Redis数据结构

1、Memcache 和 Redis 的区别

选Memcache理由

1）系统业务以**KV的缓存为主，数据量、并发业务量大**，memcache较为合适

2）memcache将所有数据存储在**物理内存**中。Redis则有**自己的VM机制**，当数据超量时，会引发swap，影响计算机服务器性能

3）memchache使用**多线程的模式**（主线程监听，work子线程工作），而Redis使用**单线程**，难以充分利用目前的多核CPU，我要求的是快快快，压榨光每一个资源的性能

选Redis的理由

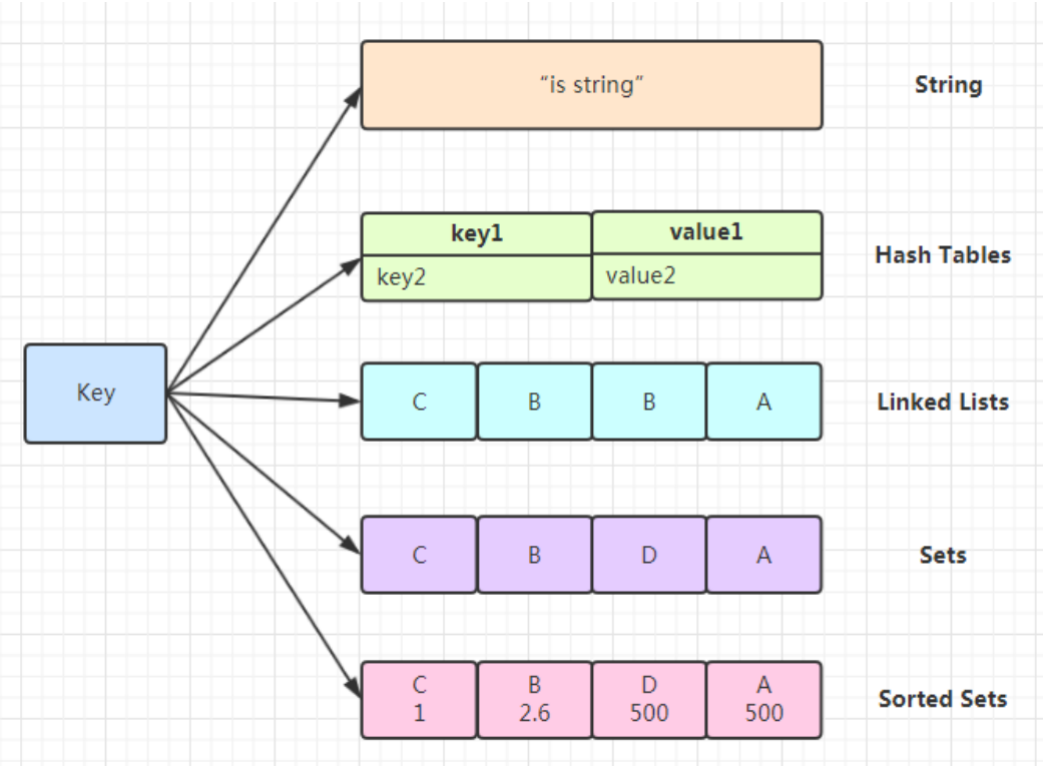
1）Redis天然高可用，官方提供了sentinel集群管理工具，释放了我们大量的工作内容

2）Redis能存储内容较Memcache的1M要大多了

3）系统不仅仅只用到KV，我们需要用到**Redis丰富的数据结构以及相关功能函数**，Memcache不适合我的系统

2、Redis 的数据结构

总览



对于redis来说，所有的Key都是字符串。

1）String

最基本的数据类型，一个key对应一个value。String类型是二进制安全的，意思是 redis 的 string 可以包含任何数据。如**数字，字符串，jpg图片或者序列化的对象。**

string使用场景一般是存储**简单的键值类型。**比如用户信息，登录信息，配置信息等。消息系统用于缓存消息内容，key 是消息的唯一 id，采用数据库号段的方式生成，value 是消息内容的 json 串。还有一种用得比较多的是string的incr/decr操作，即自减/自增操作，可以配合分布式锁来生成分布式id。调用它是原子性的，无论调用多少次，都一一计算成功。例如需要增减库存的操作。

尽管string的value可以存储很大，甚至500多MB的容量。但是在性能上来说，我们尽量存储value的值不要过1MB。

【基本操作】

使用：get 、 set 、 del 、 incr、 decr 等

【应用】

- **缓存**

  经典使用场景，把常用信息，字符串，图片或者视频等信息放到redis中，redis作为缓存层，mysql做持久化层，降低mysql的读写压力。

- **实现计数器** INCR key //INCR readcount::{帖子ID}　每阅读一次 GET key　//GET readcount::{帖子ID}　获取阅读量、访问量

- **session共享**

  用户重新刷新一次界面，可能需要访问一下数据进行重新登录，或者访问页面缓存**Cookie**，但是可以利用**Redis**将用户的**Session**集中管理，在这种模式只需要保证**Redis**的高可用，每次用户**Session**的更新和获取都可以快速完成。大大提高效率。

- **实现分布式锁**

  原子的设置key的值，并返回key的旧值。如果key不存在返回nil。应用场景：设置新值，返回旧值，配合setnx可实现分布式锁。

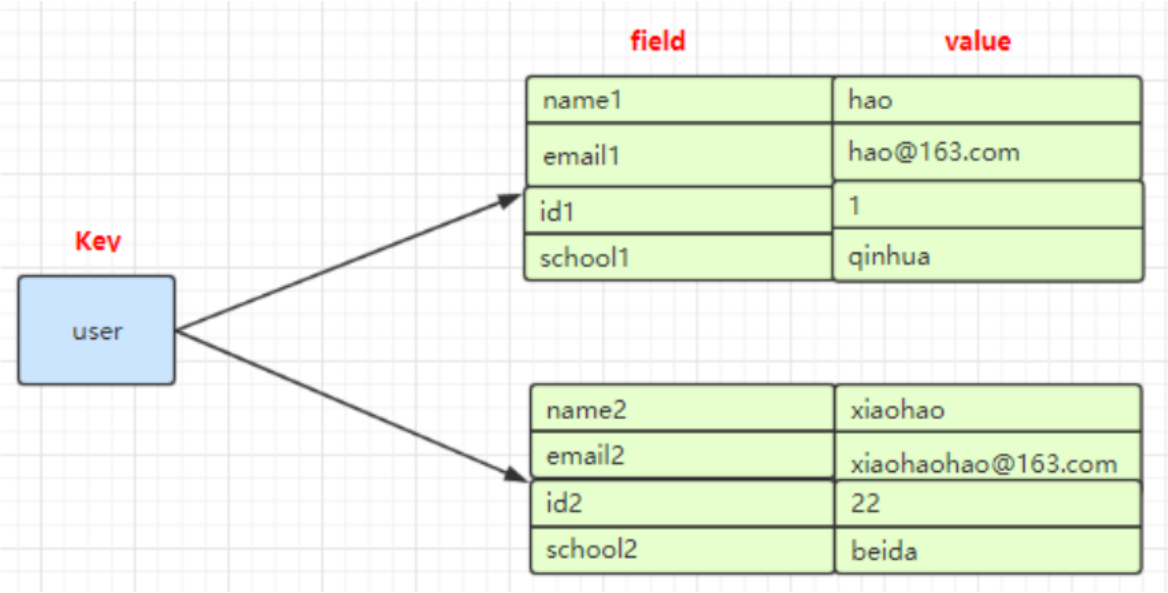
分布式锁的思路：注意该思路要保证多台Client服务器的NTP一致。

1. C3发送**SETNX** lock.foo 想要获得锁，由于C0还持有锁，所以Redis返回给C3一个0
2. C3发送**GET** lock.foo 以检查锁是否超时了，如果没超时，则等待或重试。
3. 反之，如果已超时，C3通过下面的操作来尝试获得锁：
4. **GETSET** lock.foo
5. 通过GETSET，C3拿到的时间戳如果仍然是超时的，那就说明，C3如愿以偿拿到锁了。
6. 如果在C3之前，有个叫C4的客户端比C3快一步执行了上面的操作，那么C3拿到的时间戳是个未超时的值，这时，C3没有如期获得锁，需要再次等待或重试。留意一下，尽管C3没拿到锁，但它改写了C4设置的锁的超时值，不过这一点非常微小的误差带来的影响可以忽略不计。

2）hash

是一个map，可以存储结构化的信息，而不是直接分散存储在整个Redis中。可以使你像在数据库中Update 一个属性一样只修改某一项属性值，而不用先序列化后再修改。可以减少内存占用。

需要注意的是，经过测试，在性能上来说一般hash里面的第二层key，不要超过200个为佳。尽管hash里面的key-value能达到500多MB的存储容量。



【基本操作】

使用：所有hash的命令都是 h 开头的 hget 、hset 、 hdel 等

```
127.0.0.1:6379> hset user name1 hao
(integer) 1
127.0.0.1:6379> hset user email1 hao@163.com
(integer) 1
127.0.0.1:6379> hgetall user
1) "name1"
2) "hao"
3) "email1"
4) "hao@163.com"
127.0.0.1:6379> hget user user
(nil)
127.0.0.1:6379> hget user name1
"hao"
127.0.0.1:6379> hset user name2 xiaohao
(integer) 1
127.0.0.1:6379> hset user email2 xiaohao@163.com
(integer) 1
127.0.0.1:6379> hgetall user
1) "name1"
2) "hao"
3) "email1"
4) "hao@163.com"
5) "name2"
6) "xiaohao"
7) "email2"
8) "xiaohao@163.com"
```

【应用】

- **实现秒杀服务中的购物车**

以用户id为key，商品id为field，商品数量为value，恰好构成了购物车的3个要素，如下图所示。

【String 还是 hash】

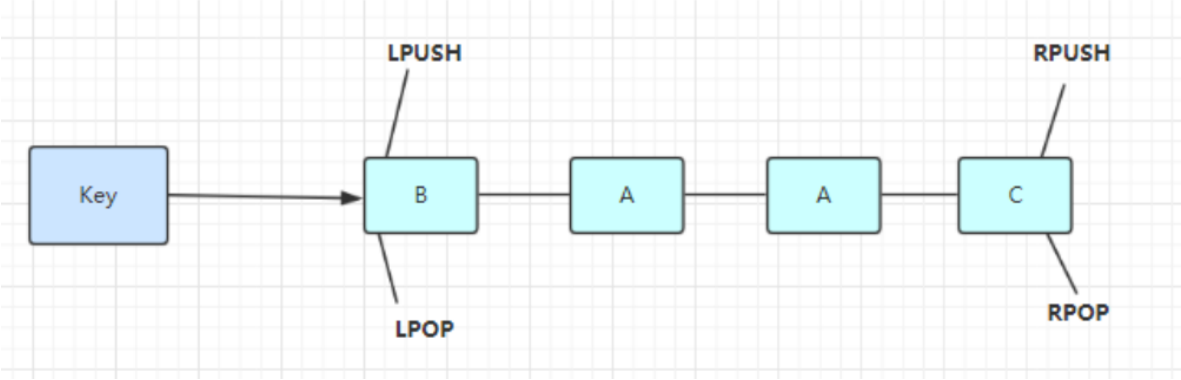
	string + json	hash
效率	很高	高
容量	低	低
灵活性	**低**	高
序列化	简单	**复杂**

当对象的某个属性需要频繁修改时，不适合用string+json，因为它不够灵活，每次修改都需要重新将整个对象序列化并赋值，如果使用hash类型，则可以针对某个属性单独修改，没有序列化，也不需要修改整个对象。比如，商品的价格、销量、关注数、评价数等可能经常发生变化的属性，就适合存储在hash类型里。

ps：短信平台在发送短信时，会先获取渠道的配置信息（账号、密码、发送的api地址等）。具体为先获取缓存（redis实现），该缓存基于redis的string实现，将配置信息（结构化的数据）转为jsonStr后存入。因为考虑到渠道的配置信息一般很固定，不会频繁修改，所以不用hash实现，降低实现的复杂度。

3）list

List 说白了就是链表（redis 使用双端链表实现的 List），是有序的，value可以重复，可以通过下标取出对应的value值，左右两边都能进行插入和删除数据。



【基本操作】

- **单个左推：**LPUSH key value
- **批量左推：**LPUSH key value1 value2...
- **单个右推：**RPUSH key value
- **单个左弹：**LPOP key
- **单个右弹：**RPOP key
- **根据范围取值：**LRANGE key start stop　eg：LRANGE key 0 -1 表示拿取所有数据

需要注意的是尽管redis可以使用推拉的队列模式，但是一定要注意场景。因为redis的队列是一种轻量级别的，没有队列重试、队列重放机制。消费完队列消息在redis代表已经删除了。

【使用列表的技巧】

- lpush+lpop=Stack(栈)
- lpush+rpop=Queue（队列）
- lpush+ltrim=Capped Collection（有限集合）
- lpush+brpop=Message Queue（消息队列）

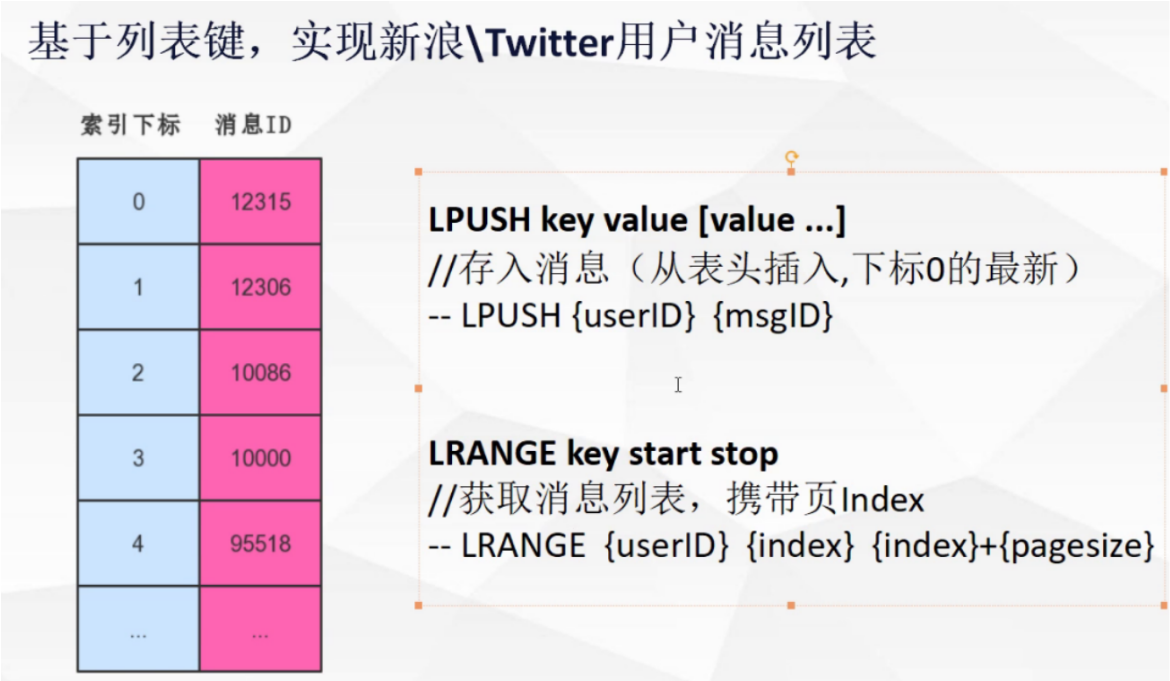
【应用】

- **获取用户的消息列表**

list 列表来缓存用户信息与消息id的关系，比如 key 就是用户名、产品名、系统名的一个组合，value 就是消息的 id。消息推送给某个用户时，首先会创建一个这样的 list 缓存，然后当客户端在用户打开消息中心时，就会带上用户的信息到消息系统来拉取消息，根据请求中的用户名、产品名、系统名为 key 来找到该用户下的所有的消息的id，再根据消息 id 去取该用户下的所有消息内容返回给客户端，同时基于

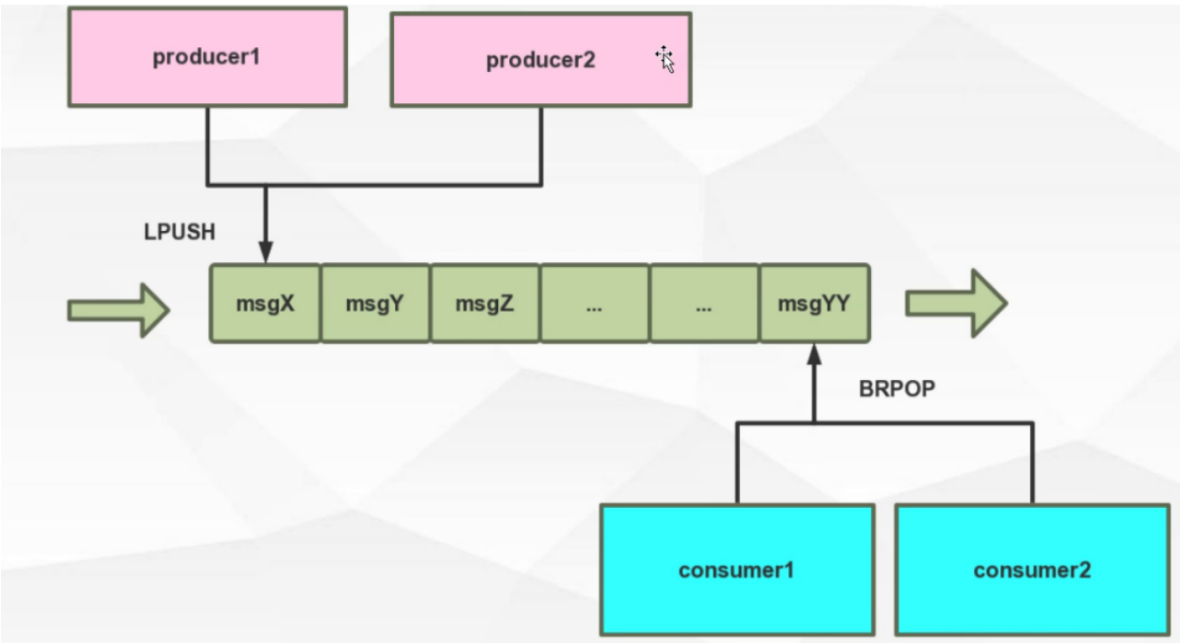
这两层缓存还实现了接口的幂等、重复拉取等拦截器的功能。

消息平台，应用登录或打开页面，需要快速获取数据并加载到消息中心，会根据用户id来取消息列表放入（过期三天）。 列表不但有序同时还支持按照范围内获取元素，可以完美解决分页查询功能。大大提高查询效率。



- **实现消息阻塞队列**

  数据的生产者可以通过**Lpush**命令从左边插入数据，多个数据消费者，可以使用**BRpop**命令阻塞的“抢”列表尾部的数据。



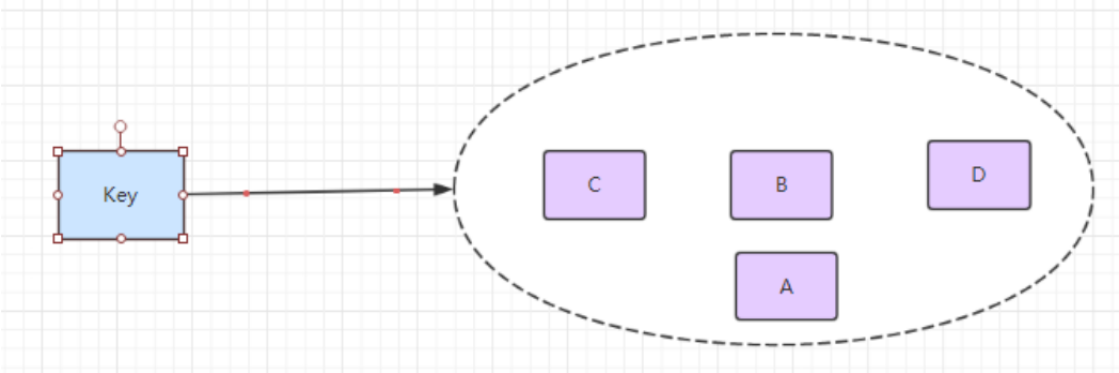
BLPOP/BRPOP 的先到先服务原则

如果有多个客户端同时因为某个列表而被阻塞，那么当有新值被推入到这个列表时，服务器会按照先到先服务（first in first service）原则，优先向最早被阻塞的客户端返回新值。举个例子，假设列表 lst 为空，那么当客户端 X 执行命令 BLPOP lst timeout 时，客户端 X 将被阻塞。在此之后，客户端 Y 也执行命令 BLPOP lst timeout ，也因此被阻塞。如果这时，客户端 Z 执行命令 RPUSH lst "hello" ，将值 "hello" 推入列表 lst ，那么这个 "hello" 将被返回给客户端 X ，而不是客户端 Y ，因为客户端 X 的被阻塞时间要早于客户端 Y 的被阻塞时间。

4）set

集合类型也是用来保存多个字符串的元素，但和列表不同的是集合中

- 不允许有重复的元素
- 集合中的元素是无序的，不能通过索引下标获取元素
- 支持集合间的操作，可以取多个集合取交集、并集、差集



【基本操作】

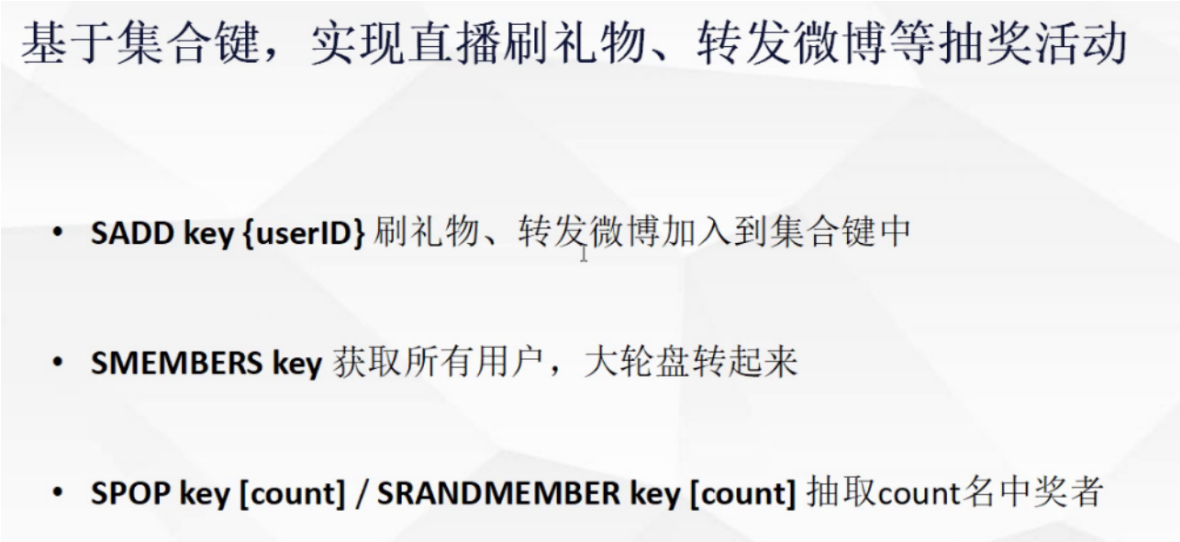
- **添加：**SADD key member [member ...]
- **删除：**SREM key member [member ...]
- **列出所有元素：**SMEMBERS key
- **求元素个数：**SCARD key
- **判断元素是否在集合里 0-不存在 1-存在：** SISMEMBER key member
- **随机弹出count个元素，默认1，保留元素在集合中：**SRANDMEMBER
- **随机弹出count个元素，并且从集合中删除：**SPOP key [count]

需要注意的是使用set一定不要存储大量的数据。value的值不宜过大，并且集合数量不宜过大。几百个集合的值，value不超过1MB为佳。

【应用】

- **实现抽奖**

  消息平台某个消息类型带有抽奖功能，点击消息时，客户端上报用户ID，然后（可以是当天某个时间）消息平台再统一进行抽奖，并返回中奖的用户ID给客户端，或者通过消息弹窗和短信的方式进行通知领奖。



- **实现点赞、签到功能**

点赞：

> **用户1给用户2点赞**
>
> **SADD 用户2的id 用户1的id**

点赞 --> SADD like::8001 1001 取消点赞 --> SREM like::8001 1001 检查用户是否点过赞 --> SISMEMBER like::8001 1001 获取点赞的用户列表 --> SMEMBERS like::8001 获取点赞用户数 --> SCARD like::8001

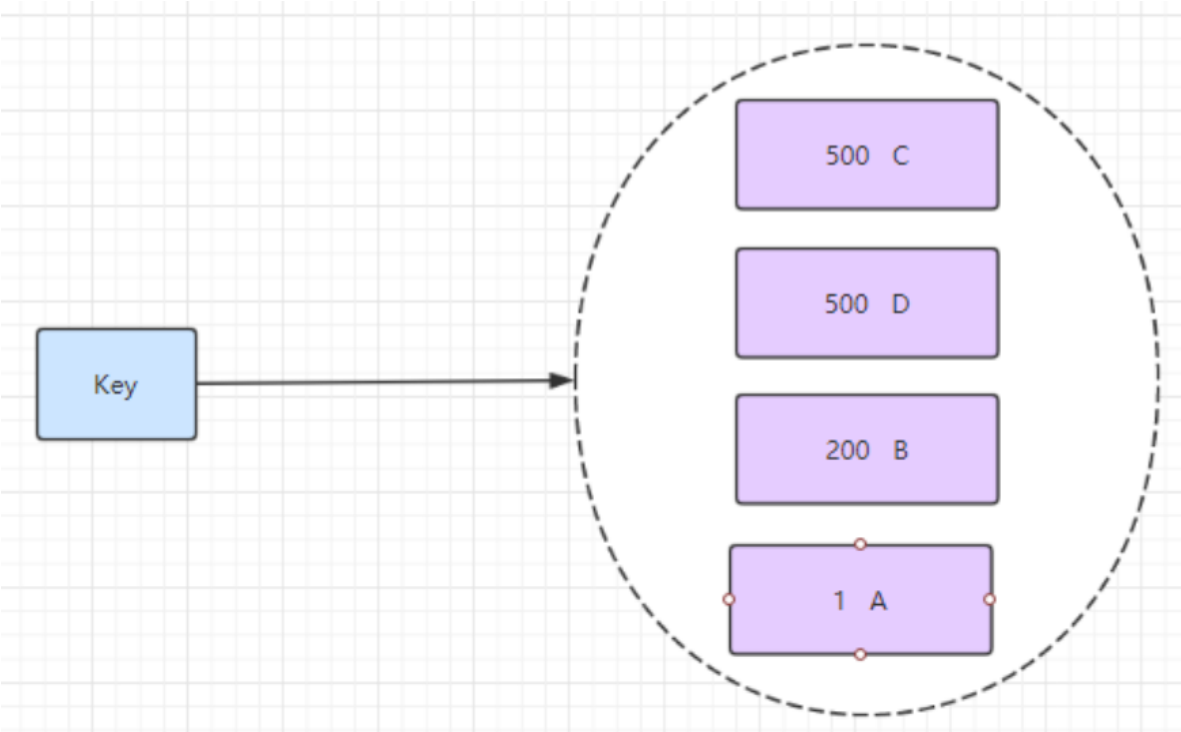
文章id、某一天日期作为key（位图）

- **实现共同好友、共同关注**

通过交集并集等集合键实现

5）sort set

有序集合和集合有着必然的联系，保留了集合不能有重复成员的特性，区别是，有序集合中的元素是可以排序的，它给每个元素设置一个分数，作为排序的依据。



【基本操作】

- **添加：**ZADD key score element [score element...]
- **删除：**ZREM key element [element...]
- **查看分值：**ZSCORE key element
- **递增：**ZINCRBY key increment element
- **元素个数：**ZCARD key
- **取出范围内元素和值 0 -1 全部：**ZRANGE key start stop [WITHSCORES]
- **反向范围：**ZREVRANGE key start stop [WITHSCORES]
- **集合运算操作** ZUNIONSTORE destkey numkeys key [key...] ZINTERSTORE destkey numkeys key [key...]

【应用】

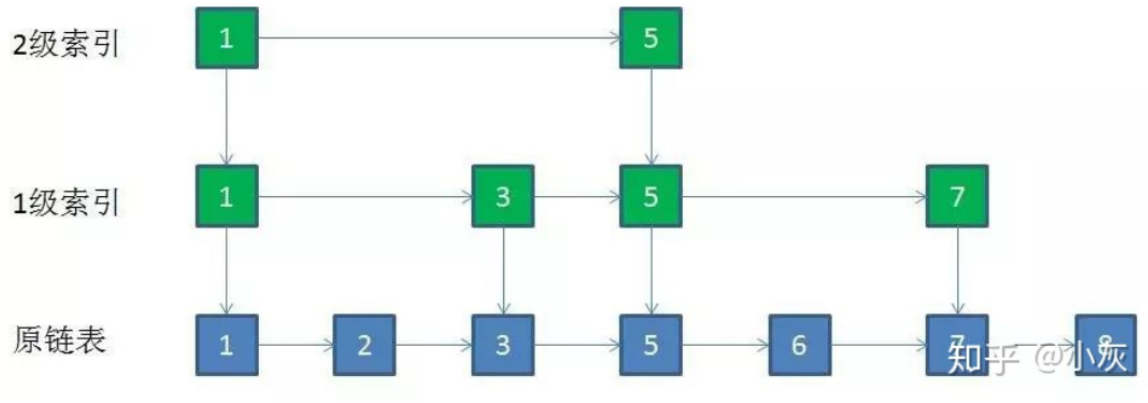
- **实现单日排行榜**（每天都不一样）key 是每一天的排行榜日期

key 是每一天的日期，score 是点击数，value 是标题

- **实现延时队列**

6）跳表

概述



跳表是为了解决在有序集合中高效插入元素的数据结构，如果没有跳表，我们要在一个链表或者数组中插入一个元素时，就需要进行一个遍历查找的过程，然后通过逐个对比元素最终找到合适的位置插入，这个时间复杂度是 O(n)。为了优化这个查找效率，通过引入多级索引，每一级索引都是从上一级的集合中抽取出一些关键字来组合，本质上用到了类似二分法的思想来构建多级索引，所以查询的时候在进行逐层索引的比较过程中，其实就是二分法的过程。所以每一层的数量都近似是上一层的一半，其查询和插入的复杂度是 O(log n)，数据结构空间占用 2n，空间复杂度 O(n)。

数据提拔

当节点插入越多，原集合元素越多时，就需要动态调整各级索引，从原集合中继续抽取部分关键元素补充到各索引层中。而由于索引的补充和删除是不可预测的，但又为了能使得索引结构更加平均均衡，所以在新增索引是用到了抛硬币的算法，这种算法在数据量充分的情况下，可以随机决定新节点是否继续向上提升，可以让决策的概率达到 50%，从而使得索引在各个层中的分布趋于均匀。

插入

- 新节点和各层索引节点逐一比较，确定原链表的插入位置。O（logN）
- 把索引插入到原链表。O（1）

- 利用抛硬币的随机方式，决定新节点是否提升为上一级索引。结果为“正”则提升并继续抛硬币，结果为“负”则停止。O（logN）

总体上，跳跃表插入操作的时间复杂度是O（logN），而这种数据结构所占空间是2N，既空间复杂度是O（N）。

删除

- 自上而下，查找第一次出现节点的索引，并逐层找到每一层对应的节点。O（logN）
- 删除每一层查找到的节点，如果该层只剩下1个节点，删除整个一层（原链表除外）。O（logN）

总体上，跳跃表删除操作的时间复杂度是O（logN）。

优点

1）在于维持索引结构平衡的开销很低，只需要依靠一种随机决策机制即可。而二叉树需要依靠节点的旋转来保持平衡，开销较大。

2）查找、插入、删除的时间复杂度是 O（logN），效率很高。

Sorted-set这种有序集合，正是对于跳跃表的改进和应用。

7）Pub /Sub

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在 Redis 中，你可以设定对某一个 key 值进行消息发布及消息订阅，当一个 key 值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

8）Transactions

虽然 Redis 的 Transactions 提供的并不是严格的 ACID 的事务（比如一串用 EXEC 提交执行的命令，在执行中服务器宕机，那么会有一部分命令执行了，剩下的没执行），但是这个 Transactions 还是提供了基本的命令打包执行的功能（在服务器不出问题的情况下，可以保证一连串的命令是顺序在一起执行的，中间不会有其它客户端命令插进来执行）。Redis 还提供了一个 Watch 功能，你可以对一个 key 进行 Watch，然后再执行 Transactions，在这过程中，如果这个 Watched 的值进行了修改，那么这个 Transactions 会发现并拒绝执行。 和 HashMap 中迭代数据发现修改了就抛异常类似。可以防止提交脏数据。

3、延时队列

【概念】

Zset本质就是Set结构上加了个排序的功能，排序通过score属性来实现。

要实现延时队列，就需要利用score这个属性，比如代表的是想要执行时间的时间戳，在某个时间将它插入Zset集合中，它变会按照时间戳大小进行排序，也就是对执行时间前后进行排序，这样的话，起一个死循环线程不断地进行取第一个key值，如果当前时间戳大于等于该key值的socre就将它取出来进行消费删除，就可以达到延时执行的目的, 注意不需要遍历整个Zset集合，以免造成性能浪费。

【应用场景】

- 快递订单超过 30 分钟未支付，则自动取消
- 外卖商家超时未接单，则自动取消

以上场景都可以用定时任务去轮询实现，但是当数据量过大的时候，高频轮询数据库会消耗大量的资源，此时用延迟队列来应对这类场景比较好。

【需求】

- 消息存储
- 过期延时消息实时获取
- 高可用性

【为什么选择redis】

- java.util.concurrent.DelayQueue 优点：JDK自身实现，使用方便，量小适用 缺点：队列消息处于jvm内存，不支持分布式运行和消息持久化
- Rocketmq 优点：消息持久化，分布式 缺点：不支持任意时间精度，只支持特定level的延时消息
- Rabbitmq 优点：消息持久化，分布式 缺点：延时相同的消息必须扔在同一个队列，每一种延时就需要建立一个队列。因为当后面的消息比前面的消息先过期，还是只能等待前面的消息过期，这里的过期检测是惰性的
- Redis
    - **消息持久化，消息至少被消费一次**
    - 实时性：存在一定的时间误差（定时任务间隔）
    - 支持指定消息 remove
    - **高可用性**
    - **Redis 的特殊数据结构 ZSet 满足延迟的特性**

使用sortedset，用时间戳作为score，使用zadd key score1 value1 命令生产消息，使用zrangebysocre key min max withscores limit 0 1消费消息最早的一条消息。

这里选用 Redis 主要的原因就是其支持高性能的 score 排序，同时 Redis 的持久化 bgsave 特性，保证了消息的消费和存贮问题。bgsave 的原理是 fork 和 cow。fork 是指 Redis 通过创建子进程来进行 bgsave 操作, cow 指的是copy on write, 子进程创建后, 父进程通过共享数据段, 父进程继续提供读写服务, 写脏的页面数据会逐渐和子进程分离开来。

【如何保证消息的可靠性（ACK方案）】

- 维护一个消息记录表，存贮消息的消费记录，用于失败时回滚消息。表中记录消息ID、消息内容、消息时间、消息状态。
- 定时任务轮询该消息表，处理消费记录表中消费状态未成功的记录，重新放入等待队列。

【集群部署问题】

集群部署即多实例问题，是指同一个服务部署在不同的地方，发挥相同的作用，此时就会导致同时消费同一个消息的问题。

一般情况下解决此类问题就需要考虑接入外部应用的辅助。常见的分布式锁的方案有：基于数据库实现分布式锁、基于缓存实现分布式锁、基于 Zookeeper 实现分布式锁，这里使用缓存也就是 Redis 解决问题。

【实现】

1）使用redis - zset

```
package cn.chinotan.service.delayQueueRedis;

import org.apache.commons.lang3.StringUtils;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.Tuple;

import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.Set;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

/**
 * @program: test
 * @description: redis实现延时队列
 * @author: xingcheng
 * @create: 2018-08-19
 **/
public class AppTest {

    private static final String ADDR = "127.0.0.1";
    private static final int PORT = 6379;
    private static JedisPool jedisPool = new JedisPool(ADDR, PORT);
    private static CountDownLatch cdl = new CountDownLatch(10);

    public static Jedis getJedis() {
        return jedisPool.getResource();
    }

    /**
     * 生产者,生成5个订单
     */
    public void productionDelayMessage() {
        for (int i = 0; i < 5; i++) {
            Calendar instance = Calendar.getInstance();
            // 3秒后执行
            instance.add(Calendar.SECOND, 3 + i);
            AppTest.getJedis().zadd("orderId", (instance.getTimeInMillis()) /
1000, StringUtils.join("000000000", i + 1));
            System.out.println("生产订单: " + StringUtils.join("000000000", i +
1) + " 当前时间: " + new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new
Date()));
            System.out.println((3 + i) + "秒后执行");
        }
    }

    //消费者，取订单
    public static void consumerDelayMessage() {
        Jedis jedis = AppTest.getJedis();
        while (true) {
```

```
            Set<Tuple> order = jedis.zrangeWithScores("orderId", 0, 0);
            if (order == null || order.isEmpty()) {
                System.out.println("当前没有等待的任务");
                try {
                    TimeUnit.MICROSECONDS.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                continue;
            }
            Tuple tuple = (Tuple) order.toArray()[0];
            double score = tuple.getScore();
            Calendar instance = Calendar.getInstance();
            long nowTime = instance.getTimeInMillis() / 1000;
            if (nowTime >= score) {
                String element = tuple.getElement();
                Long orderId = jedis.zrem("orderId", element);
                if (orderId > 0) {
                    System.out.println(new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date()) + ":redis消费了一个任务：消费的订单OrderId为" +
element);
                }
            }
        }
    }

    static class DelayMessage implements Runnable{
        @Override
        public void run() {
            try {
                cdl.await();
                consumerDelayMessage();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        AppTest appTest = new AppTest();
        appTest.productionDelayMessage();
        for (int i = 0; i < 10; i++) {
            new Thread(new DelayMessage()).start();
            cdl.countDown();
        }
    }
}
```

2）使用redis过期回调机制

修改redis 配置 redis.conf 添加notify-keyspace-events Ex

```
# you are sure you want clients from other hosts to connect to Redis
# even if no authentication is configured, nor a specific set of interfaces
# are explicitly listed using the "bind" directive.
protected-mode no

# Accept connections on the specified port, default is 6379 (IANA #815344).
# If port 0 is specified Redis will not listen on a TCP socket.
port 6379

notify-keyspace-events Ex

# TCP listen() backlog.
#
# In high requests-per-second environments you need an high backlog in order
# to avoid slow clients connections issues. Note that the Linux kernel
# will silently truncate it to the value of /proc/sys/net/core/somaxconn so
# make sure to raise both the value of somaxconn and tcp_max_syn_backlog
# in order to get the desired effect.
tcp-backlog 511

# Unix socket.
```

新建redisconfig 注入 RedisMessageListenerContainer Bean

```java
@Configuration
public class RedisListenerConfig {
    @Bean
    RedisMessageListenerContainer container(RedisConnectionFactory connectionFactory) {

        RedisMessageListenerContainer container = new RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
        return container;
    }
}
```

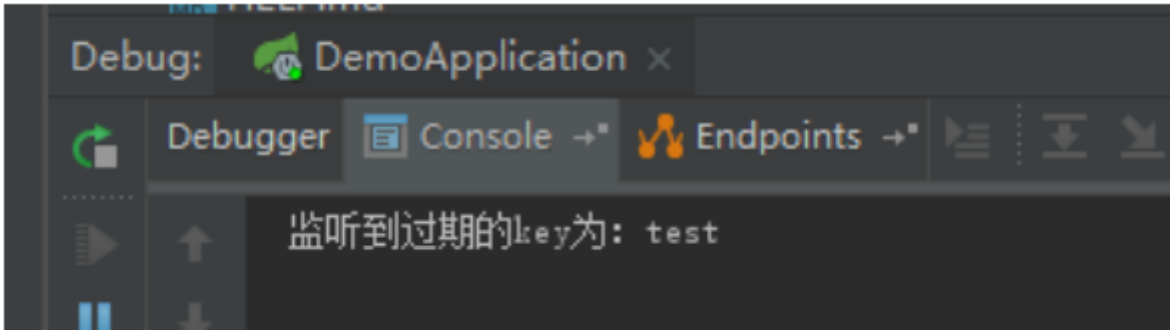
编写redis 监听类 继承KeyExpirationEventMessageListener

```java
@Component
public class RedisKeyExpirationListener extends KeyExpirationEventMessageListener {

    public RedisKeyExpirationListener(RedisMessageListenerContainer listenerContainer) {
        super(listenerContainer);
    }
    @Override
    public void onMessage(Message message, byte[] pattern) {
        String expiredKey = message.toString();
        System.out.println("监听到过期的key为: "+expiredKey);
    }
}
```

运行boot工程 使用redis 客户端 redis desktop manager添加一个key 设置过期时间

```
Connected.
39.96.77.182:0>set test abcd123 ex 3
"OK"

39.96.77.182:0>
```

延时两秒 在工程控制太看到输出

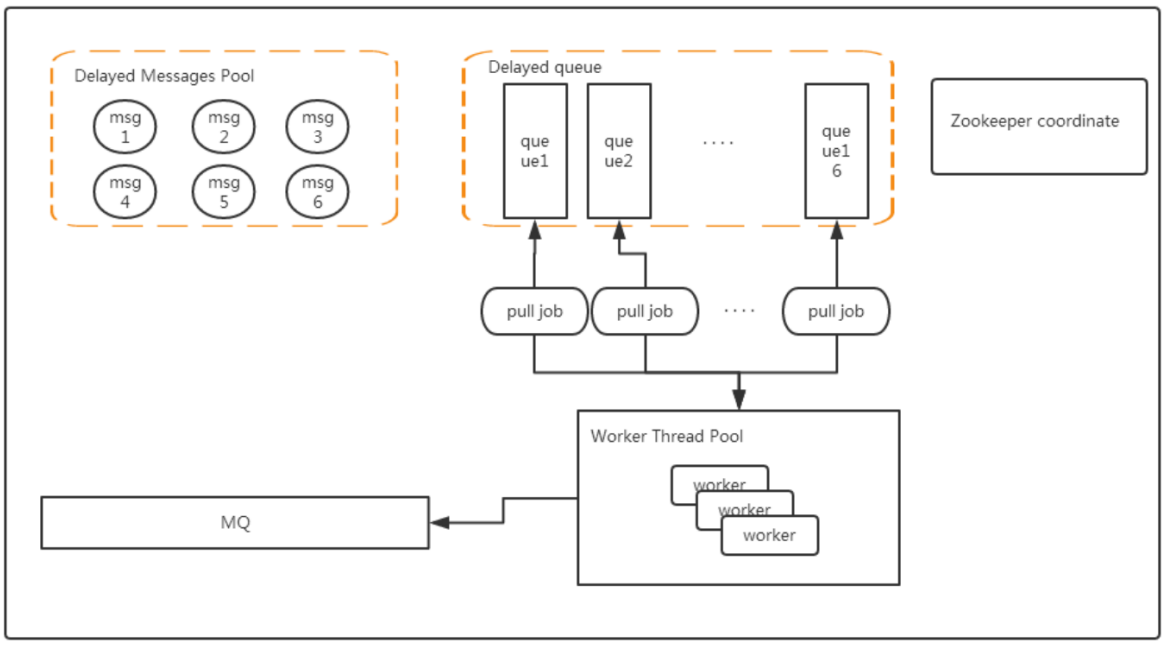


【问题点】

这种实现方式简单，但在生产环境下大多是多实例部署，所以存在并发问题，即缓存的查找和删除不具有原子性（zrangeWithScores和zrem操作不是一个命令，不具有原子性），会导致消息的多次发送问题。

【集群实现方案】

1）架构图



- **Messages Pool**

  所有延时消息存放的地方，结构为Redis Hash结构，因为hash结构能存储较大的数据量，数据较多时候会进行渐进式rehash扩容，并且对于HSET和HGET命令来说时间复杂度都是O(1)，key为消息ID，还有消息的 topic、过期时间、消息内容等属性组成的一个具体的 **Message** 消息实体。

- **Message**

  - tag：消息过期之后发送mq的tag
  - key：消息过期之后发送mq的key（topic）
  - body：消息过期之后发送mq的body，提供给消费这做具体的消息处理
  - delayTime：延时发送时间（默认，delayTime、expectDate有一个即可）
  - expectDate：期望发送时间

- **Delayed Queue**

  是16个有序的队列（队列支持水平扩展），结构为ZSET，value为 **Messages Pool** 中的消息ID，score为过期时间（分为多个队列是为了提高扫描的速度）。

- **Pull Job**

  这里分别为每一个队列创建了一个pull job thread，功能很简单，就是负责去队列中拉取过期的消息数据（这里保证一个队列有且只有一个pull job）。

- **Worker**

  pull job拉取到的过期消息会交给一个worker thread去处理，这样的好处是处理过期的消息实时性更高（pull job不必等去除过期消息全部处理完成在继续去拉取新的过期数据）。

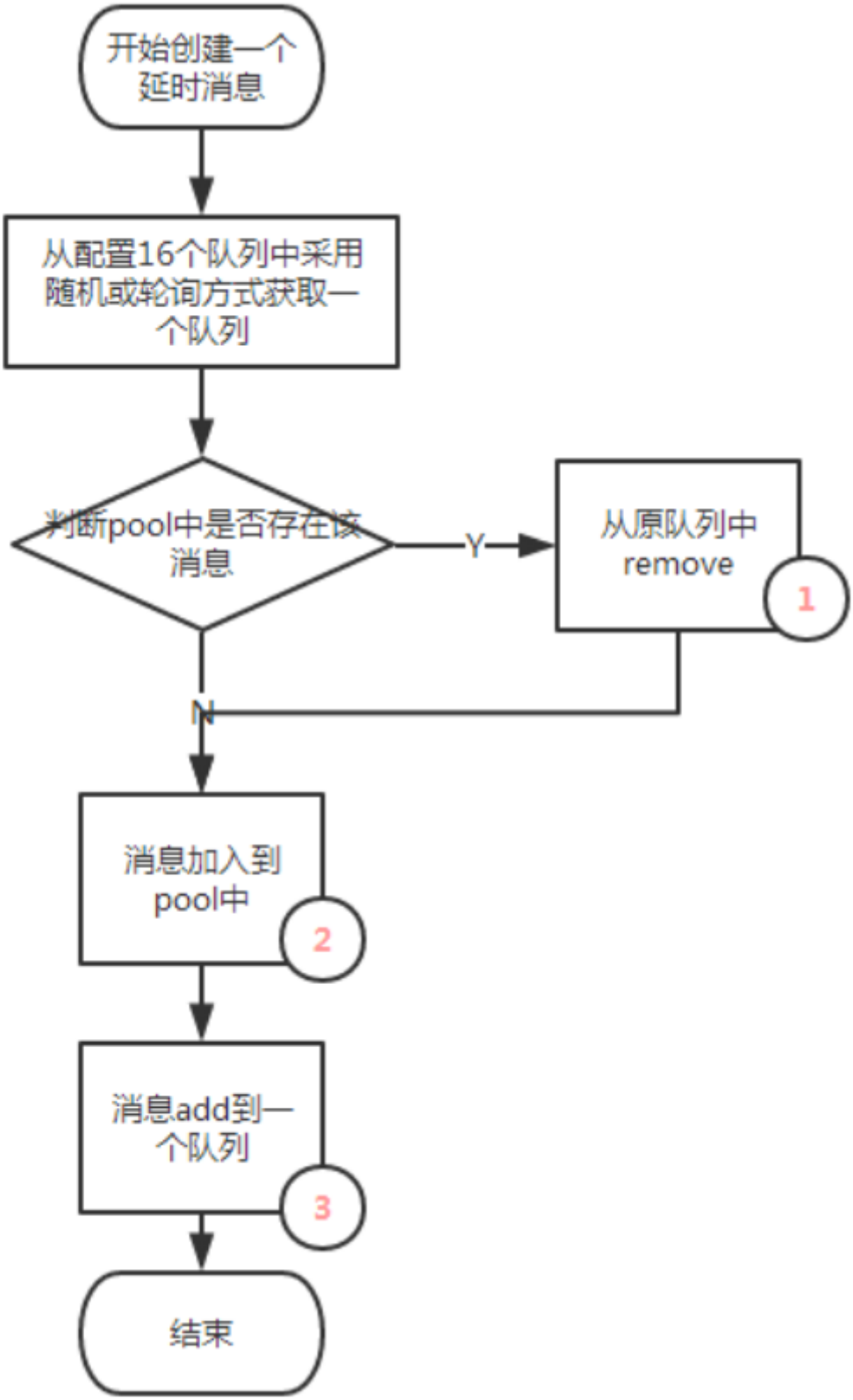
- **MQ**

  RocketMQ 消息中间件。

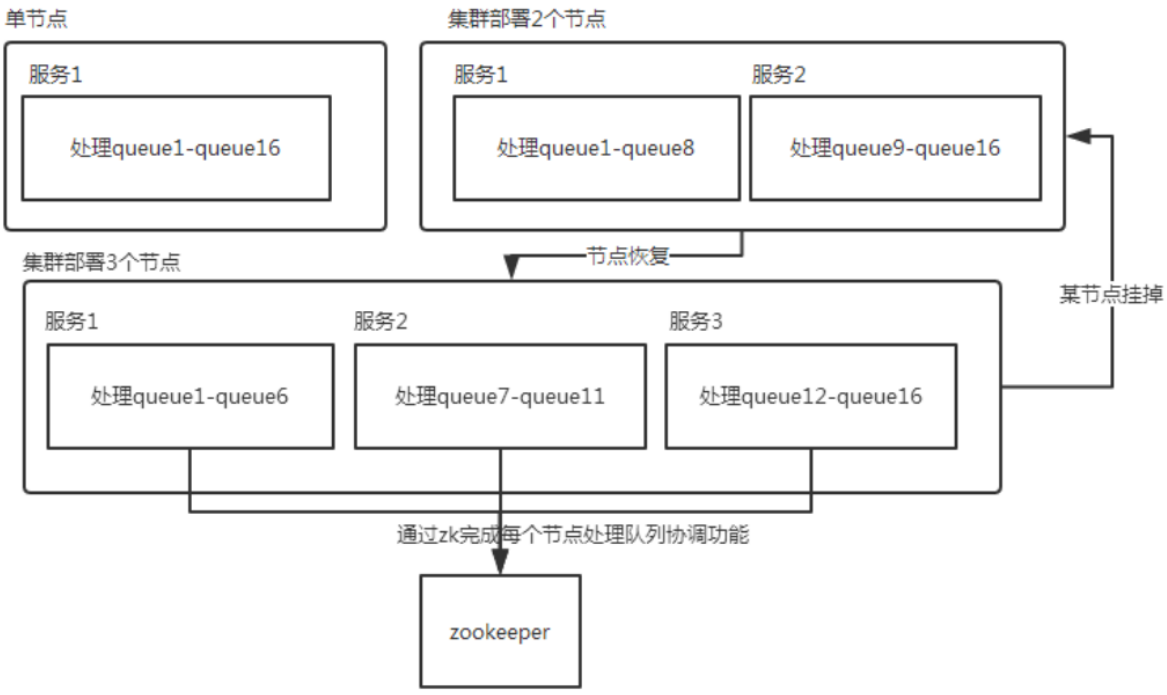
- **Zookeeper coordinate**

通过zk的操作来完成对 Redis 队列的重新分配工作，daemon thread监听zk节点的创建和删除 。

2）生产者流程



3）集群部署架构图



4）消费者流程

1）服务启动注册 zk 建立临时节点，获取分配处理的 redis queues，启动后台线程监听zk服务的临时节点

2）为每个分配queue创建一个pull job

3）pull job首先会去queue中查询是否有过期消息（查看set中第一个元素）并处理： Y：将根据消息id 从消息池中取出消息交给worker处理 N：等待

*由于过期消息发送成功才会从队列中remove，所以pull job会记录上一次查询队列的一个offset，每次获取到过期消息会将offset向前偏移，过期消息交给worker处理，当worker由于某些异常原因处理失败会重置pull job中offset，这样可以避免消息发送一次失败之后没办法在继续处理（除了新节点add || remove时候） 当部署服务有新增，延时队列服务会重新计算得到当前处理队列，并将之前创建pull job cancel，为新处理队列重新创建pull job。删除同理。

4）worker发送message到mq，remove掉 **Message Pool** 与 **Delayed Queue** 中对应的消息

5）MQ消费者进行业务消费

【延时队列总结】

因为在诸如快递或者外卖这类订单场景中，每日的订单量特别大，并发量特别大，这种通过数据库去维护订单的过期状态，然后通过查询数据库的方式的开销会特别大，而且订单的超时时间一般都是在几十分钟或者几小时之内，一般是一天内的，时效性很强，就需要考虑在缓存中维护这个过期状态而不是在数据库中了。

而考虑缓存除了考虑性能外，还需要考虑实现的复杂度。一般像订单这类场景，其数据结构往往更加丰富，相比之下redis有着更丰富的数据结构可以支持。而且若干个订单的过期处理，这种业务场景更像是若干个延时任务的处置和执行，可以基于redis的sorted set加上以时间戳为权重的模型来实现一个延时队列，而一般生产环境中往往是集群多实例的部署，就需要考虑分布式锁和负载均衡的应用，一般考虑用Zookeeper或者redis的setnx命令来实现分布式锁和负载均衡。

如上图集群部署方案的架构图中，可以基于redis的sorted set来存储score为时间戳，value为消息id的数据，然后再基于redis的hash来实现一个消息池 Message Pool 存储真正的消息体 Message 数据，当生产者生产了消息后，首先放入消息池中，然后再路由到多个延时队列中，这里使用多个延时队列的原因是，当数据量特别大时，多个延时队列同时扫描，可以大大提升消费的速度。

当服务启动时，首先到zk中注册，zk为每个服务分配若干延时队列。然后为每个队列分配一个pull job，用以不停探查队列中是否有过期的消息，通过检索比较set中第一个元素的score与当前系统时间，如果没有过期则等待，如果过期则取出其messageId，并交给工作线程处理。

工作线程根据messageId从消息池中将完整的消息体取出，然后进一步通过MQ发给消费端进行解耦，这样就可以及时取出过期的数据。如果消费端成功消费，则工作线程就删除掉消息池和延时队列中的消息记录。

4、阻塞队列

有时候，为了等待一个新元素到达数据中，需要使用轮询的方式对数据进行探查。 另一种更好的方式是，使用系统提供的阻塞原语，在新元素到达时立即进行处理，而新元素还没到达时，就一直阻塞住，避免轮询占用资源。BLPOP和BRPOP能很好解决这个问题。和Java提供的阻塞队列如LinkedBlockingQueue相比，redis提供读取阻塞，不提供写入阻塞。

【场景】

1）网络流量很大，但有紧急情况需要发送邮件通知别人，这时如果在程序中发送邮件，将会很费时，但又必须做；

2）处理客户端请求后，剩下的事情跟正常逻辑没有关系，但有些耗时的操作需要完成

遇到这样的情况，可以有几种办法：

1）开一个线程跑这些数据，主程序继续处理其他的任务；

如果这样的操作很多的话，对程序影响很大的，因为每次涉及到线程资源的申请和销毁，这些都要占用一些主程序时间，并且这样破坏了主要逻辑的流程。

2）用redis阻塞队列，将消息传到队列中，其他的事情由别的程序来从redis中读取队列中的消息，并进行处理；

实现方法：由php向redis中写人的队列写入消息，由python脚本读redis中的队列中的消息，使用该方法，当python读取队列的时候，如果为空，则阻塞，如果有消息，则唤醒并执行，效果很好。

在redis中，python调用的一个方法为blpop(key, timeout) ，当把timeout致为0时表示永久不超时，这样的话，这个python进程就是类似一个监听进程， 当有消息的时候他就努力工作，没有的时候就看着。

Spring使用Redis

```
//list集合 第一个元素为key值，第二个元素为弹出的元素值;当超时返回[null]
List<Object> obj = redisTemplate.executePipelined(new RedisCallback<Object>() {

   @Nullable
   @Override
   public Object doInRedis(RedisConnection connection) throws
DataAccessException {                //队列没有元素会阻塞操作，直到队列获取新的元素或超时
    return connection.bLPop(timeout,key.getBytes());
   }
},new StringRedisSerializer());
```

我们通过向队列添加元素来释放阻塞的线程：

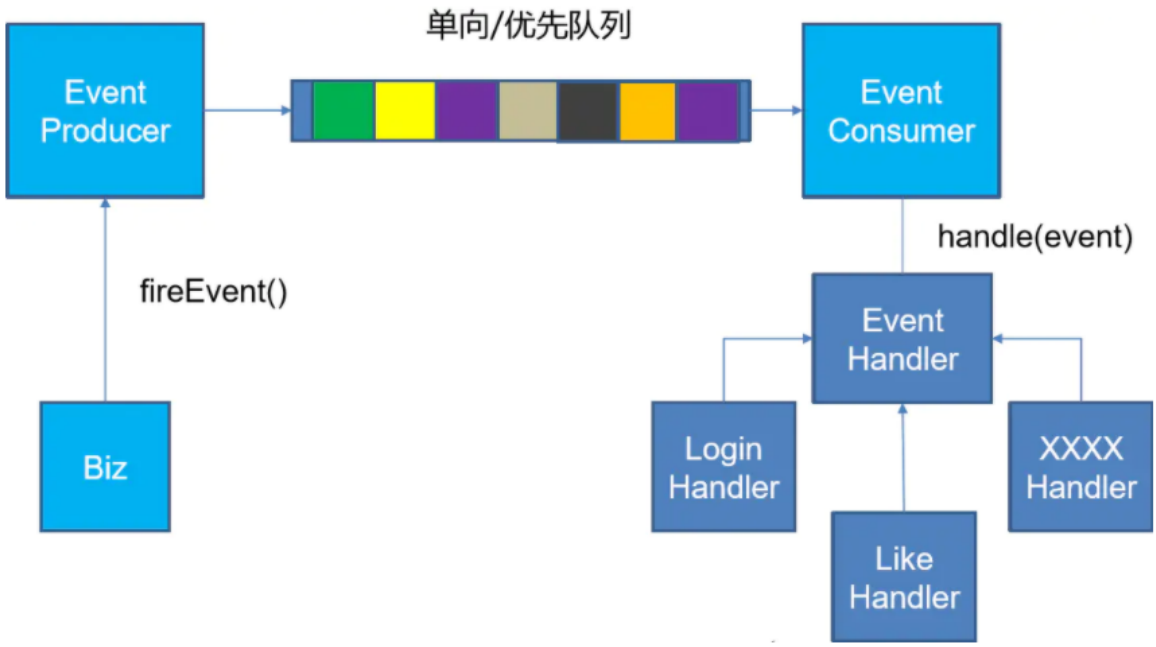
redisTemplate.opsForList().rightPush(key,value);

```
@Override
    public void run() {
        while(true) {
            List<byte[]> bytesList = null;
            try{
                //这种就是阻塞队列模式
                bytesList = jedis.blpop(0, queueKey.getBytes());
                // dosth
            } catch (Exception e) {
                 ....
            }
        }
   }
```

5、异步队列

事件驱动，比如 Linux 的 epoll IO 模型，让我们自己实现的话就可以基于异步队列来实现，用阻塞队列来装载就绪的事件，比如 accept 事件，read、write 事件等。然后起一个线程不断轮询这个阻塞队列，有事件就取出然后去执行，而不同的事件需要有不同的事件处理器来处理，这些事件处理器可以看成是对象，比如和读事件有关的事件处理器对象就封装了读数据的方法，然后可以在 spring bean 初始化后将这些处理器存储到一个 Map 中，key就是对应的事件的类型，value 就是 bean，用 map 来实现一个策略模式，这样就可以将不同事件交给不同事件处理器来处理，实现了一个事件驱动模型。

流程图



生产者

```
public class EventProducer {
    @Autowired
    private JedisAdapter jedisAdapter;
    /**
     * 发事件,把事件添加进队列
     * @param eventModel
     * @return
     */
    public boolean fireEvent(EventModel eventModel) {
        try {
            /**
             * 把事件发生现场进行序列化push进异步消息队列
             */
            String json = JSONObject.toJSONString(eventModel);
            String key = RedisKeyUtil.getEventQueueKey();
            jedisAdapter.lpush(key, json);
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}
```

消费者

```
public class EventConsumer implements InitializingBean, ApplicationContextAware
{
    private static final Logger logger =
LoggerFactory.getLogger(EventConsumer.class);
    /**
     * key  从队列取出的事件Evnet
     * value那些关注事件Handler
     */
```

```java
    private Map<EventType, List<EventHandler>> config = new HashMap<>();
    /**
     * Spring应用程序上下文，用于找出被容器管理的所有Handler
     */
    private ApplicationContext applicationContext;

    /**
     *
     */
    @Autowired
    private JedisAdapter jedisAdapter;

    @Override
    public void afterPropertiesSet() throws Exception {
        /**
         * 找出容器中所有实现该接口的类EventHandler
         */
        Map<String, EventHandler> beans =
applicationContext.getBeansOfType(EventHandler.class);
        if (beans != null) {
            for (Map.Entry<String, EventHandler> entry : beans.entrySet()) {
                /**
                 * 遍历所有的Handler拿到每个Handler关注的事件类型
                 */
                List<EventType> eventTypes =
entry.getValue().getSupportEventTypes();
                /**
                 * 遍历拿到的事件类型，注册这些事件
                 */
                for (EventType type : eventTypes) {
                    if (!config.containsKey(type)) {
                        // 如果map中没有这个key 第一次注册这个时间
                        // 则new一个ArrayList对象初始化关注该事件的Handler列表
                        config.put(type, new ArrayList<EventHandler>());
                    }
                    //添加进关注该事件的Handler列表
                    config.get(type).add(entry.getValue());
                }
            }
        }
        /**
         * 该线程   取异步时间队列
         */
        Thread thread = new Thread(() -> {
            while(true) {
                /**
                 * 拿到EventQueue的key
                 */
                String key = RedisKeyUtil.getEventQueueKey();
                /**
                 * 从队列中取出事件
                 * 这是一个阻塞操作，若当前队列为空则会一直阻塞在这里
                 */
                List<String> events = jedisAdapter.brpop(0, key);

                for (String message : events) {
                    /**
                     * brpop 返回的类型是 key和value   要把key过滤掉
```

```
                     */
                    if (message.equals(key)) {
                        continue;
                    }
                    /**
                     * 将事件转成实体
                     */
                    EventModel eventModel = JSON.parseObject(message,
EventModel.class);
                    /**
                     * 看注册中心是否有这个事件
                     */
                    if (!config.containsKey(eventModel.getType())) {
                        logger.error("不能识别的事件");
                        continue;
                    }
                    /**
                     * 找到这个事件，Handler开始处理事件
                     */
                    for (EventHandler handler :
config.get(eventModel.getType())) {
                        handler.doHandle(eventModel);
                    }
                }
            }
        });
        thread.start();
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

事件处理接口

```
public interface EventHandler {
    /**
     * 处理事件
     * @param model
     */
    void doHandle(EventModel model);

    /**
     * Handler关心哪些事件Event
     * @return
     */
    List<EventType> getSupportEventTypes();
}
```

实现EventHandler接口，具体的事件处理以LikeHandler为例

```java
public class LikeHandler implements EventHandler {
    /**
     * 注入站内信服务
     */
    @Autowired
    private IMessageService IMessageService;
    /**
     * 注入用户服务
     */
    @Autowired
    private IUserService IUserService;

    @Override
    public void doHandle(EventModel model) {
        /**
         * 点完赞后给实体拥有者发站内信
         */
        Message message = new Message();
        /**
         * 当前用户发出
         */
        message.setFromId(QaaUtil.SYSTEM_USERID);
        /**
         * 发给被点赞实体的拥有者
         */
        message.setToId(model.getEntityOwnerId());
        /**
         * 设置私信时间
         */
        message.setCreatedDate(new Date());
        /**
         * 通过触发者id找到用户
         */
        User user = IUserService.getUser(model.getActorId());
        /**
         * 设置消息内容
         */
        message.setContent("用户" + user.getName()
                + "赞了你的评论,http://127.0.0.1:8080/question/" +
model.getExt("questionId"));
        /**
         * 调用站内信服务添加消息
         */
        IMessageService.addMessage(message);
    }

    @Override
    public List<EventType> getSupportEventTypes() {
        /**
         * 关注了like事件
         */
        return Arrays.asList(EventType.LIKE);
    }
}
```

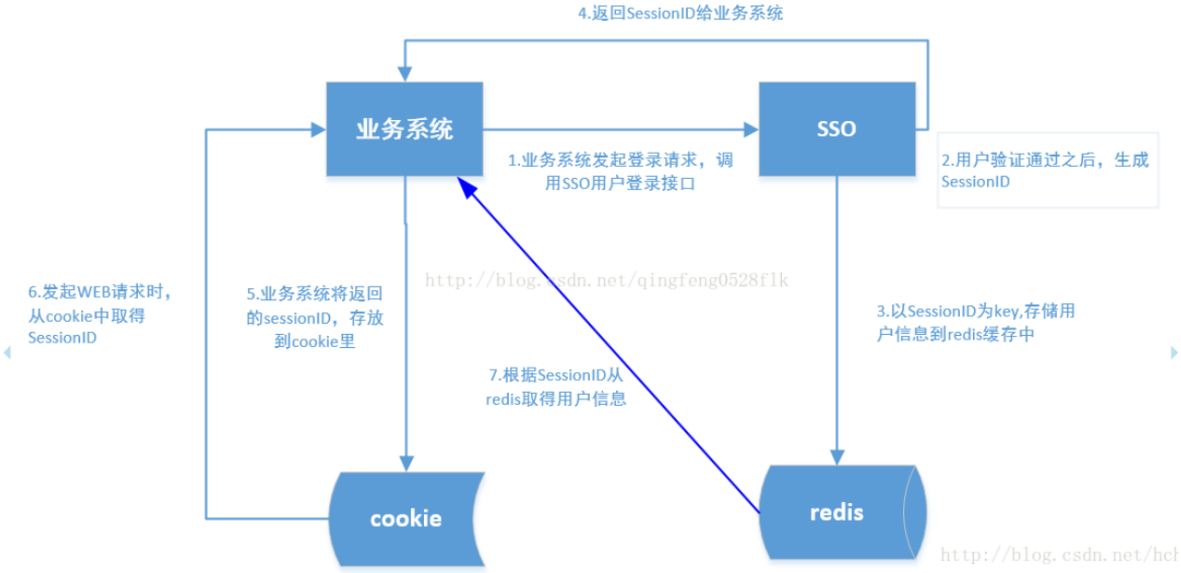
一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

如果不用sleep，list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。

如果想要生产一次消费多次，可以使用pub/sub主题订阅者模式，可以实现1:N的消息队列，但在消费者下线后，生产的消息会丢失，想要持久化的话，需要使用消息队列如rabbitmq等。

6、Redis缓存Session

通过redis缓存和cookie实现单点登录思路图：



步骤

```
未登录用户访问子站 a.com 进行登录，自动跳转到账户中心的统一登录页 account.com/login
用户在统一登录页进行登录，登录成功后显示登录跳转页
显示登录跳转页后自动跳转回 a.com，单点登录完成
用户在访问 b.com 时无需再次登录
```

Session的缓存机制

1.业务系统发起登录请求，调用SSO用户登录接口;

2.SSO登录接口的处理逻辑：

```
2.1 根据用户名和密码去数据库验证用户是否合法。
2.2 用户验证通过之后，生成SessionID,并返回给业务系统。
    同时以SessionID为key,存储用户信息到redis缓存中
```

SSO登录接口代码：

```
public JSONObject userLogin(@RequestBody JSONObject jsonObject){
        UserLoginResponse userLoginResponss = new UserLoginResponse();
        try {
            logger.info("处理用户登录业务逻辑,接收报文"+jsonObject);
            String msgWithDigesta=SecurityUtil.scfMatch(jsonObject.toString(),
newXService.getPrivateKey());
            //生成实体
            User user = JSONObject.parseObject(msgWithDigesta,User.class);
```

```
                //是否验证用户的密码
                boolean isChechPassword = true;
                User userInfo =
anaService.loginCheckUserInfo(user,isChechPassword);
                // 存储用户信息到redis缓存中
                String ticket =
anaService.storeUserLoginSessionInRedis(userInfo,user.getModuleCode());
                userLoginResponss.setRetCode(RetCode.LOGIN_SUCCESS.getCode());
                userLoginResponss.setRetMessage("用户登录成功");
                userLoginResponss.setTicket(ticket);
                userLoginResponss.setStatus(userInfo.getStatus());

 userLoginResponss.setIsModifyPassword(userInfo.getIsModifyPassword());
            } catch (Exception e) {
                userLoginResponss.setRetCode(RetCode.LOGIN_FAILED.getCode());
                userLoginResponss.setRetMessage(e.getMessage());
                logger.info("插入用户数据到表中失败，原因："+e.getMessage());
            }
            logger.info("返回处理用户登录业务逻辑结果，
Result{[]}"+JSONObject.toJSONString(userLoginResponss));
            return JSON.parseObject(JSONObject.toJSONString(userLoginResponss));
        }
```

存储用户信息到redis缓存的代码：

```
/**
        * 存储用户登录session信息到redis中
        * @param userInfo
        * @return
        */
       public String storeUserLoginSessionInRedis(User userInfo,String
moduleCode) {
            // 存储用户ticket信息
            // 使用AES加密登录用户ID生成SessionID,加密密码是配置文件里定义的64位字符串
            String sessionId =
AesUtil.encrypt(String.valueOf(userInfo.getUserId()), newXService.getBizkey());

            String unique_ticket = userInfo.getSystemId()+sessionId+"_USER_LOGIN";

            //
            String ticket =
userInfo.getSystemId()+sessionId+System.currentTimeMillis()+"_USER_LOGIN";

            UserLoginSession userLoginSession = new UserLoginSession();
            userLoginSession.setUserId(String.valueOf(userInfo.getUserId()));
            userLoginSession.setUserName(userInfo.getUserName());
            userLoginSession.setUserLoginName(userInfo.getUserLoginName());
            // 获取权限
            List<Permission> permissions =
getUserPermissions(userInfo.getUserId());
            userLoginSession.setPermissions(permissions);


userLoginSession.setModuleCode(StringUtils.killNull(userInfo.getModuleCode()));

            userLoginSession.setLastLoginTime(userInfo.getLastLoginTime());
            userLoginSession.seteId(StringUtils.killNull(userInfo.geteId()));
```

```
            userLoginSession.setSessionId(ticket);
            userLoginSession.setUserInfo(userInfo);

            //限制唯一登录，删除上一个用户信息
            if (redisService.exists(unique_ticket))
                redisService.del(redisService.get(unique_ticket));

            redisService.set(unique_ticket, ticket);

            logger.info("访问AnaController的login方法:放进redis"+ticket);
            redisService.setKeyExpire((ticket).getBytes(),1800);

            logger.info("userloginsession result
="+JSONObject.toJSONString(userLoginSession));
            return ticket;
        }
```

3.业务系统将返回的sessionID，存放到cookie里

业务系统controller的代码：

```
@RequestMapping("/login.ajax")
     @ResponseBody
     public  Map<String, Object> login(@RequestParam("username2") String
username2,
                                       @RequestParam("moduleCode2") String
moduleCode2,
                                       @RequestParam("password2") String password2,
String requestUrl, HttpServletResponse response) {
                // 其他业务逻辑省略
                String sessionId = userBySso.getTicket();
                Cookie cookie = new Cookie("CORE_SESSION", sessionId);
                cookie.setPath("/");
                response.addCookie(cookie);
                // 其他业务逻辑省略
    }
```

4.业务系统取得Session信息，并检验用户信息

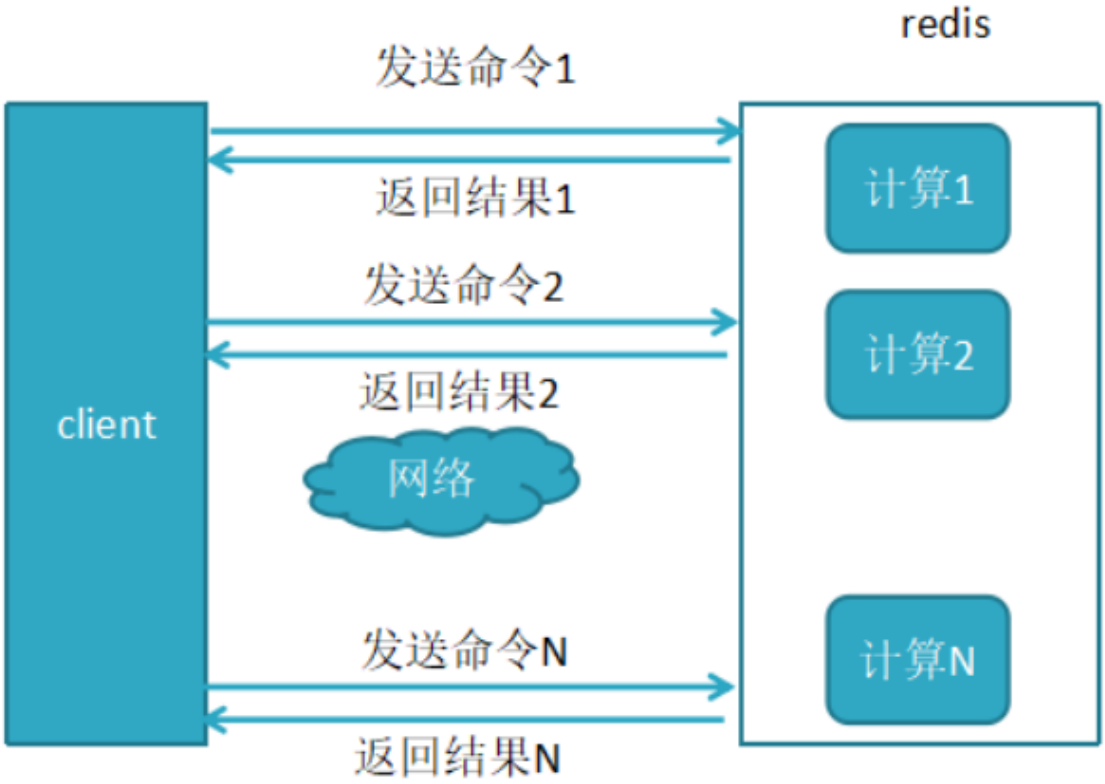
业务系统的页面发起web请求时，在自定义拦截器(继承自HandlerInterceptor)的preHandle方法里取得session信息，并检查用户是否登录。 Session信息取得时，首先从cookie中取得SessionId，然后根据SessionId从redis取得用户信息

```
    public UserLoginSession getUserLoginSession(HttpServletRequest req) {
            logger.info("访问getUserLoginSession");

            String sessionId = "";
            Cookie[] cookie = req.getCookies();
            if (cookie == null) {
                return null;
            }
            for (int i = 0; i < cookie.length; i++) {
                Cookie cook = cookie[i];
                if (cook.getName().equals("CORE_SESSION")) {
                    sessionId = cook.getValue().toString();
                }
            }
```

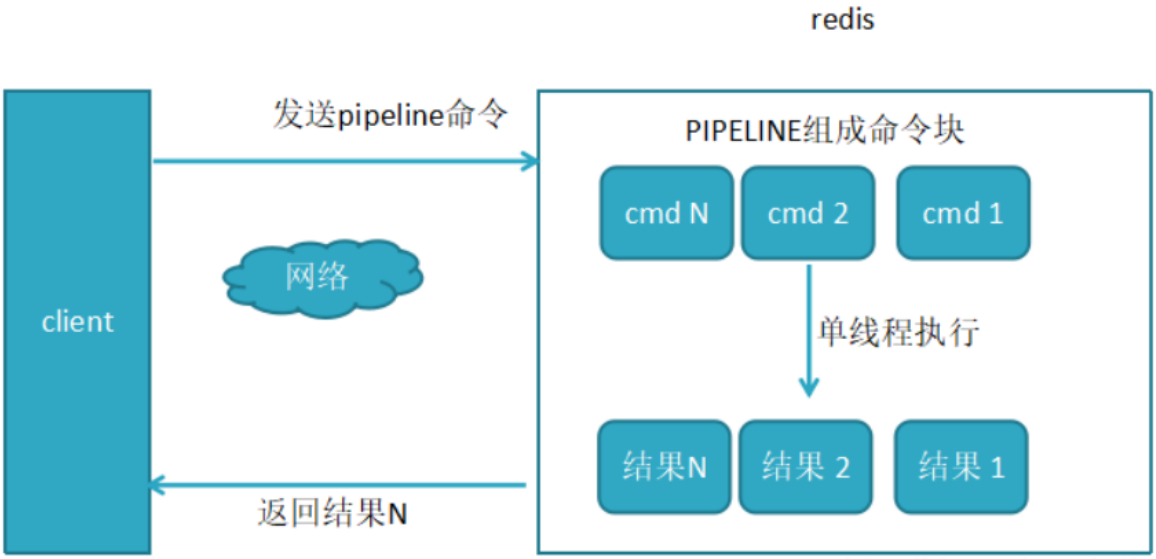
```
        logger.info("访问getUserLoginSession获取sessionId:  " + sessionId);

        if ("".equals(sessionId)) {

            return null;
        }

        String UserLoginSessionStr = redisService.get(sessionId);

        logger.info("访问getUserLoginSession获取USERLOGINSESSION:  " +
UserLoginSessionStr);

        if (null == UserLoginSessionStr || "".equals(UserLoginSessionStr)) {

            return null;
        }

        UserLoginSession userLoginSession = (UserLoginSession)
JSONObject.toJavaObject(JSONObject.parseObject(UserLoginSessionStr),
UserLoginSession.class);
        logger.info("访问getUserLoginSession获取USER_ID成功:  " +
userLoginSession.getUserId());
        redisService.setKeyExpire((sessionId).getBytes(), 1800);

 redisService.setKeyExpire((userLoginSession.getTicketRole()).getBytes(),1800);

        return userLoginSession;
    }
```

7、Pipeline

redis客户端执行一条命令分4个过程： 发送命令 - 〉命令排队 - 〉命令执行 - 〉返回结果



使用pipeline后的执行过程：



性能对比：

网络	延迟	非Pipeline	Pipeline
本机	0.17ms	573ms	134ms
内网服务器	0.41ms	1610ms	240ms
异地机房	7ms	80 000ms	1104ms

原生批命令(mset, mget)与Pipeline对比：

- 原生批命令是原子性，pipeline 是非原子性
- 原生批命令一命令多个key, 但 pipeline 支持多命令（存在事务），非原子性
- 原生批命令是服务端实现，而 pipeline 需要服务端与客户端共同完成

Redis Mset 命令用于同时设置一个或多个 key-value 对：

```
redis 127.0.0.1:6379> MSET key1 "Hello" key2 "World"
OK
redis 127.0.0.1:6379> GET key1
"Hello"
redis 127.0.0.1:6379> GET key2
1) "World"
```

Pipeline 管道正确使用方式：

使用pipeline组装的命令个数不能太多，不然数据量过大，增加客户端的等待时间，还可能造成网络阻塞，可以将大量命令的拆分多个小的pipeline命令完成。

使用步骤：

- 获取jedis对象（一般从连接池中获取）
- 获取jedis对象的pipeline对象
- 添加指令（pipe.del(key)）
- 执行指令（pipe.sync()）

```
public boolean mdel(List<String> keys) {
        Jedis jedis = null;
        boolean flag = false;
        try {
            jedis = pool.getResource();//从连接借用Jedis对象
            Pipeline pipe = jedis.pipelined();//获取jedis对象的pipeline对象
            for(String key:keys){
                pipe.del(key); //将多个key放入pipe删除指令中
            }
            pipe.sync(); //执行命令，完成此时pipeline对象的远程调用
            flag = true;
        } catch (Exception e) {
            pool.returnBrokenResource(jedis);
            e.printStackTrace();
        } finally {
            returnResource(pool, jedis);
        }
        return flag;
    }
```

小结：

使用管道可以将多个请求合并在一次网络 IO 中去发送，相比起普通命令一次 IO 携带一个命令，并且下个命令要等上个命令执行完才能继续发送的方式来说，降低了 IO 开销，提高了系统的吞吐量。除此以外，使用管道还可以降低操作系统从用户态向内核态转换的开销，因为一次 IO 相当于一次向服务端发起的一次网络请求，那么是需要服务端调用 read write 去读写数据的，如果 IO 次数少了，那么这些系统调用就少了。

8、Redis事务

- 批量操作在发送 EXEC 命令前被放入队列缓存。
- 收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行。
- 在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中。

命令	描述
MULTI	用于标记事务的开始，其后执行的命令都将被存入命令队列，直到执行EXEC时，这些命令才会被原子的执行。
EXEC	执行在一个事务内命令队列中的所有命令，同时将当前连接的状态恢复为正常状态，即非事务状态。如果在事务中执行了WATCH命令，那么只有当WATCH所监控的Keys没有被修改的前提下，EXEC命令才能执行事务队列中的所有命令，否则EXEC将放弃当前事务中的所有命令。
DISCARD	回滚事务队列中的所有命令，同时再将当前连接的状态恢复为正常状态，即非事务状态。如果WATCH命令被使用，该命令将UNWATCH所有的Keys。
WATCH *key [key ...]*	在MULTI命令执行之前，可以指定待监控的Keys，然而在执行EXEC之前，如果被监控的Keys发生修改，EXEC将放弃执行该事务队列中的所有命令。
UNWATCH	取消当前事务中指定监控的Keys，如果执行了EXEC或DISCARD命令，则无需再手工执行该命令了，因为在此之后，事务中所有被监控的Keys都将自动取消。

```
127.0.0.1:6379> multi 事务开始
OK
127.0.0.1:6379> sadd user:name james   业务操作1
QUEUED
127.0.0.1:6379> sadd user:age 24    业务操作2
QUEUED
127.0.0.1:6379> get user:age   若在另一客户端取user:age，返回0，事务还没结束
QUEUED
127.0.0.1:6379> exec   事务结束
1) (integer) 1
2) (integer) 1
3) (error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379> get age  此时才能查到数据
"4"
```

https://blog.csdn.net/w1lgy

```
127.0.0.1:6379> multi   事务开始
OK
127.0.0.1:6379> sadd tt 1    业务操作
QUEUED
127.0.0.1:6379> discard    停止事务
OK
127.0.0.1:6379> exec
(error) ERR EXEC without MULTI
127.0.0.1:6379> get tt   查不到数据
(nil)
```

原理：

pipeline是多条命令的组合，为了保证它的原子性，redis提供了简单的事务。 一组需要一起执行的命令放到multi和exec两个命令之间，其中multi代表事务开始，exec代表事务提交。然后提供了 watch 命令来监控某个 key，在执行EXEC命令之前，如果Redis检测到至少有一个键被修改了，那么整个事务便会中止运行，然后EXEC命令会返回一个Null值，提醒用户事务运行失败。

项目：

消息系统中用到了事务，是在对接口的访问频率上做的限制。接口控制在一分钟访问3次，然后用 redis 的 String 的 incr 来实现计数器，每来一次请求原子自增1，第一次请求在自增的同时要设置 key 的过期时间，这两步需要保证一个原子性，所以用事务包裹起来执行。

```
//每次递增1，如果在60s内访问超过了3次 则 访问频率过高(防止抓取)  模拟的时候 连续执行多次junit来实现打印用户访问频率超限的效果
    private void website(String ip) {
        String value = jedis.get(ip);
        if(value==null){
            Transaction transaction = jedis.multi();
            transaction.incr(ip);
            transaction.expire(ip, 60);
            transaction.exec();
        }else{
            int intvalue = Integer.parseInt(value);
            if(intvalue<=3){
                jedis.incr(ip);
            }else{
                System.err.println("用户访问频率超限!");
            }
      }
```

通过watch机制实现CAS乐观锁：

WATCH的机制是：在事务EXEC命令执行时，Redis会检查被WATCH的key，只有被WATCH的key从WATCH起始时到执行前没有发生过变更，EXEC才会被执行。如果WATCH的key在WATCH命令到EXEC命令之间发生过变化，则EXEC命令会返回失败。

Redis使用WATCH命令实现事务的“检查再设置”（CAS）行为。

CAS乐观锁实现伪代码

```
WATCH mykey
val = GET mykey
val = val + 1 //先set
MULTI
SET mykey $val
EXEC
```

由上述伪码可知，如果存在竞争状态，并且有另一个客户端在我们调用WATCH命令和EXEC命令之间的时间内修改了val变量的结果，那么事务将会运行失败。

我们只需要重复执行上述伪码的操作，希望此次运行不会再出现竞争状态。这种形式的锁就被称为乐观锁，它是一种非常强大的锁。在许多用例中，多个客户端可能会访问不同的键，因此不太可能发生冲突 —— 也就是说，通常没有必要重复执行上述伪码的操作。

为什么Redis不支持事务回滚?

因为 redis 对自己的架构十分自信，认为导致需要回滚的一般都是业务程序自身的问题，和 redis 本身没有关系。而这种程序错误多半是在开发期间就会遇到并得以解决，并不会进入生产环境，所以redis不采取回滚也是为了更简单高效地实现当错误发生时的一些逻辑。如果这种由于程序错误导致的命令执行失败，最终还需要开发人员在开发期迭代解决，单纯回滚并不能解决问题。

9、Lua脚本

Lua是大小写敏感的

Lua的变量默认是全局变量，局部变量需要显示声明

do-end标记代码块的范围

Lua的单行注释使用双横线“--”表示，双横线后的内容为注释内容。

Lua是一种动态类型语言，它有8种基本类型：nil，Boolean，number，string，userdata，**function**，**thread**和**table**（结构化的数据类型，下标从1开始）。type函数可以返回指定值的类型：

Lua中函数可以接收的**参数是list**， **可以返回多个结果**。

使用Lua脚本的优点：

- 减少网络开销。可以将多个请求通过脚本的形式一次发送，减少网络时延
- 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。因此在编写脚本的过程中无需担心会出现竞态条件，无需使用事务。
- 复用。客户端发送的脚步会永久存在redis中，这样，其他客户端可以复用这一脚本而不需要使用代码完成相同的逻辑。
- 从定义上来说， Redis 中的脚本本身就是一种事务， 所以任何在事务里可以完成的事， 在脚本里面也能完成。 并且一般来说， 使用脚本要来得更简单，并且速度更快。

使用Lua脚本：

在redis里面使用lua脚本主要用三个命令

- eval
- evalsha

- script load

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

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

- key代表要操作的rediskey
- arg可以传自定义的参数
- numkeys用来确定key有几个
- script就是你写的lua脚本

区别

Redis Script Load 命令用于将脚本 script 添加到脚本缓存中，但并不立即执行这个脚本。

eval 命令也会将脚本添加到脚本缓存中，但是它会立即对输入的脚本进行求值。

在脚本被加入到缓存之后，通过 evalsha 命令，可以使用脚本的 SHA1 校验和来调用这个脚本。

脚本可以在缓存中保留无限长的时间，直到执行 scrip_flush 为止。

```
redis 127.0.0.1:6379> SCRIPT LOAD "return 'hello moto'"
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis 127.0.0.1:6379> EVALSHA "232fd51614574cf0867b83d384a5e898cfd24e5a" 0
"hello moto"
```

lua脚本里面使用KEYS[1]和ARGV[1]来获取传递的key和arg，在用eval命令的时候，可以注意到每次都要把执行的脚本发送过去，这样势必会有一定的网络开销，所以redis lua脚本做了缓存，通过script load 和 evalsha实现，script load命令会在redis服务器缓存你的lua脚本，并且返脚本内容的SHA1校验和，然后通过evalsha 传递SHA1校验和来找到服务器缓存的脚本进行调用。

spring-data-redis操作lua：

这个框架把lua脚本封装成RedisScript对象，并且可以将lua脚本执行的结果自动转换为配置的java类型，然后只要直接调用execute方法即可，并且这个execute逻辑中封装了evalsha的优化,源码如下

```
protected <T> T eval(RedisConnection connection, RedisScript<T> script,
ReturnType returnType, int numKeys,
            byte[][] keysAndArgs, RedisSerializer<T> resultSerializer) {

        Object result;
        try {
            result = connection.evalSha(script.getSha1(), returnType, numKeys,
keysAndArgs);
        } catch (Exception e) {

            if (!exceptionContainsNoScriptError(e)) {
                throw e instanceof RuntimeException ? (RuntimeException) e : new
RedisSystemException(e.getMessage(), e);
            }
```

```
            result = connection.eval(scriptBytes(script), returnType, numKeys,
keysAndArgs);
        }

        if (script.getResultType() == null) {
            return null;
        }

        return deserializeResult(resultSerializer, result);
    }
```

Redis生成全局id实战：

```
 @Autowired
    private RedisTemplate<String,Long> redisTemplate;


    public  String nextID(){
        String key = Prefix+simpleDateFormatThreadLocal.get().format(new
Date());
        Long existedID = redisTemplate.opsForValue().get(key);
        if(existedID!=null){
            redisTemplate.opsForValue().set(key,existedID+1);
            return key+String.format("%04d",existedID+1);
        }else{
            redisTemplate.opsForValue().set(key,1L);
            return key+"0001";
        }
    }
```

这段代码是存在问题的，在并发的情况下，get方法可以访问到相同的key，就会出现id重复的问题，测试代码如下：

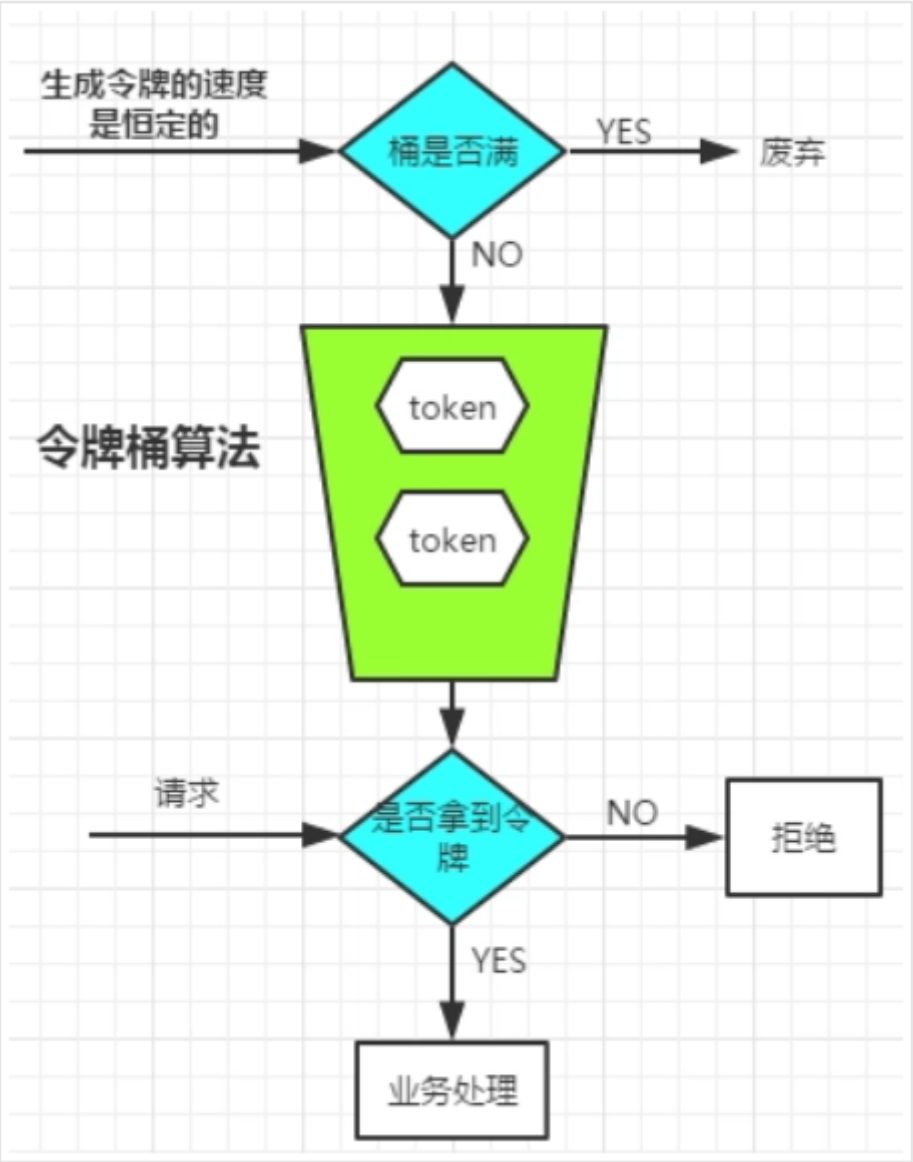
```
System.out.println("current:"+idGenerator.currentID());
        Integer threadSize =5;
        final CountDownLatch countDownLatch = new CountDownLatch(threadSize);
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                for(int i =0 ;i<100;i++){

 System.out.println(Thread.currentThread().getName()+":"+idGenerator.nextID());
                }
                countDownLatch.countDown();
            }
        };
        for(int i =0;i<threadSize;i++){
            new Thread(runnable,"Thread"+i).start();
        }
        countDownLatch.await();
        System.out.println("current:"+idGenerator.currentID());
```

当然这边我们也可以使用乐观锁或者分布式锁来实现，但是锁自旋的逻辑还是有潜在危险的 如果用lua来实现，把这个阻塞动作放在redis服务器，那我们的代码就会很健壮了，新建一个lua脚本：

```
local key = KEYS[1]
local id = redis.call('get',key)
if(id == false)
then
    redis.call('set',key,1)
    return key.."0001"
else
    redis.call('set',key,id+1)
    return key..string.format('%04d',id + 1)
end
```

```
public String nextIDLua(){
        String key = Prefix+simpleDateFormatThreadLocal.get().format(new
Date());
        DefaultRedisScript<String> redisScript =new DefaultRedisScript<>();
        redisScript.setLocation(new ClassPathResource("lua/genID.lua"));
        redisScript.setResultType(String.class);
        //System.out.println(redisScript.getSha1());
        return redisTemplate.execute(redisScript,(RedisSerializer<?>)
redisTemplate.getKeySerializer(),
(RedisSerializer<String>)redisTemplate.getKeySerializer(),Lists.newArrayList(key
));
}
```

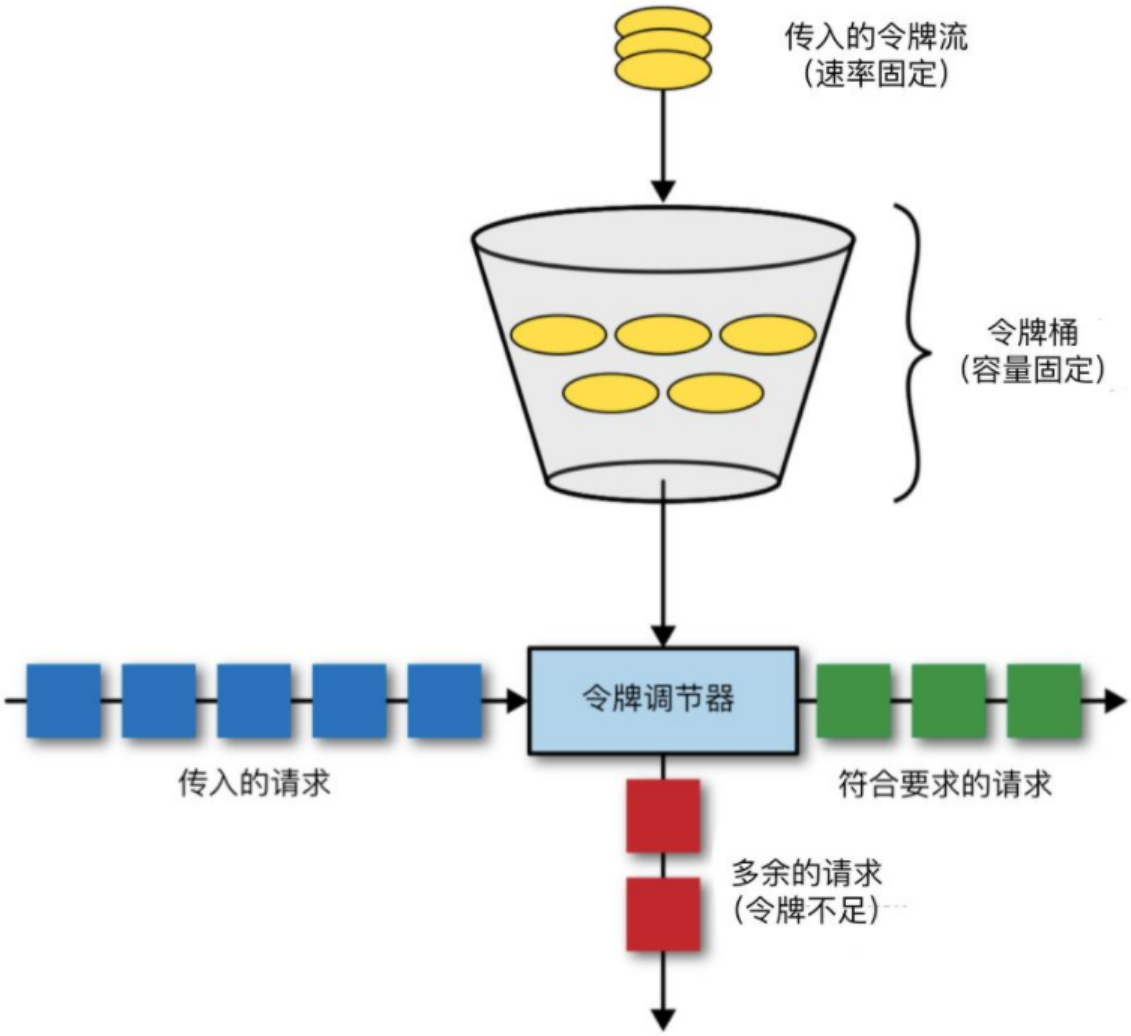
10、Redis + Lua 实现分布式限流桶



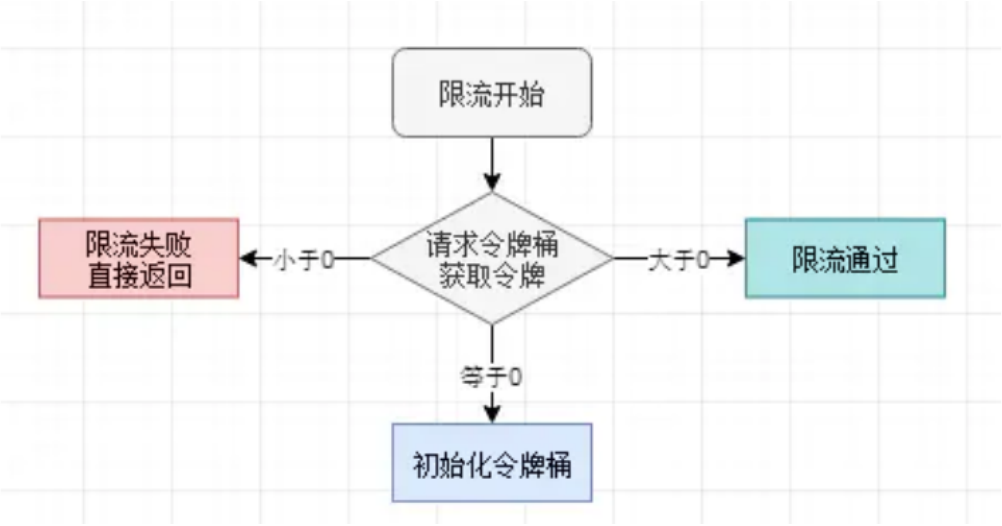
令牌桶算法是常见的限流算法，用来控制发送到网络上的数据的数目，并允许突发数据的发送，其原理也很简单：

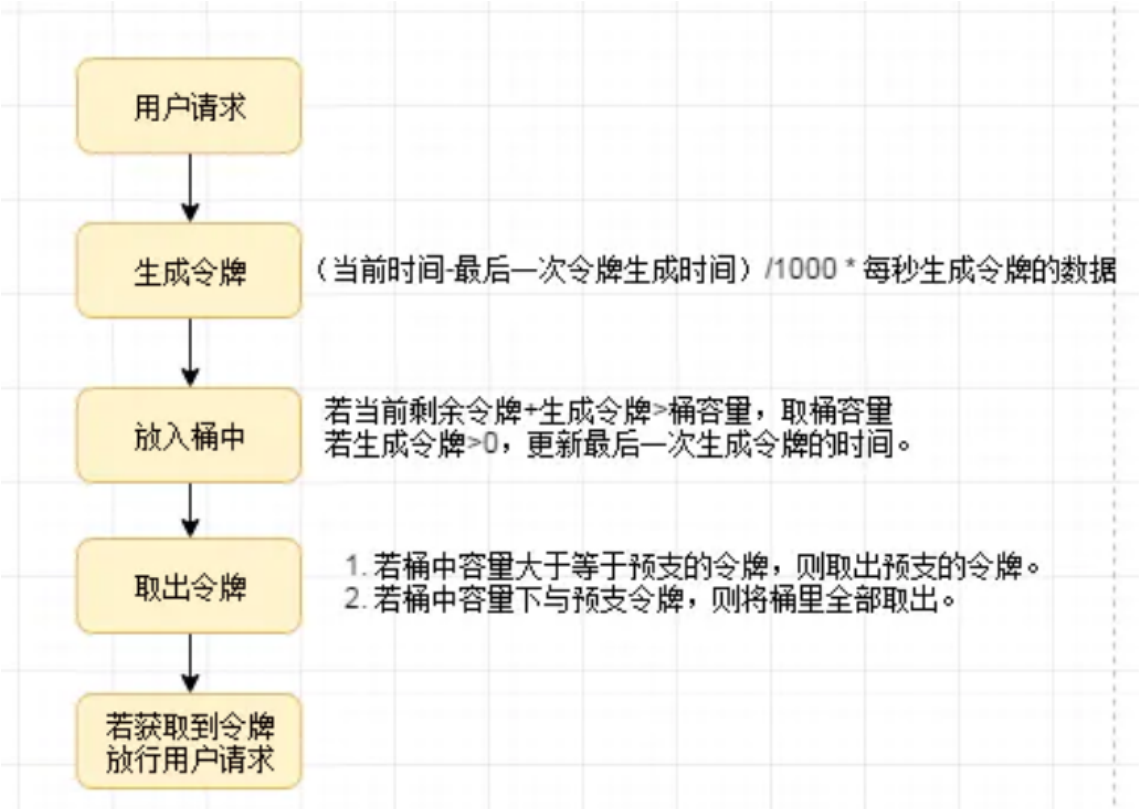
1. 首先设有一个令牌桶，桶内存放令牌，一开始令牌桶内的令牌是满的（桶内令牌的数量可根据服务器情况设定）；
2. 每次访问从桶内取走一个令牌，当桶内令牌为0，则不允许再访问;
3. 每隔一段时间，再放入令牌，最多使桶内令牌满额。（可以根据实际情况，每隔一段时间放入若干个令牌，或直接补满令牌桶）；

我们可以使用redis的队列作为令牌桶容器使用，使用lPush（入队），rPop（出队），实现令牌加入与消耗的操作。



有一个容量有限的桶，令牌以固定的速率添加到这个桶里面。由于桶的容量是有限的，所以不可能无限制的往里面添加令牌，如果令牌到达桶的时候，桶是满的，那么这个令牌就被抛弃了。每次请求，n个数量的令牌从桶里面被移除，如果桶里面的令牌数少于n，那么该请求就会被拒绝。





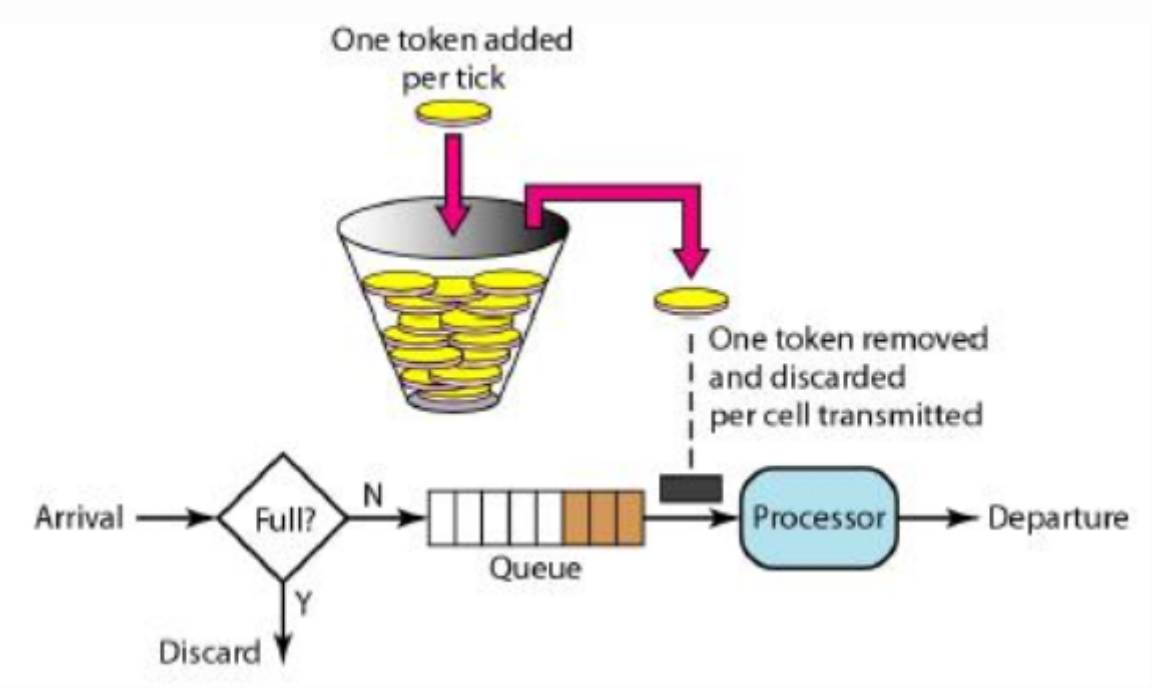
比如说你想把API的速率限制到每分钟20次，同时允许偶尔的流量爆发，最大不超过每分钟50次。那么你可以使用一个Key-Value的内存数据存储，例如Redis，然后：

1. 用户的第一次请求时，初始化一个桶，它能装50个令牌，把请求的时间戳和令牌的数量存在Redis里，使用用户ID作为Key。
2. 在后续的请求里，按照预定的固定速率和上次请求后经过的时间向桶里面填充新的令牌。
3. 然后从桶里面删除一个令牌，并把时间戳更新为现在的时间。
4. 最后，如果可用的令牌数为0，那么拒绝请求。

注意：

通过用户ID作为key，是因为限流是对不同用户进行的，也是对每个用户的请求频率来进行控制。通过用户ID就可以区分不同的用户，采取不同的限流策略，具体的限流策略是通过令牌数来实施的，因为令牌数直接关系到请求的速率，比如QPS设定为500，即500次请求/s，那么其令牌速率是1个/2ms，即令牌生成的间隔是2ms，如果要再进一步进行降速，比如要降速一倍，那么可以将原本一次取1个令牌消耗的时间，提升为取2个令牌，这样的话取令牌的时间消耗就翻倍了，间接实现了限流。

而记录每次请求的时间戳，是为了下次取的时候，可以获取两次时间的时间差，再去除以生成令牌的时间间隔，来补充这个过程要生成的新的令牌数。



令牌桶算法(Token Bucket)和 Leaky Bucket 效果一样但方向相反的算法,更加容易理解.随着时间流逝,系统会按恒定1/QPS时间间隔(如果QPS=100,则间隔是10ms)往桶里加入Token(想象和漏洞漏水相反,有个水龙头在不断的加水),如果桶已经满了就不再加了.新请求来临时,会各自拿走一个Token,如果没有Token可拿了就阻塞或者拒绝服务.

令牌桶的另外一个好处是可以方便的改变速度. **一旦需要提高速率,则按需提高放入桶中的令牌的速率（也即缩短上文中提到的生成令牌的间隔时间）。** 一般会定时(比如100毫秒)往桶中增加一定数量的令牌, 有些变种算法则实时的计算应该增加的令牌的数量.。

【消息与短信项目的限流】

线上是分布式系统，布署了多个节点，而且多个节点最终调用的是同一个短信服务商接口。虽然我们对单个节点能做到将QPS限制在400/s，但是多节点条件下，如果每个节点均是400/s，那么到服务商那边的总请求就是节点数x400/s，于是限流效果失效。使用该方案对单节点的阈值控制是难以适应分布式环境的，至少目前我还没想到更为合适的方式。

因为这两个系统都是高并发系统，通常业务方会在极短的时间内有大量的消息和短信并发请求，此时如果在api-controller层作限流的话，就会影响整个系统的吞吐量，所以我们项目中是采取kafka和rocketMq进行异步地解耦和削峰，收到请求后迅速丢到消息中间件的队列中，然后立马给业务方返回接口成功，继而快速处理大量并发请求，不至于阻塞甚至瘫痪系统，另一方面，消息的消费端我们放在一个业务处理层去进行，该层主要是包含一些和业务相关的几个核心的处理模块，其中就可以包含我们的限流模块，把限流放在这里面来做，可以合理地利用系统的资源限流，把流量限制在渠道方合理安全的范围之内。

所以最终使用redis进行限流，其很好地解决了分布式环境下多实例所导致的并发问题。因为使用redis设置的计时器和计数器均是全局唯一的，不管多少个节点，它们使用的都是同样的计时器和计数器，因此可以做到非常精准的流控。同时，这种方案编码并不复杂，可能需要的代码不超过10行。

【短信项目】

使用 Redis 和 Lua 脚本实现了短信发送的流量控制。短信平台对接的许多第三方渠道都有每秒的请求限制数，有的渠道限制比较宽松，所以我们之前采用了一种比较粗糙的方式，就是起一个线程数少的小线程池，来消费 kafka 中的短信数据，并设置等待时间，以此来限速，但这种方式不够精确，只能是大概不超过第三方渠道限制地数量级，但不能精确到每秒多少个请求，这样的话，相当于其实还有很大的提

速空间。所以后面为了精准地控制短信平台发请求的流量，基于 Redis 的 hash 结构来实现一个限速器。

这个模型简单来说就是用哈希数据结构来充当一个令牌桶容器，容器中预先放置一定数量的令牌，当请求到来时，先在桶中获取一个令牌，如果是一批请求的话，就获取n个令牌，只有获取到令牌的请求才可以继续执行，否则就等待直到有令牌生成为止，这里没有采取直接拒绝的策略，而是选择阻塞继续等待有令牌生成。令牌的生成是在每次要获取令牌时，计算一下从上一次获取到当前获取的时间间隔，再乘以令牌的生成速率，就计算出此次需要补充的令牌数，先补充，再获取。

比如渠道方的流量控制是500QPS，即500次请求/s，那么其令牌速率是1个/2ms，即令牌生成的间隔是2ms。那么现在来一个请求，通过计算出一个当前时间（redis），减去上次获取令牌的时间，得到前后两次获取令牌的间隔是20ms，那么此时需要补充的令牌数就是20ms*令牌生成速度是2ms每个=10个，当然如果此时令牌桶的最大容量是5个的话，那么此时就是取桶的最大容量5，因为超过的令牌桶是要被溢出掉的，因为桶的最大容量代表是一个限速的极限值。

此时因为桶中还有10个，那么该请求就放行了，如果在这10ms内陆续来了100个请求，即请求的速度远远大于令牌生成的速度，那么前10个拿到令牌的请求放行，后面90个就要继续等待直到有令牌生成，再放行。这就实现了令牌桶，这种方式的好处还在于可以随时通过修改令牌的生成速度来调节限流桶的速度。

而采用 Lua 脚本是因为短信发送模块是分布式多实例部署的，在令牌的获取和补充的过程中，会存在并发问题，而 Lua 脚本可以保证这两个操作的原子性，所以把令牌的 get 和 set 放到脚本中执行。

主要步骤：

- 初始化令牌桶容器，主要参数是令牌桶最大容量、当前容量、令牌生成速率、最后一次获取令牌的时间
- 当请求到来时，生成当前时间（redis 服务器时间），获取上次取牌时间（即最后一次获取令牌的时间）计算得一个间隔时间，乘以令牌生成速率得到要补充到桶中的令牌数，然后基于 Lua 脚本的 pcall 命令执行 HSET 更新当前令牌数
- 开始获取令牌，基于 Lua 脚本执行 HGET，判断令牌是否充足，如果充足则返回 true，请求放行；如果令牌不充足，则计算需要等待令牌的时间，然后让当前线程进入 sleep，阻塞等待令牌

【实现】

根据令牌桶算法，桶中的令牌是持续生成存放的，有请求时需要先从桶中拿到令牌才能开始执行，谁来持续生成令牌存放呢？

延迟计算，如上resync函数。该函数会在每次获取令牌之前调用，其实现思路为，若当前时间晚于nextFreeTicketMicros，则计算该段时间内可以生成多少令牌，将生成的令牌加入令牌桶中并更新数据。这样一来，只需要在获取令牌时计算一次即可。

使用redis实现的原因

- redis作为高性能缓存系统，性能上能够满足多机之间高并发访问的要求
- redis有比较好的api来支持限流器令牌桶算法的实现
- 对于我们的系统来说，通过spring data redis来操作比较简单和常见，避免了引入新的中间件带来的风险

但是我们也知道，限流器在每次请求令牌和放入令牌操作中，存在一个协同的问题，即获取令牌操作要尽可能保证原子性，否则无法保证限流器是否能正常工作。在RateLimiter的实现中使用了mutex作为互斥锁来保证操作的原子性，那么在redis中就需要一个类似于事务的机制来保证获取令牌中多重操作的原子性。 **面对这样的需求，我们有几个选择：**

- 用redis实现分布式锁来保证操作的原子性，这个方案实现起来应该比较简单，分布式锁有现成的例子，然后就是把Rate Limiter的代码套用分布式锁就行了，这样的话效率会显得不太高，特别是在大量访问的情况下。
- 用redis的transaction，在我查阅redis官方文档和stackoverflow之后发现redis的transaction官方并不推荐，并且有可能在未来取消事务，因此不可取。
- **通过将获取锁封装到lua脚本中，提交给redis进行eval和evalsha操作来完成lua脚本的执行，由于lua脚本在redis中天然的原子性**，我们的需求能够比较好的满足，问题是将业务逻辑封装在lua中，对于开发人员自身的能力和调试存在一定的问题。

经过权衡，我采用了第四种方式，通过redis和lua来编写令牌桶算法来完成分布式限流的需求。

【Lua脚本】

```
-- 返回码 1:操作成功 0:未配置 -1: 获取失败 -2:修改错误，建议重新初始化 -500:不支持的操作
-- redis hashmap 中存放的内容:
-- last_mill_second 上次放入令牌或者初始化的时间
-- stored_permits 目前令牌桶中的令牌数量
-- max_permits 令牌桶容量
-- interval 放令牌间隔
-- app 一个标志位，表示对于当前key有没有限流存在

local SUCCESS = 1
local NO_LIMIT = 0
local ACQUIRE_FAIL = -1
local MODIFY_ERROR = -2
local UNSUPPORT_METHOD = -500

local ratelimit_info = redis.pcall("HMGET", KEYS[1], "last_mill_second",
"stored_permits", "max_permits", "interval", "app")
local last_mill_second = ratelimit_info[1]
local stored_permits = tonumber(ratelimit_info[2])
local max_permits = tonumber(ratelimit_info[3])
local interval = tonumber(ratelimit_info[4])
local app = ratelimit_info[5]

local method = ARGV[1]

--获取当前毫秒
--考虑主从策略和脚本回放机制，这个time由客户端获取传入
--local curr_time_arr = redis.call('TIME')
--local curr_timestamp = curr_time_arr[1] * 1000 + curr_time_arr[2]/1000
local curr_timestamp = tonumber(ARGV[2])


-- 当前方法为初始化
if method == 'init' then
    --如果app不为null说明已经初始化过，不要重复初始化
    if(type(app) ~='boolean' and app ~=nil) then
        return SUCCESS
    end

    redis.pcall("HMSET", KEYS[1],
        "last_mill_second", curr_timestamp,
        "stored_permits", ARGV[3],
        "max_permits", ARGV[4],
        "interval", ARGV[5],
        "app", ARGV[6])
```

```
    --始终返回成功
    return SUCCESS
end

-- 当前方法为修改配置
if method == "modify" then
    if(type(app) =='boolean' or app ==nil) then
        return MODIFY_ERROR
    end
    --只能修改max_permits和interval
    redis.pcall("HMSET", KEYS[1],
        "max_permits", ARGV[3],
        "interval", ARGV[4])

    return SUCCESS

end

-- 当前方法为删除
if method == "delete" then
    --已经清除完毕
    if(type(app) =='boolean' or app ==nil) then
        return SUCCESS
    end
    redis.pcall("DEL", KEYS[1])
    return SUCCESS
end

-- 尝试获取permits
if method == "acquire" then
    -- 如果app为null说明没有对这个进行任何配置，返回0代表不限流
    if(type(app) =='boolean' or app ==nil) then
        return NO_LIMIT
    end
    --需要获取令牌数量
    local acquire_permits = tonumber(ARGV[3])
    --计算上一次放令牌到现在的时间间隔中，一共应该放入多少令牌
    local reserve_permits = math.max(0, math.floor((curr_timestamp -
last_mill_second) / interval))

    local new_permits = math.min(max_permits, stored_permits + reserve_permits)
    local result = ACQUIRE_FAIL
    --如果桶中令牌数量够则放行
    if new_permits >= acquire_permits then
        result = SUCCESS
        new_permits = new_permits - acquire_permits
    end
    --更新当前桶中的令牌数量
    redis.pcall("HSET", KEYS[1], "stored_permits", new_permits)
    --如果这次有放入令牌，则更新时间
    if reserve_permits > 0 then
        redis.pcall("HSET", KEYS[1], "last_mill_second", curr_timestamp)
    end
    return result
end


return UNSUPPORT_METHOD
```

我在这个脚本中编写了4种函数：

- init 初始化限流器
- modify 修改限流器配置（主要针对限流器的桶大小和放令牌间隔，即1/QPS）
- delete 删除限流器配置
- acquire 尝试获取制定数目的令牌

代码基本上仿照了Guava RateLimiter的逻辑，实现了触发式的放令牌策略。

【java集成】

在完成了lua的调试工作之后，我们就开始java部分的集成代码编写，我们使用的是spring boot来完成开发。 第一部分是redis配置：

```
    @Bean("rateLimitLua")
    public DefaultRedisScript<Long> getRateLimitScript() {
        DefaultRedisScript<Long> rateLimitLua = new DefaultRedisScript<>();
        rateLimitLua.setLocation(new ClassPathResource("rate_limit.lua"));
        rateLimitLua.setResultType(Long.class);
        return rateLimitLua;
    }
```

然后是一些与lua适配的枚举和一些bean：

```
/**
 * @author: Yuanqing Luo
 * @date: 2018/10/22
 *
 * 限流的具体方法
 */
public enum RateLimitMethod {

    //initialize rate limiter
    init,

    //modify rate limiter parameter
    modify,

    //delete rate limiter
    delete,

    //acquire permits
    acquire;
}
/**
 * @author: Yuanqing Luo
 * @date: 2018/10/22
 * rate limite result
 **/
public enum RateLimitResult {

    SUCCESS(1L),
    NO_LIMIT(0L),
    ACQUIRE_FAIL(-1L),
    MODIFY_ERROR(-2L),
```

```
    UNSUPPORT_METHOD(-500L),
    ERROR(-505L);

    private Long code;

    RateLimitResult(Long code){
        this.code = code;
    }

    public static RateLimitResult getResult(Long code){
        for(RateLimitResult enums: RateLimitResult.values()){
            if(enums.code.equals(code)){
                return enums;
            }
        }
        throw new IllegalArgumentException("unknown rate limit return code:" +
code);
    }
}
/**
 * @author: Yuanqing Luo
 * @date: 2018/10/22
 **/
@Getter
@Setter
public class RateLimitVo {

    private String url;

    private boolean isLimit;

    private Double interval;

    private Integer maxPermits;

    private Integer initialPermits;

}
```

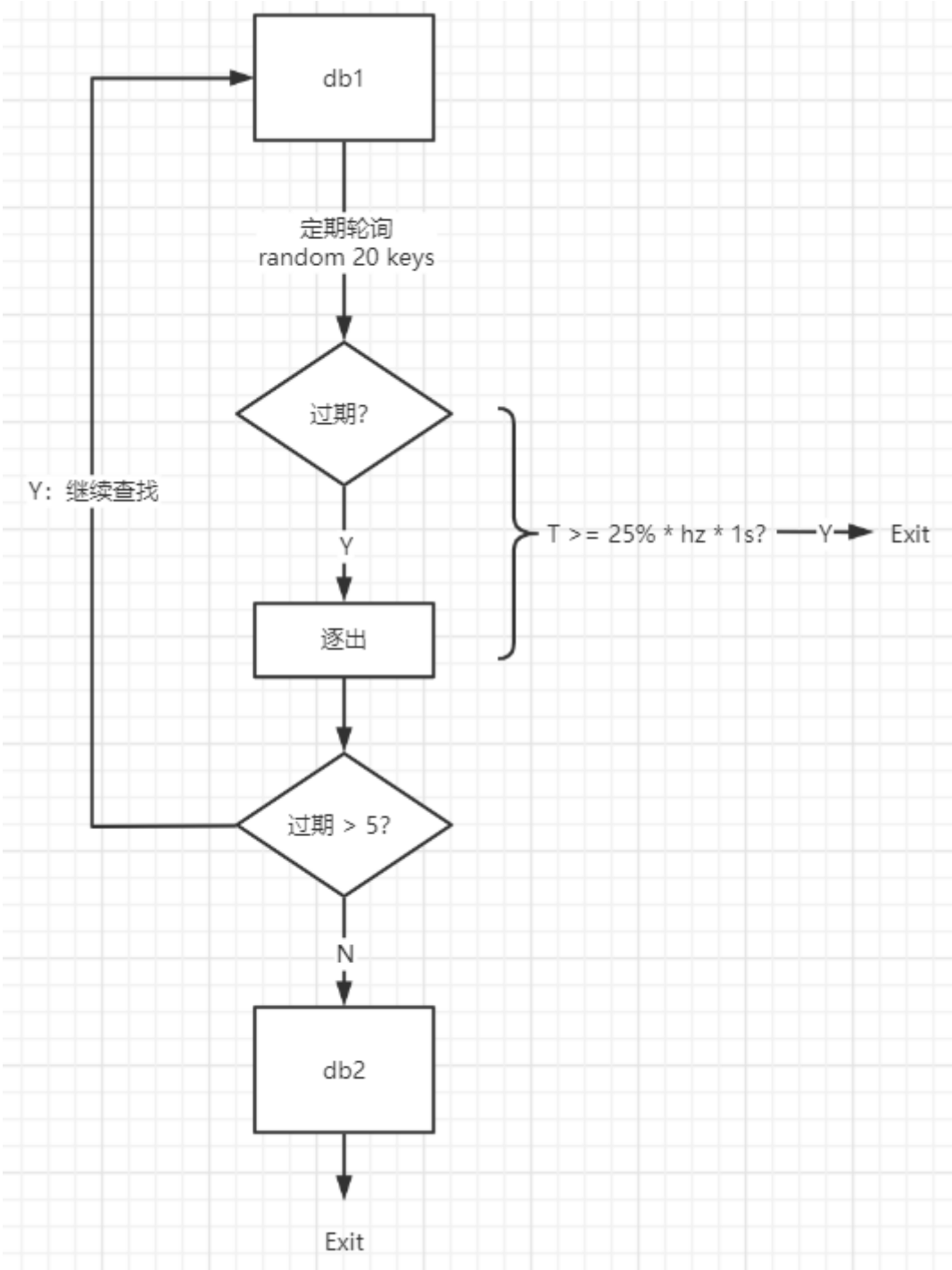
第三部分就是限流器的调用组装部分：

```
/**
 * @author: Yuanqing Luo
 * @date: 2018/10/22
 **/
@Service
@Slf4j
public class RateLimitClient {

    private static final String RATE_LIMIT_PREFIX = "ratelimit:";

    @Autowired
    StringRedisTemplate redisTemplate;

    @Resource
    @Qualifier("rateLimitLua")
    RedisScript<Long> rateLimitScript;
```

```java
    public RateLimitResult init(String key, RateLimitVo rateLimitInfo){
        return exec(key, RateLimitMethod.init,
                rateLimitInfo.getInitialPermits(),
                rateLimitInfo.getMaxPermits(),
                rateLimitInfo.getInterval(),
                key);
    }

    public RateLimitResult modify(String key, RateLimitVo rateLimitInfo){
        return exec(key, RateLimitMethod.modify, key,
                rateLimitInfo.getMaxPermits(),
                rateLimitInfo.getInterval());
    }

    public RateLimitResult delete(String key){
        return exec(key, RateLimitMethod.delete);
    }

    public RateLimitResult acquire(String key){
        return acquire(key, 1);
    }

    public RateLimitResult acquire(String key, Integer permits){
        return exec(key, RateLimitMethod.acquire, permits);
    }

    /**
     * 执行redis的具体方法，限制method,保证没有其他的东西进来
     * @param key
     * @param method
     * @param params
     * @return
     */
    private RateLimitResult exec(String key, RateLimitMethod method, Object...
params){
        try {
            Long timestamp = getRedisTimestamp();
            String[] allParams = new String[params.length + 2];
            allParams[0] = method.name();
            allParams[1] = timestamp.toString();
            for(int index = 0;index < params.length; index++){
                allParams[2 + index] = params[index].toString();
            }
            Long result = redisTemplate.execute(rateLimitScript,
                    Collections.singletonList(getKey(key)),
                    allParams);
            return RateLimitResult.getResult(result);
        } catch (Exception e){
            log.error("execute redis script fail, key:{}, method:{}",
                    key, method.name(), e);
            return RateLimitResult.ERROR;
        }
    }

    private Long getRedisTimestamp(){
        Long currMillSecond = redisTemplate.execute(
                (RedisCallback<Long>) redisConnection -> redisConnection.time()
```

```
        );
        return currMillSecond;
    }
    private String getKey(String key){
        return RATE_LIMIT_PREFIX + key;
    }
}
```

11、过期与内存淘汰策略



【过期策略】

Redis的过期策略，是有**定期清除+惰性清除**两种。

定期清除：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是后两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

惰性清除：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。

定时清除：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。

【定期清除流程】

每隔一段时间就逐个遍历 Redis 中16个数据库，先随机取出一定数量的 key 比如20个，然后逐个判断是否过期，如果过期了则从内存删除，在这20个 key 的遍历过程中，如果遍历时间超过了某个预设值的四分之一，就结束此次定期清除过程。某个预设值是 CPU 每秒执行任务的次数，之所以这样做，是为了避免定期清除占用太多的 CPU 资源而影响到正常的业务。

如果都没有超过 CPU 预设时间的话，遍历完20个 key 之后，再统计一下一共有多少个 key 被逐出了，如果超过一定的 key 被逐出，说明此次随机抽样的结果其过期的比重较大，说明数据库有较大概率还存在较多过期 key，比如20个里面有5个被逐出，那么将对当前的数据库再执行一遍遍历和清除，如果没有超过，则遍历下个数据库，以此类推。

【内存淘汰策略】

noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。

allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。

allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。

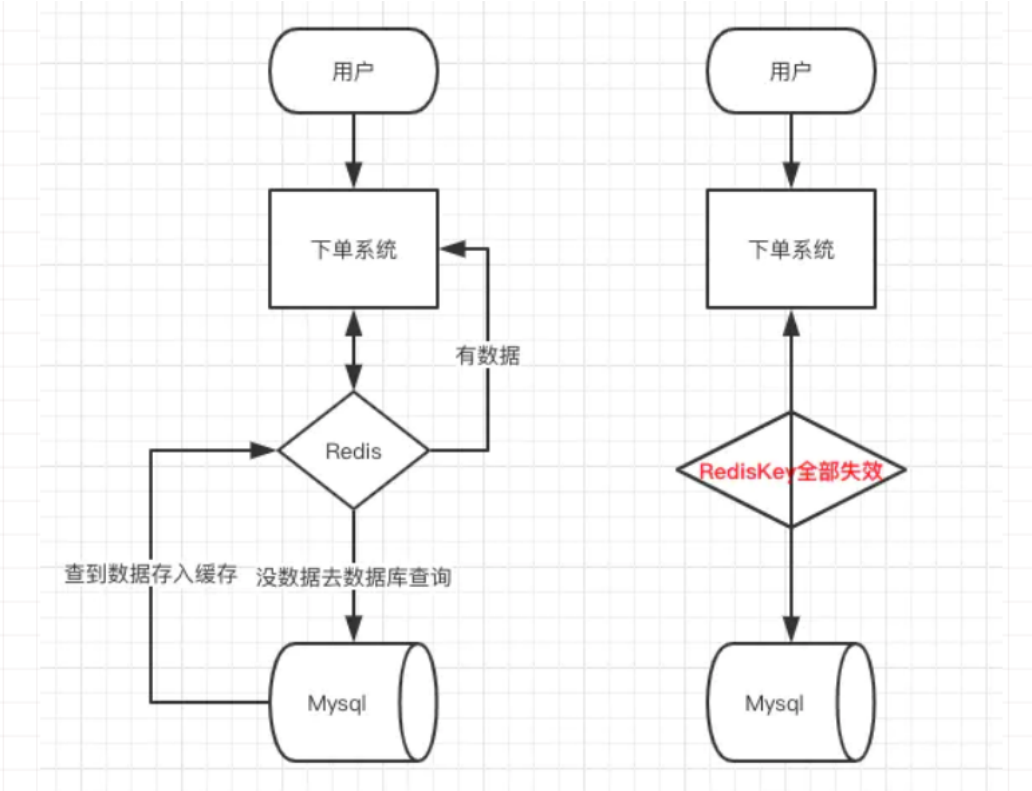
volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。

volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。

volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

12、缓存雪崩、穿透、击穿

【缓存雪崩】



概述

大量的缓存由于过期时间设置得太过紧凑，导致在同一时间都失效了，就是缓存雪崩。其结果可能会造成失效的时候大量的请求直接打到数据库，让数据库负载过重而奔溃。如果有其他服务依赖了这个库，同时又没有做限流熔断或者降级，则会造成更多的级联反应导致所有服务奔溃。

解决

在批量往Redis存数据的时候，把每个Key的失效时间都加个随机值就好了，这样可以保证数据不会在同一时间大面积失效。

```
setRedis（Key，value，time + Math.random() * 10000）；
```

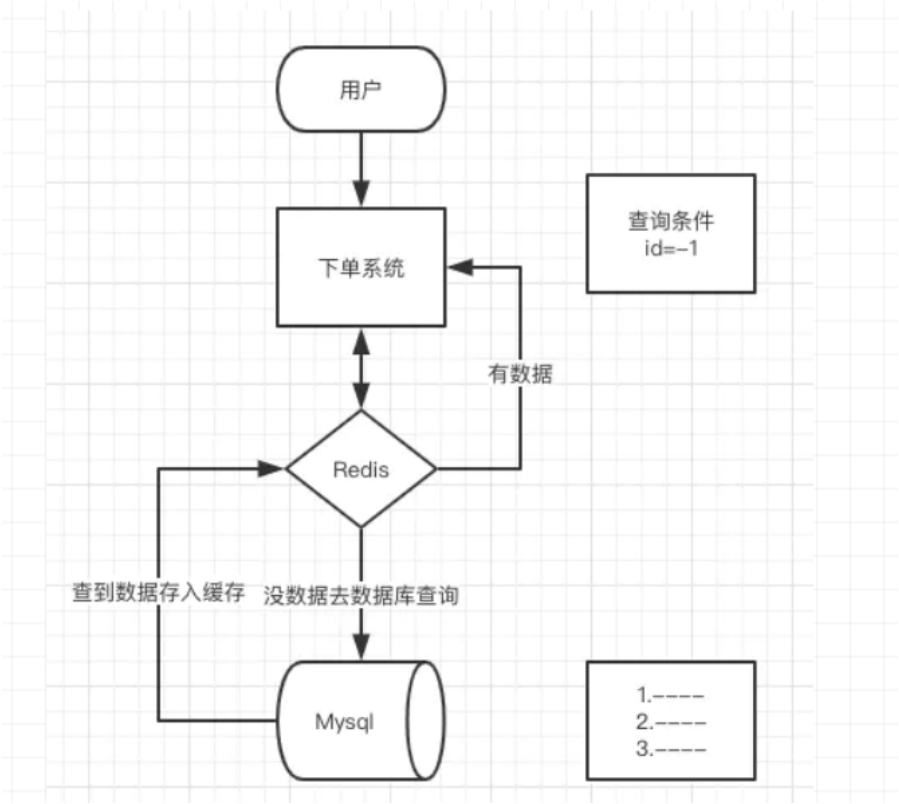
或者在 Redis 容量还够的时候，设置热点数据永远不过期，有更新操作就更新缓存就好了。

如果Redis是集群部署，将热点数据均匀分布在不同的Redis库中也能避免全部失效的问题，

举例

如果所有首页的Key失效时间都是12小时，中午12点刷新的，我零点有个秒杀活动大量用户涌入，假设当时每秒 6000 个请求，本来缓存在可以扛住每秒 5000 个请求，但是缓存当时所有的Key都失效了。此时 1 秒 6000 个请求全部落数据库，数据库必然扛不住，它会报一下警，真实情况可能DBA都没反应过来就直接挂了。此时，如果没用什么特别的方案来处理这个故障，DBA 很着急，重启数据库，但是数据库立马又被新的流量给打死了。这就是我理解的缓存雪崩。

【缓存穿透】



概述

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，这时的用户很可能是攻击者，制造一些非法数据，但这些数据又可以避开系统的检查，攻击会导致数据穿过缓存，直接到达数据库，导致数据库压力过大而奔溃。一般是由于参数校验没有做到位引起的。

解决

在接口层增加校验，比如用户鉴权校验，参数做校验，不合法的参数直接代码Return，比如：id 做基础校验，id <=0的直接拦截等。

从缓存取不到的数据，在数据库中也没有取到，这时也可以将对应Key的Value对写为null、位置错误、稍后重试这样的值具体取啥问产品，或者看具体的场景，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。

还有一个高级用法**布隆过滤器（Bloom Filter）**这个也能很好的防止缓存穿透的发生，他的原理也很简单就是利用高效的数据结构和算法快速判断出你这个Key是否在数据库中存在，不存在你return就好了，存在你就去查了DB刷新KV再return。

举例

像这种你如果不对参数做校验，数据库id都是大于0的，我一直用小于0的参数去请求你，每次都能绕开Redis直接打到数据库，数据库也查不到，每次都这样，并发高点就容易崩掉了。

【缓存击穿】

概述

至于**缓存击穿**，这个跟**缓存雪崩**有点像，但是又有一点不一样，缓存雪崩是因为大面积的缓存失效，打崩了DB，而缓存击穿不同的是**缓存击穿**是指一个Key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个Key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个完好无损的桶上凿开了一个洞。

解决

设置热点数据永远不过期。或者加上分布式锁，降低数据库的读写压力。

代码

```java
/**
 * 获取数据
 * @param Key                          查询参数
 * @return data                        数据
 * @throws InterruptedException        异常
 * @author 敖丙
 */
public static String getData(String Key) throws InterruptedException {
    // 从redis查询数据
    String result = getDataByKV(Key);
    // 参数校验
    if (StringUtils.isBlank(result)) {
        // 获取锁
        if (reenLock.tryLock()) {
            // 去数据库查询
            result = getDataByDB(Key);
            // 校验
            if (StringUtils.isNotBlank(result)) {
                // 搞进缓存
                setDataToKV(Key, result);
            }
            // !!!释放锁 正常会在finally里面释放
            reenLock.unLock();
        } else {
            // 睡一会再拿
            Thread.sleep(100L);
            result = getData(Key);
        }
    }
    return result;
}
// 这里面的锁都是单机玩玩，分布式锁还是得靠lua脚本这样的
```

13、其他

1）使用jedisPool的好处：

不适用连接池：当有很多任务需要采用线程执行的时候，而且有时可能会创建很多线程的时候，最好使用下线程池。 不使用线程池的话，所创建的线程数无法控制，比如一下子创建了几百几千个线程，电脑一下子就崩溃了，因为创建销毁线程，消耗资源较多。。 使用连接池：提高效率 创建好一定数量的线程放在池中，等需要使用的时候就从池中拿一个，这要比需要的时候创建一个线程对象要快的多。 方便管理 可以编写线程池管理代码对池中的线程统一进行管理，比如说系统启动时由该程序创建100个线程，每当有请求的时候，就分配一个线程去工作， 如果刚好并发有101个请求，那多出的这一个请求可以排队等候，避免因无休止的创建线程导致系统崩溃

2）zrange是传入的值是索引，因此查询的值“1”代表是查询第二个值，也就是说，索引的情况下，索引0是第一个元素，而zrangebyscore传入的值就是值本身的含义，也就是“score”

参考资料

- Redis数据结构以及应用场景

  https://www.cnblogs.com/sky-chen/p/10166368.html

- Redis五种数据结构

  https://www.cnblogs.com/haoprogrammer/p/11065461.html
- hash类型的应用场景 - Redis实战经验

  https://www.cnblogs.com/pangzizhe/p/10657801.html
- Redis系列十：Pipeline详解

  https://blog.csdn.net/w1lgy/article/details/84455579
- Redis Lua实战

  https://www.jianshu.com/p/366d1b4f0d13

Redis集群

0、前言

Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。

所谓的集群，就是通过添加服务器的数量，提供相同的服务，从而让服务器达到一个稳定、高效的状态。

Redis单点吞吐量

单点TPS达到8万/秒，QPS达到10万/秒，补充下TPS和QPS的概念

1. QPS: 应用系统每秒钟最大能接受的用户访问量。每秒钟处理完请求的次数，注意这里是处理完，具体是指发出请求到服务器处理完成功返回结果。可以理解在server中有个counter，每处理一个请求加1，1秒后counter=QPS。
2. TPS： 每秒钟最大能处理的请求数。每秒钟处理完的事务次数，一个应用系统1s能完成多少事务处理，一个事务在分布式处理中，可能会对应多个请求，对于衡量单个接口服务的处理能力，用QPS比较合理。

Redis Sentinal着眼于高可用，在master宕机时会自动将slave提升为master，继续提供服务。

Redis Cluster着眼于扩展性，在单个redis内存不足时，使用Cluster进行分片存储。

1、分区和槽slot

redis cluster中有一个16384（2^4 * 2^10）长度的槽的概念。通过哈希算法再加上取模运算可以将一个值固定地映射到某个区间，区间由连续的slot组成。

redis cluster采用虚拟槽分区，所有的键根据哈希函数(CRC16[key]&16383)映射到0 - 16383槽内，共16384个槽位，每个节点维护部分槽及槽所映射的键值数据。

哈希函数: Hash()=CRC16[key]&16383 按位与。

redis用虚拟槽分区原因：解耦数据与节点关系，节点自身维护槽映射关系，分布式存储。

比如节点1记录自己负责0-5000个槽位，节点2记录自己负责5000-10000个槽位等，不用记录和数据的关系，解耦了数据和节点的关系。

2、集群高可用

a、一个集群里面有M1、M2、M3三个节点，其中节点 M1包含 0 到 5500号哈希槽，节点M2包含5501 到 11000 号哈希槽，节点M3包含11001 到 16384号哈希槽。如果M2宕掉了，就会导致5501 到 11000 号哈希槽不可用，从而使整个集群不可用。

b、一个集群里面有M1-S1、M2-S2、M3-S3六个主从节点，其中节点 M1包含 0 到 5500号哈希槽，节点M2包含5501 到 11000 号哈希槽，节点M3包含11001 到 16384号哈希槽。如果是M2宕掉，集群便会选举S2为新节点继续服务，整个集群还会正常运行。当M2、S2都宕掉了，这时候集群就不可用了。

所以， redis集群至少需要一个备份节点，才能更好的保证集群的高可用。

3、主从模式

主从模式是三种模式中最简单的，在主从复制中，数据库分为两类：主数据库(master)和从数据库(slave)。

其中主从复制有如下特点：

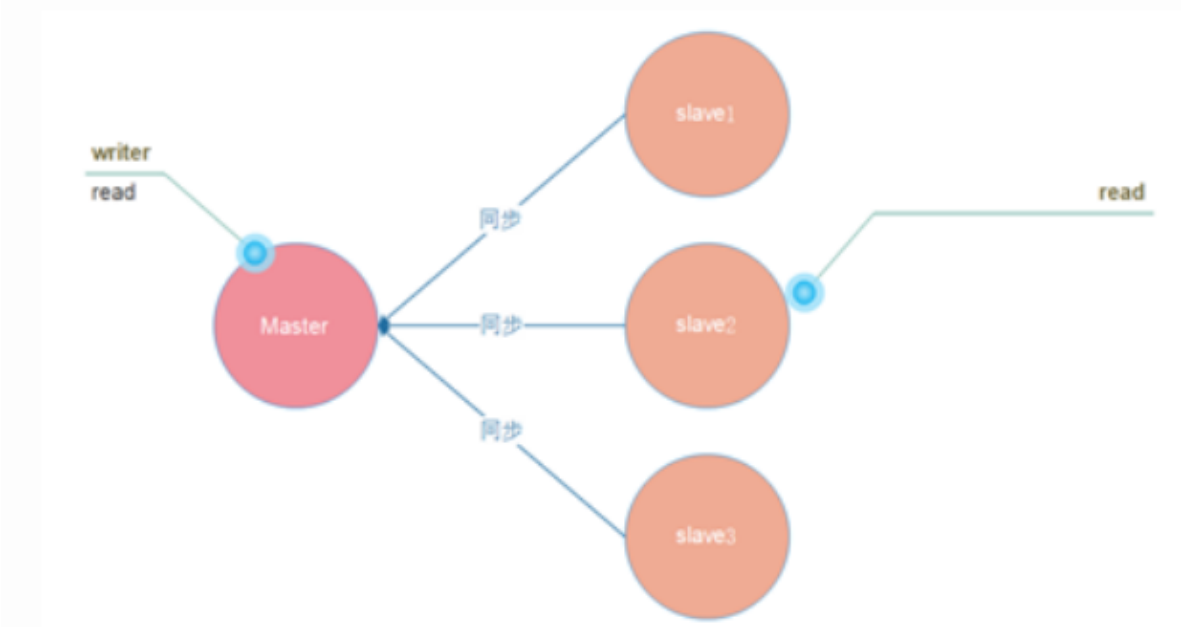
- 主数据库可以进行读写操作，当读写操作导致数据变化时会自动将数据同步给从数据库
- 从数据库一般都是只读的，并且接收主数据库同步过来的数据
- 一个master可以拥有多个slave，但是一个slave只能对应一个master
- slave挂了不影响其他slave的读和master的读和写，重新启动后会将数据从master同步过来
- master挂了以后，不影响slave的读，但redis不再提供写服务，master重启后redis将重新对外提供写服务
- master挂了以后，不会在slave节点中重新选一个master

工作机制：

当slave启动后，主动向master发送SYNC命令。master接收到SYNC命令后在后台保存快照（RDB持久化）和缓存保存快照这段时间的命令，然后将保存的快照文件和缓存的命令发送给slave。slave接收到快照文件和命令后加载快照文件和缓存的执行命令。

复制初始化后，master每次接收到的写命令都会同步发送给slave，保证主从数据一致性。

只要网络连接正常，Master会一直将自己的数据更新同步给Slaves，保持主从同步。



因此，主从模型可以提高**读的能力**，在一定程度上缓解了写的压力。因为能写仍然只有Master节点一个，可以将读的操作全部移交到从节点上，变相提高了写能力。

安全设置：

当master节点设置密码后，

- 客户端访问master需要密码
- 启动slave需要密码，在配置文件中配置即可
- 客户端访问slave不需要密码

缺点：

从上面可以看出，master节点在主从模式中唯一，若master挂掉，则redis无法对外提供写服务。

环境准备：

```
master节点                  192.168.30.128

slave节点                   192.168.30.129

slave节点                   192.168.30.130
```

修改配置（129、130省略）：

```
# mkdir -p /data/redis

# vim /usr/local/redis/redis.conf

bind 192.168.30.128                #监听ip，多个ip用空格分隔
daemonize yes               #允许后台启动
logfile "/usr/local/redis/redis.log"                #日志路径
dir /data/redis                 #数据库备份文件存放目录
masterauth 123456               #slave连接master密码，master可省略
requirepass 123456              #设置master连接密码，slave可省略

appendonly yes                  #在/data/redis/目录生成appendonly.aof文件，将每一次写操作请求都追加到appendonly.a

# echo 'vm.overcommit_memory=1' >> /etc/sysctl.conf

# sysctl -p
```

查看集群状态：

```
# redis-cli -h 192.168.30.128 -a 123456
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe

192.168.30.128:6379> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=192.168.30.129,port=6379,state=online,offset=168,lag=1
slave1:ip=192.168.30.130,port=6379,state=online,offset=168,lag=1
master_replid:fb4941e02d5032ad74c6e2383211fc58963dbe90
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:168
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:168
```

在master节点写入的数据，很快就同步到slave节点上，而且在slave节点上无法写入数据。

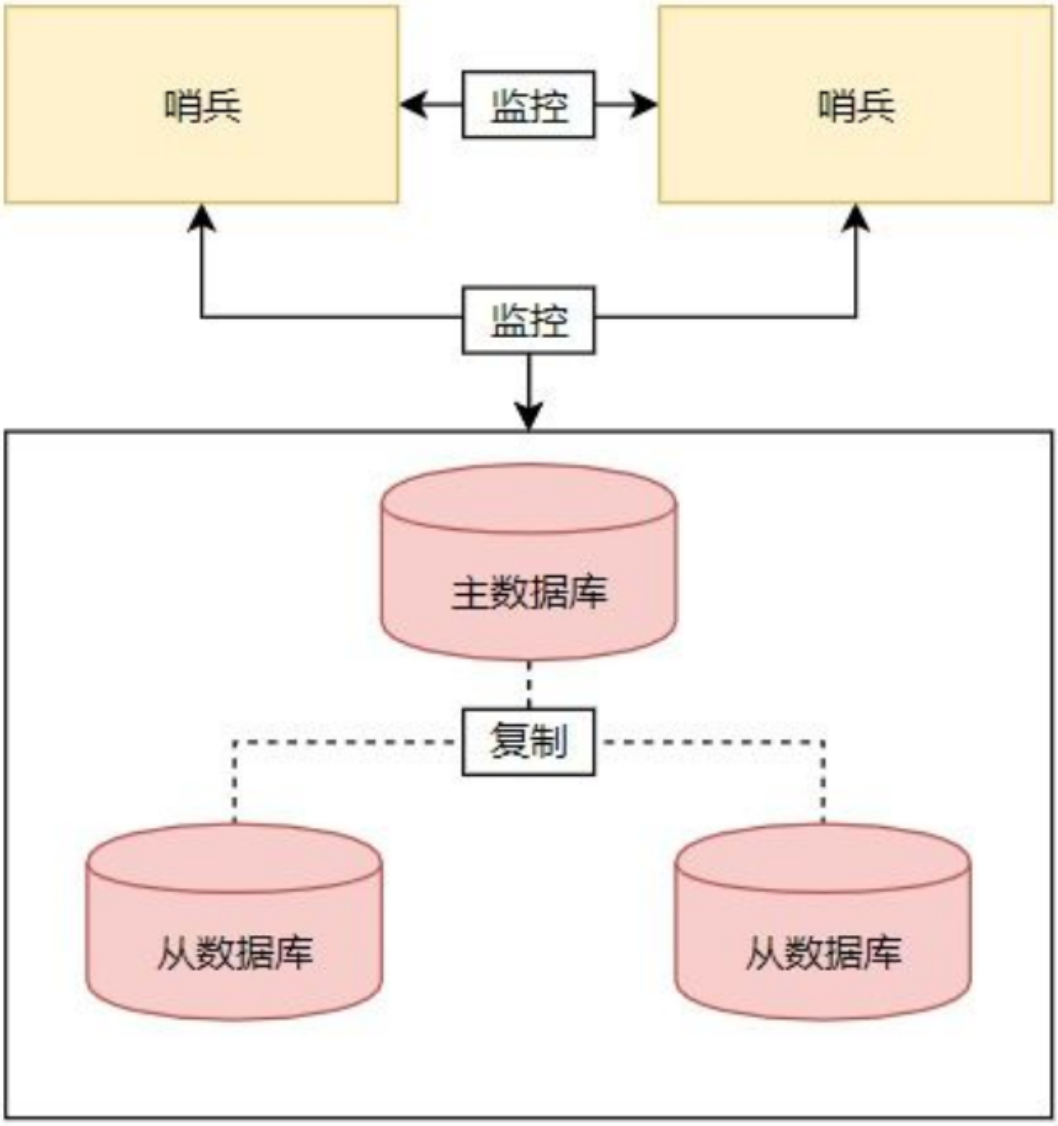
4、Sentinel模式

主从模式的弊端就是不具备高可用性，当master挂掉以后，Redis将不能再对外提供写入操作，因此sentinel应运而生。

哨兵的任务：

Redis 的 Sentinel 系统用于管理多个 Redis 服务器（instance）， 该系统执行以下三个任务：

- 监控（Monitoring）： Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。
- 提醒（Notification）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
- 自动故障迁移（Automatic failover）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作， 它会进行选举，将其中一个从服务器升级为新的主服务器， 并让失效主服务器的其他从服务器改为复制新的主服务器； 当客户端试图连接失效的主服务器时， 集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器。



sentinel中文含义为哨兵，顾名思义，它的作用就是监控redis集群的运行状况，**特点如下：**

- sentinel模式是建立在主从模式的基础上，如果只有一个Redis节点，sentinel就没有任何意义
- 当master挂了以后，sentinel会在slave中选择一个做为master，并修改它们的配置文件，其他slave的配置文件也会被修改，比如slaveof属性会指向新的master
- 当master重新启动后，它将不再是master而是做为slave接收新的master的同步数据
- sentinel因为也是一个进程有挂掉的可能，所以sentinel也会启动多个形成一个sentinel集群
- 多sentinel配置的时候，sentinel之间也会自动监控
- 当主从模式配置密码时，sentinel也会同步将配置信息修改到配置文件中，不需要担心
- 一个sentinel或sentinel集群可以管理多个主从Redis，多个sentinel也可以监控同一个redis
- sentinel最好不要和Redis部署在同一台机器，不然Redis的服务器挂了以后，sentinel也挂了

当使用sentinel模式的时候，客户端就不要直接连接Redis，而是连接sentinel的ip和port，由sentinel来提供具体的可提供服务的Redis实现，这样当master节点挂掉以后，sentinel就会感知并将新的master节点提供给使用者。

工作机制：

哨兵启动后，会与要监控的主数据库建立两条连接：

1. 一条用来用来订阅**sentinel**:hello频道以获取其他哨兵节点的信息；
2. 另一条用来定期向主数据库发送INFO等命令来获取主数据库本身的信息；

在和主数据库建立连接后，哨兵会定时执行下面3个操作：

1. 每10秒哨兵会向主数据库和从数据库发送INFO命令；
2. 每2秒哨兵会向主数据库和从数据库的**sentinel**:hello频道发送自己的信息；
3. 每1秒哨兵会向主数据库和从数据库和其他哨兵发送PING命令；

第一个操作是发送INFO命令，目的是获取主数据库的信息，以及主数据库的从数据库的信息，从而实现新节点的自动发现，并对从数据库也建立两条连接。

第二个操作是订阅**sentinel**:hello频道，并发送哨兵本身的信息，与同样监控该数据库的其他哨兵分享自己的信息，同时也能识别哨兵是否是新哨兵。哨兵与哨兵之间也会建立一个链接，用来发送PING命令，用于哨兵发现；

第三个操作是发送PING命令，在发现了从数据库和其他哨兵后，要做的就是定时监控Redis服务是否停止，时间间隔与配置文件中的down-after-milliseconds有关，当这个值小于1秒时，哨兵会每隔该值的时间发送PING命令，当这个值大于1秒时，哨兵会每隔1秒发送一次PING命令。

配置方式是在sentinel.conf文件中加入：

```
sentinel down-after-milliseconds mymaster 600 # 600毫秒发送一个PING
```

当超过down-after-milliseconds时，如果PING的数据库未回复，则哨兵认为其**主观下线**。主观下线可以理解为当前的哨兵认为该节点下线了。

如果该节点是主数据库，则哨兵们会进一步判断是否需要对其进行故障修复：

哨兵会发送SENTINEL is-master-down-by-addr命令询问其他哨兵，判断他们是否也认为该主数据库下线，如果达到quorum参数，也就是我们在配置哨兵时的命令：

```
sentinel monitor mymaster 192.168.2.101 6379 3
```

的最后一个参数3，哨兵们会认为这个主数据库**客观下线**，并选举一个领头哨兵对主从系统发起故障恢复。

领头哨兵选举：

要进行故障恢复，则需要选举出一个领头哨兵，领头哨兵的选择算法是Raft算法，具体过程如下：

发现主数据库客观下线的哨兵节点（A节点）想每个哨兵节点发送命令，要求对方选择自己成为领头哨兵；

如果目标哨兵节点没有选择过其他人，则会同意将A设置成领头哨兵；

如果A发现超过半数且超过quorum参数个哨兵节点同意选择自己，则A成功成为领头哨兵；

当有多个哨兵同时参选，则会出现没有任何节点当选的可能，此时每个参选节点将等待一个随即时间重新发起竞选，直到选举成功。

故障恢复：

选择出领头哨兵后，会把从数据库中的一个挑选出来升级为主数据库：

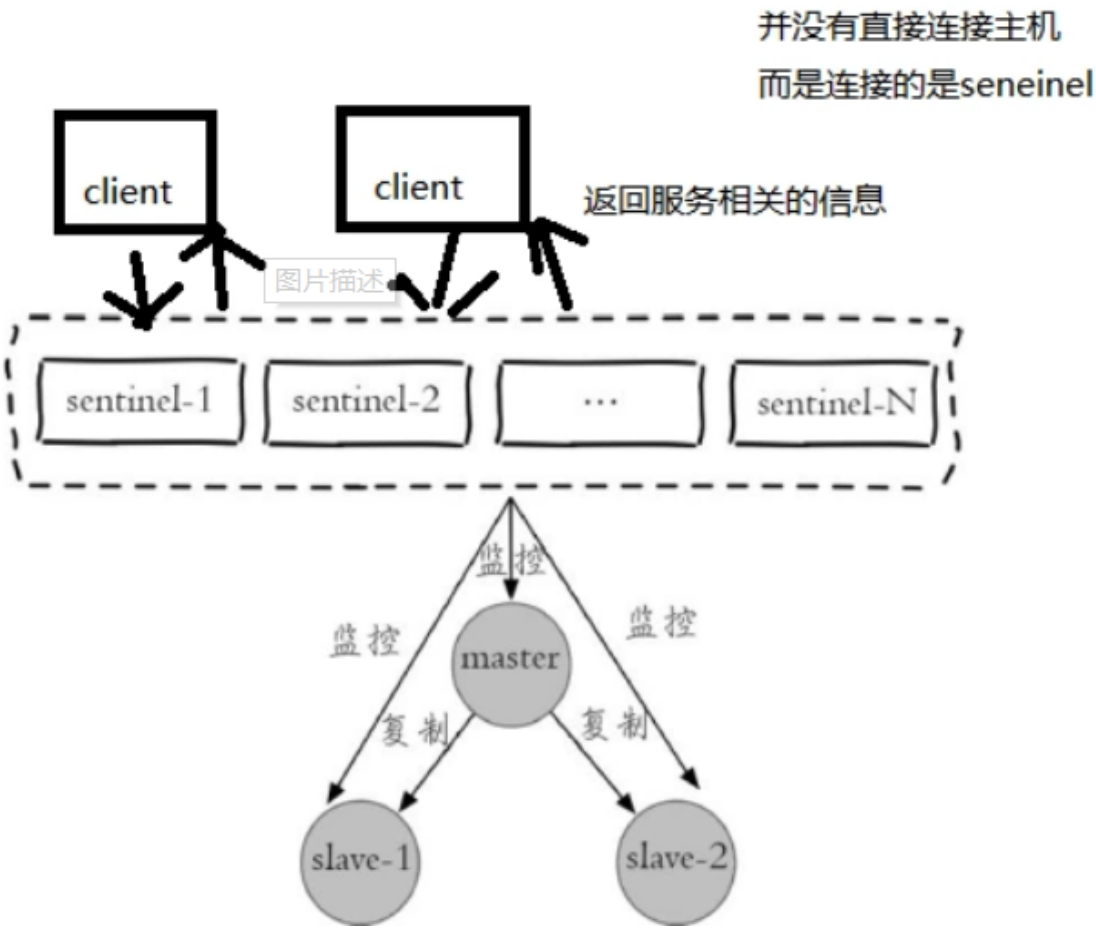
1. 所有先线的从数据库中，选择**优先级**最高的，优先级可以通过slave-priority来设置；
2. 如果有多个一样优先级的从数据库，则复制的命令**偏移量**越大，越优先（与down掉的主数据库最接近，数据量越接近）；
3. 如果还有多个备选，则选择**运行ID较小**的（运行ID不会重复）；

选择好节点后，领头哨兵将想这个节点发送slaveof no one，升级他为主数据库。

然后想其他从数据库发送slaveof命令切换主数据库。

最后更新内部的记录，将已经停止服务的旧的主数据库更新为新的主数据库的从数据库，当其回复后自动以从数据库的身份加入到主从架构中。

哨兵网络：



故障转移：

- **投票（半数原则）**

当任何一个Sentinel发现被监控的Master下线时，会通知其它的Sentinel开会，投票确定该Master是否下线（半数以上，所以sentinel通常配奇数个）。

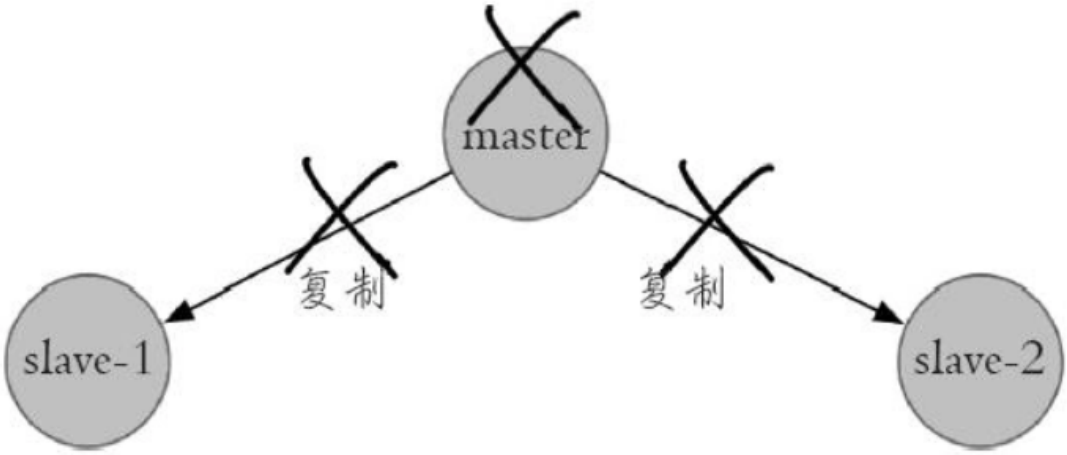
- **选举**

（2）选举

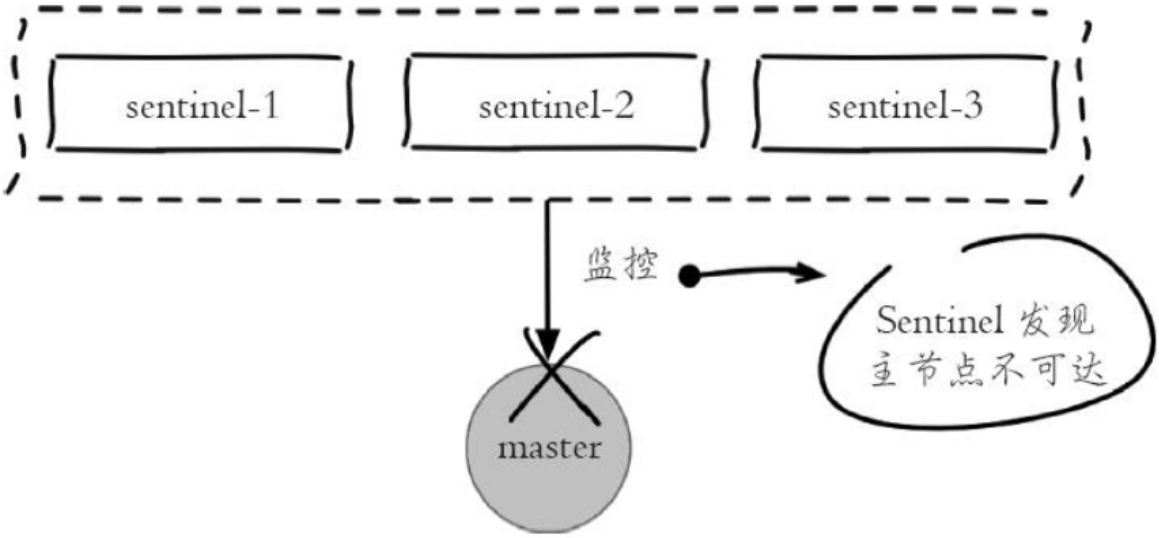
当Sentinel确定Master下线后，会在所有的Slaves中，选举一个新的节点，升级成Master节点。其它Slaves节点，转为该节点的从节点。

过程：

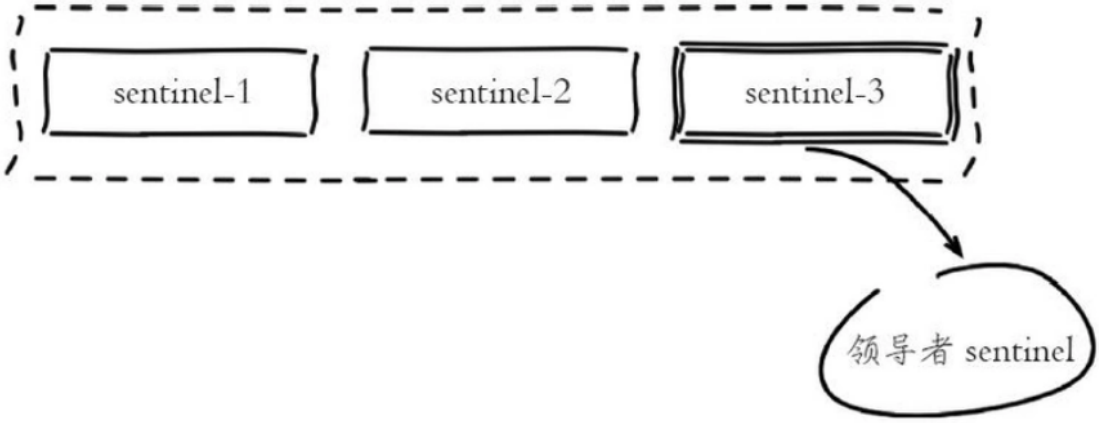
- 主节点出现故障，此时两个从节点与主节点失去连接，主从复制失败



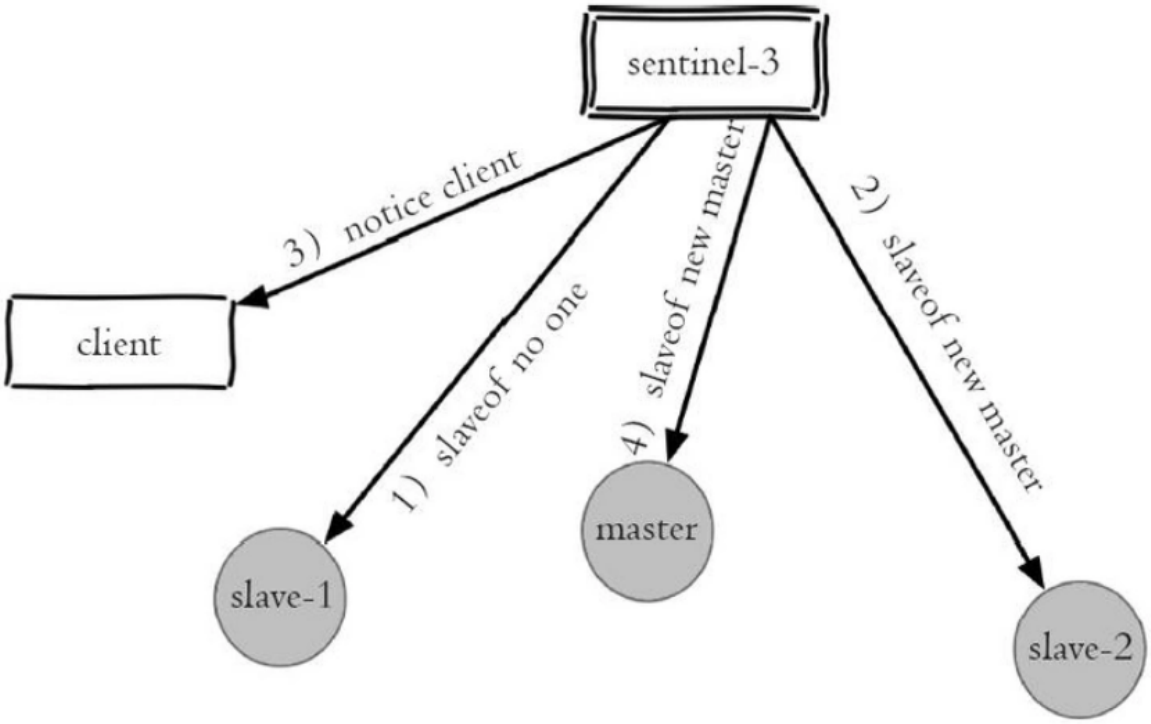
- 每个 Sentinel 节点通过定期监控发现主节点出现了故障



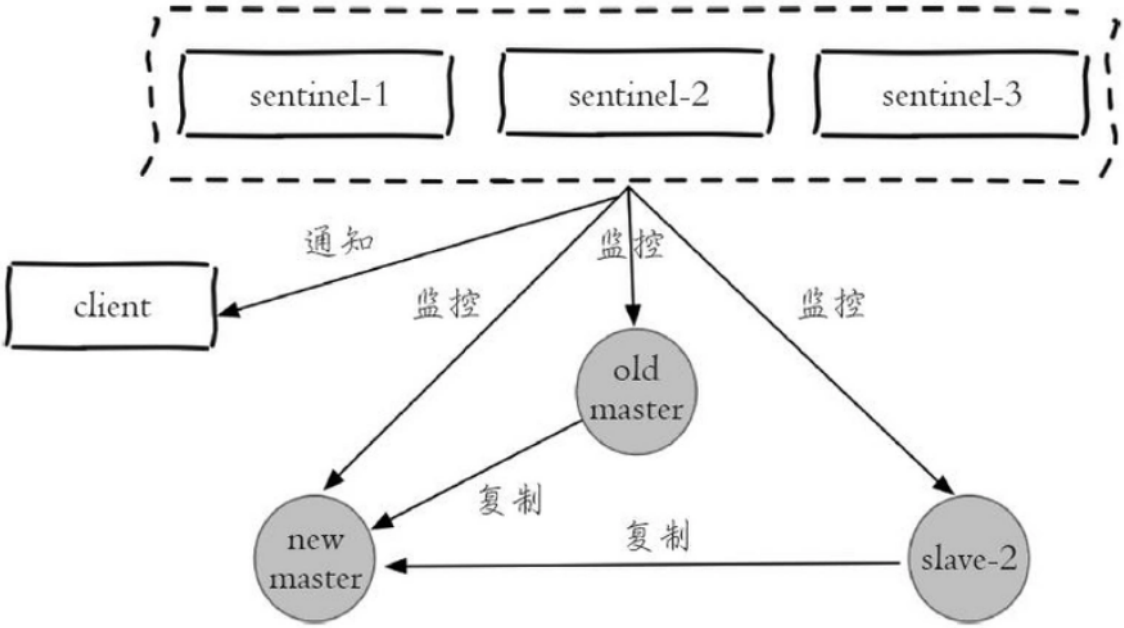
- 多个 Sentinel 节点对主节点的故障达成一致会选举出其中一个节点作为领导者负责故障转移



- Sentinel 领导者节点执行了故障转移，整个过程基本是跟我们手动调整一致的，只不过是自动化完成的



- 故障转移后整个 Redis Sentinel 的结构,重新选举了新的主节点



环境准备：

- master节点　　192.168.30.128　　sentinel端口：26379
- slave节点　　192.168.30.129　　sentinel端口：26379
- slave节点　　192.168.30.130　　sentinel端口：26379

修改配置（129、130省略）：

192.168.30.128

```
# vim /usr/local/redis/sentinel.conf

daemonize yes
logfile "/usr/local/redis/sentinel.log"
dir "/usr/local/redis/sentinel"                #sentinel工作目录
sentinel monitor mymaster 192.168.30.128 6379 2                #判断master失效至少需要2个sentinel同意，建议设
sentinel auth-pass mymaster 123456
sentinel down-after-milliseconds mymaster 30000                #判断master主观下线时间，默认30s
```

后台启动、log文件、工作目录、监听的主机地址以及判断主机客观下线的哨兵票数、密码和心跳连接超时时间

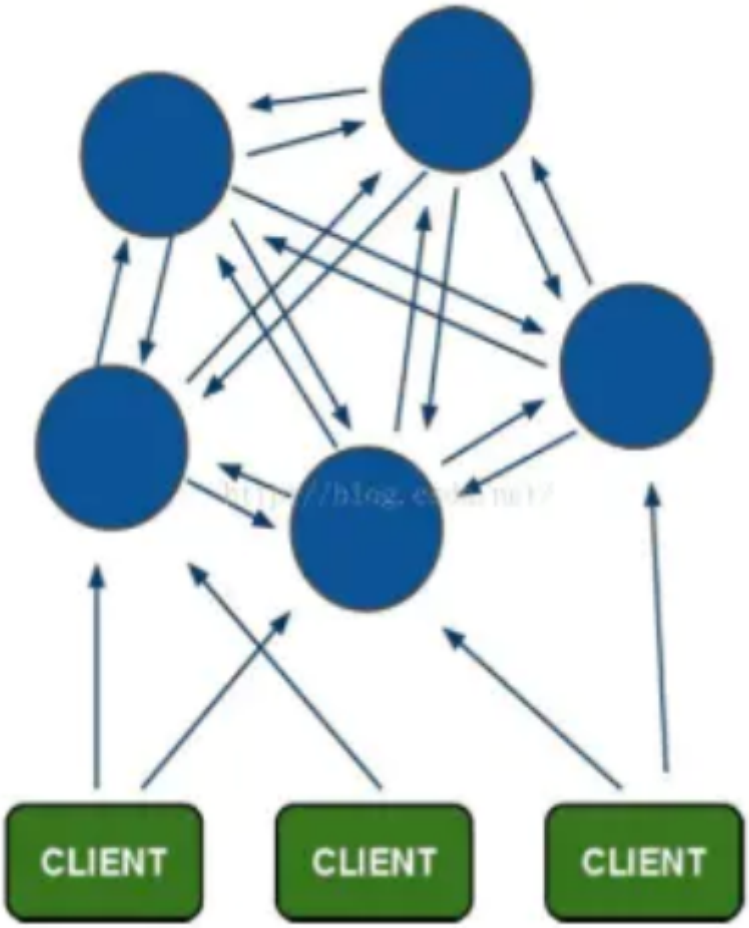
查看集群信息：

```
# /usr/local/bin/redis-cli -h 192.168.30.129 -p 6379 -a 123456
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe

192.168.30.129:6379> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=192.168.30.130,port=6379,state=online,offset=291039,lag=1
master_replid:757aff269236ed2707ba584a86a40716c1c76d74
master_replid2:47a862fc0ff20362be29096ecdcca6d432070ee9
master_repl_offset:291182
second_repl_offset:248123
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:291182

192.168.30.129:6379> set key4 linux
OK
```

即使192.168.30.128重新启动redis服务，也是作为slave加入redis集群，192.168.30.129仍然是master。

5、Cluster模式

Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。



sentinel模式基本可以满足一般生产的需求，具备高可用性。但是当数据量过大到一台服务器存放不下的情况时，主从模式或sentinel模式就不能满足需求了，这个时候需要对存储的数据进行分片，将数据存储到多个Redis实例中。cluster模式的出现就是为了解决单机Redis容量有限的问题，将Redis的数据根据一定的规则分配到多台机器。

cluster可以说是sentinel和主从模式的结合体，通过cluster可以实现主从和master重选功能，所以如果配置两个副本三个分片的话，就需要六个Redis实例。因为Redis的数据是根据一定规则分配到cluster的不同机器的，当数据量过大时，可以新增机器进行扩容。

cluster集群特点：

- 所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽。
- 节点的fail是通过集群中超过半数的节点检测失效时才生效。
- 客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可。
- redis-cluster把所有的物理节点映射到[0-16383]slot上（不一定是平均分配）,cluster 负责维护node<->slot<->value。
- Redis集群预分好16384个桶，当需要在 Redis 集群中放置一个 key-value 时，根据 CRC16(key) mod 16384的值，决定将一个key放到哪个桶中。
- 所有的节点都是一主一从（也可以是一主多从），其中从不提供服务，仅作为备用
- 支持在线增加、删除节点

节点分配：

现在我们是三个主节点分别是：A, B, C 三个节点，它们可以是一台机器上的三个端口，也可以是三台不同的服务器。那么，采用哈希槽 (hash slot)的方式来分配16384个slot 的话，它们三个节点分别承担的slot 区间是：

- 节点A覆盖0 - 5460;
- 节点B覆盖5461 - 10922;
- 节点C覆盖10923 - 16383.

  获取数据: 如果存入一个值，按照redis cluster哈希槽的算法： CRC16('key')384 = 6782。 那么就会把这个key 的存储分配到 B 上了。同样，当我连接(A,B,C)任何一个节点想获取'key'这个key时，也会这样的算法，然后内部跳转到B节点上获取数据

  新增一个主节点: 新增一个节点D，redis cluster的这种做法是从各个节点的前面各拿取一部分slot到D上，我会在接下来的实践中实验。大致就会变成这样：

- 节点A覆盖1365-5460
- 节点B覆盖6827-10922
- 节点C覆盖12288-16383
- 节点D覆盖0-1364,5461-6826,10923-12287

同样删除一个节点也是类似，移动完成后就可以删除这个节点了。

Redis Cluster主从模式 redis cluster 为了保证数据的高可用性，加入了主从模式，一个主节点对应一个或多个从节点，主节点提供数据存取，从节点则是从主节点拉取数据备份，当这个主节点挂掉后，就会有这个从节点选取一个来充当主节点，从而保证集群不会挂掉

上面那个例子里, 集群有ABC三个主节点, 如果这3个节点都没有加入从节点，如果B挂掉了，我们就无法访问整个集群了。A和C的slot也无法访问。

所以我们在集群建立的时候，一定要为每个主节点都添加了从节点, 比如像这样, 集群包含主节点A、B、C, 以及从节点A1、B1、C1, 那么即使B挂掉系统也可以继续正确工作。

B1节点替代了B节点，所以Redis集群将会选择B1节点作为新的主节点，集群将会继续正确地提供服务。 当B重新开启后，它就会变成B1的从节点。

不过需要注意，如果节点B和B1同时挂了，Redis集群就无法继续正确地提供服务了。

实现原理：

- 主观下线　集群中每个节点都会定期向其他节点发送ping消息，接受节点回复ping消息作为响应。如果在cluster-node-timeout时间内通信一直失败，则发送节点会认为接收节点存在故障，把接受节点标记为主观下线(pfail)状态。
- 客观下线
  - 当某个节点判断另一个节点主观下线后，相应的节点状态会跟随消息在集群内传播。
  - 假设节点a标记节点b为主观下线，一段时间后节点a通过消息把节点b的状态发送到其他节点，当其他节点收到消息并解析出消息体中含有b的pfail状态，把节点b加入下线报告链表；
  - 当某一节点c收到节点b的pfail状态时，此时有超过一半的槽主节点都标记了节点b为pfail状态时，则标记故障节点b为客观下线；
  - 向集群广播一条pfail消息，通知集群内的所有节点标记故障节点b为客观下线状态并立刻生效，同时通知故障节点b的从节点触发故障转移流程。

故障恢复:

- 资格检查

  若从节点与主节点断线时间超过一定时间，则不具备资格
- 准备选举时间

  当从节点符合故障转移资格后，要等待一段选举时间后才开始选举，在故障节点的所有从节点中，复制**偏移量**最大的那个从节点最先开始（与主节点的数据最一致）进行选举，然后是次大的节点开始选举.....剩下其余的从节点等待到它们的选举时间到达后再进行选举
- 发起选举
- 选举投票

  只有持有槽的主节点才具有一张唯一的选票，从节点收集到N/2 + 1个持有槽的主节点投票时，从节点可以执行替换主节点操作
- 替换主节点

  当从节点收集到足够的选票之后，触发替换主节点操作，**当前从节点取消复制变为主节点，撤销故障主节点负责的槽，并把这些槽委派给自己，向集群广播自己的pong消息，告诉他们已经接管了主节点以及它拥有的槽位。**(变主、撤主、委派、通知)

环境准备：

在redis-cluster下 修改bin文件夹为redis01,复制redis.conf配置文件 创建目录redis-cluster并在此目录下再创建7000 7001 7002 7003 7004 7005共6个目录，在7000中创建配置文件redis.conf，内容如下：

```
daemonize yes #后台启动
port 7001 #修改端口号，从7001到7006
cluster-enabled yes #开启cluster，去掉注释
cluster-config-file nodes.conf #自动生成
cluster-node-timeout 15000 #节点通信时间
appendonly yes #持久化方式
```

后台启动、端口号、集群开关、节点通信时间、持久化方式

启动节点：

```
cd 7000
redis-server redis.conf
cd ..
cd 7001
redis-server redis.conf
cd ..
cd 7002
redis-server redis.conf
cd ..
cd 7003
redis-server redis.conf
cd ..
cd 7004
redis-server redis.conf
cd ..
cd 7005
redis-server redis.conf
cd ..
```

设置权限启动：

```
[root@localhost redis-cluster]# chmod 777 start-all.sh
[root@localhost redis-cluster]# ./start-all.sh
```

查看redis进程启动状态：

```
[root@localhost redis-4.0.2]# ps -ef|grep cluster
root      54956      1  0 19:17 ?        00:00:00 redis-server *:7000 [cluster]
root      54961      1  0 19:17 ?        00:00:00 redis-server *:7001 [cluster]
root      54966      1  0 19:17 ?        00:00:00 redis-server *:7002 [cluster]
root      54971      1  0 19:17 ?        00:00:00 redis-server *:7003 [cluster]
root      54976      1  0 19:17 ?        00:00:00 redis-server *:7004 [cluster]
root      54981      1  0 19:17 ?        00:00:00 redis-server *:7005 [cluster]
root      55071  24089  0 19:24 pts/0    00:00:00 grep --color=auto cluster
```

可以看到redis的6个节点已经启动成功，**注意：这里并没有创建集群**

使用redis-trib.rb创建集群 注意：redis-trib.rb在redis/src目录下。

```
./redis-trib.rb create --replicas 1 127.0.0.1:7001 127.0.0.1:7002 127.0.0.1:7003
127.0.0.1:7004 127.0.0.1:7005 127.0.0.1:7000
```

使用create命令 --replicas 1 参数表示为每个主节点创建一个从节点，其他参数是实例的地址集合。

```
[root@localhost redis]# ./src/redis-trib.rb create --replicas 1 127.0.0.1:7001
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 127.0.0.1:7000
>>> Creating cluster
>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
127.0.0.1:7001
127.0.0.1:7002
```

```
127.0.0.1:7003
Adding replica 127.0.0.1:7004 to 127.0.0.1:7001
Adding replica 127.0.0.1:7005 to 127.0.0.1:7002
Adding replica 127.0.0.1:7000 to 127.0.0.1:7003
M: f4ee0a501f9aaf11351787a46ffb4659d45b7bd7 127.0.0.1:7001
   slots:0-5460 (5461 slots) master
M: 671a0524a616da8b2f50f3d11a74aaf563578e41 127.0.0.1:7002
   slots:5461-10922 (5462 slots) master
M: 18948dab5b07e3726afd1b6a42d5bf6e2f411ba1 127.0.0.1:7003
   slots:10923-16383 (5461 slots) master
S: 34e322ca50a2842e9f3664442cb11c897defba06 127.0.0.1:7004
   replicates f4ee0a501f9aaf11351787a46ffb4659d45b7bd7
S: 62a00566233fbff4467c4031345b1db13cf12b46 127.0.0.1:7005
   replicates 671a0524a616da8b2f50f3d11a74aaf563578e41
S: 2cb649ad3584370c960e2036fb01db834a546114 127.0.0.1:7000
   replicates 18948dab5b07e3726afd1b6a42d5bf6e2f411ba1
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join...
>>> Performing Cluster Check (using node 127.0.0.1:7001)
M: f4ee0a501f9aaf11351787a46ffb4659d45b7bd7 127.0.0.1:7001
   slots:0-5460 (5461 slots) master
   1 additional replica(s)
M: 671a0524a616da8b2f50f3d11a74aaf563578e41 127.0.0.1:7002
   slots:5461-10922 (5462 slots) master
   1 additional replica(s)
S: 2cb649ad3584370c960e2036fb01db834a546114 127.0.0.1:7000
   slots: (0 slots) slave
   replicates 18948dab5b07e3726afd1b6a42d5bf6e2f411ba1
S: 34e322ca50a2842e9f3664442cb11c897defba06 127.0.0.1:7004
   slots: (0 slots) slave
   replicates f4ee0a501f9aaf11351787a46ffb4659d45b7bd7
M: 18948dab5b07e3726afd1b6a42d5bf6e2f411ba1 127.0.0.1:7003
   slots:10923-16383 (5461 slots) master
   1 additional replica(s)
S: 62a00566233fbff4467c4031345b1db13cf12b46 127.0.0.1:7005
   slots: (0 slots) slave
   replicates 671a0524a616da8b2f50f3d11a74aaf563578e41
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

上面显示创建成功，有3个主节点，3个从节点，每个节点都是成功连接状态。

redis集群的测试：

测试存取值，客户端连接集群redis-cli需要带上 -c ， redis-cli -c -p 端口号

```
[root@localhost redis]# ./redis-cli -c -p 7001
127.0.0.1:7001> set name andy
-> Redirected to slot [5798] located at 127.0.0.1:7002
OK
127.0.0.1:7002> get name
"andy"
127.0.0.1:7002>
```

根据redis-cluster的key值分配，name应该分配到节点7002[5461-10922]上，上面显示redis cluster自动从7001跳转到了7002节点。

测试一下7000从节点获取name值

```
[root@localhost redis]# ./redis-cli -c -p 7000
127.0.0.1:7000> get name
-> Redirected to slot [5798] located at 127.0.0.1:7002
"andy"
127.0.0.1:7002>
```

集群节点选举

现在模拟将7002节点挂掉，按照redis-cluster原理会选举会将 7002的从节点7005选举为主节点。

```
[root@localhost redis-cluster]# ps -ef | grep redis
root       7966      1  0 12:50 ?        00:00:29 ./redis-server 127.0.0.1:7000
[cluster]
root       7950      1  0 12:50 ?        00:00:28 ./redis-server 127.0.0.1:7001
[cluster]
root       7952      1  0 12:50 ?        00:00:29 ./redis-server 127.0.0.1:7002
[cluster]
root       7956      1  0 12:50 ?        00:00:29 ./redis-server 127.0.0.1:7003
[cluster]
root       7960      1  0 12:50 ?        00:00:29 ./redis-server 127.0.0.1:7004
[cluster]
root       7964      1  0 12:50 ?        00:00:29 ./redis-server 127.0.0.1:7005
[cluster]
root      11346  10581  0 14:57 pts/2    00:00:00 grep --color=auto redis
[root@localhost redis-cluster]# kill 7952
```

在查看集群中的7002节点

```
[root@localhost src]# ./redis-trib.rb check 127.0.0.1:7002
>>> Performing Cluster Check (using node 127.0.0.1:7002)
S: 671a0524a616da8b2f50f3d11a74aaf563578e41 127.0.0.1:7002
   slots: (0 slots) slave
   replicates 62a00566233fbff4467c4031345b1db13cf12b46
M: 18948dab5b07e3726afd1b6a42d5bf6e2f411ba1 127.0.0.1:7003
   slots:10923-16383 (5461 slots) master
   1 additional replica(s)
M: 62a00566233fbff4467c4031345b1db13cf12b46 127.0.0.1:7005
   slots:5461-10922 (5462 slots) master
```

```
   1 additional replica(s)
M: f4ee0a501f9aaf11351787a46ffb4659d45b7bd7 127.0.0.1:7001
   slots:0-5460 (5461 slots) master
   1 additional replica(s)
S: 34e322ca50a2842e9f3664442cb11c897defba06 127.0.0.1:7004
   slots: (0 slots) slave
   replicates f4ee0a501f9aaf11351787a46ffb4659d45b7bd7
S: 2cb649ad3584370c960e2036fb01db834a546114 127.0.0.1:7000
   slots: (0 slots) slave
   replicates 18948dab5b07e3726afd1b6a42d5bf6e2f411ba1
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

可以看到集群连接不了7002节点，而7005有原来的S转换为M节点，代替了原来的7002节点。我们可以获取name值：

```
[root@localhost redis]# ./redis-cli -c -p 7001
127.0.0.1:7001> get name
-> Redirected to slot [5798] located at 127.0.0.1:7005
"andy"
127.0.0.1:7005>
127.0.0.1:7005>
```

从7001节点连入，自动跳转到7005节点，并且获取name值。

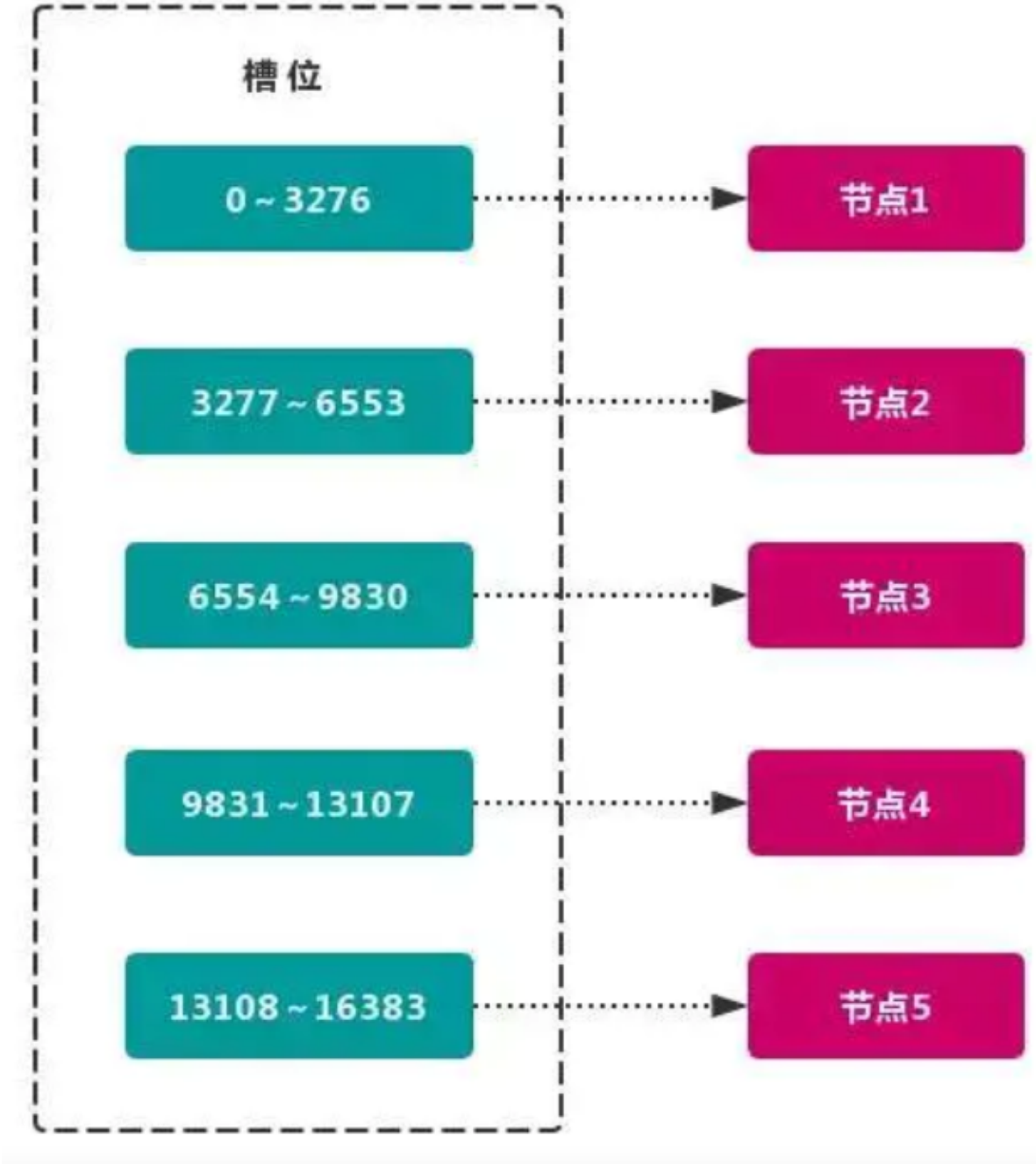
现在我们将7002节点恢复，看是否会自动加入集群中以及充当的M还是S节点。

```
[root@localhost redis-cluster]# cd 7002
[root@localhost 7002]# ./redis-server redis.conf
[root@localhost 7002]#
```

再check一下7002节点，可以看到7002节点变成了7005的从节点。

6、虚拟槽分片算法

虚拟槽分区 巧妙地使用了 **哈希空间**，使用 **分散度良好** 的 **哈希函数** 把所有数据 **映射** 到一个 **固定范围** 的 **整数集合** 中，整数定义为 **槽**（slot）。这个范围一般 **远远大于** 节点数，比如 Redis Cluster 槽范围是 0 ~ 16383。**槽** 是集群内 **数据管理** 和 **迁移** 的 **基本单位**。采用 **大范围槽** 的主要目的是为了方便 **数据拆分** 和 **集群扩展**。每个节点会负责 **一定数量的槽**，如图所示：



当前集群有 5 个节点，每个节点平均大约负责 3276 个 **槽**。由于采用 **高质量** 的 **哈希算法**，每个槽所映射的数据通常比较 **均匀**，将数据平均划分到 5 个节点进行 **数据分区**。Redis Cluster 就是采用 **虚拟槽分区**。

节点1： 包含 0 到 3276 号哈希槽。

节点2：包含 3277 到 6553 号哈希槽。

节点3：包含 6554 到 9830 号哈希槽。

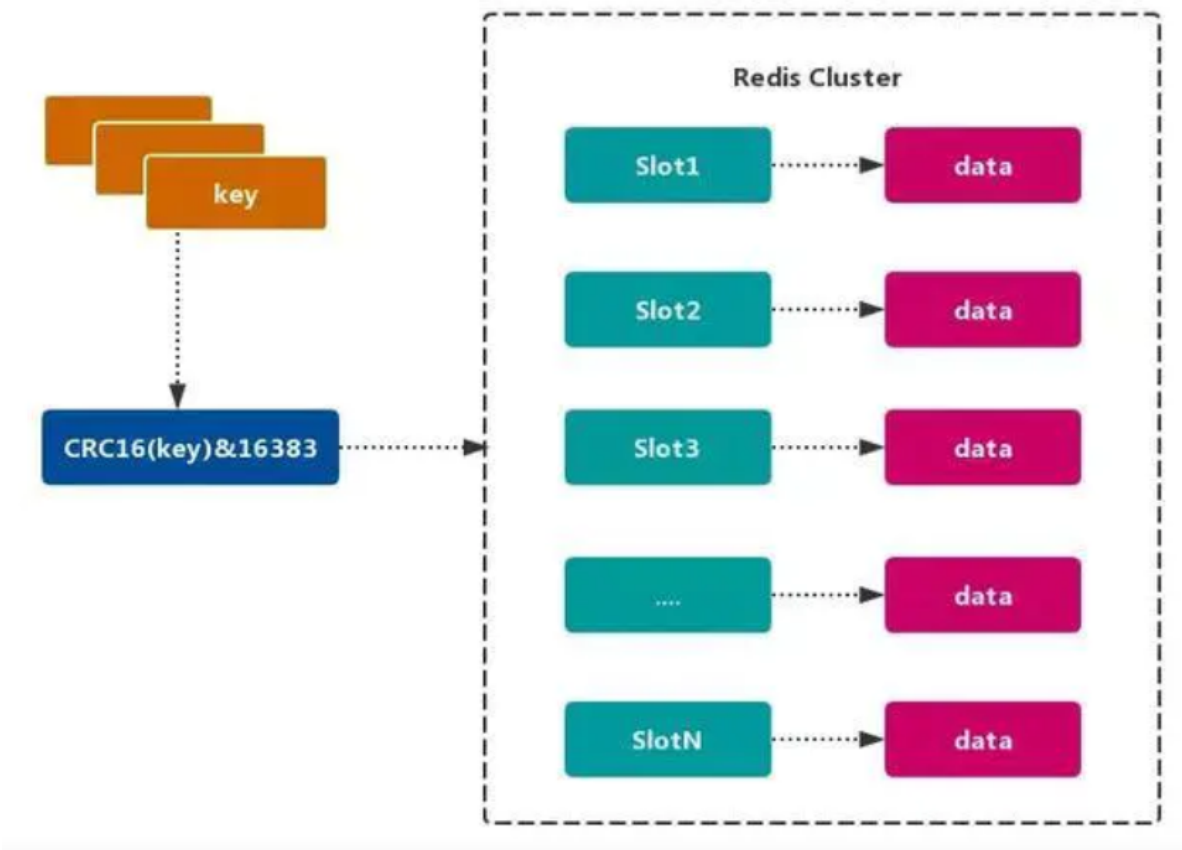
节点4：包含 9831 到 13107 号哈希槽。

节点5：包含 13108 到 16383 号哈希槽。

这种结构很容易 **添加** 或者 **删除** 节点。如果 **增加** 一个节点 6，就需要从节点 1 ~ 5 获得部分 **槽** 分配到节点 6 上。如果想 **移除** 节点 1，需要将节点 1 中的 **槽** 移到节点 2 ~ 5 上，然后将 **没有任何槽** 的节点 1 从集群中 **移除** 即可。

由于从一个节点将 **哈希槽** 移动到另一个节点并不会 **停止服务**，所以无论 **添加删除**或者 **改变** 某个节点的 **哈希槽的数量** 都不会造成 **集群不可用** 的状态.

Redis Cluster 采用 **虚拟槽分区**，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。每个节点负责维护一部分槽以及槽所映射的 **键值数据**，如图所示：



Redis虚拟槽分区的特点：

解耦 **数据** 和 **节点** 之间的关系，简化了节点 **扩容** 和 **收缩** 难度。

节点自身 维护槽的 **映射关系**，不需要 **客户端** 或者 **代理服务** 维护 **槽分区元数据。**

支持 **节点**、**槽**、**键** 之间的 **映射查询**，用于 **数据路由**、**在线伸缩** 等场景。

参考资料

Redis集群详解 https://blog.csdn.net/miss1181248983/article/details/90056960

Redis-Cluster集群 https://www.jianshu.com/p/813a79ddf932

深入剖析Redis - Redis集群模式搭建与原理详解 https://www.jianshu.com/p/84dbb25cc8dc

Redis哨兵机制 https://segmentfault.com/a/1190000018278099

Redis主从同步

Redis全量同步，又称快照同步。可能有些人不太理解快照的意思，快照，就跟照片一样，记录着在某一个时间点所有键值状态。在Redis中，获取一个快照的命令并不难，我们可以手动创建一个快照，也可以通过配置，后台根据相关规则创建快照。在Redis中，我们可以通过save命令，就可以创建快照，因为Redis是单进程的，使用save命令会影响性能，所以通常我们会使用bgsave。

```
179
180 ################################ SNAPSHOTTING  ################################
181 #
182 # Save the DB on disk:
183 #
184 #   save <seconds> <changes>
185 #
186 #   Will save the DB if both the given number of seconds and the given
187 #   number of write operations against the DB occurred.
188 #
189 #   In the example below the behaviour will be to save:
190 #   after 900 sec (15 min) if at least 1 key changed
191 #   after 300 sec (5 min) if at least 10 keys changed
192 #   after 60 sec if at least 10000 keys changed
193 #
194 #   Note: you can disable saving completely by commenting out all "save" lines.
195 #
196 #   It is also possible to remove all the previously configured save
197 #   points by adding a save directive with a single empty string argument
198 #   like in the following example:
199 #
200 #   save ""
201
202 save 900 1
203 save 300 10
204 save 60 10000
```

在redis.conf中，有着自动保存的配置，上述是Redis的默认配置，说的是在900秒内，如果有1个key变更，或者300秒内，有300个key变更，或者60秒内，有10000个key变更，就会后台出发生成快照功能。

与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

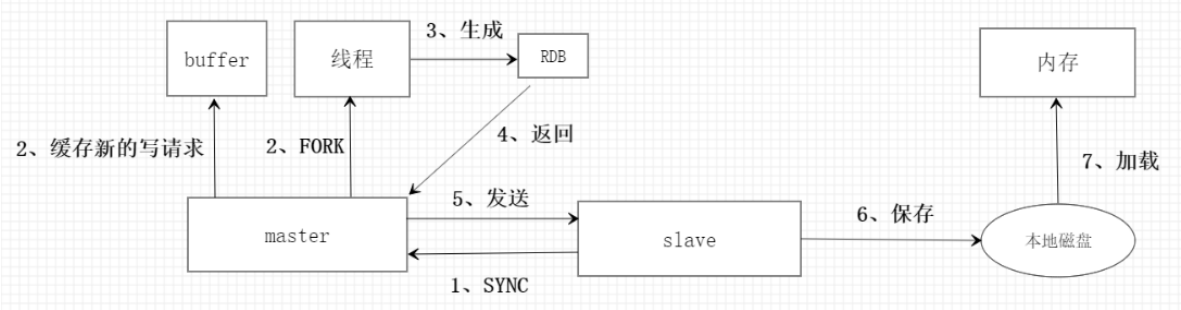
定时：data->data文件；操作->日志文件

1、工作机制：

Redis的主从结构可以采用一主多从或者级联结构，Redis主从复制可以根据是否是全量分为全量同步和增量同步。

全量同步 Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。具体步骤如下：

- 从服务器连接主服务器，发送SYNC命令；
- 主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；
- 主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；
- 从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；
- 主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；
- 从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；



上述的过程对于一个从机，新加入到集群里面的时候，比较合适，但是如果因为断线重连，则需要重新复制**主机上所有的内容**，主机也因此要进行一次**全量级的RDB**，比较耗费性能。

在2.8版本之后，采用了一个新的增量复制的过程，流程如下：

1. 前面的`sync`的过程还是一样，没有差异
2. 当从机断线重连之后，会发送`PSYNC`,要求增量同步，并包括一个**offset**
3. 主机根据offset的位置，对之后的数据进行一次增量同步，到从机上面

这里面的offset是由主机和从机共同维护的，相当于一个乐观锁，描述了两方的版本差异。

那决定能否增量同步的主要因素，包括如下三个：

1. 是否具备偏移量，与主机进行对比
2. 主服务器的复制积压缓冲区（Replication backlog）
3. 从服务器的ID

增量同步 Redis增量复制是指Slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。

增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。

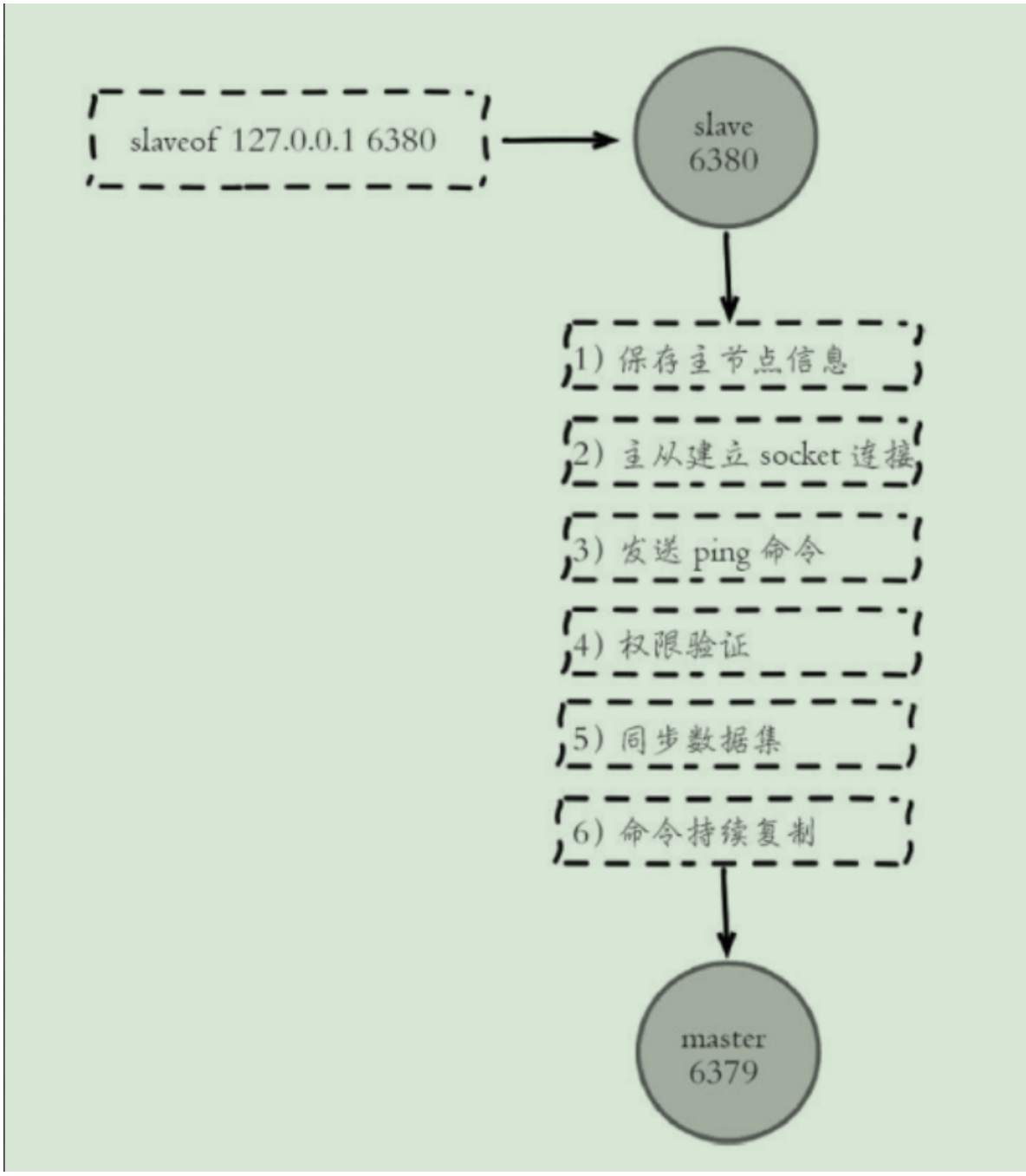
Redis主从同步策略 主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，slave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。

注意点 如果多个Slave断线了，需要重启的时候，因为只要Slave启动，就会发送sync请求和主机全量同步，当多个同时出现的时候，可能会导致Master IO剧增宕机。

时机点：断线重连、正常连接时，主服务器接收到写命令会及时同步给从服务器

2、具体过程

复制过程大致分为6个过程：



1）保存主节点信息

执行slaveof后从节点只保存主节点的地址信息便直接返回，这时建立复制流程还没有开始，在从节点执行info replication可以看到如下信息：

```
master_host:xxx
master_port:xxx
master_link_status:down
```

从统计信息可以看出，主节点的ip和port被保存下来，但是主节点的连接状态master_link_status是下线状态，需要执行slaveof。

2）建立socket连接

从节点内部通过每秒运行的定时任务维护复制相关逻辑，当定时任务发现存在新的主节点后，会尝试与该节点建立网络连接。从节点会建立一个socket套接字，专门用于接受主节点发送的复制命令。 如果从节点无法建立连接，定时任务会无限重试直到连接成功或执行slaveof no one取消复制。

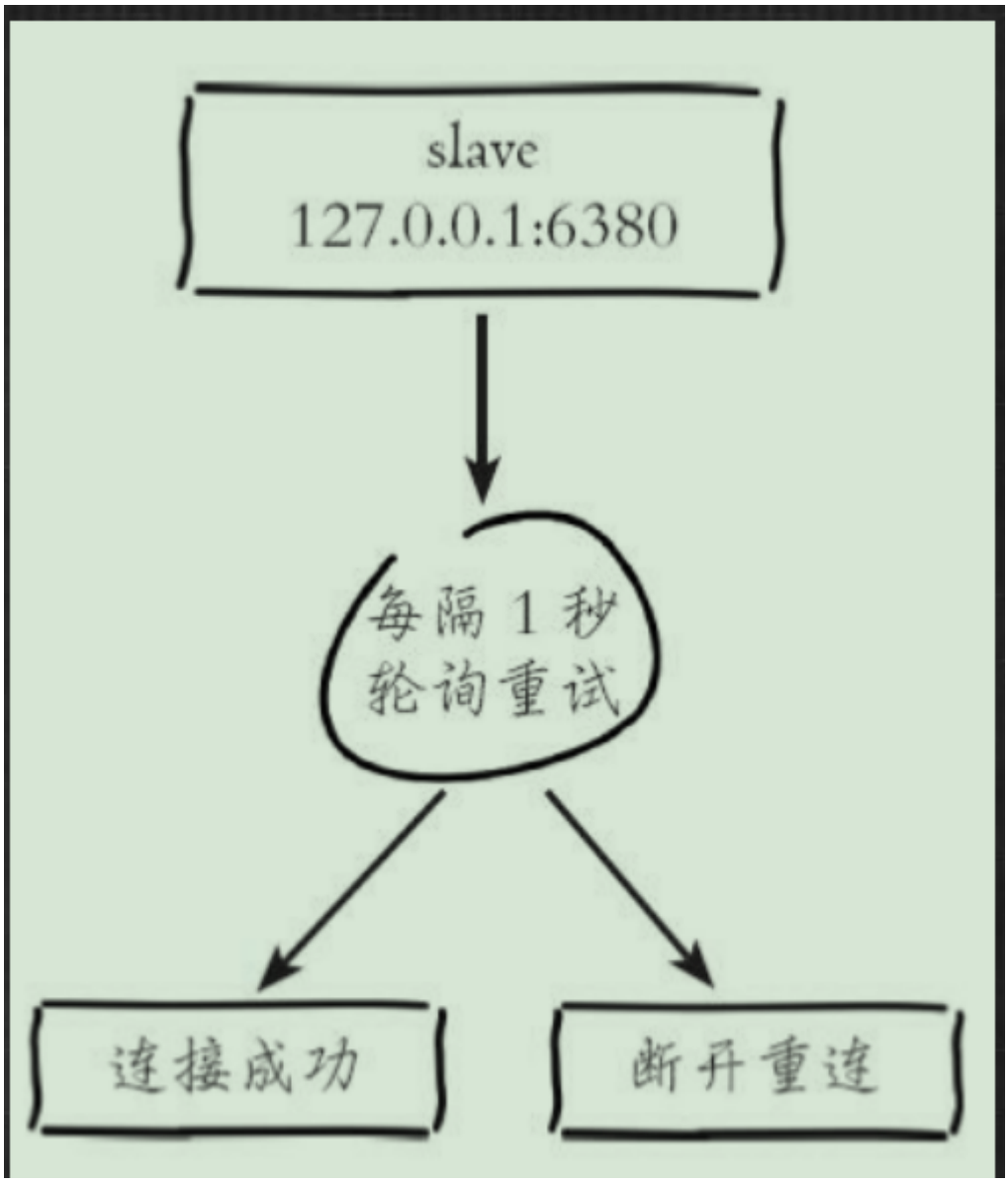
3） 发送ping命令

连接建立成功后从节点发送ping请求进行首次通信，ping请求主要目的如下：

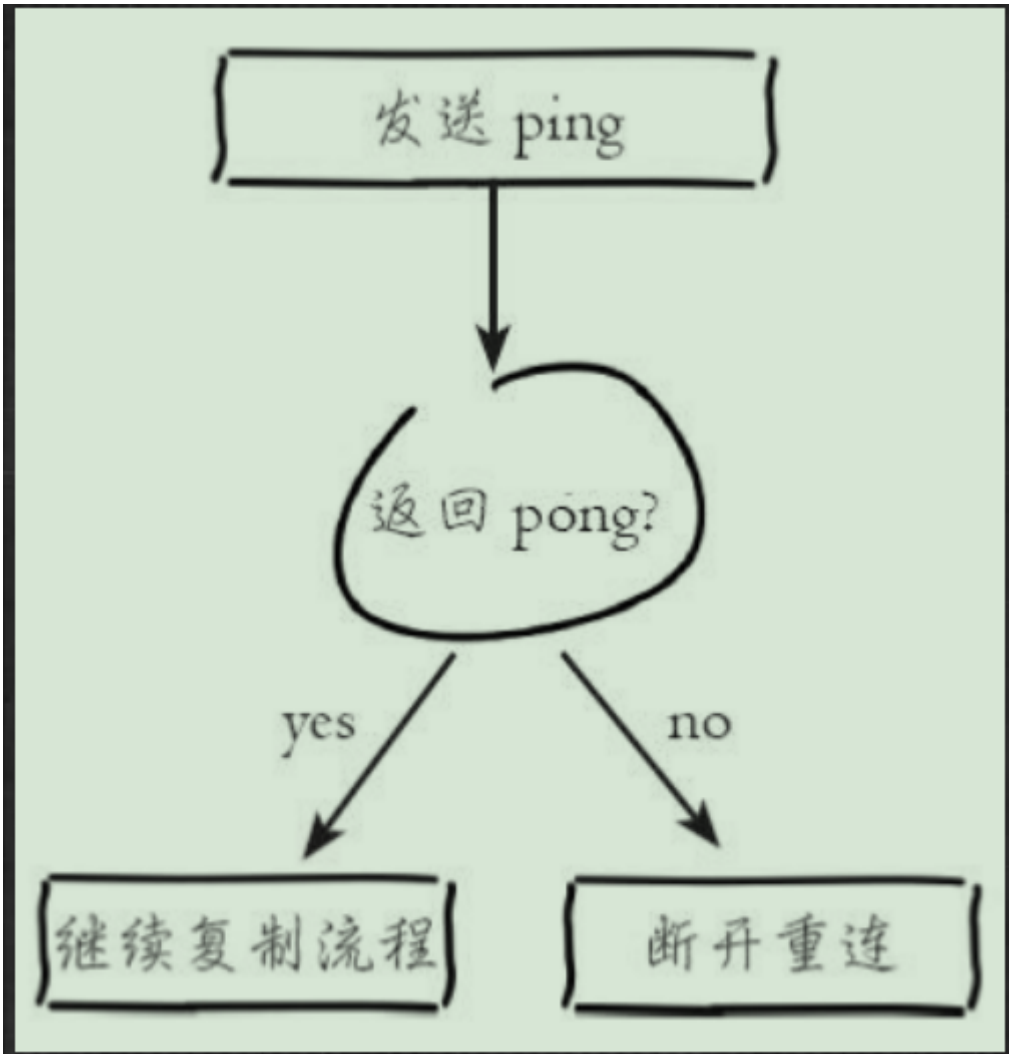
- 检测主从之间网络套接字是否可用
- 检测主节点当前是否可接受处理命令

如果发送ping命令后，从节点没有收到主节点的pong回复或者超时，比如网络超时或者主节点正在阻塞无法响应命令，从节点会断开复制链接，下次定时任务时任务会重连。

从节点与主节点建立连接流程如图：



从节点发送ping命令流程：



4）权限验证

如果主节点设置了requirepass参数，则需要密码验证，从节点必须配置masterauth参数保证与主节点相同的密码才能通过验证；如果验证失败复制将终止，从节点重新发起复制流程。

5）同步数据集

主从复制链接正常通信后，对于首次建立复制的场景，主节点会把持有的数据全部发送给从节点，这部分是耗时最长的步骤。Redis在2.8版本以后采用新复制命令psync进行数据同步，原来的sync命令也同样支持，为了新旧版本的兼容性。新同步划分为两种情况：全量同步和部分同步。

6）命令持续复制

当主节点把当前数据同步给从节点或，便完成了复制的建立流程，接下来主节点会持续地把写命令发送给从节点，保证主从数据一致性。

3、数据同步

Redis在2.8及以上版本使用psync命令完成主从数据同步，同步过程分为全量复制和部分复制。

全量复制：一般用于初次复制场景，Redis早期支持的复制功能只有全量复制，它会把从节点全部数据一次性发送给从节点，当数据较大时，会对主从节点和网络造成很大的开销。

部分复制：用于处理在主从复制中因网络闪断等原因造成的数据丢失场景，当从节点再次连上主节点后，如果条件允许，主节点会补发丢失数据给从节点。因为补发的数据远远小于全量数据，可以有效避免全量复制的开销。

部分复制是对老版复制的重大优化，有效避免了不必要的全量复制操作。因此当使用复制功能时，尽量采用2.8以上版本的Redis。

psync命令运行需要以下组件支持：

- 主从节点各自复制偏移量
- 主节点复制积压缓冲区
- 主节点运行id

1）复制偏移量

参与复制的主从节点都会维护自身复制偏移量，主节点处理完写入命令后，会把命令的字节长度做累加记录，统计信息在info replication中的master_repl_offset指标中。

从节点每秒上报自身的复制偏移量给主节点，因此主节点也会保存从节点的复制偏移量。均可以通过info replication查看。

从节点在接收到主节点发送的命令后，也会累加记录自身的偏移量。统计信息在info replication中的slave_repl_offset指标中 通过对比主从节点的复制偏移量，可以判断主从节点数据是否一致。

提示：可以通过主节点的统计信息，计算出master_repl_offset - slave_offset字节量，判断主从节点复制相差的数据量，根据这个差值判定当前复制的健康度。

2）复制积压缓冲区

复制积压缓冲区是保存在主节点上的一个固定长度的队列，默认大小为1MB，当主节点有连接的从节点时被创建，这时主节点响应写命令时，不但会把命令发送给从节点，还会写入复制积压缓冲区。

由于缓冲区本质上是先进先出的定长队列，所以能实现保存最近已复制数据的功能，用于部分复制和复制命令丢失的数据补救。复制缓冲区相关统计信息也保存在主节点的info replication中：

3）主节点运行ID

每个Redis节点启动后都会动态分配一个40位的十六进制字符串作为运行ID。**运行ID的主要作用是用来唯一识别Redis节点，比如从节点保存主节点的运行ID识别自己正在复制的是哪个主节点。**如果只使用ip+port的方式识别主节点，那么主节点重启变更了整体数据集（如替换RDB/AOF），从节点再基于偏移量复制数据将是不安全的，因此**当运行ID变化后从节点将做全量复制。**可以运行info server命令查看当前节点的运行ID。

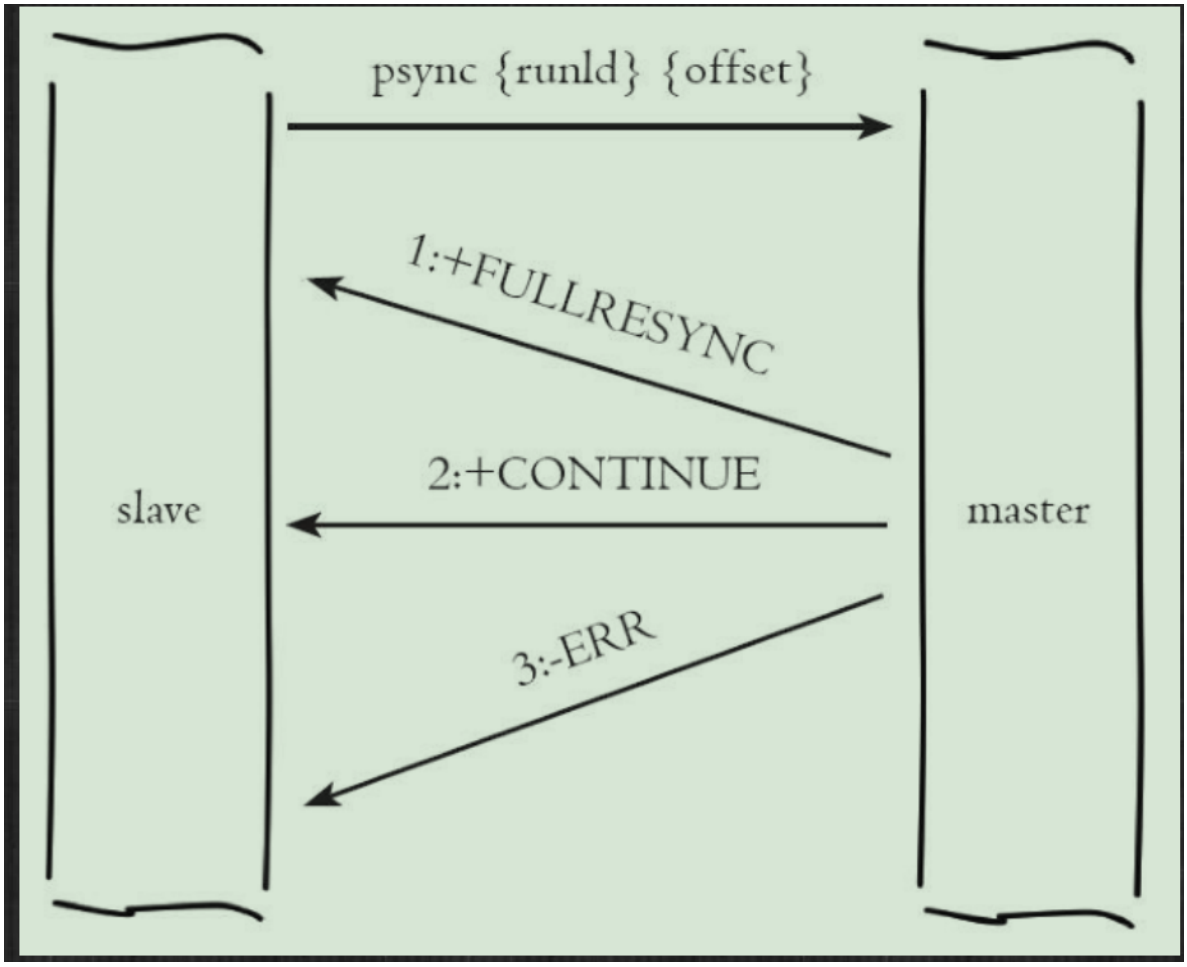
需要注意的是Redis关闭再启动后，运行ID会随之改变。Redis不改变ID重启，可以使用redis-cli debug reload

注意：debug reload命令会阻塞当前Redis节点主线程，阻塞期间会生成本地RDB快照并清空数据之后再加载RDB文件，因此对于大数据量的节点和无法容忍阻塞的应用场景，谨慎使用。

4）psync命令

从节点使用psync命令完成部分复制和全量复制功能，命令格式：psync {runid} {offset}

- runid：从节点所复制主节点的运行id
- offset：当前从节点已复制的数据偏移量



流程说明：

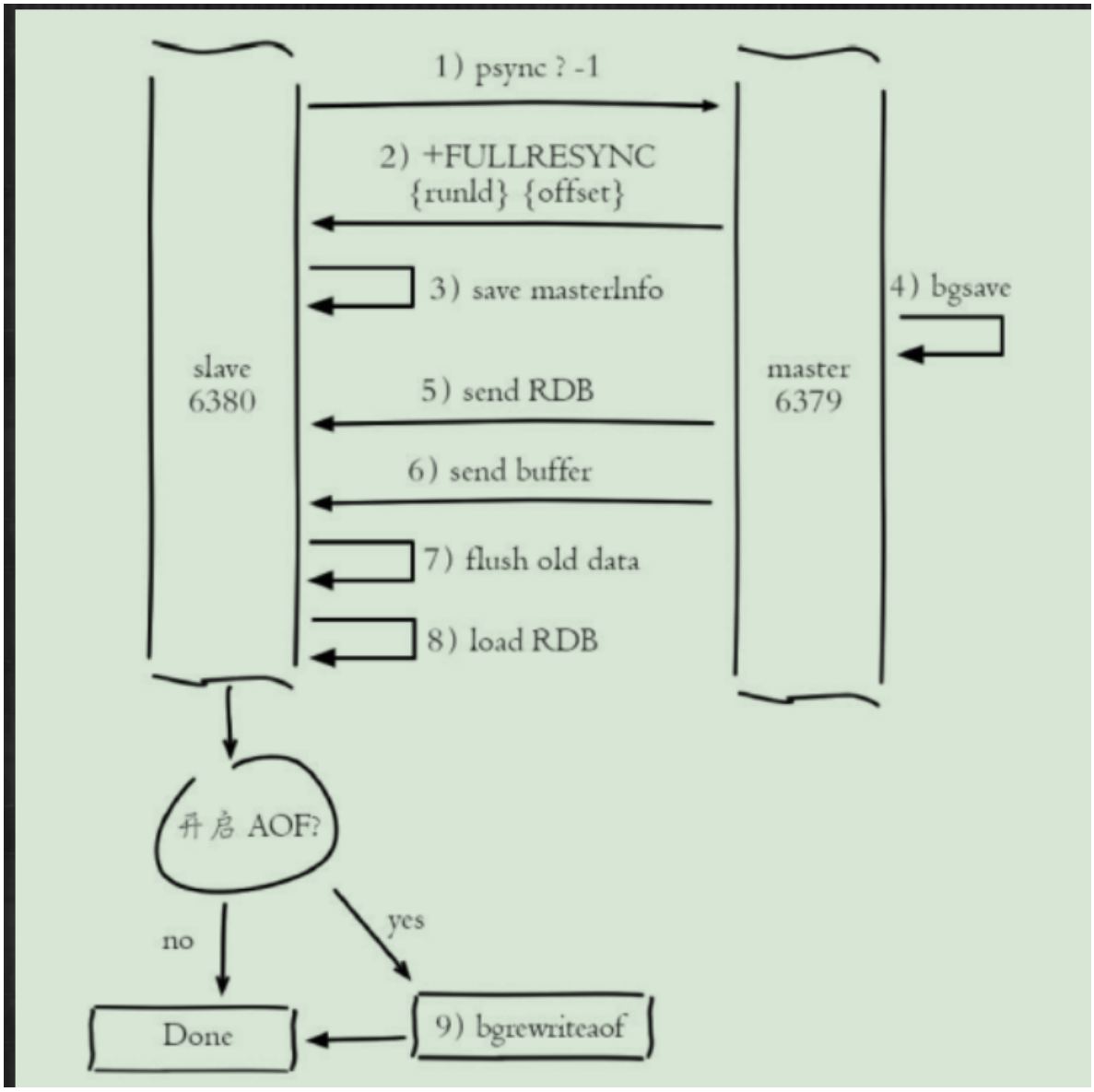
1）从节点发送psync命令给主节点，参数runid是当前从节点保存的主节点运行ID，参数offset是当前从节点保存的复制偏移量，如果是第一次复制则为-1。

2）主节点根据psync参数和自身数据情况决定响应结果：

- 如果回复 FULLRESYNC {runid} {offset}，那么从节点触发全量复制
- 如果回复CONTINUE，从节点将触发部分复制流程
- 如果回复ERR，说明主节点版本低于2.8，无法识别psync命令，从节点将发送旧版的sync命令触发全量复制流程。

全量复制

全量复制是Redis最早支持的复制方式，也是主从第一次建立复制时必须经历的阶段。触发全量复制的命令是sync和psync。 用于初次复制或其它无法进行部分复制的情况，将主节点中的所有数据都发送给从节点，是一个非常重型的操作，当数据量较大时，会对主从节点和网络造成很大的开销。



流程说明：

1）发送psync命令进行数据同步，由于是第一次进行复制，从节点没有复制偏移量和主节点的运行ID，所以发送psync-1。

2）主机会向从机发送run_id和offset，因为slave并没有对应的 offset，所以是全量复制

3）从节点接收并保存主节点的响应数据保存运行ID和偏移量offse。

4）主节点执行bgsave（异步执行），在后台生成RDB文件（快照），并使用一个缓冲区（称为复制缓冲区）记录后续的所有写命令

5）主机发送RDB文件给从机，从节点把接收的RDB文件保存在本地并直接作为从节点的数据文件。

注意：对于数据量大的主节点，比如生成的RDB文件超过6GB以上传输文件这一步操作非常耗时，速度取决于主从节点之间的网络带宽，针对数据量较大的节点，建议调大repl-timeout参数防止出现全量同步数据超时。

关于无盘复制：为了降低主节点的磁盘开销，Redis支持无盘复制，生成的RDB文件不保存到硬盘而是直接通过网络发送给从节点，通过repl-diskless-sync参数控制，默认关闭。无盘复制适用于主节点所在机器磁盘性能较差但网络带宽较充裕的场景。注意无盘复制目前依然处于试验阶段。

6）发送缓冲区数据给从节点。（对于从节点开始接收RDB快照到接收完成期间，**主节点仍然响应读写命令，因此主节点会把这期间写命令数据保存在复制客户端缓冲区内，当从节点加载完RDB文件后，主节点再把缓冲区内的数据发送给从节点，保证主从之间数据一致性。**）

7）从节点接收完主节点传送来的全部数据后**清空自身旧数据。**

8）从节点清空数据后开始**加载RDB文件**，对于较大的RDB依然耗时。对于读写分离的场景，可以设置slave-serve-stale-data参数为no，关闭命令执行，在完成同步前不响应读命令

9）从节点成功加载完RDB后，如果当前节点开启了AOF持久化功能，它会立刻做bgrewriteaof操作，为了保证全量复制后AOF持久化文件立刻可用。

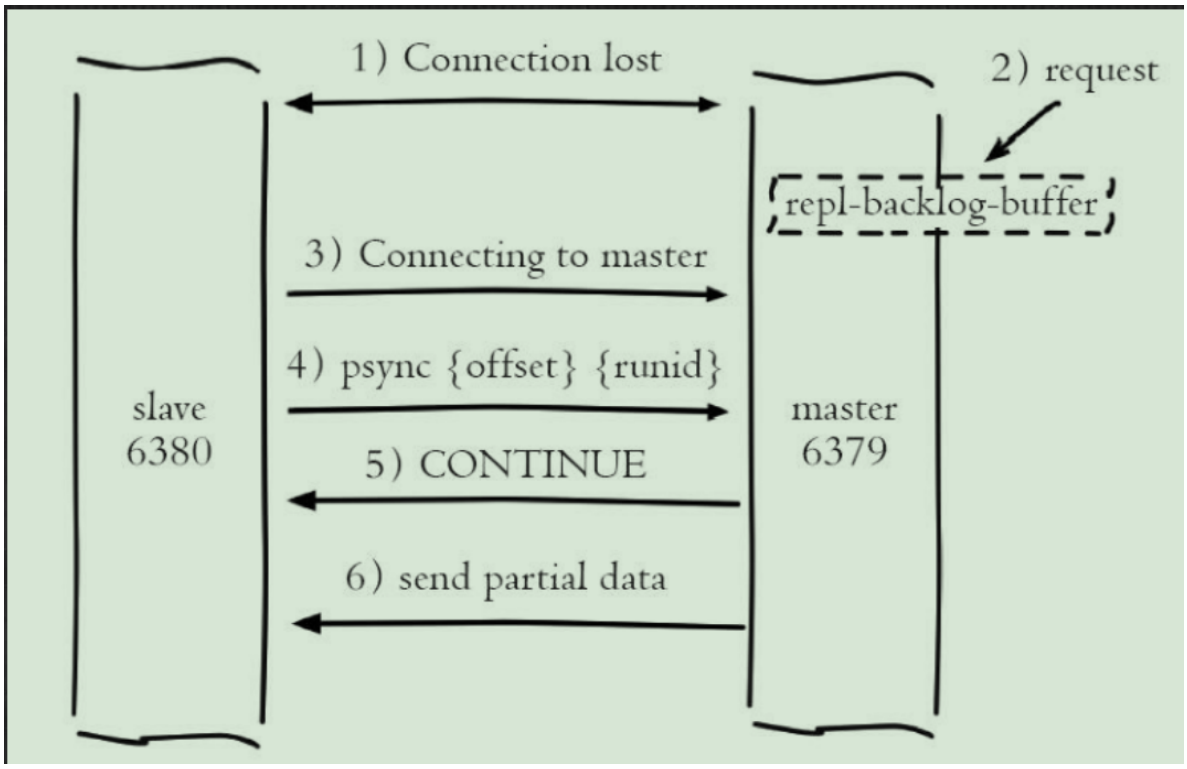
RDB和AOF区别：RDB是将Redis某一时刻的数据持久化到磁盘上，类似于快照；AOF是将redis执行过的所有写指令记录下来，在下次redis重新启动时，只要把这些写指令从前到后再重复执行一遍，就可以实现数据恢复，默认每秒钟fsync一次。

开销：

- 主节点需要bgsave
- RDB文件网络传输占用网络io
- 从节点要清空数据
- 从节点加载RDB
- 全量复制会触发从节点AOF重写

部分复制

部分复制主要是Redis针对全量复制的过高开销做出的一种优化措施，使用psync {runid} {offset}命令实现，当从节点正在复制主节点时，如果出现异常情况，从节点要求补发丢失的命令数据，如果主节点的复制积压缓冲区内存在这部分数据则直接发送给从节点，这样就可以保持主从节点复制的一致性。



流程说明：

1）当主从节点之间网络出现中断时，如果超过repl-timeout时间，主节点会认为从节点故障并中断复制链接。

2）主从连接中断期间主节点依然响应命令，写命令保存在复制积压缓冲区，默认最大缓存1MB。

3）当主从节点网络恢复后，从节点会再次连接主节点。

4）连接恢复后，由于从节点保存了自身已复制的偏移量和主节点的运行ID，因此会把他们当做psync参数发送给主节点，要求进行部分复制操作。

5）主节点接收到psync命令后首**先核对参数runid，如果一致**，说明之前复制的是当前主节点，之后根据参数offset在自身复制积压缓冲区查找，找到发送+CONTINUE响应。

6）**主节点根据偏移量把复制积压缓冲区里的数据发送给从节点，保证主从复制进入正常状态。**

4、注意点

- 从服务器也可以接受其他从服务器的连接
- **Redis主从复制不阻塞主服务器端。**也就是说当若干个从服务器在进行初始同步时，主服务器仍然可以处理请求。
- **主从复制也不阻塞从服务器端**
- 主从复制可以用来增强扩展性，使用多个从服务器来处理只读的请求（比如，繁重的排序操作可以放到从服务器去做），避免单个redis服务器的读写访问压力过大的问题，同时也可以简单的用来做数据冗余

1）无盘复制

在上面的全量同步过程中，master会将数据保存在rdb文件中然后发送给slave服务器，但是如果master上的磁盘空间很小怎么办呢？

那么此时全部同步对于master来说，将是一份十分有压力的操作了。此时可以通过无盘复制来达到目的，由master直接开启一个socket将rdb文件发送给slave服务器。（无盘复制一般应用在磁盘空间有限但是网 络状态良好的情况下）

2）关于slave服务器上过期键的处理

由master服务器负责键的过期删除处理，然后将相关删除命令已数据同步的方式同步给slave服务器，slave服务器根据删除命令删除本地的key。

3）当主服务器不进行持久化时复制的安全性

在进行主从复制设置时，**强烈建议在主服务器上开启持久化**，当不能这么做时，比如考虑到延迟的问题，**应该将实例配置为避免自动重启。**为什么不持久化的主服务器自动重启非常危险呢？ 为了更好的理解这个问题，看下面这个失败的例子，其中主服务器和从服务器中数据库都被删除了。设置节点A为主服务器，关闭持久化，节点B和C从节点A复制数据。 这时出现了一个崩溃，但Redis具有自动重启系统，重启了进程，因为关闭了持久化，节点重启后只有一个空的数据集。节点B和C从节点A进行复制，现在节点A是空的，所以节点B和C上的复制数据也会被删除。 当在高可用系统中使用Redis Sentinel，关闭了主服务器的持久化，并且允许自动重启，这种情况是很危险的。 比如主服务器可能在很短的时间就完成了重启，以至于Sentinel都无法检测到这次失败，那么上面说的这种失败的情况就发生了。如果数据比较重要，并且在使用主从复制时关闭了主服务器持久化功能的场景中，**都应该禁止实例自动重启。**

4）读写分离

- 复制数据存在延迟（如果从节点发生阻塞）
- 从节点可能发生故障

5）主从配置不一致

- 例如maxmemory不一致，可能会造成丢失数据

6）规避全量复制

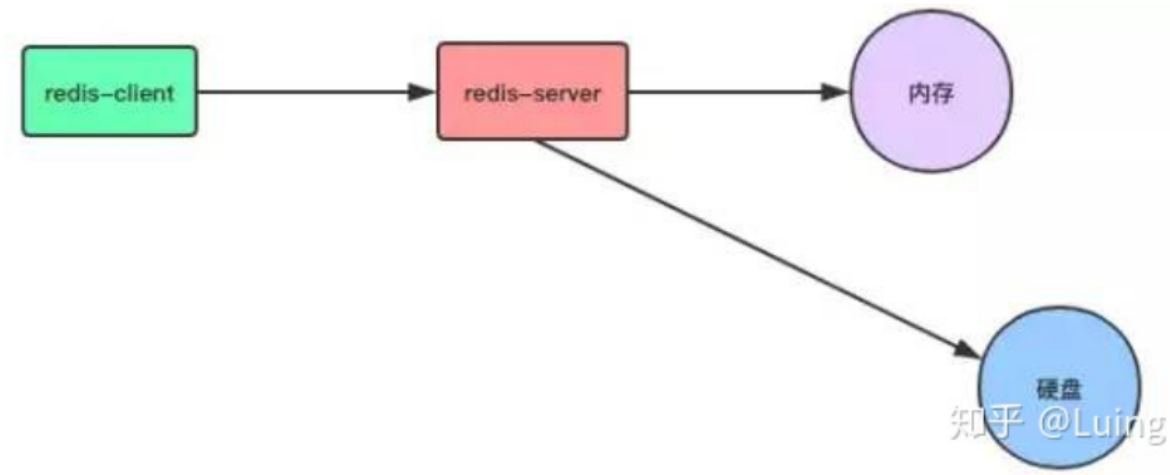
- 第一次全量复制不可避免，所以分片的maxmemory减小，同时选择在**低峰（夜间）**时，做全量复制。
- 复制积压**缓冲区**不足 增大复制缓冲区配置rel_backlog_size

7）复制风暴

master节点重启，master节点生成一份rdb文件，但是要给所有从节点发送rdb文件。对cpu，内存，带宽都造成很大的压力。

5、RDB 与 AOF 文件

为了避免内存中数据丢失，Redis提供了对持久化的支持，我们可以选择不同的方式将数据从内存中保存到硬盘当中，使数据可以持久化保存。



Redis本身，数据的持久化有两种形式，一种是**AOF**，另一种就是**RDB文件**的格式，**作为主从同步的模型，Redis采用的就是RDB的文件格式。**这里简单介绍一下RDB 和 AOF。

RDB

RDB是一种**快照存储**持久化方式，具体就是将Redis**某一时刻**的内存数据保存到**硬盘**的文件当中，默认保存的文件名为**dump.rdb**，而在Redis服务器**启动**时，会**重新加载**dump.rdb文件的数据到内存当中恢复数据。

相较于AOF的快照行为，RDB是对于内存中的数据，直接进行一次压缩和文件写入的过程，这种形式还原数据对于Redis来说，比AOF要快很多，但是对于生成来讲，可能性能就不尽如人意了，他是将数据，按照对应的**类型**（string，list，set，map，zset）按照特定的**格式**，存入文件里面，添加一些版本号和唯一标示，来识别RDB文件。

生成慢（添加一些版本号和唯一标示，来识别RDB文件），还原快

执行

- SAVE：**同步操作，会阻塞redis，整个Redis不工作来生成数据**，当客户端向服务器发送save命令请求进行持久化时，服务器会阻塞save命令之后的其他客户端的请求，直到数据同步完成。如果数据量太大，同步数据会执行很久，而这期间Redis服务器也无法接收其他请求，所以，最好不要在生产环境使用save命令。
- BGSAVE：开启另一个**线程**来处理，不影响Redis的正常工作。性能低下，注定他不能实时存储，通常情况下都是设置一些条件触发生成操作。
- 自动保存。Redis配置文件中设置了自动保存的触发机制，可以自定义修改，运行原理同bgsave。

与save命令相比，Redis服务器在处理bgsave采用子线程进行IO写入，而主进程仍然可以接收其他请求，而且RDB备份时通过fork子线程来完成，对主线程的性能影响更小。**但forks子进程是同步的，所以forks子进程时，一样不能接收其他请求，这意味着，如果forks一个子进程花费的时间太久**(一般是很快的)，bgsave命令仍然有阻塞其他客户的请求的情况发生。

save和bgsave的对比

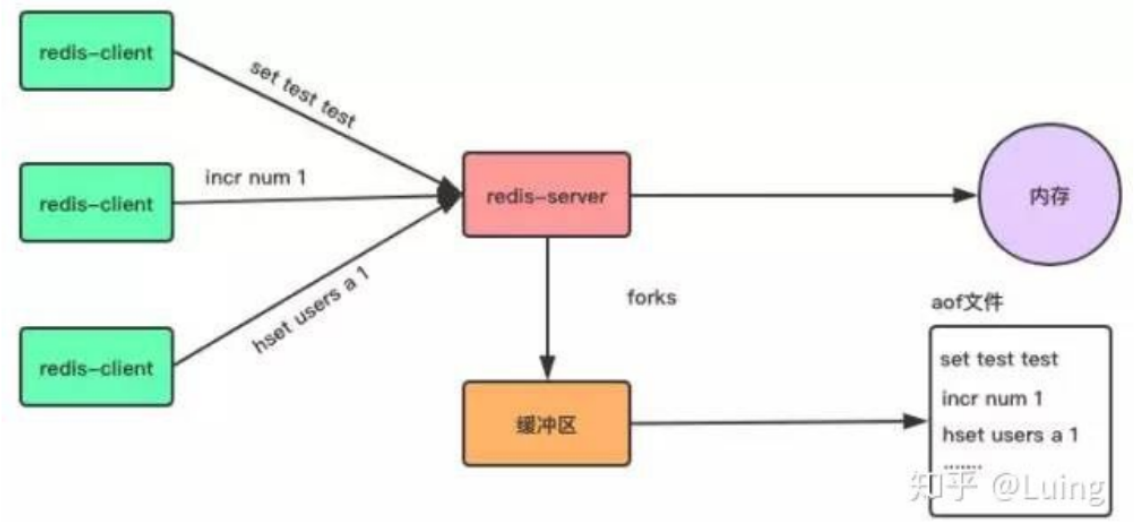
命令	Save	Bgsave
IO 类型	同步	异步
阻塞?	是	是(阻塞发生在fork)
复杂度	O(n)	0（n）
优点	不会消耗额外内存	不阻塞客户端命令
缺点	阻塞客户端命令	需要fork,消耗内存

AOF

AOF持久化方式会记录客户端对服务器的每一次写操作命令，并将这些写操作以Redis协议追加保存到以后缀为aof文件末尾，在Redis服务器重启时，会加载并运行aof文件的命令，以达到恢复数据的目的。

但是有两点点要注意一下：

1. **写入文件的过程不是实时的**，毕竟写入文件是有性能消耗的，所以会有一个**AOF缓冲区**来存储**最近**的操作，然后**定时写入**文件。
2. 对于一些命令，比如说反复对list的操作，结果集会越来越大，极端的例子是我们**for循环一组数据，逐一添加到list里面，对于AOF来说，就是几千条命令**，所以AOF会有一个**重写**的过程，具体的内容在这里不介绍了，可以参阅官网和相关书籍。



开启AOF持久化

```
# 开启aof机制
appendonly yes

# aof文件名
appendfilename "appendonly.aof"

# 写入策略,always表示每个写操作都保存到aof文件中,也可以是everysec或no
appendfsync always
```

```
# 默认不重写aof文件
no-appendfsync-on-rewrite no

# 保存目录
dir ~/redis/
```

执行方式

- always。Redis的每条写命令都写入到系统缓冲区，**然后每条写命令都使用fsync“写入”硬盘。**
- everysec。过程与always相同，**只是fsync的频率为1秒钟一次。这个是Redis默认配置**，如果系统宕机，会丢失一秒左右的数据
- no。由操作系统决定什么时候从系统缓冲区刷新到硬盘。

三种写入策略

在上面的配置文件中，我们可以通过appendfsync选项指定写入策略,有三个选

```
appendfsync always
# appendfsync everysec
# appendfsync no
```

1. always 客户端的每一个写操作都保存到aof文件当，这种策略很安全，但是每个写请注都有IO操作，所以也很慢。

2. everysec

appendfsync的默认写入策略，每秒写入一次aof文件，因此，最多可能会丢失1s的数据。

3. no

Redis服务器不负责写入aof，而是交由操作系统来处理什么时候写入aof文件。更快，但也是最不安全的选择，不推荐使用。

AOF文件重写 AOF将客户端的每一个写操作都追加到aof文件末尾，比如对一个key多次执行incr命令，这时候，aof保存每一次命令到aof文件中，aof文件会变得非常大。

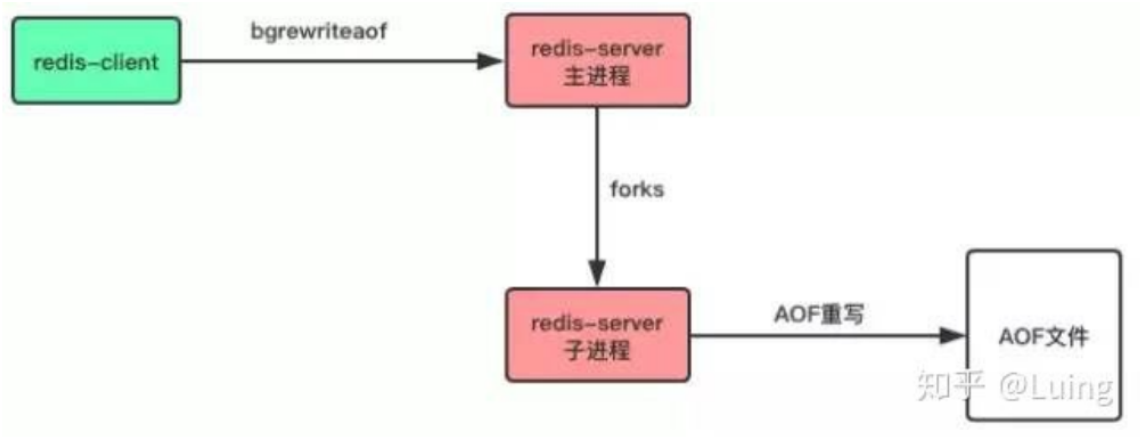
```
incr num 1
incr num 2
incr num 3
incr num 4
incr num 5
incr num 6
...
incr num 100000
```

aof文件太大，加载aof文件恢复数据时，就会非常慢，为了解决这个问题，Redis支持aof文件重写，通过重写aof，可以生成一个恢复当前数据的最少命令集，比如上面的例子中那么多条命令，可以重写为：

```
set num 100000
```

aof文件是一个二进制文件，并不是像上面的例子一样，直接保存每个命令，而使用Redis自己的格式，上面只是方便演示。

AOF重写方式也是异步操作，即如果要写入aof文件，则Redis主进程会forks一个子进程来处理，如下所示：



重写aof文件的好处

- 压缩aof文件，减少磁盘占用量。
- 将aof的命令压缩为最小命令集，加快了数据恢复的速度。

两者对比

AOF的优点

- AOF只是追加日志文件，因此**对服务器性能影响较小**，速度比RDB要快，消耗的内存较少

AOF的缺点

- AOF方式生成的日志文件太大，即使通过AFO重写，文件体积仍然很大。
- 恢复数据的速度比RDB慢。

速度快、恢复慢、体积大

RDB的优点

- 与AOF方式相比，通过rdb文件恢复数据比较快。
- rdb文件非常紧凑，适合于数据备份。
- 通过RDB进行数据备，**由于使用子进程生成，所以对Redis服务器性能影响较小。**

RDB的缺点

- **数据丢失的问题上，RDB由于不是实时的**，所以注定的在宕机的时候，会有一段时间空隙，比如我们设置十分钟同步一次，在这期间宕机了，那么这段时间内的数据会产生丢失的情况，
- 使用save命令会造成服务器阻塞，直接数据同步完成才能接收后续请求。
- 使用bgsave命令在forks子进程时，如果数据量太大，**forks的过程也会发生阻塞**，另外，forks子进程会耗费内存。

命令	RDB	AOF
启动优先级	低	高
体积	小	大
恢复速度	快	慢
数据安全性	丢数据	根据策略决定
轻重	重	轻

知乎 @Luing

自动保存

除了通过客户端发送命令外，还有一种方式，就是在Redis配置文件中的save指定到达触发RDB持久化的条件，比如【多少秒内至少达到多少写操作】就开启RDB数据同步。 例如我们可以在配置文件redis.conf指定如下的选项：

```
# 900s内至少达到一条写命令
save 900 1
# 300s内至少达至10条写命令
save 300 10
# 60s内至少达到10000条写命令
save 60 10000
```

这种通过服务器配置文件触发RDB的方式，与bgsave命令类似，达到触发条件时，会forks一个子进程进行数据同步。

6、总结

Redis主从复制可以根据是否是全量分为全量同步和增量同步。

【全量同步】

Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。具体步骤如下：
1）从节点连接上主节点，发送 SYNC 命令
2）主节点收到后，给从节点发送主节点的id和数据的偏移量；然后主节点 fork 出一个线程，用来执行 RDB 命令生成一个数据的快照文件，同时缓存生成快照文件这段时间继续收到的写命令
3）快照文件生成后发给从节点，从节点收到快照文件后，清空本地旧数据，加载新的快照文件
4）主节点发完快照文件后，继续发送缓存区中缓存的写命令，从节点收到后执行这些命令，保持数据同步

【增量同步】

上述的过程对于一个从机，新加入到集群里面的时候，比较合适，但是如果因为断线重连，则需要重新复制**主机上所有的内容**，主机也因此要进行一次**全量级的RDB**，比较耗费性能。
在2.8版本之后，采用了一个新的增量复制的过程，流程如下：
1）从节点根据保存的主节点的id找到主节点并连接，然后发送本地数据偏移量，要求从该偏移量处开始同步数据
2）主节点收到后，判断请求中的节点id是否是自己，是就接收其偏移量，然后从偏移量开始向从节点

同步数据
3）之后主节点每收到一个写命令，就会及时同步给从节点

参考资料

Redis 的主从同步，及两种高可用方式 https://blog.csdn.net/weixin_42711549/article/details/83061052

Redis主从同步及哨兵原理 https://www.cnblogs.com/dukuan/p/10119348.html

Redis主从复制原理总结 https://www.cnblogs.com/daofaziran/p/10978628.html

聊一聊RDB、AOF在redis持久化里的底层原理 https://zhuanlan.zhihu.com/p/80302920

Redis多路复用

0、前言

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消耗），且Redis在内存中操作数据的速度非常快（内存内的操作不会成为这里的性能瓶颈），主要以上两点造就了Redis具有很高的吞吐量。

有没有人和我一样, 自打知道了redis, 就一直听说什么redis单线程, 使用了多路复用等等. 天真的我以为多路复用是redis实现的技术. 今天才发现, 我被自己骗了, **多路复用是系统来实现的**. 对不起自己的专业了。

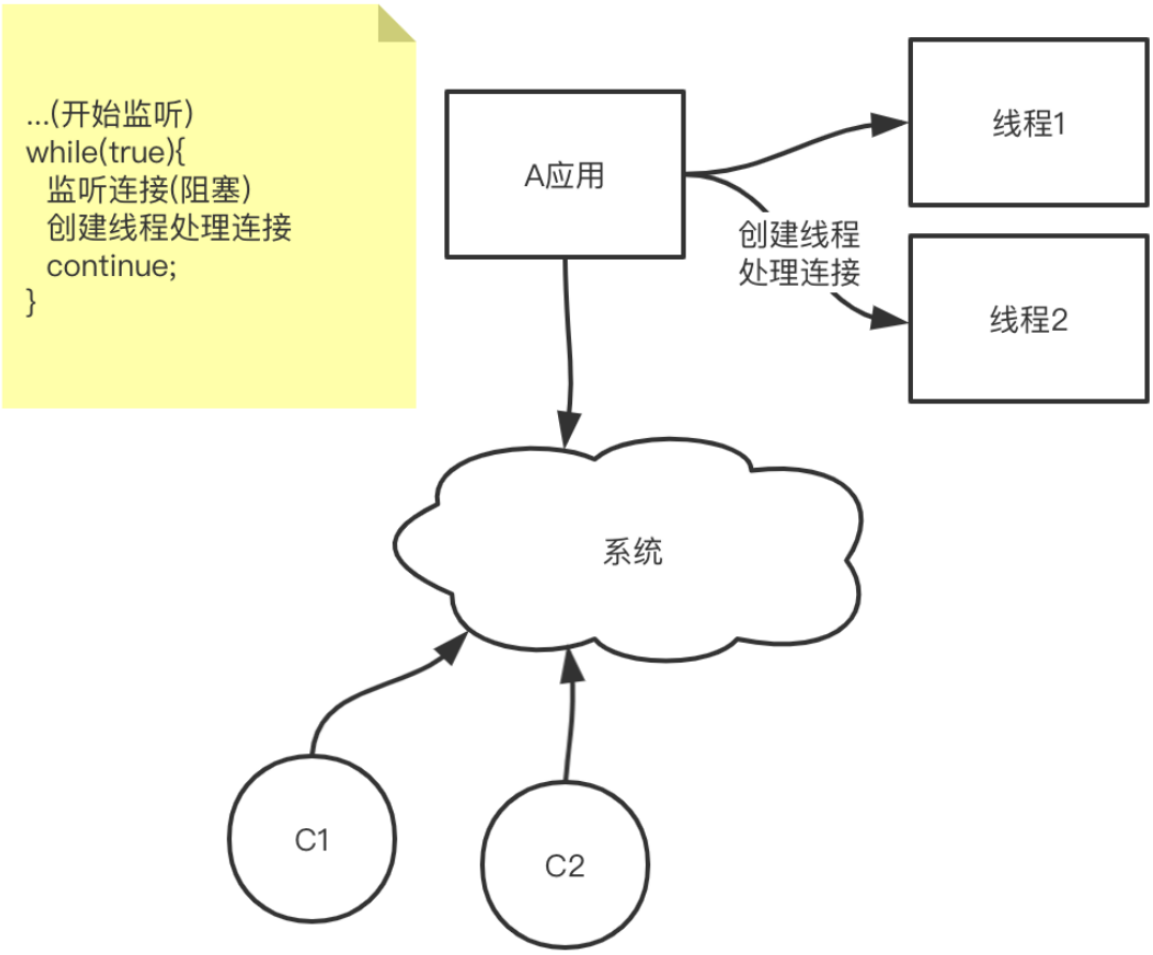
为了引出多路复用, 我来大胆设想一下技术的发展路程。

1、原始版本

一个应用程序, 想对外提供服务, 一般都是通过建立套接字监听端口来实现, 也就是`socket`. 在这个时候, 应用对外提供服务的过程大概是这样.

1. 创建套接字
2. 绑定端口号
3. 开始监听
4. 当监听到连接时, 调用系统read去读取内容, 但是读取操作是阻塞的(也就是说,如果主线程处理read,就不能接收其他连接了, 所以只能开新的线程去处理这个事情)

画个丑丑的流程图:



问题分析

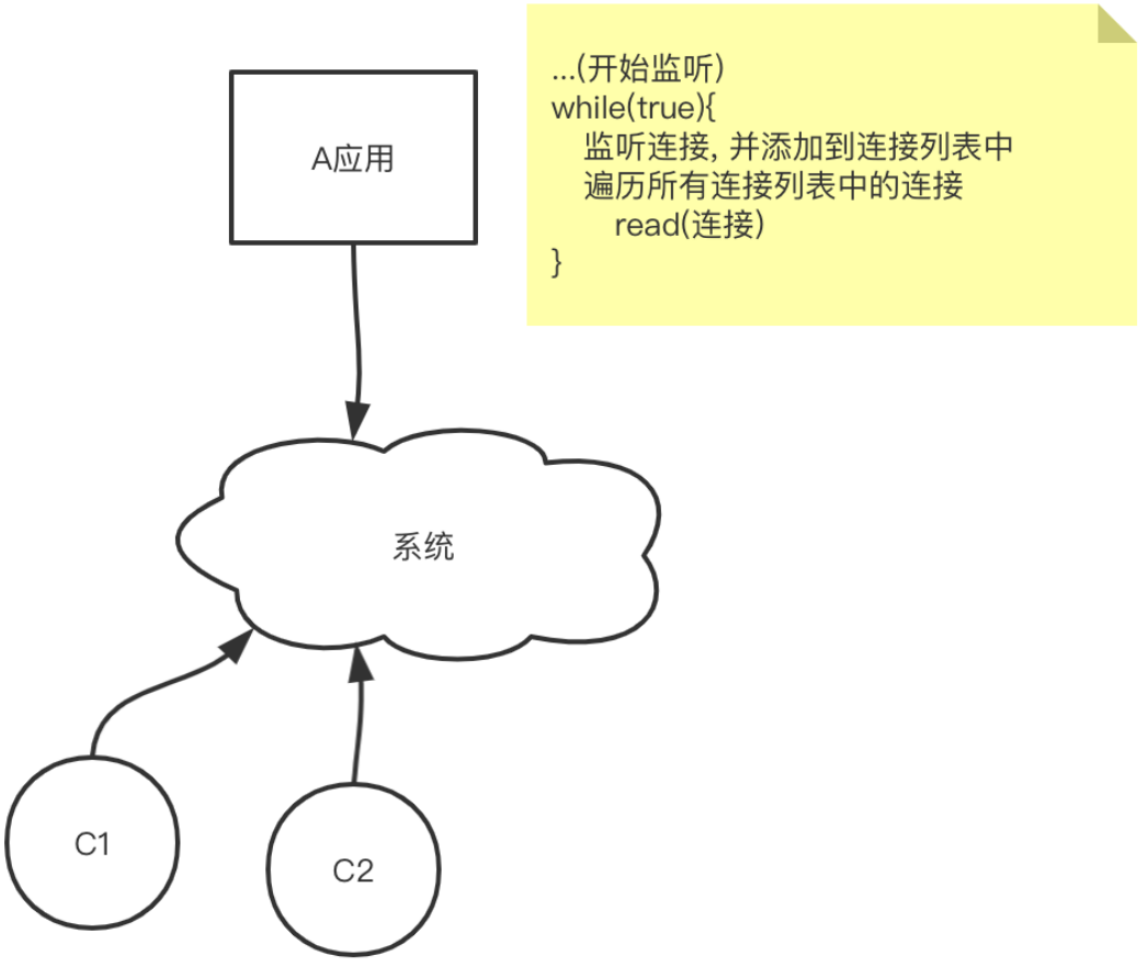
这个流程的问题很明显, 会不停的创建线程, 当然, 可以维护一个线程池. 但是**线程之间的不停切换也是消耗资源的**. 而且**也不可能无限的创建线程**. 那我如果想一个线程处理呢? 从上面流程图能看的出来, 问题出在`阻塞`上面. 如果read操作**可以立刻返回结果**, 如果没有读到数据, 就可以继续处理后边的事情了.

2、改进版本

整个简单版本. 主线程维护一个所有连接的列表, 每次循环读取所有列表, 有数据就处理, 没有就跳过.

1. 创建套接字
2. 绑定端口号
3. 开始监听
4. 监听到连接, 将连接加到连接列表中, 循环读取连接列表中的所有连接, 对有数据的进行处理

画个丑图:



问题分析

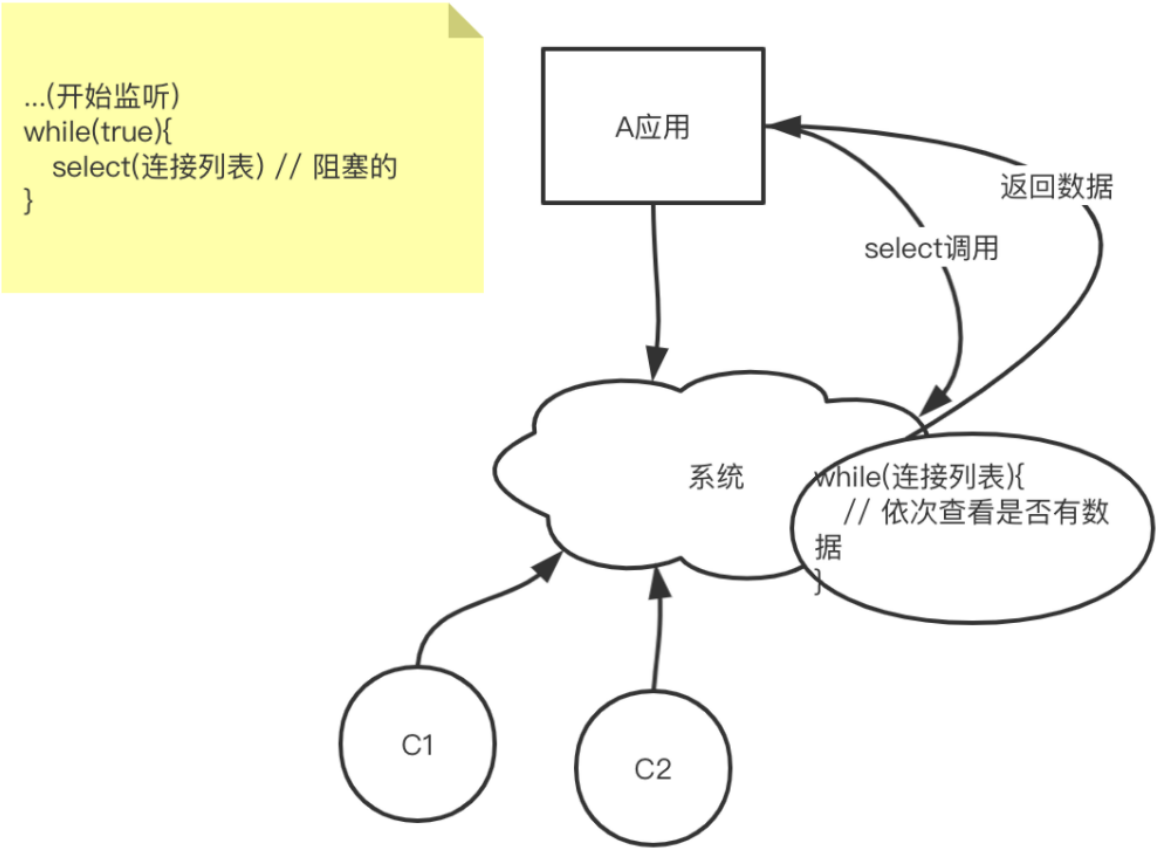
现在这样处理貌似是比开线程要好一些了, 但是事实是这样么? 众所周知, 其中的**read操作是调用系统函数**, 简单说就是要进行**进程的切换, 从用户进程切换到系统进程**. 连接少还好, 如果有十万个连接, 甚至更多呢? 每次循环都会频繁的调用系统函数, 可能十万次调用, 甚至**其中有数据的只有一次**, 其余调用都浪费掉了. 这无疑降低了性能.

其实解决方法说起来也很容易想到. 问题出在系统调用上, 频繁的系统调用导致了问题的出现. 如果能够**一次性**将需要查询的所有数据都发给系统, 让系统进行查询, 那不就只需要一次切换就可以了么?

3、Select版本

为了解决上面的问题. 可以批量将待查询的连接发给系统. 出现了`select`, 这就是说的`多路复用`了. 在系统 中, 无论是监听端口还是建立了连接, 程序拿到的都是一个`文件描述符`, 将这些`文件描述符`批量查询就是了.

直接上丑图:



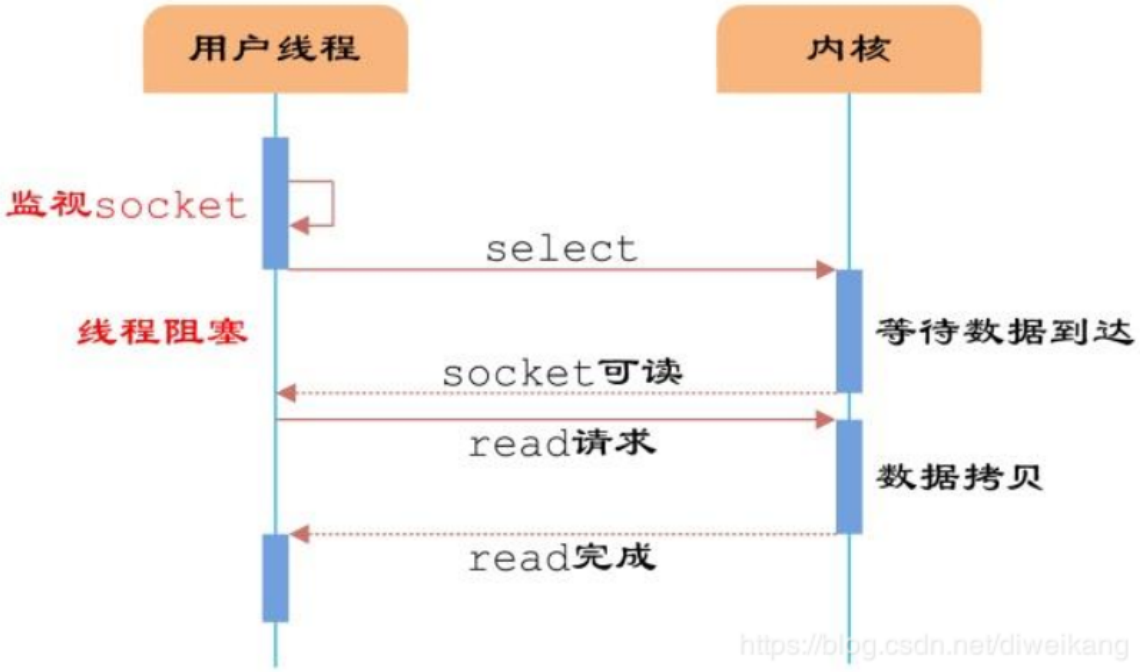
这里和上一个版本相比, **将循环检查交到了系统去做, 只发生一次进程的切换**. `select` 简单说, 就是你告诉系统, 你需要哪些数据, 你同帮你遍历找找, 然后将结果返回给你.

问题分析

这样确实要好上一些, 但是上面的问题还没有完全解决. 比如有10万个连接, **其中有数据的只有一个**, 那就回有9999次无效的操作, 虽然这些无效的操作是由系统做的, 但不一样么, **系统做的无效操作, 你应用程序也得等着**啊. 而且每次查询**都要把所有的需要的都传过去**, 10万个就要传10万了, 蓝瘦.

问题出在哪呢? 需要循环遍历, 是因为不知道哪些连接是有数据的, 所以只能一个一个的看. 如果**可以不需要遍历,** 直接知道哪些连接是有数据的, 然后直接拿到数据返回就好了.

原理分析



当client连接到server后，server会创建一个该连接的描述符fd，fd有三种状态，分别是读、写、异常，存放在三个fd_set（其实就是三个long型数组）中。

如上图所示，

1) 用户线程发起请求的时候，使用copy_from_user从用户空间拷贝fd_set到内核空间，这时**阻塞**等待select函数返回。

2) **内核遍历[0,nfds)范围内的每个fd，调用fd所对应的设备的驱动poll函数，poll函数可以检测fd的可用流（读流、写流、异常流）。当数据到达时，select**被**激活**，**select函数返回，否则阻塞住。**

3）此时用户线程才正式发起**read请求**，读取数据并继续执行。

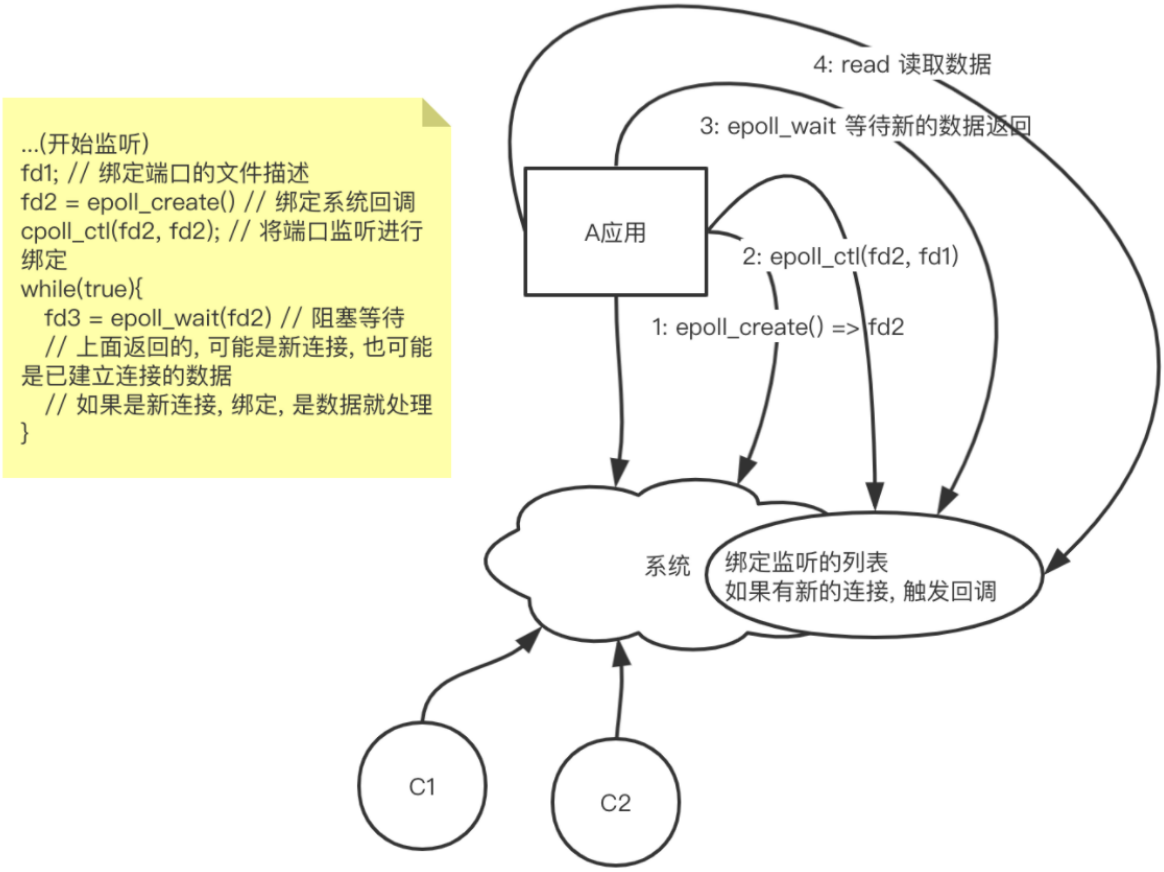
使用select最大的优势是用户可以在一个线程内同时处理多个socket的I/O请求。因为用户可以**注册多个**socket，然后**不断地调用select**读取被激活的socket（**但每次调用都需要将所有fd发给系统内核去轮询socket的状态，以查询数据是否到达**），即可达到在**同一个线程内同时处理多个I/O请求**的目的。

而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

4、Epoll版本

跟上一个版本相比, 现在不通过批量查询的方式了, 而是**通过回调的方式**. 简单说, 建立一个需要回调的连接, 将需要监听的文件描述符**都扔给他**, 当有新数据到达时, 会返回给你.

上丑图:



看着是不是有点晕了? 其实它和 `select` 版本的区别简单来说, `epoll` **是将你需要监听的列表交给系统维护**, 这样当有新数据来的时候, 系统知道这是你要的, 等你下次来拿的时候, 直接给你了, 少去了上面的系统遍历. 同时, 也没有 `select` 查询时那一大堆参数, **每次都只调用一次进行绑定即可**.

那系统是怎么知道新数据的到来呢? 这里靠的是**事件中断**, 忘得差不多了, 回头再看看.

`epoll` 简单说就是, 你告诉系统, 你需要哪些数据, 然后等着, 有数据了系统就**通知**你, 然后你去读.

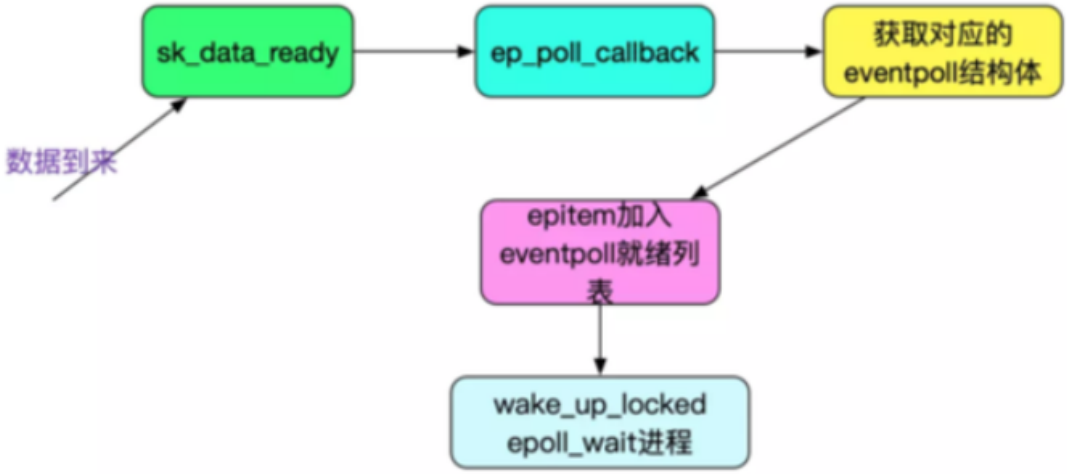
【数据结构】

eventpoll： 用于存放需要监控的fd事件，这些事件都会挂载在结构体重的红黑树中。而所有添加到eventpoll结构体中，**都会与设备(网卡)建立回调**

- rb： 红黑树的根节点，存储了所有添加到epoll中所需要监控的事件
- rdlist： 双向链表中存储着所有的通过epoll_wait返回给用户满足条件的事件

epoll_wait： 收集事件发生的fd

当有事件发生时候，相应的事件会回调这个回调方法。这个回调方法就是ep_poll_callback，他将发生的事件添加到eventpoll结构体中的rdlist双向链表中



调用epoll_create：linux内核会在epoll文件系统创建一个file节点，同时创建一个eventpoll结构体，结构体中有两个重要的成员：rbr是一棵红黑树，用于存放epoll_ctl注册的socket和事件；rdllist是一条双向链表，用于存放准备就绪的事件供epoll_wait调用。

调用epoll_ctl：会检测rbr中是否已经存在节点，有就返回，没有则新增，同时会向内核注册回调函数ep_poll_callback，**当有事件中断来临时，调用回调函数向rdllist中插入数据**，epoll_ctl也可以增删改事件。

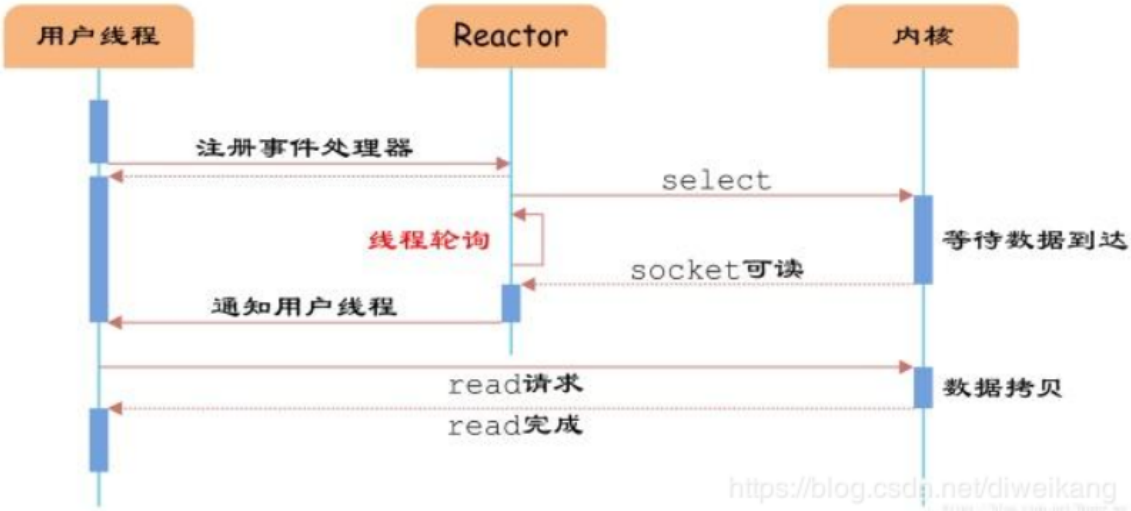
调用epoll_wait：返回或者判断rdllist中的数据即可。

【总结】

当应用程序调用 epoll_create时，系统会开辟一块属于epoll的空间，存储一个节点，节点中有一颗红黑树用来存储将要注册的监听事件以及fd，还有一个就绪队列，用来存储已经准备的就绪事件。

然后应用程序调用 epoll_ctl 开始向系统内核注册监听的事件和fd，然后系统会将监听事件放入红黑树中，然后建立一个到网卡的回调函数，之后应用程序再调用 epoll_wait 阻塞等待系统的就绪事件到来，当有数据到来时，就会通过回调方法将就绪事件放入就绪列表中，然后从 wait 函数中返回就绪的 fd 和事件，相当于激活阻塞的应用程序，应用程序再通过相应的fd和事件，进行对应的操作，比如读写等。

5、Reactor（反应器模式）



如上图，I/O多路复用模型使用了Reactor设计模式实现了这一机制。

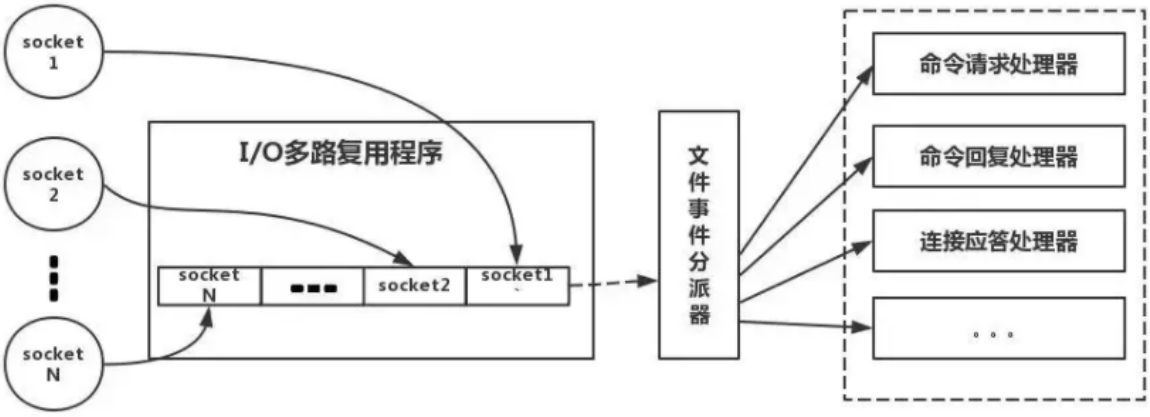
通过Reactor的方式，可以将用户线程轮询I/O操作状态的工作统一交给handle_events**事件处理器循环进行处理**

用户线程注册事件处理器之后可以继续**执行做其他的工作（异步）**，而Reactor线程负责调用内核的**select**函数检查socket状态。当有socket被激活时，则**通知**相应的用户线程（或执行用户线程的**回调**函数），执行handle_event进行**数据读取、处理**的工作。

由于select函数是阻塞的，因此多路I/O复用模型也被称为**异步阻塞I/O模型**。注意，这里的所说的阻塞是指select函数执行时线程被阻塞，而不是指socket。一般在使用I/O多路复用模型时，socket都是设置为NONBLOCK的，不过这并不会产生影响，因为用户发起I/O请求时，数据已经到达了，用户线程一定不会被阻塞。

【基本原理】

- 同步事件分离器，阻塞等待Handles中的事件发生，就是不停的轮询fd中已就绪的事件。
- 初始分派器，作用为添加和删除Event handler（事件处理器）以及分派事件给Event handler。同步事件分离器负责等待新事件发生，事件发生时通知Initiation Dispatcher，然后Initiation Dispatcher调用event handler相应的事件处理器处理事件。



Redis-client 在操作的时候，会产生具有不同事件类型的 Socket。在服务端，有一段 I/O 多路复用程序，将其置入队列之中。然后，文件事件分派器，依次去队列中取，转发到不同的事件处理器中，再根据不同的处理逻辑，或通知用户线程执行相应的动作。

6、与多线程的比较

- 跟多线程相比较，线程切换需要切换到内核进行线程切换，需要消耗时间和资源。
- 而I/O多路复用不需要切换线/进程，效率相对较高，特别是对高并发的应用nginx就是用I/O多路复用，故而性能极佳。
- 但多线程编程逻辑和处理上比I/O多路复用简单，而I/O多路复用处理起来较为复杂。

7、几种模型（select、poll、epoll）的比较

【select、poll】

首先select、poll都是传递一个 **fd的集合**调用操作系统的API。其中select 模式有大小的限制最大只能是**1024**的fd，而 poll是传递一个pollfd数组，没有大小限制。

select/poll的几大缺点：

1.每次调用select/poll,都需要把**fd集合**从**用户态copy到内核态**，如果是fd集合很大的时候，开销会很大

2.每次调用select/poll,都要在内核态中**循环遍历**fd集合，去判断哪些fd准备好了数据，这个是在fd集合很大的情况下面，开销会很大

3.针对select支持的文件描述符数量太小了，默认是1024

4.select**返回的是整个句柄的数组**，需要**全部遍历一遍**才知道那些fd有事件的产生

相比于poll,因为poll使用的是链表保存fd，所以没有fd大小数量的限制，但是其他的缺点依然存在

【epoll】

epoll 支持的fd上限为**文件打开数**。这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右。

设想一下如下场景：

有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的(事实上大部分场景都是这种情况)。如何实现这样的高并发?

在select/poll时代，**会把这些fd从用户态复制到内核态**，然后**让操作系统去查询**这些套接字上是否有事件发生，轮询完以后**，在将这些句柄集合返回**，从**内核态复制会用户态**，让**用户应用程序遍历轮询**去获取有哪些句柄有事件的发上，这个过程大量的**浪费了性能**。因此，select和poll 一般只能处理几千的并发。但是epoll的设计和select/poll的设计完全不同。他不需要用户应用程序循环遍历所有的句柄数组才能知道有哪些句柄有事件的发生，因此epoll能支持更大的并发。

epoll总结：

使用红黑树而不是数组存放描述符和事件，增删改查非常高效，轻易可处理大量并发连接。红黑树及双向链表都在内核cache中，避免拷贝开销。采用回调机制，事件的发生只需关注rdllist双向链表即可。

8、总结

BIO：每来一个请求，生成一个线程来处理，线程调用系统的 read 函数阻塞式地等待客户端准备好数据。缺点就是线程创建太多，资源的占用高。如果采用线程池的方式，则会造成线程频繁切换的开销。

NIO：非阻塞 IO ，通常是用一个列表来保存客户端的连接，然后用一个线程遍历所有的连接，依次调用 read 函数来获取数据，如果数据还没有准备好，则不用阻塞式地等待，过段时间再来轮询。缺点是请求太多时，频繁调用系统函数，会造成很大开销，因为要从用户态转向内核态。

select：只执行一次系统调用，然后把所有的文件描述符的连接发给内核，内核去遍历，有数据到达便从 select 中返回。好处是只需要进行一次系统调用，缺点是每次都需要把很多请求发给内核，而内核发回来的数组中即包含了有数据的fd，也包含了没有数据的fd，而且大部分都是没有数据的，用户程序收到以后还要去遍历再探查有没有数据。

参考资料

redis的多路复用是什么鬼 https://www.cnblogs.com/hujingnb/p/12439661.html

Redis I/O 多路复用 https://blog.csdn.net/diweikang/article/details/90346020

redis多路复用的理解 https://www.cnblogs.com/histlyb/p/12155934.html

IO多路复用：Redis中经典的Reactor设计模式 https://baijiahao.baidu.com/s?id=1663589997196141368&wfr=spider&for=pc

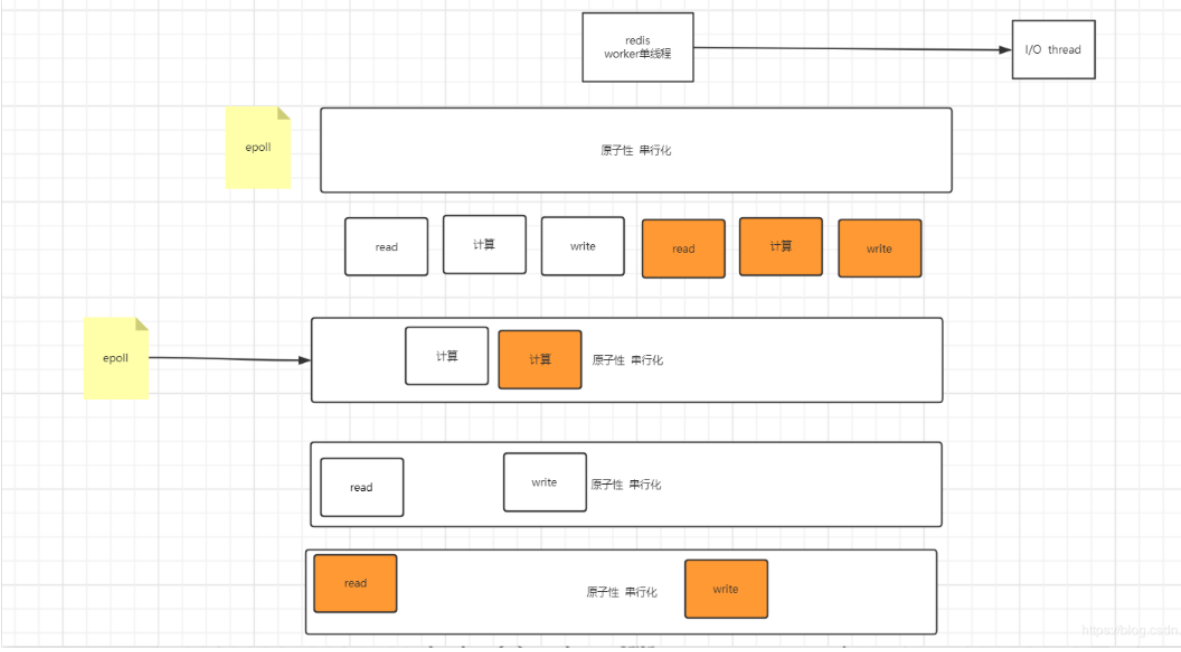
Redis IO多路复用 https://www.jianshu.com/p/511b98074052

Redis线程模型

0、前言

Redis内部使用文件事件处理器来处理客户端的请求，而这个文件事件处理器是单线程的，所以又称其为Redis的单线程模型。

它采用非阻塞的IO多路复用机制同时监听多个socket，根据socket上的事件来选择对应的事件处理器进行处理。



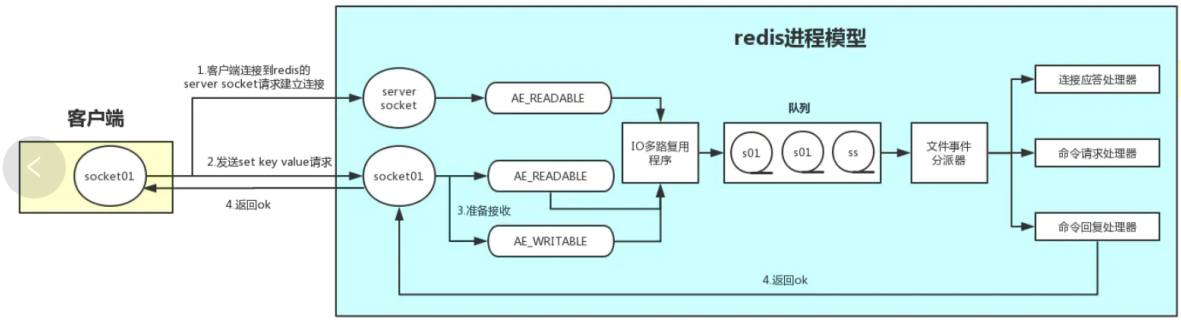
redis采用epoll_wait轮询地处理任务。

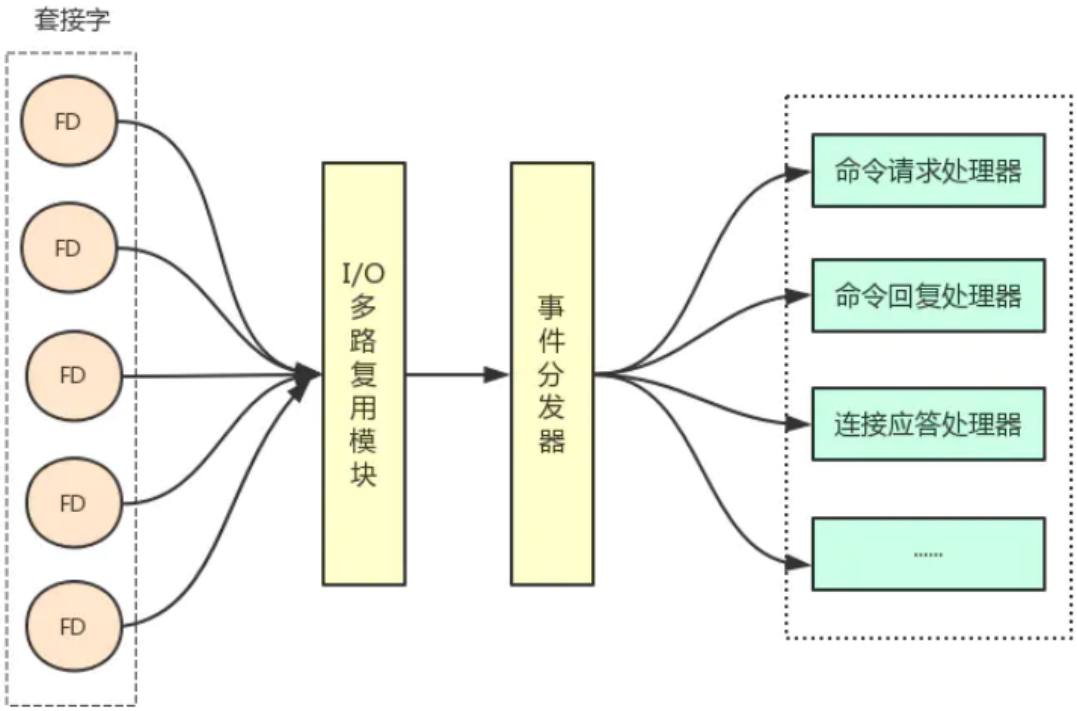
1、线程模型

线程模型（文件事件处理器）包括以下部分：

- 多个Socket
- IO多路复用程序
- 文件事件分发器
- 事件处理器（**连接应答处理器、命令请求处理器、命令回复处理器**）

多个 socket 可能会并发产生不同的操作，每个操作对应不同的文件事件，但是 IO 多路复用程序会监听多个 socket，会将 socket 产生的事件放入队列中排队，事件分派器每次从队列中取出一个事件，把该事件交给对应的事件处理器进行处理。





代码：

```
//从 epoll 中获关注的事件
numevents = aeApiPoll(eventLoop, tvp);
for (j = 0; j < numevents; j++) {
    // 从已就绪数组中获取事件
    aeFileEvent *fe = &eventLoop->events[eventLoop->fired[j].fd];

    int mask = eventLoop->fired[j].mask;
    int fd = eventLoop->fired[j].fd;
    int rfired = 0;

    // 读事件
    if (fe->mask & mask & AE_READABLE) {
        // rfired 确保读/写事件只能执行其中一个
        rfired = 1;
        fe->rfileProc(eventLoop,fd,fe->clientData,mask);
    }
    // 写事件
    if (fe->mask & mask & AE_WRITABLE) {
        if (!rfired || fe->wfileProc != fe->rfileProc)
            fe->wfileProc(eventLoop,fd,fe->clientData,mask);
    }

    processed++;
```

```
}
```

1）事先将要监听的事件挂在树上，建立起事件与IO设备的回调，如果事件就绪了，就会产生硬中断，根据中断号判断是哪个硬件，然后多路复用程序将树上的与该硬件的匹配的fd和事件搬到就绪队列中，相当于是回调的方式。比如客户端 socket01 向 redis 的 server socket 请求建立连接，此时 server socket 会产生一个 AE_READABLE 事件，IO 多路复用程序监听到 server socket 产生的事件后（**epoll网卡回调到epoll**），将该事件压入队列中（**epoll空间中的就绪队列**）。文件事件分派器从队列中获取该事件（**epoll_wait轮询到队列有事件产生**），交给连接应答处理器（**激活相应的处理器**）。连接应答处理器会创建一个能与客户端通信的 socket01，**并将该 socket01 的 AE_READABLE 事件与命令请求处理器关联。**

2）假设此时客户端发送了一个 set key value 请求，此时 redis 中的 socket01 会产生 AE_READABLE 事件，IO 多路复用程序将事件压入队列，此时事件分派器从队列中获取到该事件，由于前面 socket01 的 AE_READABLE 事件已经与命令请求处理器关联，因此事件分派器将事件交给命令请求处理器来处理。命令请求处理器读取 socket01 的 key value 并在自己内存中完成 key value 的设置。操作完成后，**它会将 socket01 的 AE_WRITABLE 事件与命令回复处理器关联。**

3）如果此时客户端准备好接收返回结果了，那么 redis 中的 socket01 会产生一个 AE_WRITABLE 事件，同样压入队列中，事件分派器找到相关联的命令回复处理器，由命令回复处理器对 socket01 输入本次操作的一个结果，比如 ok，**之后解除 socket01 的 AE_WRITABLE 事件与命令回复处理器的关联。**

这样便完成了一次通信。

总结

Redis 底层通过 IO 多路复用模型将客户端产生的事件放入一个就绪队列中，Redis 单线程程序不断轮询该队列，一旦发现有事件就绪了，就将事件取出，然后根据事件的类型分派给不同的具体的事件处理器去处理。所以事件驱动机制是一种编程的范式，通常是基于策略模式将持有处理逻辑的各个处理器先准备好，比如可以在 bean 初始化之后注册到一个 Map 中，key 就是对应的事件的类型。

2、文件事件

当 Socket 变得可读时（比如客户端对redis执行**write**操作，或者**close**操作），或者有新的可以应答的 Sccket 出现时（客户端对redis执行**connect**操作），Socket就会产生一个**AE_READABLE**事件。当 Socket 变得可写的时候（客户端对redis执行**read**操作），Socket 会产生一个**AE_WRITABLE**事件。

IO 多路复用程序可以同时监听 AE_REABLE 和 AE_WRITABLE 两种事件，如果一个Socket同时产生了这两种事件，那么文件事件分派器优先处理 **AE_READABLE** 事件，然后才是 **AE_WRITABLE** 事件。

3、文件事件处理器

如果是客户端要连接redis，那么会为 Socket 关联连接应答处理器。

如果是客户端要写数据到redis，那么会为 Socket 关联命令请求处理器。

如果是客户端要从redis读数据，那么会为 Socket 关联命令回复处理器。

4、Redis效率高的原因

- **纯内存操作**

- **核心是基于非阻塞的 IO 多路复用机制**

有了非阻塞 IO 意味着线程在读写 IO 时可以不必再阻塞了，读写可以瞬间完成然后线程可以继续干别的事了（**epoll_wait轮询就绪队列，有则取出处理，无则处理其他事情**）。

redis 需要处理多个 IO 请求，同时把每个请求的结果返回给客户端。由于 redis 是单线程模型，同一时间只能处理一个 IO 事件，于是 redis 需要在合适的时间暂停对某个 IO 事件的处理，转而去处理另一个 IO 事件，这就需要用到IO多路复用技术了。

- **单线程反而避免了多线程的频繁上下文切换问题**

redis 为什么快?

完全基于内存,绝大部分请求是纯粹的内存操作，数据在内存的字典中,字典的结构类似于HashMap,查找和操作的时间复杂度都是`o(1)`

数据结构简单,对数据结构的操作也简单,Redis基于C语言精心设计了很多数据结构。

采用了单线程,避免了不必要的上下文切换和竞态条件，也不存在多线程切换消耗CPU，不用考虑各种并发锁的问题，也不会出现死锁导致性能的消耗。

使用I/O多路复用模型，非堵塞的IO

为什么是单线程?

而且这个单线程指的是接收网络请求只用一个线程处理，实际上还会有一些子线程的，比如刷盘、主从同步、交换心跳信息等

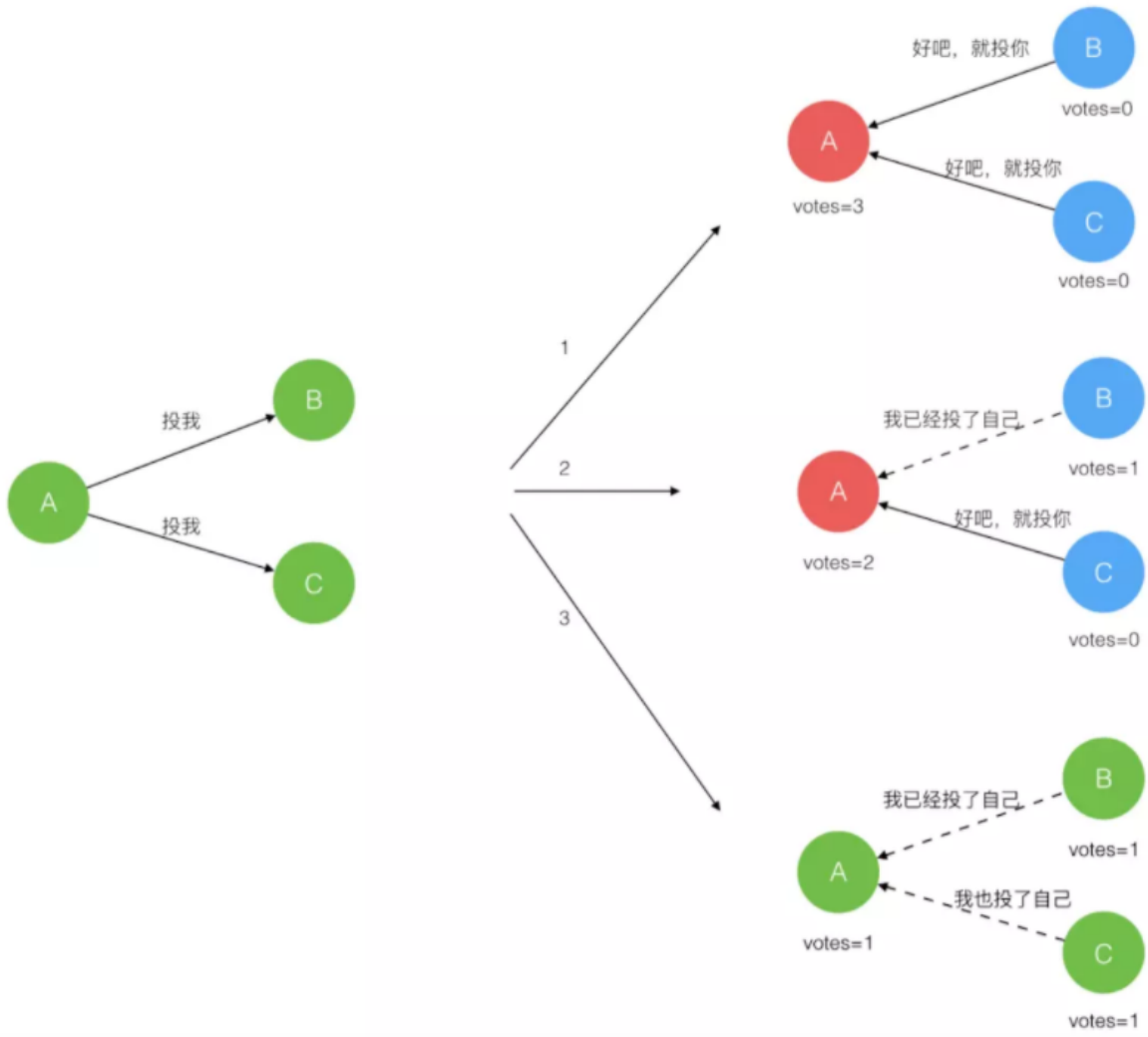
参考资料

Redis线程模型 https://www.jianshu.com/p/8f2fb61097b8

「面试突击」Redis的线程模型了解吗？为啥单线程效率还这么高？ https://baijiahao.baidu.com/s?id=1655867650424958128&wfr=spider&for=pc

Raft算法

1、原理



在Raft中，每个结点会处于下面三种状态中的一种：

1、follower：所有结点都以follower的状态开始。如果没收到leader的心跳消息则会变成candidate状态。

2、candidate：会向其他结点“拉选票”，如果得到大部分的票则成为leader。这个过程就叫做Leader选举(Leader Election)。

3、leader：所有对系统的修改都会先经过leader。每个修改都会写一条日志(log entry)。leader收到修改请求后的过程如下，这个过程叫做日志复制(Log Replication)：和zk一样

- **复制日志到所有follower结点(replicate entry)**
- **大部分结点响应**
- **leader提交日志**
- **通知所有follower结点日志已提交**
- **所有follower也提交日志**
- **现在整个系统处于一致的状态**

2、leader选举

当follower在选举超时时间（150~300ms之间的随机数）内未收到leader的心跳消息(append entries)，则变成candidate状态，然后开始进行 leader 选举。

成为candidate的结点首先会重置自己的选举计时器，然后投自己一票，接着发送拉票信息给其他节点。

如果这个过程中收到其他节点发送的已当选 leader 的信息后，当前节点就会变成 follower，选举结束。

如果接收结点在新term内没有投过票那它就会投给此candidate，并重置它自己的选举超时时间。candidate拉到大部分选票就会成为leader，并发送消息告诉其他候选者自己已当选的信息。

如果恰好有两个候选者都去拉取选票，而选票恰好一样，那么此轮选举结束，然后等待哪个节点的随机超时选举时间到，再发起新一轮的选举，直到选举出结果为止。

3、日志复制

当发生改变时，leader会复制日志给follower结点，这也是通过Append Entries心跳消息完成的。前面已经列举了Log Replication的过程，这里就不重复了。

Raft能够正确地处理网络分区（“脑裂”）问题。假设A~E五个结点，B是leader。如果发生“脑裂”，A、B成为一个子分区，C、D、E成为一个子分区。此时C、D、E会发生选举，选出C作为新term的leader。这样我们在两个子分区内就有了不同term的两个leader。这时如果有客户端写A时，因为B无法复制日志到大部分follower所以日志处于uncommitted未提交状态。而同时另一个客户端对C的写操作却能够正确完成，因为C是新的leader，它只知道D和E。

当网络通信恢复，B能够发送心跳给C、D、E了，却发现“改朝换代”了，因为C的term值更大，所以B自动降格为follower。然后A和B都回滚未提交的日志，并从新leader那里复制最新的日志。

4、哨兵选举流程

Sentinel集群正常运行的时候每个节点epoch相同，**当需要故障转移的时候会在集群中选出Leader执行故障转移操作。**Sentinel采用了Raft协议实现了Sentinel间选举Leader的算法，不过也不完全跟论文描述的步骤一致。**Sentinel集群运行过程中故障转移完成，所有Sentinel又会恢复平等。Leader仅仅是故障转移操作出现的角色。**

流程

- 1、某个Sentinel认定master客观下线的节点后，该Sentinel会先看看自己有没有投过票，如果自己已经投过票给其他Sentinel了，在2倍故障转移的超时时间自己就不会成为Leader。相当于它是一个Follower。
- 2、如果该Sentinel还没投过票，那么它就成为Candidate。
- 3、和Raft协议描述的一样，成为Candidate，Sentinel需要完成几件事情
  - 1）**更新故障转移状态为start**
  - 2）当前epoch加1，相当于进入一个新term，在Sentinel中epoch就是Raft协议中的term。
  - 3）**重置超时定时器。**更新自己的超时时间为当前时间随机加上一段时间，随机时间为1s内的随机毫秒数。
  - 4）向其他节点发送`is-master-down-by-addr`命令请求投票。命令会带上自己的epoch（拉选票）。
  - 5）给自己投一票，在Sentinel中，投票的方式是把自己master结构体里的leader和leader_epoch改成投给的Sentinel和它的epoch。
- 4、其他Sentinel会收到Candidate的`is-master-down-by-addr`命令。如果Sentinel当前epoch和Candidate传给他的epoch一样，说明他已经把自己master结构体里的leader和leader_epoch改成其他Candidate，相当于把票投给了其他Candidate。投过票给别的Sentinel后，在当前epoch内自己就只能成为Follower。

- 5、Candidate会不断的统计自己的票数，直到他发现认同他成为Leader的票数超过一半而且超过它配置的quorum（quorum可以参考《redis sentinel设计与实现》）。Sentinel比Raft协议增加了quorum，这样一个Sentinel能否当选Leader还取决于它配置的quorum。
- 6、如果在一个选举时间内，Candidate没有获得超过一半且超过它配置的quorum的票数，自己的这次选举就失败了。
- 7、**如果在一个epoch内，没有一个Candidate获得更多的票数。那么等待超过2倍故障转移的超时时间后，Candidate增加epoch重新投票。**
- 8、如果某个Candidate获得超过一半且超过它配置的quorum的票数，那么它就成为了Leader。
- 9、**与Raft协议不同，Leader并不会把自己成为Leader的消息发给其他Sentinel。其他Sentinel等待Leader从slave选出master后，检测到新的master正常工作后，就会去掉客观下线的标识，从而不需要进入故障转移流程。**

参考资料

raft动画演示 http://thesecretlivesofdata.com/raft/

两分钟带你了解分布式一致性协议—Raft协议 https://www.bilibili.com/video/BV1Tb411L7Cb/?spm_id_from=333.788.videocard.3

好玩的Raft动画演示，原理秒懂 https://www.jianshu.com/p/98e2b1f1c1fa

分布式锁

1.什么是分布式锁

要介绍分布式锁，首先要提到与分布式锁相对应的是线程锁、进程锁。

1.线程锁

主要用来给方法、代码块加锁。当某个方法或代码使用锁，在同一时刻仅有一个线程执行该方法或该代码段。线程锁只在同一JVM中有效果，因为线程锁的实现在根本上是依靠线程之间共享内存实现的，比如Synchronized、Lock等。

2.进程锁

为了控制同一操作系统中多个进程访问某个共享资源，因为进程具有独立性，各个进程无法访问其他进程的资源，因此无法通过synchronized等线程锁实现进程锁。

3.分布式锁

当多个进程不在同一个系统中，用分布式锁控制多个进程对资源的访问。

2.分布式锁的由来

在传统单机部署的情况下，可以使用Java并发处理相关的API(如ReentrantLcok或synchronized)进行互斥控制。

但是在分布式系统后，由于分布式系统多线程、多进程并且分布在不同机器上，这将使原单机并发控制锁策略失效，为了解决这个问题就需要一种跨JVM的互斥机制来控制共享资源的访问，这就是分布式锁的由来。

当多个进程不在同一个系统中，就需要用分布式锁控制多个进程对资源的访问。

3.分布式锁的特点

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

1）可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。

2）这把锁要是一把可重入锁（避免死锁）

3）这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）

4）有高可用的获取锁和释放锁功能

5）获取锁和释放锁的性能要好

不该释放的锁

但是，直接执行 `del mylock` 是有问题的，我们不能直接执行 `del mylock` 为什么？—— 会导致“信号错误”，**释放了不该释放的锁** 。假设如下场景：

时间线	线程1	线程2	线程3
时刻1	执行 setnx mylock val1 加锁	执行 setnx mylock val2 加锁	执行 setnx mylock val2 加锁
时刻2	加锁成功	加锁失败	加锁失败
时刻3	执行任务...	尝试加锁...	尝试加锁...
时刻4	任务继续（锁超时，自动释放了）	setnx 获得了锁（因为线程1的锁超时释放了）	仍然尝试加锁...
时刻5	任务完毕，del mylock 释放锁	执行任务中...	获得了锁（因为线程1释放了线程2的）
...			

4.CAP理论

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

所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

5.实现方案

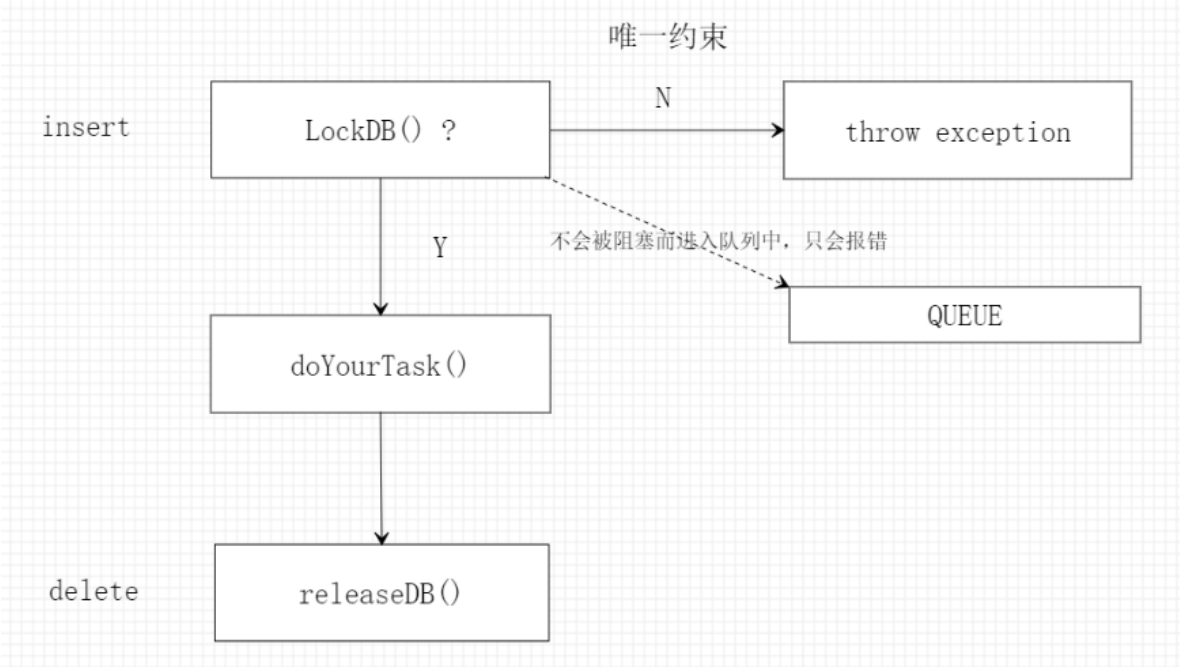
1.数据库实现

2.基于缓存（redis，memcached等）实现

3.Zookeeper实现分布式锁

6.基于数据库实现

1）基于唯一约束



问题：

a. 单点、强依赖

b. 死锁问题（无失效时间）

c. 非阻塞锁

d. 不可重入（唯一索引）

解决：

a. 主从同步

b. 定时任务、定期清理

c. while循环，insert成功才返回

d. 表加字段，记录主机ip与线程信息，是同一个的话就允许先删除再插入

要实现分布式锁，最简单的方式可能就是直接创建一张锁表，然后通过操作该表中的数据来实现了。

当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

创建这样一张数据库表：

```
CREATE TABLE `methodLock` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `method_name` varchar(64) NOT NULL DEFAULT '' COMMENT '锁定的方法名',
  `desc` varchar(1024) NOT NULL DEFAULT '备注信息',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '保存数据时间，自动生成',
  PRIMARY KEY (`id`),
  UNIQUE KEY `uidx_method_name` (`method_name `) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';
```

当我们想要锁住某个方法时，执行以下SQL：

```
insert into methodLock(method_name,desc) values ('method_name','desc')
```

因为我们对method_name做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

当方法执行完毕之后，想要释放锁的话，需要执行以下Sql:

```
delete from methodLock where method_name ='method_name'
```

上面这种简单的实现有以下几个问题：

> 1、这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。
>
> 2、这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。
>
> 3、这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。
>
> 4、这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

当然，我们也可以有其他方式解决上面的问题。

- 数据库是单点？搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。
- 没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。
- 非阻塞的？搞一个while循环，直到insert成功再返回成功。
- 非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

2）基于排它锁

innodb：排它锁 = for update + 唯一索引

释放：commit()

问题：

mysql查询优化（小表）、连接池连接资源的问题

解决：

解决阻塞、解决死锁

除了可以通过增删操作数据表中的记录以外，其实还可以借助数据中自带的锁来实现分布式的锁。

我们还用刚刚创建的那张数据库表。可以通过数据库的排他锁来实现分布式锁。 基于MySql的InnoDB引擎，可以使用以下方法来实现加锁操作：

```
public boolean lock(){
    connection.setAutoCommit(false)
    while(true){
        try{
            result = select * from methodLock where method_name=xxx for update;
            if(result==null){
                return true;
            }
        }catch(Exception e){

        }
        sleep(1000);
    }
    return false;
}
```

在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁（**这里再多提一句，InnoDB引擎在加锁的时候，只有通过索引进行检索的时候才会使用行级锁，否则会使用表级锁。这里我们希望使用行级锁，就要给method_name添加索引，值得注意的是，这个索引一定要创建成唯一索引，否则会出现多个重载方法之间无法同时被访问的问题。重载方法的话建议把参数类型也加上。**）。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。

我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，再通过以下方法解锁：

```
public void unlock(){
    connection.commit();
}
```

通过connection.commit()操作来释放锁。

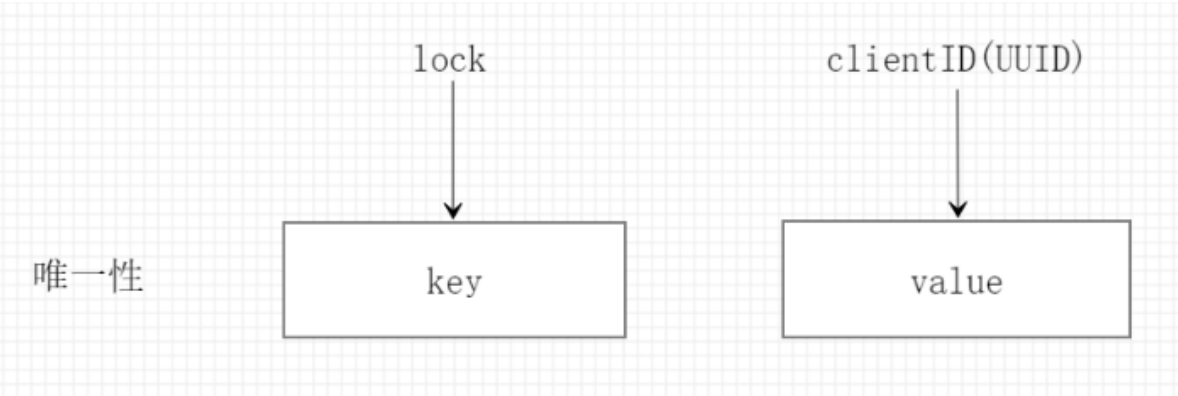
这种方法可以有效的解决上面提到的无法释放锁和阻塞锁的问题。

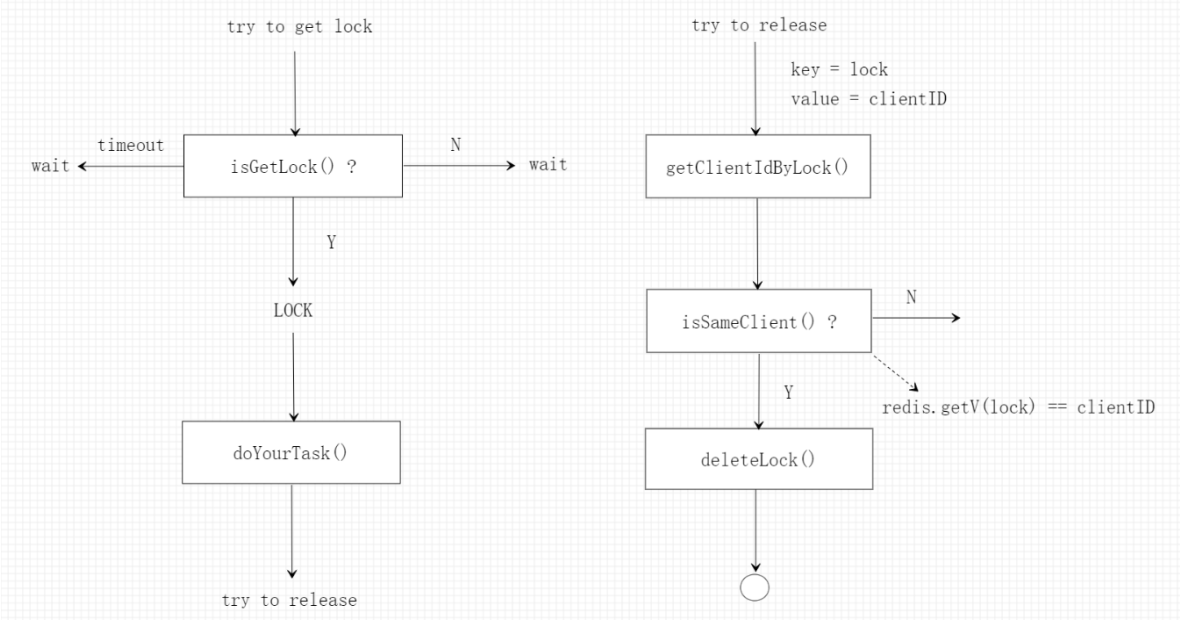
- 阻塞锁？ for update语句会在执行成功后立即返回，在执行失败时一直处于阻塞状态，直到成功。
- 锁定之后服务宕机，无法释放？使用这种方式，服务宕机之后数据库会自己把锁释放掉。

但是还是无法直接解决数据库单点和可重入问题。

这里还可能存在另外一个问题，虽然我们对method_name 使用了唯一索引，并且显示使用for update来使用行级锁。但是，MySql会对查询进行优化，即便在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，**如果 MySQL 认为全表扫效率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 将使用表锁，而不是行锁。如果发生这种情况就悲剧了。。。**

7.基于Redis实现





基于Redis命令：SET key value NX EX max-lock-time

这里补充下： 从2.6.12版本后, 就可以使用set来获取锁, Lua 脚本来释放锁。setnx是老黄历了， set命令nx,xx等参数, 是为了实现 setnx 的功能。

1.加锁

```
public class RedisTool {

private static final String LOCK_SUCCESS = "OK";

private static final String SET_IF_NOT_EXIST = "NX";

private static final String SET_WITH_EXPIRE_TIME = "PX";

/**

\* 尝试获取分布式锁

\* @param jedis Redis客户端

\* @param lockKey 锁

\* @param requestId 请求标识

\* @param expireTime 超期时间

\* @return 是否获取成功

*/

public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String
requestId, int expireTime) {

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

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

```
}
```

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

这个set()方法一共有五个形参：

第一个为key，我们使用key来当锁，因为key是唯一的。

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

第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；

第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。

第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

2.解锁

```
public class RedisTool {

private static final Long RELEASE_SUCCESS = 1L;

/**

\* 释放分布式锁

\* @param jedis Redis客户端

\* @param lockKey 锁

\* @param requestId 请求标识

\* @return 是否释放成功

*/

public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String
requestId) {

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

Object result =
jedis.eval(script,Collections.singletonList(lockKey),Collections.singletonList(r
equestId));
    if (RELEASE_SUCCESS.equals(result)) {
        return true;
    }
    return false;
  }
```

```
}
```

那么这段Lua代码的功能是什么呢？其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。

【不靠谱的情况】

上面我们说的是redis，是单点的情况。如果是在redis sentinel集群中情况就有所不同了。关于redis sentinel 集群可以看这里。在redis sentinel集群中，我们具有多台redis，他们之间有着主从的关系，例如一主二从。我们的set命令对应的数据写到主库，然后同步到从库。当我们申请一个锁的时候，对应就是一条命令 `setnx mykey myvalue` ，在redis sentinel集群中，这条命令先是落到了主库。假设这时主库down了，而这条数据还没来得及同步到从库，sentinel将从库中的一台选举为主库了。这时，我们的新主库中并没有mykey这条数据，若此时另外一个client执行 `setnx mykey hisvalue`，也会成功，即也能得到锁。这就意味着，此时有两个client获得了锁。这不是我们希望看到的，虽然这个情况发生的记录很小，只会在主从failover的时候才会发生，大多数情况下、大多数系统都可以容忍，但是不是所有的系统都能容忍这种瑕疵。

【redlock】

为了解决故障转移情况下的缺陷，Antirez 发明了 Redlock 算法，使用redlock算法，**需要多个redis实例，加锁的时候，它会想多半节点发送 `setex mykey myvalue` 命令，只要过半节点成功了，那么就算加锁成功了。释放锁的时候需要想所有节点发送del命令。这是一种基于【大多数都同意】的一种机制。**感兴趣的可以查询相关资料。在实际工作中使用的时候，我们可以选择已有的开源实现，python有redlock-py，java 中有Redisson redlock。

redlock确实解决了上面所说的“不靠谱的情况”。但是，它解决问题的同时，也带来了代价。你需要多个redis实例，你需要引入新的库 代码也得调整，性能上也会有损。所以，果然是不存在“完美的解决方案”，我们更需要的是能够根据实际的情况和条件把问题解决了就好。

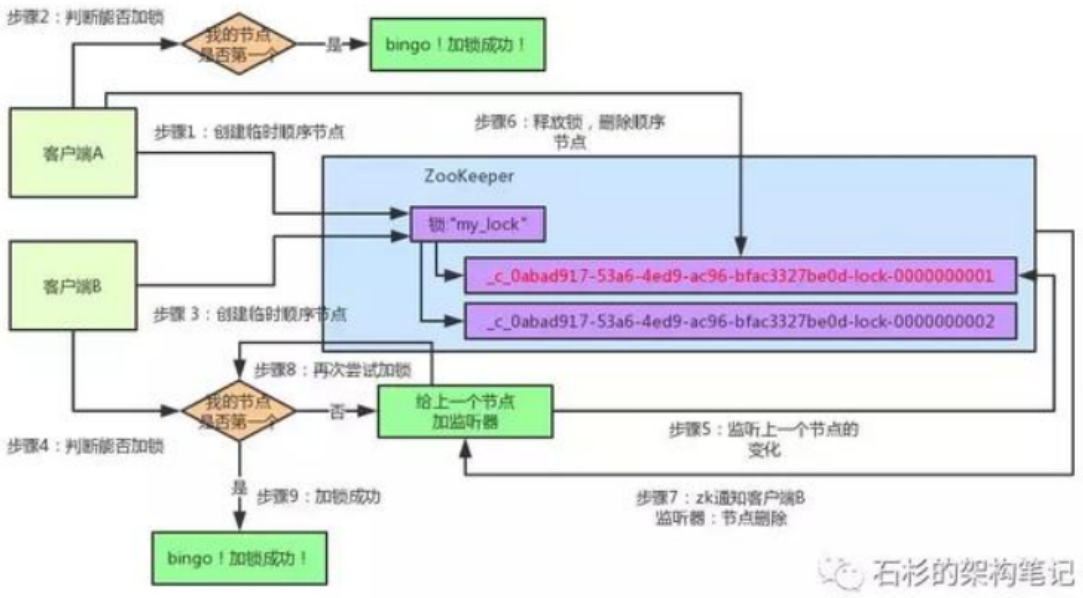
【小结】

redis是kv结构，每对kv组成的数据结构在内存中其实也是一个jvm对象，只要是jvm对象，在对象头就会有锁，当多个线程同时要访问改变这个对象时，就需要加线程同步操作。有且仅当获取到锁的线程，才可以操作和访问这个对象，其他的线程只能等待锁的释放。

在redis中，由于redis是单线程执行，所以对于redis中的任何共享资源的访问，都必须是互斥同步的。而redis中虽然没有提供像jvm-sy*这样的互斥锁，**但是却提供了互斥的实现工具，比如setnx命令，就是利用了其设置的kv对在内存中是唯一存在的特性，且该操作是原子的，才能够实现多线程中有且仅有一个线程可以setnx成功，这样就实现了互斥同步。而redis中锁的释放即setnx中的key被删除，只有这个key被删除了，其他的线程才有继续setnx成功的可能，这就变相实现了锁的释放，让其他线程通过setnx成功得以继续独占redis中的共享资源。**

形象地比喻就是，把redis中的key看作是锁的别名，而value则对应着被上了锁的盒子里面的东西。当一个线程想要拿这个东西，首先需要这个盒子，前提是需要过来看看这个盒子有没有被锁，如果没有，则这个线程先获得这个锁，然后锁上盒子，他独占着这个盒子的所有权。但是每个线程相当于一个拿着万能解锁器的人，随时都可以对这个简易的锁装置进行解锁，如果没有进行身份验证，那么任何线程都轻而易举地把锁解掉，然后修改盒子里面的东西。所以就需要多加一个身份验证的步骤，当一个线程想要过来解锁时，首先需要一个保安，询问它盒子里面装了什么，如果答得上来，则说明这个线程就是上锁的线程，如果不是，则拒绝给他放行去解锁。这样就保证了分布式锁的实现。

8.基于Zookeeper实现



基于Zookeeper实现分布式锁

基于zookeeper临时有序节点可以实现的分布式锁。

大致思想即为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

上面使用zk实现的简单的分布式锁，实现比较简单，但是存在性能问题，从上面的打印的结果可以看出、每一次客户端释放锁的时候，其他的客户端都会去抢锁，这就造成了不必要的浪费。那么如果提升性能呢？

客户端在抢锁的时候进行排队，客户端只要监听它前一个节点的变化就行，如果前一个节点释放了锁，客户端才去进行抢锁操作，这个时候我们就需要创建顺序节点了。

来看下Zookeeper能不能解决前面提到的问题。

- 锁无法释放？使用Zookeeper可以有效的解决锁无法释放的问题，因为在创建锁的时候，客户端会在ZK中创建一个临时节点，一旦客户端获取到锁之后突然挂掉（Session连接断开），那么这个临时节点就会自动删除掉。其他客户端就可以再次获得锁。
- 非阻塞锁？使用Zookeeper可以实现阻塞的锁，客户端可以通过在ZK中创建顺序节点，并且在节点上绑定监听器，一旦节点有变化，Zookeeper会通知客户端，客户端可以检查自己创建的节点是不是当前所有节点中序号最小的，如果是，那么自己就获取到锁，便可以执行业务逻辑了。
- 不可重入？使用Zookeeper也可以有效的解决不可重入的问题，客户端在创建节点的时候，把当前客户端的主机信息和线程信息直接写入到节点中，下次想要获取锁的时候和当前最小的节点中的数据比对一下就可以了。如果和自己的信息一样，那么自己直接获取到锁，如果不一样就再创建一个临时的顺序节点，参与排队。
- 单点问题？使用Zookeeper可以有效的解决单点问题，ZK是集群部署的，只要集群中有半数以上的机器存活，就可以对外提供服务。

可以直接使用zookeeper第三方库Curator客户端，这个客户端中封装了一个可重入的锁服务。

```java
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException {
    try {
        return interProcessMutex.acquire(timeout, unit);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return true;
}
public boolean unlock() {
    try {
        interProcessMutex.release();
    } catch (Throwable e) {
        log.error(e.getMessage(), e);
    } finally {
        executorService.schedule(new Cleaner(client, path), delayTimeForClean, TimeUnit.MILLISECONDS);
    }
    return true;
}
```

Curator提供的InterProcessMutex是分布式锁的实现。acquire方法用户获取锁，release方法用于释放锁。

使用ZK实现的分布式锁好像完全符合了本文开头我们对一个分布式锁的所有期望。但是，其实并不是，Zookeeper实现的分布式锁其实存在一个缺点，那就是性能上可能并没有缓存服务那么高。因为每次在创建锁和释放锁的过程中，都要动态创建、销毁瞬时节点来实现锁功能。ZK中创建和删除节点只能通过Leader服务器来执行，然后将数据同不到所有的Follower机器上。

其实，使用Zookeeper也有可能带来并发问题，只是并不常见而已。考虑这样的情况，由于网络抖动，客户端可ZK集群的session连接断了，那么zk以为客户端挂了，就会删除临时节点，这时候其他客户端就可以获取到分布式锁了。就可能产生并发问题。这个问题不常见是因为zk有重试机制，一旦zk集群检测不到客户端的心跳，就会重试，Curator客户端支持多种重试策略。**多次重试之后还不行的话才会删除临时节点。**（所以，选择一个合适的重试策略也比较重要，要在锁的粒度和并发之间找一个平衡。）

Zookeeper实现分布式锁总结

Zookeeper实现分布式锁的优点

- 有效的解决单点问题，不可重入问题，非阻塞问题以及锁无法释放的问题。实现起来较为简单。

Zookeeper实现分布式锁的缺点

- 性能上不如使用缓存实现分布式锁。 需要对ZK的原理有所了解。

伪代码

```java
public void lock() {
    if (tryLock()) {
        System.out.println("获得锁");
    } else {
        // 尝试加锁
        // 进入等待 监听
        waitForLock();
        // 再次尝试
        lock();
    }

}

public synchronized boolean tryLock() {
    // 第一次就进来创建自己的临时节点
    if (StringUtils.isBlank(path)) {
        path = zkClient.createEphemeralSequential( path: Z_NODE + "/",  data: "lock");
    }

    // 对节点排序
    List<String> children = zkClient.getChildren(Z_NODE);
    Collections.sort(children);

    // 当前的是最小节点就返回加锁成功
    if (path.equals(Z_NODE + "/" + children.get(0))) {
        System.out.println(" i am true");
        return true;
    } else {
        // 不是最小节点 就找到自己的前一个 依次类推 释放也是一样
        int i = Collections.binarySearch(children, path.substring(Z_NODE.length() + 1));
        beforePath = Z_NODE + "/" + children.get(i - 1);
    }
    return false;
}
```

头条 @敖丙Kirk

```java
public void waitForLock() {

    IZkDataListener listener = new IZkDataListener() {
        public void handleDataChange(String s, Object o) throws Exception {
        }

        public void handleDataDeleted(String s) throws Exception {
            System.out.println(Thread.currentThread().getName() + ":监听到节点删除事件! ----------------------------");
            cdl.countDown();
        }
    };
    // 监听
    this.zkClient.subscribeDataChanges(beforePath, listener);
    if (zkClient.exists(beforePath)) {
        try {
            System.out.println("加锁失败 等待");
            cdl.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    // 释放监听
    zkClient.unsubscribeDataChanges(beforePath, listener);
```

头条 @敖丙Kirk

9.Redis分布式锁代码实现

【加锁】

正确代码

```
public class RedisTool {

    private static final String LOCK_SUCCESS = "OK";
```

```
    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";

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

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

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

    }
}
```

可以看到，我们加锁就一行代码：`jedis.set(String key, String value, String nxxx, String expx, int time)`，这个set()方法一共有五个形参：

- 第一个为key，我们使用key来当锁，因为key是唯一的。
- 第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用`UUID.randomUUID().toString()`方法生成。
- 第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
- 第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。
- 第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件。首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

错误1：

```
public static void wrongGetLock1(Jedis jedis, String lockKey, String requestId,
int expireTime) {

    Long result = jedis.setnx(lockKey, requestId);
    if (result == 1) {
        // 若在这里程序突然崩溃，则无法设置过期时间，将发生死锁
        jedis.expire(lockKey, expireTime);
    }

}
```

setnx()方法作用就是SET IF NOT EXIST，expire()方法就是给锁加一个过期时间。乍一看好像和前面的set()方法结果一样，然而由于这是两条Redis命令，不具有原子性，如果程序在执行完setnx()之后突然崩溃，导致锁没有设置过期时间。那么将会发生死锁。网上之所以有人这样实现，是因为低版本的jedis并不支持多参数的set()方法。

错误2：

```
public static boolean wrongGetLock2(Jedis jedis, String lockKey, int expireTime)
{

    long expires = System.currentTimeMillis() + expireTime;
    String expiresStr = String.valueOf(expires);

    // 如果当前锁不存在，返回加锁成功
    if (jedis.setnx(lockKey, expiresStr) == 1) {
        return true;
    }

    // 如果锁存在，获取锁的过期时间
    String currentValueStr = jedis.get(lockKey);
    if (currentValueStr != null && Long.parseLong(currentValueStr) <
System.currentTimeMillis()) {
        // 锁已过期，获取上一个锁的过期时间，并设置现在锁的过期时间
        String oldValueStr = jedis.getSet(lockKey, expiresStr);
        if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {
            // 考虑多线程并发的情况，只有一个线程的设置值和当前值相同，它才有权利加锁
            return true;
        }
    }

    // 其他情况，一律返回加锁失败
    return false;

}
```

这一种错误示例就比较难以发现问题，而且实现也比较复杂。实现思路：使用`jedis.setnx()`命令实现加锁，其中key是锁，value是锁的过期时间。执行过程：1. 通过setnx()方法尝试加锁，如果当前锁不存在，返回加锁成功。2. 如果锁已经存在则获取锁的过期时间，和当前时间比较，如果锁已经过期，则设置新的过期时间，返回加锁成功。代码如下：

那么这段代码问题在哪里？1. 由于是客户端自己生成过期时间，所以需要强制要求分布式下每个客户端的时间必须同步。 2. 当锁过期的时候，如果多个客户端同时执行`jedis.getSet()`方法，那么虽然最终只有一个客户端可以加锁，但是这个客户端的锁的过期时间可能被其他客户端覆盖。3. 锁不具备拥有者标识，即任何客户端都可以解锁。

【解锁】

正确代码：

```
public class RedisTool {

    private static final Long RELEASE_SUCCESS = 1L;

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

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

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

    }

}
```

可以看到，我们解锁只需要两行代码就搞定了！第一行代码，我们写了一个简单的Lua脚本代码，上一次见到这个编程语言还是在《黑客与画家》里，没想到这次居然用上了。第二行代码，我们将Lua代码传到`jedis.eval()`方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。

那么这段Lua代码的功能是什么呢？其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。那么为什么要使用Lua语言来实现呢？因为要确保上述操作是原子性的。关于非原子性会带来什么问题，可以阅读【解锁代码-错误示例2】。那么为什么执行eval()方法可以确保原子性，源于Redis的特性，下面是官网对eval命令的部分解释：

简单来说，就是在eval命令执行Lua代码的时候，Lua代码将被当成一个命令去执行，并且直到eval命令执行完成，Redis才会执行其他命令。

错误1：

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

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

错误2：

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

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

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

}
```

如代码注释，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。

参考资料：

阿里P8架构师谈：分布式锁的3种实现（数据库、缓存、Zookeeper）

https://www.toutiao.com/i6601005667241165326/?tt_from=weixin&utm_campaign=client_share&wxshare_count=2&from=singlemessage&timestamp=1592130260&app=news_article&utm_source=weixin&isappinstalled=0&utm_medium=toutiao_ios&use_new_style=0&req_id=202006141824200101941000394C3B1E1F&group_id=6601005667241165326&pbid=6777745738342401549

高并发架构系列：什么是分布式锁？Redis实现分布式锁详解

https://www.toutiao.com/i6636601284021453319/?tt_from=weixin&utm_campaign=client_share&wxshare_count=2&from=singlemessage&timestamp=1592127033&app=news_article&utm_source=weixin&isappinstalled=0&utm_medium=toutiao_ios&use_new_style=0&req_id=202006141730320100140481412 86C814B&group_id=6636601284021453319&pbid=6777745738342401549

彻底理解ZooKeeper分布式锁实现原理

https://m.toutiaocdn.com/i6738532213022786051/?app=news_article&timestamp=1592132513&use_new_style=0&req_id=202006141901530100140481442A827D05&group_id=6738532213022786051&tt_from=weixin&utm_source=weixin&utm_medium=toutiao_ios&utm_campaign=client_share&wxshare_count=2&from=singlemessage&isappinstalled=0&pbid=6777745738342401549

Lua脚本在redis分布式锁场景的运用

https://www.cnblogs.com/demingblog/p/9542124.html