(key);
    //执行具体业务逻辑
    dosoming
    //释放锁
    DistributedRedisLock.release(key);
    //返回结果
    return soming;
}

```

Redisson分布式锁的实现原理



加锁机制

如果该客户端面对的是一个redis cluster集群，他首先会根据hash节点选择一台机器。

发送lua脚本到redis服务器上，脚本如下：

```
"if (redis.call('exists',KEYS[1])==0) then "+           --看有没有锁
    "redis.call('hset',KEYS[1],ARGV[2],1) ; "+       --无锁 加锁
    "redis.call('pexpire',KEYS[1],ARGV[1]) ; "+
    "return nil; end ;" +
    "if (redis.call('hexists',KEYS[1],ARGV[2]) ==1 ) then "+ --我加的锁
        "redis.call('hincrby',KEYS[1],ARGV[2],1) ; "+ --重入锁
        "redis.call('pexpire',KEYS[1],ARGV[1]) ; "+
        "return nil; end ;" +
    "return redis.call('pttl',KEYS[1]) ;" --不能加锁，返回锁的时间
```

lua的作用：保证这段复杂业务逻辑执行的原子性。

lua的解释：

KEYS[1])： 加锁的key

ARGV[1]： key的生存时间，默认为30秒

ARGV[2]： 加锁的客户端ID (UUID.randomUUID()) + ":" + threadId)

第一段if判断语句，就是用“exists myLock”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，用下面的命令：

hset myLock

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

myLock :{"8743c9c0-0795-4907-87fd-6c719a6b4586:1":1 }

上述就代表“8743c9c0-0795-4907-87fd-6c719a6b4586:1”这个客户端对“myLock”这个锁key完成了加锁。

接着会执行“pexpire myLock 30000”命令，设置myLock这个锁key的生存时间是30秒。

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 myLock”，发现myLock这个锁key已经存在了。

接着第二个if判断，判断一下，myLock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 myLock返回的一个数字，这个数字代表了myLock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。

自动延时机制

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。

可重入锁机制

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

第二个if判断会成立，因为myLock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 myLock
```

```
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。数据结构会变成：

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

释放锁机制

执行lua脚本如下：

```
#如果key已经不存在，说明已经被解锁，直接发布（publish）redis消息
"if (redis.call('exists', KEYS[1]) == 0) then " +
    "redis.call('publish', KEYS[2], ARGV[1]); " +
    "return 1; " +
    "end;" +
# key和field不匹配，说明当前客户端线程没有持有锁，不能主动解锁。不是我加的锁 不能解锁
"if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " +
    "return nil;" +
    "end;" +
# 将value减1
"local counter = redis.call('hincrby', KEYS[1], ARGV[3],
-1); " +
# 如果counter>0说明锁在重入，不能删除key
"if (counter > 0) then " +
    "redis.call('pexpire', KEYS[1], ARGV[2]); " +
    "return 0; " +
# 删除key并且publish 解锁消息
"else " +
    "redis.call('del', KEYS[1]); " + #删除锁
    "redis.call('publish', KEYS[2], ARGV[1]); " +
```

```
        "return 1; "+
        "end; " +
        "return nil;",
```

- KEYS[1]：需要加锁的key，这里需要是字符串类型。
- KEYS[2]：redis消息的ChannelName,一个分布式锁对应唯一的一个channelName:
"redisson_lockchannel{" + getName() + "}"
- ARGV[1]：reids消息体，这里只需要一个字节的标记就可以，主要标记redis的key已经解锁，再结合redis的Subscribe，能唤醒其他订阅解锁消息的客户端线程申请锁。
- ARGV[2]：锁的超时时间，防止死锁
- ARGV[3]：锁的唯一标识，也就是刚才介绍的 id (UUID.randomUUID()) + ":" + threadId

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，就是每次都对我myLock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 myLock"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。

分布式锁特性

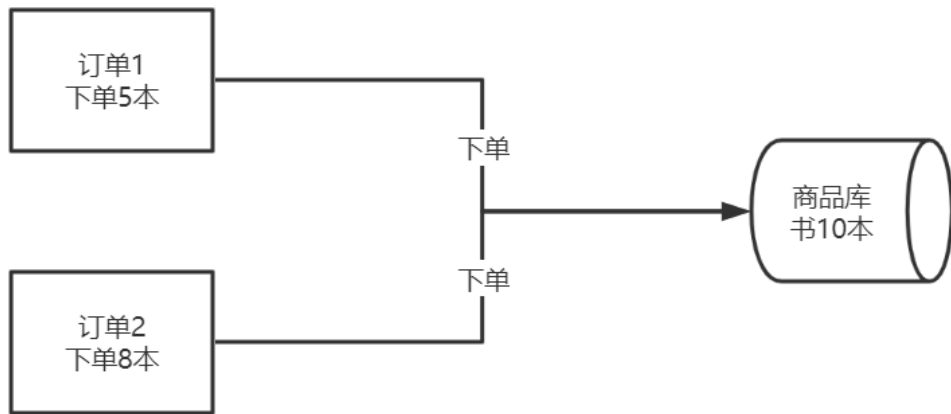
- 互斥性
任意时刻，只能有一个客户端获取锁，不能同时有两个客户端获取到锁。
- 同一性
锁只能被持有该锁的客户端删除，不能由其它客户端删除。
- 可重入性
持有某个锁的客户端可继续对该锁加锁，实现锁的续租
- 容错性
锁失效后（超过生命周期）自动释放锁（key失效），其他客户端可以继续获得该锁，防止死锁

分布式锁的实际应用

- 数据并发竞争

利用分布式锁可以将处理串行化，前面已经讲过了。

- 防止库存超卖

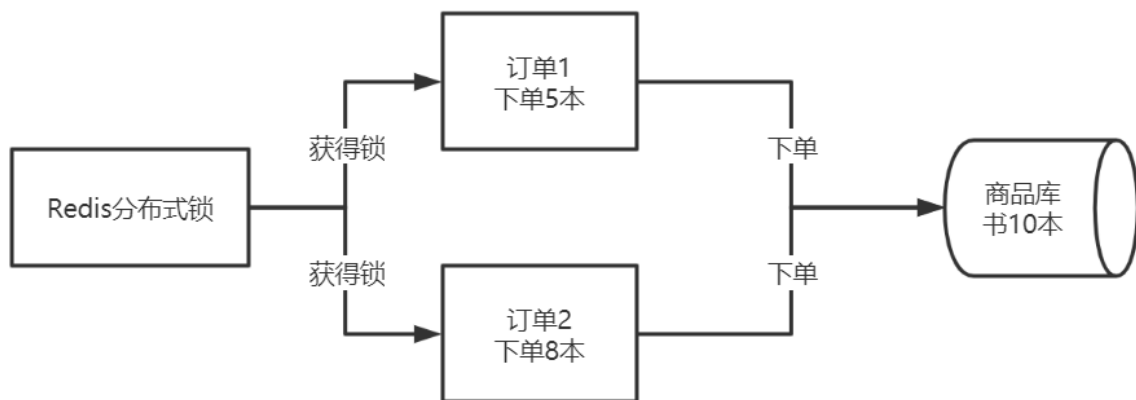


订单1下单前会先查看库存，库存为10，所以下单5本可以成功；

订单2下单前会先查看库存，库存为10，所以下单8本可以成功；

订单1和订单2 同时操作，共下单13本，但库存只有10本，显然库存不够了，这种情况称为库存超卖。

可以采用分布式锁解决这个问题。



订单1和订单2都从Redis中获得分布式锁(setnx)，谁能获得锁谁进行下单操作，这样就把订单系统下单的顺序串行化了，就不会出现超卖的情况了。伪码如下：

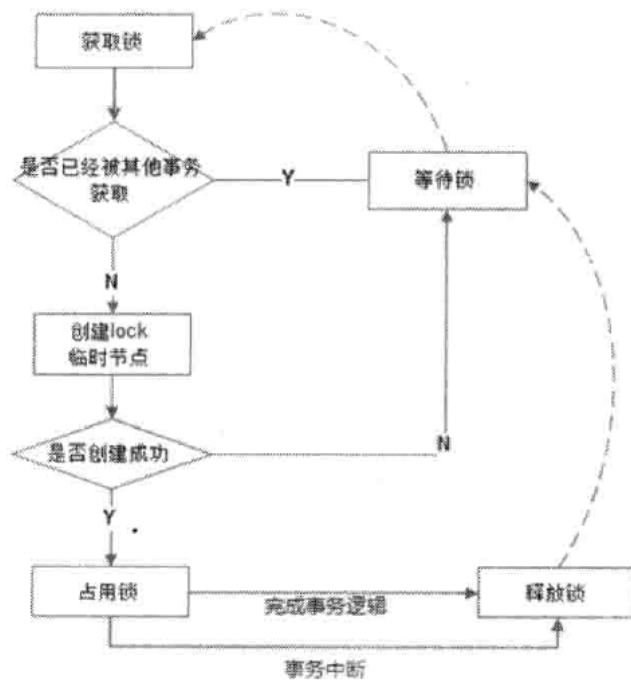
```

//加锁并设置有效期
if(redis.lock("RDL",200)){
    //判断库存
    if (orderNum<getCount()){
        //加锁成功 ,可以下单
        order(5);
        //释放锁
        redis.unlock("RDL");
    }
}
  
```

注意此种方法会降低处理效率，这样不适合秒杀的场景，秒杀可以使用CAS和Redis队列的方式。

Zookeeper分布式锁的对比

- 基于Redis的set实现分布式锁
- 基于zookeeper临时节点的分布式锁



- 基于etcd实现

三者的对比，如下表

	Redis	zookeeper	etcd
一致性算法	无	paxos (ZAB)	raft
CAP	AP	CP	CP
高可用	主从集群	n+1 (n至少为2)	n+1
接口类型	客户端	客户端	http/grpc
实现	setNX	createEphemeral	restful API

分布式集群架构中的session分离

传统的session是由tomcat自己进行维护和管理，但是对于集群或分布式环境，不同的tomcat管理各自的session，很难进行session共享，通过传统的模式进行session共享，会造成session对象在各个tomcat之间，通过网络和Io进行复制，极大的影响了系统的性能。

可以将登录成功后的Session信息，存放在Redis中，这样多个服务器(Tomcat)可以共享Session信息。

利用spring-session-data-redis (SpringSession)，可以实现基于redis来实现的session分离。这个知识点在讲Spring的时候可以讲过了，这里就不再赘述了。

阿里Redis使用手册

本文主要介绍在使用阿里云Redis的开发规范，从下面几个方面进行说明。

- 键值设计
- 命令使用
- 客户端使用

· 相关工具

通过本文的介绍可以减少使用Redis过程带来的问题。

一、键值设计

1、key名设计

可读性和可管理性

以业务名(或数据库名)为前缀(防止key冲突), 用冒号分隔, 比如业务名:表名:id

```
ugc:video:1
```

简洁性

保证语义的前提下, 控制key的长度, 当key较多时, 内存占用也不容忽视, 例如:

```
user:{uid}:friends:messages:{mid} 简化为 u:{uid}:fr:m:{mid}。
```

不要包含特殊字符

反例: 包含空格、换行、单双引号以及其他转义字符

2、value设计

拒绝bigkey

防止网卡流量、慢查询, string类型控制在10KB以内, hash、list、set、zset元素个数不要超过5000。

反例: 一个包含200万个元素的list。拆解

非字符串的bigkey, 不要使用del删除, 使用hscan、sscan、zscan方式渐进式删除, 同时要注意防止bigkey过期时间自动删除问题(例如一个200万的zset设置1小时过期, 会触发del操作, 造成阻塞, 而且该操作不会不出现在慢查询中(latency可查)), 查找方法和删除方法

选择适合的数据类型

例如: 实体类型(要合理控制和使用数据结构内存编码优化配置,例如ziplist, 但也要注意节省内存和性能之间的平衡)

反例:

```
set user:1:name tom
set user:1:age 19
set user:1:favor football
```

正例:

```
hmset user:1 name tom age 19 favor football
```

控制key的生命周期

redis不是垃圾桶, 建议使用expire设置过期时间(条件允许可以打散过期时间, 防止集中过期), 不过期的数据重点关注idle time。

二、命令使用

1、O(N)命令关注N的数量

例如hgetall、lrange、smembers、zrange、sinter等并非不能使用, 但是需要明确N的值。有遍历的需求可以使用hscan、sscan、zscan代替。

2、禁用命令

禁止线上使用keys、flushall、flushdb等，通过redis的rename机制禁掉命令，或者使用scan的方式渐进式处理。

3、合理使用select

redis的多数据库较弱，使用数字进行区分，很多客户端支持较差，同时多业务用多数据库实际还是单线程处理，会有干扰。

4、使用批量操作提高效率

1.原生命令：例如mget、mset。

2.非原生命令：可以使用pipeline提高效率。

但要注意控制一次批量操作的**元素个数**(例如500以内，实际也和元素字节数有关)。

注意两者不同：

1.原生是原子操作，pipeline是非原子操作。

2.pipeline可以打包不同的命令，原生做不到

3.pipeline需要客户端和服务端同时支持。

5、不建议过多使用Redis事务功能

Redis的事务功能较弱(不支持回滚)，而且集群版本(自研和官方)要求一次事务操作的key必须在一个slot上(可以使用hashtag功能解决)

6、Redis集群版本在使用Lua上有特殊要求

1、所有key都应该由 KEYS 数组来传递，redis.call/pcall 里面调用的redis命令，key的位置，必须是KEYS array, 否则直接返回error, "-ERR bad lua script for redis cluster, all the keys that the script uses should be passed using the KEYS array\r\n"

2、所有key，必须在1个slot上，否则直接返回error, "-ERR eval/evalsha command keys must in same slot\r\n"

7、monitor命令

必要情况下使用monitor命令时，要注意不要长时间使用。

三、客户端使用

1、避免多个应用使用一个Redis实例

不相干的业务拆分，公共数据做服务化。

2、使用连接池

可以有效控制连接，同时提高效率，标准使用方式：

执行命令如下：

```
Jedis jedis = null;
try {
    jedis = jedisPool.getResource();
    //具体的命令
    jedis.executeCommand()
} catch (Exception e) {
    logger.error("op key {} error: " + e.getMessage(), key, e);
} finally {
    //注意这里不是关闭连接，在JedisPool模式下，Jedis会被归还给资源池。
    if (jedis != null)
        jedis.close();
}
```

3、熔断功能

高并发下建议客户端添加熔断功能(例如netflix hystrix)

4、合理的加密

设置合理的密码，如有必要可以使用SSL加密访问（阿里云Redis支持）

5、淘汰策略

根据自身业务类型，选好maxmemory-policy(最大内存淘汰策略)，设置好过期时间。

默认策略是volatile-lru，即超过最大内存后，在过期键中使用lru算法进行key的剔除，保证不过期数据不被删除，但是可能会出现OOM问题。

其他策略如下：

- allkeys-lru：根据LRU算法删除键，不管数据有没有设置超时属性，直到腾出足够空间为止。
- allkeys-random：随机删除所有键，直到腾出足够空间为止。
- volatile-random:随机删除过期键，直到腾出足够空间为止。
- volatile-ttl：根据键值对象的ttl属性，删除最近将要过期数据。如果没有，回退到noeviction策略。
- noeviction：不会剔除任何数据，拒绝所有写入操作并返回客户端错误信息"(error) OOM command not allowed when used memory"，此时Redis只响应读操作。

四、相关工具

1、数据同步

redis间数据同步可以使用：redis-port

2、big key搜索

redis大key搜索工具

3、热点key寻找

内部实现使用monitor，所以建议短时间使用facebook的redis-faina

阿里云Redis已经在内核层面解决热点key问题

五、删除bigkey

1.下面操作可以使用pipeline加速。

2.redis 4.0已经支持key的异步删除，欢迎使用。

1、Hash删除: hscan + hdel

```
public void delBigHash(String host, int port, String password, String bigHashKey) {
    Jedis jedis = new Jedis(host, port);
    if (password != null && !"".equals(password)) {
        jedis.auth(password);
    }
    ScanParams scanParams = new ScanParams().count(100);
    String cursor = "0";
    do {
        ScanResult<Entry<String, String>> scanResult = jedis.hscan(bigHashKey, cursor, scanParams);
        List<Entry<String, String>> entryList = scanResult.getResult();
        if (entryList != null && !entryList.isEmpty()) {
            for (Entry<String, String> entry : entryList) {
                jedis.hdel(bigHashKey, entry.getKey());
            }
        }
        cursor = scanResult.getStringCursor();
    } while (!"0".equals(cursor));

    //删除bigkey
    jedis.del(bigHashKey);
}
```

2、List删除: ltrim

```
public void delBigList(String host, int port, String password, String bigListKey) {
    Jedis jedis = new Jedis(host, port);
    if (password != null && !"".equals(password)) {
        jedis.auth(password);
    }
    long llen = jedis.llen(bigListKey);
    int counter = 0;
    int left = 100;
    while (counter < llen) {
        //每次从左侧截掉100个
        jedis.ltrim(bigListKey, left, llen);
        counter += left;
    }
    //最终删除key
    jedis.del(bigListKey);
}
```

3、Set删除: sscan + srem

```

public void delBigSet(String host, int port, String password, String bigSetKey) {
    Jedis jedis = new Jedis(host, port);
    if (password != null && !"".equals(password)) {
        jedis.auth(password);
    }
    ScanParams scanParams = new ScanParams().count(100);
    String cursor = "0";
    do {
        ScanResult<String> scanResult = jedis.sscan(bigSetKey, cursor, scanParams);
        List<String> memberList = scanResult.getResult();
        if (memberList != null && !memberList.isEmpty()) {
            for (String member : memberList) {
                jedis.srem(bigSetKey, member);
            }
        }
        cursor = scanResult.getStringCursor();
    } while (!"0".equals(cursor));

    // 删除 bigkey
    jedis.del(bigSetKey);
}

```

4、SortedSet删除: zscan + zrem

```

public void delBigZset(String host, int port, String password, String bigZsetKey) {
    Jedis jedis = new Jedis(host, port);
    if (password != null && !"".equals(password)) {
        jedis.auth(password);
    }
    ScanParams scanParams = new ScanParams().count(100);
    String cursor = "0";
    do {
        ScanResult<Tuple> scanResult = jedis.zscan(bigZsetKey, cursor, scanParams);
        List<Tuple> tupleList = scanResult.getResult();
        if (tupleList != null && !tupleList.isEmpty()) {
            for (Tuple tuple : tupleList) {
                jedis.zrem(bigZsetKey, tuple.getElement());
            }
        }
        cursor = scanResult.getStringCursor();
    } while (!"0".equals(cursor));

    // 删除 bigkey
    jedis.del(bigZsetKey);
}

```