# 课前准备

- 准备redis安装包

# 课堂主题

Redis和lua整合、Redis消息模式、Redis实现分布式锁、缓存穿透、缓存雪崩、缓存击穿、缓存双写一致性

# 课堂目标

- 理解lua概念，能够使用Redis和lua整合使用
- 理解redis消息原理
- 掌握redis分布式锁的原理、本质
- 掌握Redisson的原理和实现
- 理解缓存穿透、缓存雪崩、缓存击穿、缓存双写一致性并掌握解决方案

# 知识要点

# Redis和lua整合

## 什么是lua

lua是一种轻量小巧的**脚本语言**，用标准**C语言**编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

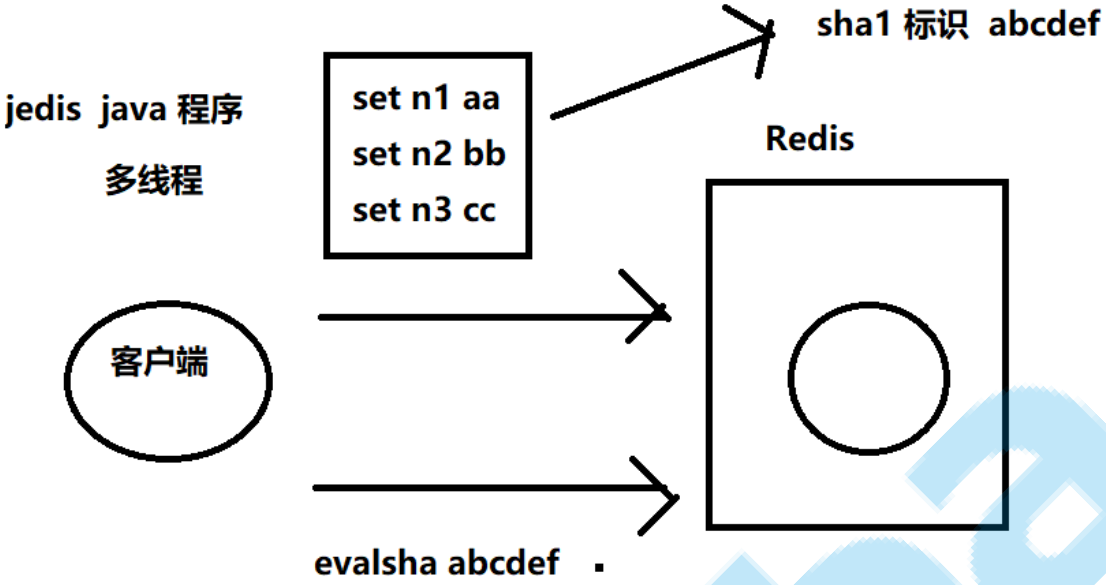
## Redis中使用lua的好处

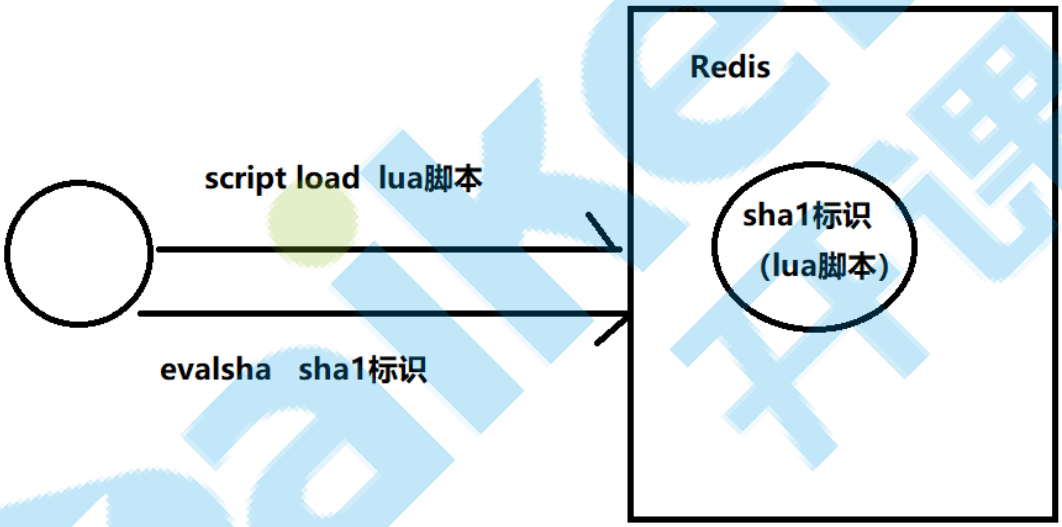
1、减少网络开销，在Lua脚本中可以把多个命令放在同一个脚本中运行

2、原子操作，redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。换句话说，编写脚本的过程中无需担心会出现竞态条件

3、复用性，客户端发送的脚本会永远存储在redis中，这意味着其他客户端可以复用这一脚本来完成同样的逻辑

在内存中生成一个sha1 标识 （script load ）




## lua的安装（了解）

- 下载

  地址：http://www.lua.org/download.html

  可以本地下载上传到linux，也可以使用curl命令在linux系统中进行在线下载

  ```
  curl -R -O http://www.lua.org/ftp/lua-5.3.5.tar.gz
  ```

- 安装

```
yum -y install readline-devel ncurses-devel
tar -zxvf lua-5.3.5.tar.gz
make linux
make install
```

如果报错，说找不到readline/readline.h, 可以通过yum命令安装

```
yum -y install readline-devel ncurses-devel
```

安装完以后再

```
make linux  / make install
```

最后，直接输入 lua命令即可进入lua的控制台

# lua常见语法（了解）

详见http://www.runoob.com/lua/lua-tutorial.html

# Redis整合lua脚本

从Redis2.6.0版本开始，通过**内置的lua编译/解释器**，可以使用EVAL命令对lua脚本进行求值。

## EVAL命令

通过执行redis的eval命令，可以运行一段lua脚本。

```
EVAL script numkeys key [key ...] arg [arg ...]
```

命令说明：**

- **script参数：**是一段Lua脚本程序，它会被运行在Redis服务器上下文中，这段脚本不必(也不应该)定义为一个Lua函数。
- **numkeys参数：**用于指定键名参数的个数。
- **key [key ...]参数：** 从EVAL的第三个参数开始算起，使用了numkeys个键（key），表示在脚本中所用到的那些Redis键(key)，这些键名参数可以在Lua中通过全局变量**KEYS**数组，用1为基址的形式访问( KEYS[1]， KEYS[2]，以此类推)。
- **arg [arg ...]参数：**可以在Lua中通过全局变量**ARGV**数组访问，访问的形式和KEYS变量类似( ARGV[1]、 ARGV[2]，诸如此类)。

```
eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
```

## lua脚本中调用Redis命令

- redis.call()：
  - 返回值就是redis命令执行的返回值
  - 如果出错，则返回错误信息，不继续执行
- **redis.pcall()：**
  - 返回值就是redis命令执行的返回值
  - 如果出错，则记录错误信息，继续执行
- **注意事项**
  - 在脚本中，使用return语句将返回值返回给客户端，如果没有return，则返回nil

```
eval "return redis.call('set',KEYS[1],ARGV[1])" 1 n1 zhaoyun
```

## SCRIPT命令

- **SCRIPT FLUSH：** 清除所有脚本缓存
- **SCRIPT EXISTS：** 根据给定的脚本校验和，检查指定的脚本是否存在于脚本缓存
- **SCRIPT LOAD：** 将一个脚本装入脚本缓存，**返回SHA1摘要**，但并不立即运行它

```
192.168.24.131:6380> script load "return redis.call('set',KEYS[1],ARGV[1])"
"c686f316aaf1eb01d5a4de1b0b63cd233010e63d"
192.168.24.131:6380> evalsha c686f316aaf1eb01d5a4de1b0b63cd233010e63d 1 n2
zhangfei
OK
192.168.24.131:6380> get n2
```

- **SCRIPT KILL：** 杀死当前正在运行的脚本

## EVALSHA

EVAL 命令要求你在每次执行脚本的时候都发送一次脚本主体(script body)。

Redis 有一个内部的缓存机制，因此它不会每次都重新编译脚本，不过在很多场合，付出无谓的带宽来传送脚本主体并不是最佳选择。

为了减少带宽的消耗， Redis 实现了 EVALSHA 命令，它的作用和 EVAL 一样，都用于对脚本求值，但它接受的第一个参数不是脚本，而是脚本的 SHA1 校验和(sum)

## redis-cli --eval

直接执行lua脚本

test.lua

```
return redis.call('set',KEYS[1],ARGV[1])

./redis-cli -h 192.168.24.131 -p 6380  --eval test.lua n3 , 'liubei'
```

list.lua

```
local key=KEYS[1]

local list=redis.call("lrange",key,0,-1);

return list;

./redis-cli --eval list.lua list
```

利用Redis整合Lua，主要是为了性能以及事务的原子性。因为redis帮我们提供的事务功能太差。

# Redis消息模式

## 队列模式

MQ主要是用来：

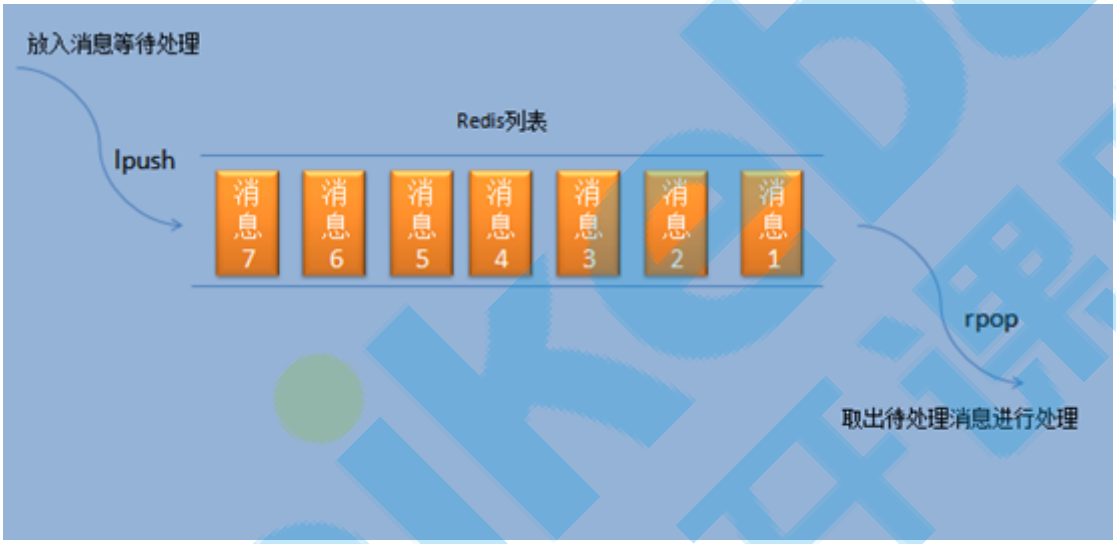
- 解耦应用
- 异步化消息
- 流量削峰填谷

典型的消息服务是一个生产者和消费者模式的服务。一般是有生产者产生消息，将消息发送到队列中。而消息的消费者则监听消息，对消息进行处理。

有很多非常优秀的消息队列服务的产品。例如 RabbitMQ、RocketMQ、Kafka 等。这些产品都具备非常高级的功能。可靠性、扩展性都非常的好。

但是 redis 自身也能够很简单的实现消息队列的生产者和消费者模式。

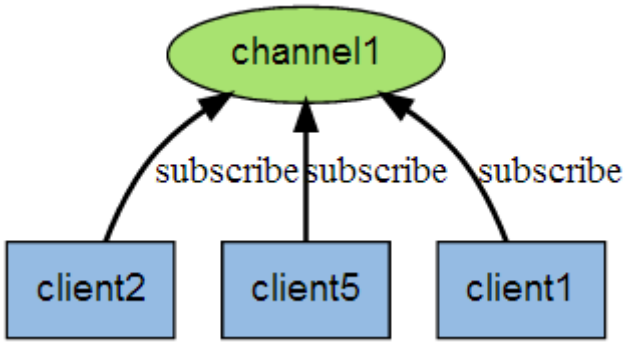
使用list类型的**lpush和rpop**实现消息队列

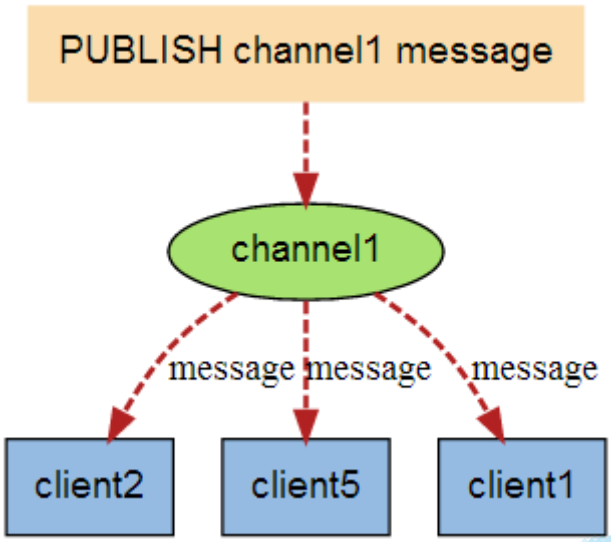



**注意事项：**

- 消息接收方如果不知道队列中是否有消息，会一直发送rpop命令，如果这样的话，会每一次都建立一次连接，这样显然不好。
- 可以使用**brpop**命令，它如果从队列中取不出来数据，会一直阻塞，在一定范围内没有取出则返回null、

## 发布订阅模式






# 即时消费

## 生产消费者关系为一对一

A的任务由B执行

使用Redis中list的操作BLPOP或BRPOP，即列表的阻塞式(blocking)弹出。

```
BRPOP key [key ...] timeout
```

此命令的说明是：

> 1、当给定列表内没有任何元素可供弹出的时候，连接将被 BRPOP 命令阻塞，直到等待超时或发现可弹出元素为止。 2、当给定多个key参数时，按参数 key 的先后顺序依次检查各个列表，弹出第一个非空列表的尾部元素。

另外，BRPOP 除了弹出元素的位置和 BLPOP 不同之外，其他表现一致。

以此来看，**列表的阻塞式弹出有两个特点：**

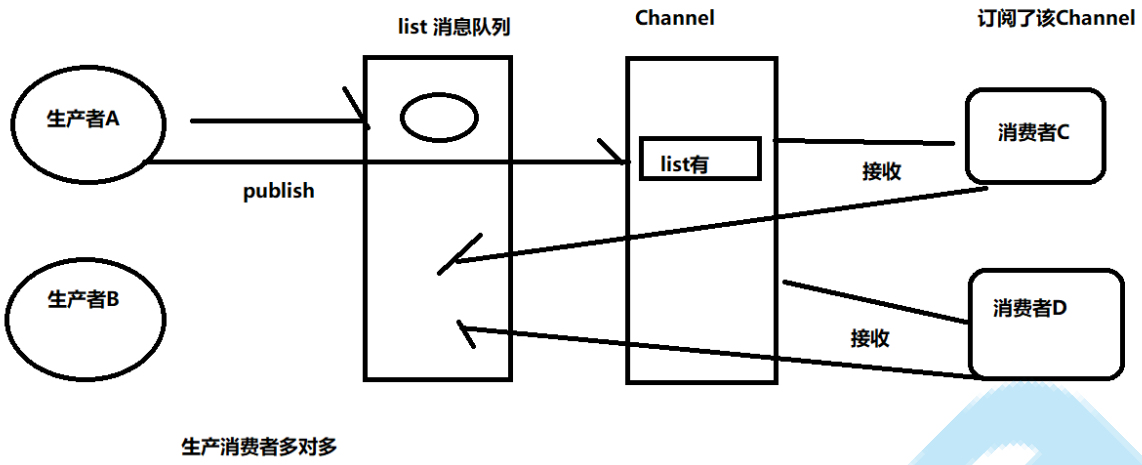
> 1、如果list中没有任务的时候，该连接将会被阻塞 2、连接的阻塞有一个超时时间，当超时时间设置为0时，即可无限等待，直到弹出消息

## 生产消费者关系为多对多

A和Z的任务，B和C都能执行

使用订阅/发布模式

在消息A入队list的同时发布（PUBLISH）消息B到频道channel，此时已经订阅channel的worker就接收到了消息B，知道了list中有消息A进入，即可循环lpop或rpop来消费list中的消息。




## 实现ACK机制

**ack**，即消息确认机制(Acknowledge)

用Redis实现消息队列的ack机制

1、work处理失败后，要回滚消息到原始队列

2、假如worker挂掉，也要回滚消息到原始队列

## 实现方案

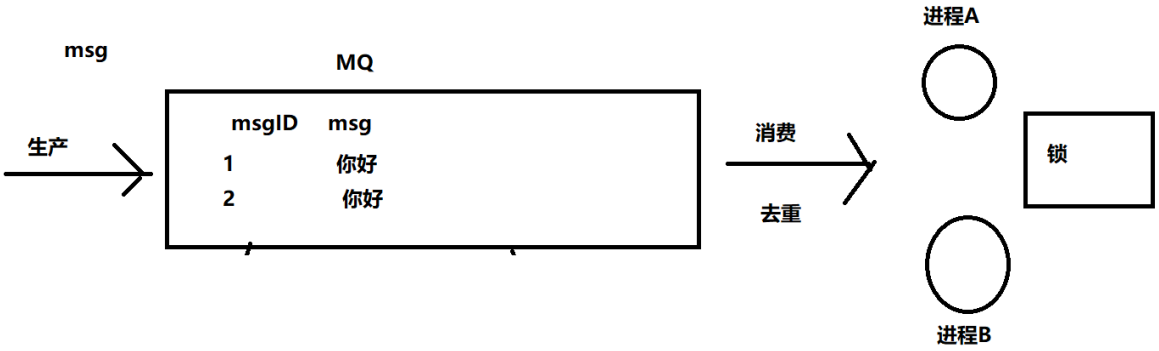
1. 维护两个队列：pending队列和doing表（hash表）。
2. workers定义为ThreadPool。
3. 由pending队列出队后，workers分配一个线程（单个worker）去处理消息——给目标消息append一个当前时间戳和当前线程名称，将其写入doing表，然后该worker去消费消息，完成后自行在doing表擦除信息。
4. 启用一个定时任务，每隔一段时间去扫描doing队列，检查每隔元素的时间戳，如果超时，则由worker的ThreadPoolExecutor去检查线程是否存在，如果存在则取消当前任务执行，并把事务rollback。最后把该任务从doing队列中pop出，再重新push进pending队列。
5. 在worker的某线程中，如果处理业务失败，则主动回滚，并把任务从doing队列中移除，重新push进pending队列。

# Redis实现分布式锁

## 业务场景

1、防止用户重复下单

2、MQ消息去重




3、订单操作变更

4、库存超卖

。。。。

分析：

业务场景共性：

共享资源

用户id、订单id、商品id。。。

解决方案

共享资源互斥

共享资源串行化

问题转化

锁的问题 （将需求抽象后得到问题的本质）

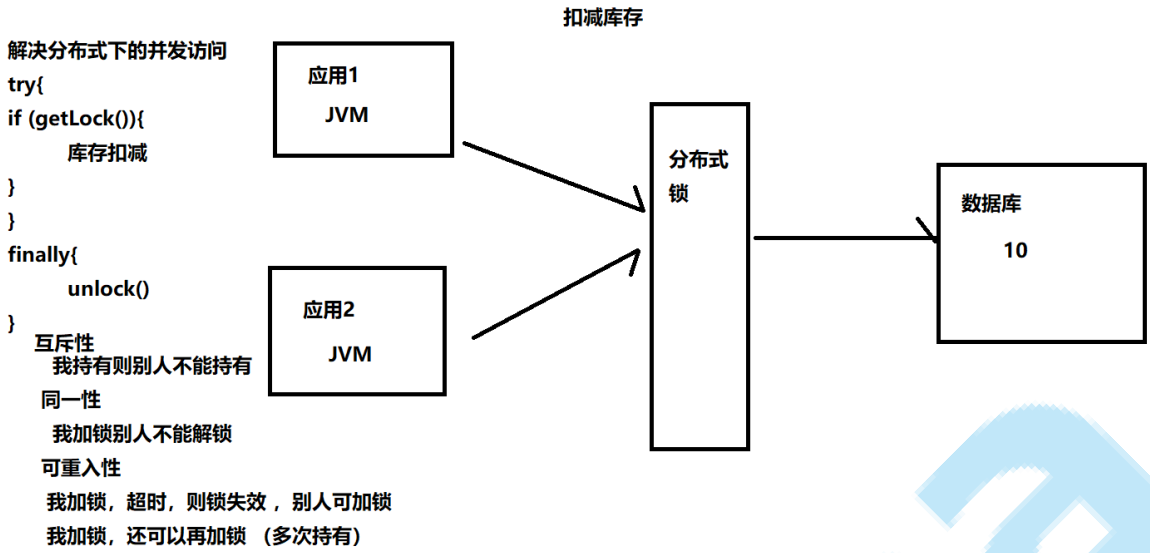
# 锁的处理

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。




# Redis实现分布式锁

## 原理

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
        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 失效时间
            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);
        }
    }
```

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;
}
```

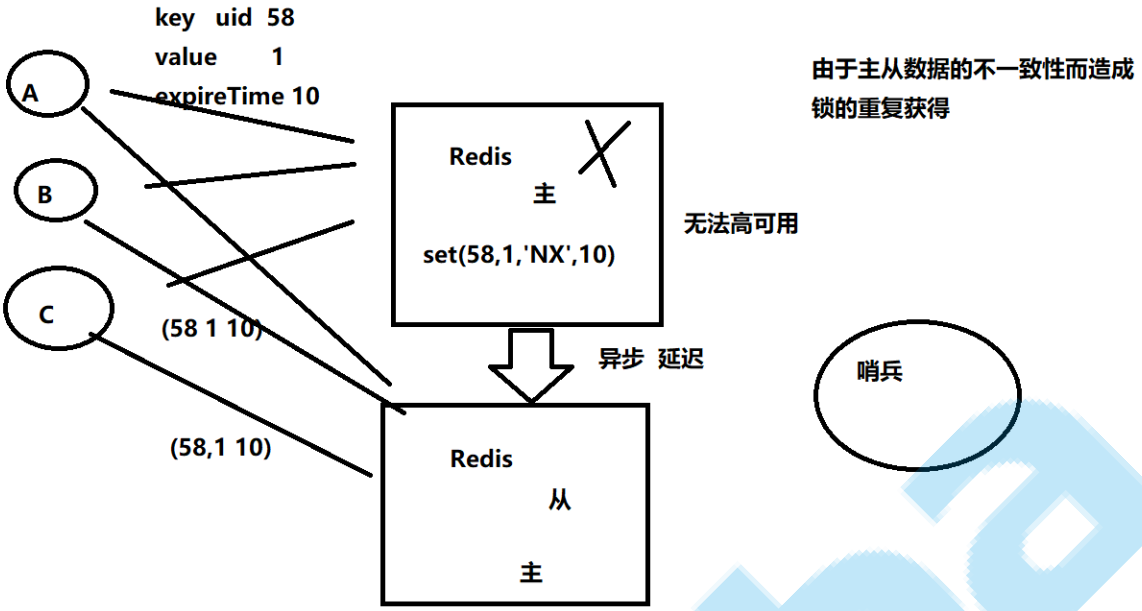
## 存在问题

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

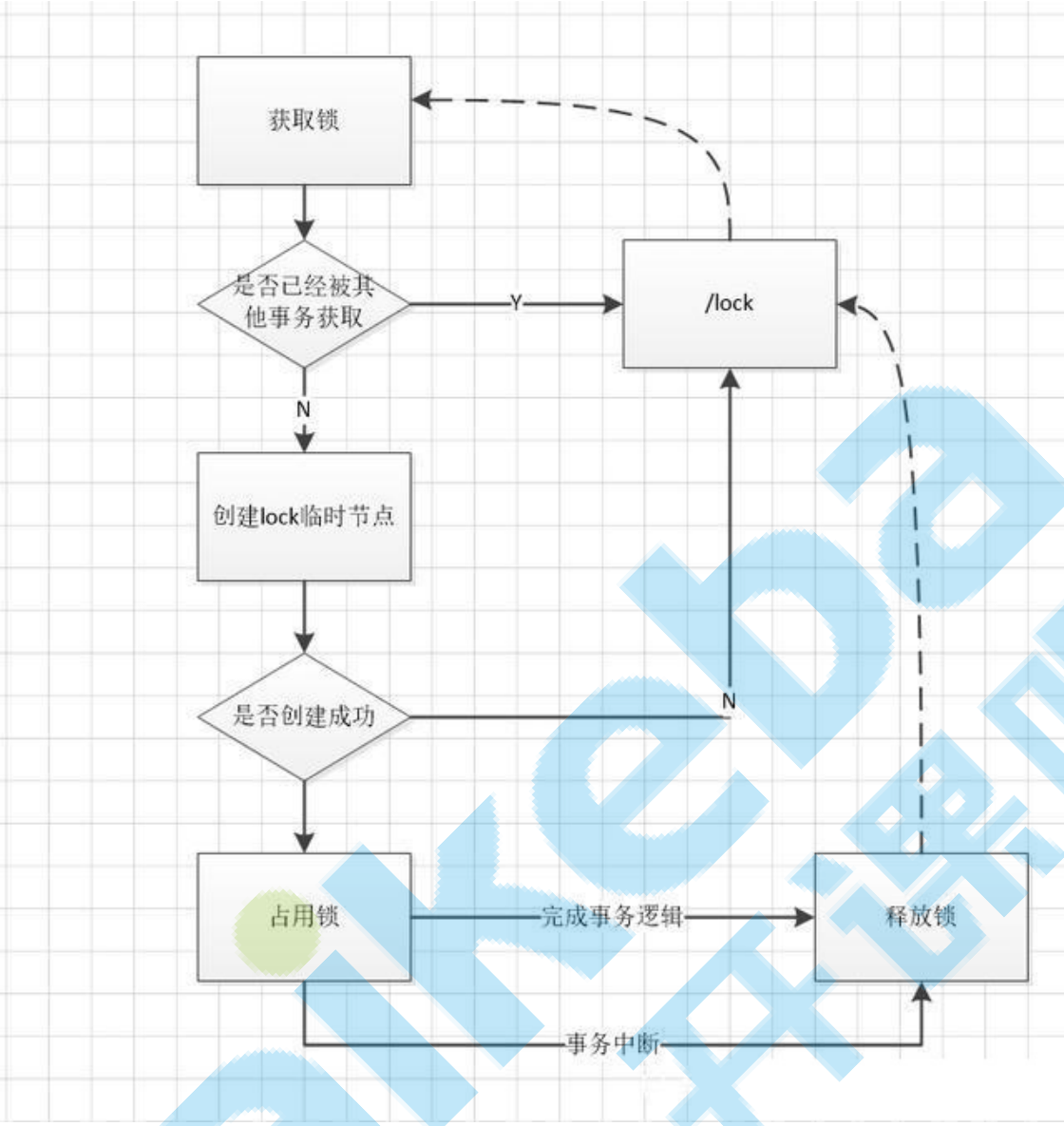
当业务不需要数据强一致性时，比如：社交场景，就可以使用Redis实现分布式锁

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。

# 分布式锁的实现方式

- 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 |

# 生产环境中的分布式锁

落地生产环境用分布式锁，一般采用开源框架，比如Redisson。下面来讲一下Redisson对Redis分布式锁的实现。

## Redisson分布式锁的使用

### 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
```

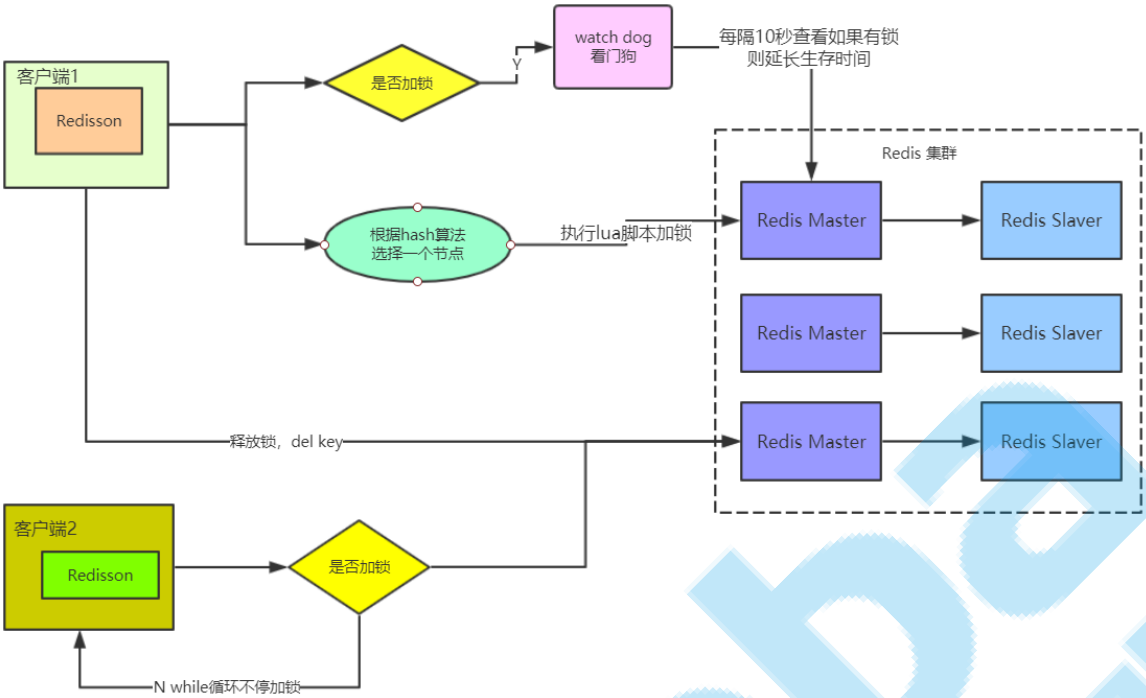
```
        String key = LOCK_TITLE + lockName;
       //获取锁对象
        RLock mylock = redisson.getLock(key);
       //加锁，并且设置锁过期时间3秒，防止死锁的产生    uuid+threadId
        mylock.lock(2,3,TimeUtil.SECOND);
       //加锁成功
        return  true;
    }
  //锁的释放
    public static void release(String lockName){
       //必须是和加锁时的同一个key
        String key = LOCK_TITLE + lockName;
       //获取所对象
        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 = "test123";
        //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 "+
    "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]) ;"
```

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-6c71a6b4586:1 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。
                    "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]); " +
                        "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_lock**channel**{” + 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。

# 常见缓存问题

## 数据读

### 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）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造成很大的压力。

也就是说，对不存在的key进行高并发访问，导致数据库压力瞬间增大，这就叫做【缓存穿透】。

解决方案：

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

### 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

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

3、高可用

## 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。

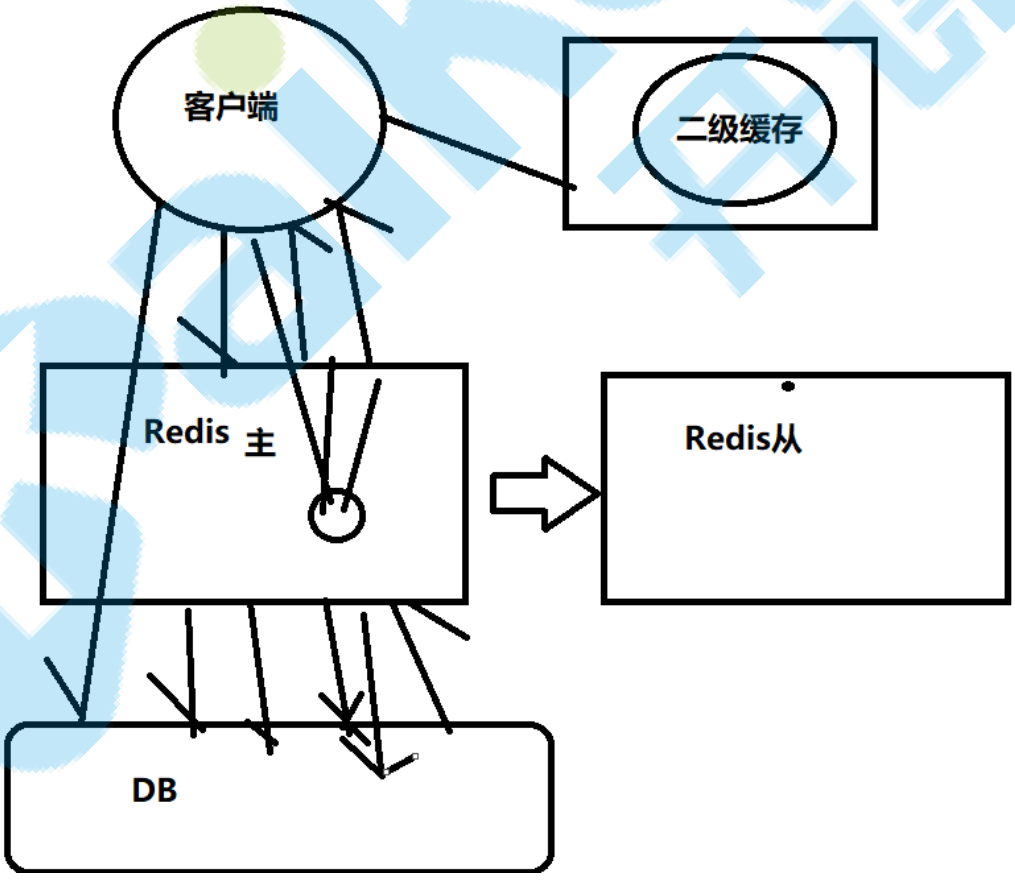
解决方案：

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。

if(redis.sexnx()==1){ //先查询缓存 //查询数据库 //加入缓存 }

不设超时时间，写一致问题




## 数据写

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

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

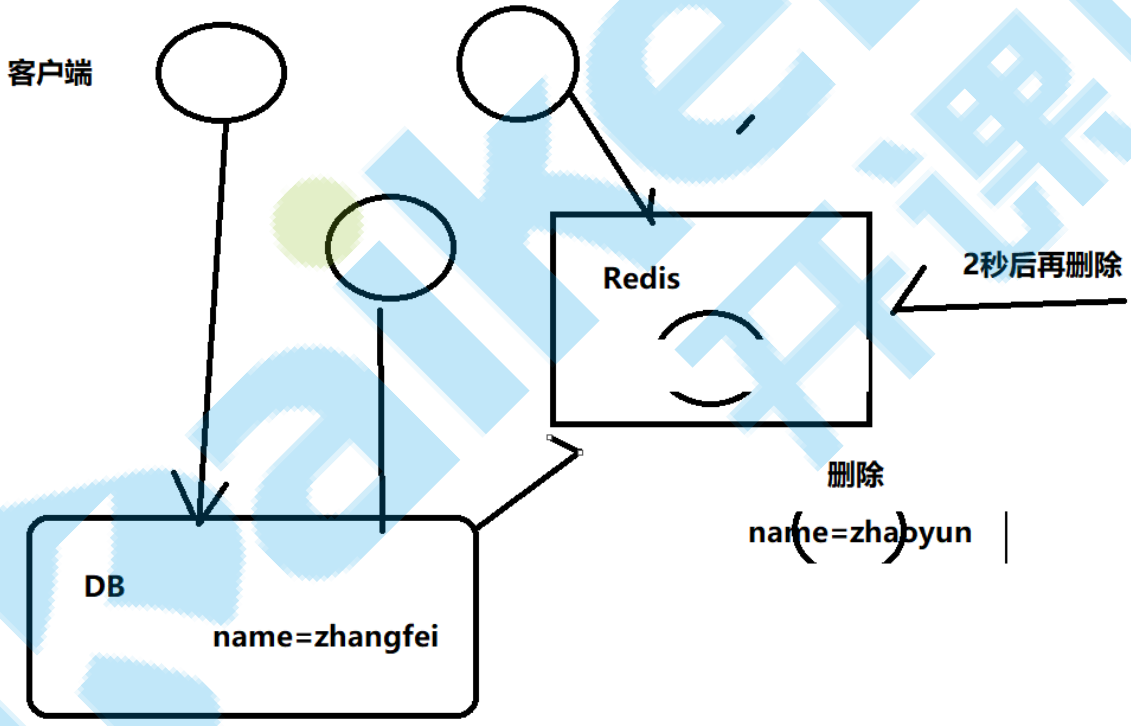
## 保证数据的最终一致性(延时双删)

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。