

1.Redis 录制计划

2.Redis 简介

3.Redis 安装

- 3.1 直接编译安装
- 3.2 通过 Docker 安装
- 3.3 直接安装
- 3.4 在线体验

4. Redis 五种基本数据类型

- 4.1 Redis 启动
- 4.2 String
 - 4.2.1 BIT 命令
- 4.3 List
- 4.4 Set
- 4.5 Hash
- 4.6 ZSet
- 4.7 key
- 4.8 补充

5.Redis 的 Java 客户端

- 5.1 开启远程连接
- 5.2 Jedis
 - 5.2.1 基本使用
 - 5.2.2 连接池
- 5.3 Lettuce

6.Redis 做分布式锁

- 6.1 基本用法
- 6.2 解决超时问题

7.Redis 做消息队列

- 7.1 消息队列
- 7.2 延迟消息队列

8.再谈 Bit 操作

- 8.1 基本介绍
- 8.2 基本操作
 - 8.2.1 零存整取
 - 8.2.1 整存零取
- 8.3 统计
- 8.4 Bit 批处理

9.HyperLogLog

10.布隆过滤器

- 10.1 场景重现
- 10.2 Bloom Filter 介绍
- 10.3 Bloom Filter 原理
- 10.4 Bloom Filter 安装
- 10.5 基本用法
- 10.6 典型场景

11.Redis 限流

- 11.1 预备知识
- 11.2 简单限流
- 11.3 深入限流操作
- 11.4 Lettuce扩展

12.Redis 之 Geo

- 1.GeoHash
- 2. Redis 中使用

13.Redis 之 Scan

- 1.简单介绍
- 2.基本用法

- 3.原理
- 4.其他用法

14.Redis 单线程如何处理高并发

- 1.阻塞 IO 与非阻塞 IO
- 2.Redis 的线程模型

15.Redis 通信协议

- 1.准备工作
- 2. 实战

16.Redis 持久化

- 1.快照
 - 1.1 原理
 - 1.2 具体配置
 - 1.3 备份流程
- 2.AOF

17.Redis 事务

- 1.原子性
- 2.Java代码实现

18.Redis 主从同步

- 1.CAP
- 2.主从复制
 - 2.1 配置方式
 - 2.2 主从复制注意点
 - 2.3 复制原理
 - 2.4 一场接力赛
 - 2.5 哨兵模式
 - 2.6 注意问题
- 3. Jedis 操作哨兵模式
- 4.SpringBoot

19.Redis 集群

- 集群原理
 - 怎么样投票
 - 怎么样判定节点不可用
- ruby 环境
- 集群搭建
- 查询集群信息
- 添加主节点
- 删除节点
- Jedis 操作 RedisCluster

1.Redis 录制计划

公众号后台回复 redis，获取 Redis 教程，视频将以该教程为蓝本，增加更多使用场景。

2.Redis 简介

Redis 是我们在互联网应用中使用最广泛的一个 NoSQL 数据库，基于 C 开发的键值对存储数据库，Redis 这个名字是 Remote Dictionary Service 字母缩写。

很多人想到 Redis，就想到缓存。但实际上 Redis 除了缓存之外，还有许多更加丰富的使用场景。比如分布式锁，限流。

特点：

- 支持数据持久化

- 支持多种不同的数据结构类型之间的映射
- 支持主从模式的数据备份
- 自带了发布订阅系统
- 定时器、计数器

3.Redis 安装

四种方式获取一个 Redis:

1. 直接编译安装 (推荐使用)
2. 使用 Docker
3. 也可以直接安装
4. 还有一个在线体验的方式, 通过在线体验, 可以直接使用 Redis 的功能<http://try.redis.io/>

3.1 直接编译安装

提前准备好 gcc 环境。

```
yum install gcc-c++
```

接下来下载并安装 Redis:

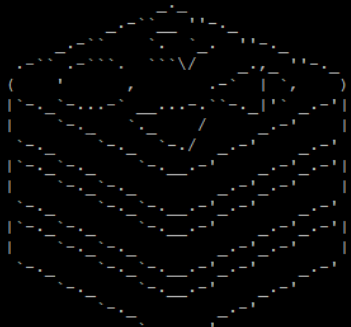
```
wget http://download.redis.io/releases/redis-5.0.7.tar.gz
tar -zxvf redis-5.0.7.tar.gz
cd redis-5.0.7/
make
make install
```

安装完成后, 启动 Redis:

```
redis-server redis.conf
```

启动成功页面如下:

```
[root@localhost redis-5.0.7]# redis-server redis.conf
7265:C 06 Mar 2020 11:18:43.993 # oOoOoOoOoOoOo Redis is starting oOoOoOoOoOoOo
7265:C 06 Mar 2020 11:18:43.993 # Redis version=5.0.7, bits=64, commit=00000000, modified=0, pid=7265, just started
7265:C 06 Mar 2020 11:18:43.993 # Configuration loaded
7265:M 06 Mar 2020 11:18:43.997 * Increased maximum number of open files to 10032 (it was originally set to 1024).
```



```
Redis 5.0.7 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 7265

http://redis.io
```

3.2 通过 Docker 安装

提前准备好 Docker (公众号【江南一点雨】后台回复 docker)。

Docker 安装好之后，启动 Docker，直接运行安装命令即可：

```
docker run --name javaboy-redis -d -p 6379:6379 redis --requirepass 123
```

Docker 上的 Redis 启动成功之后，可以从宿主机上连接（前提是宿主机上存在 redis-cli）：

```
redis-cli -a 123
```

如果宿主机上没有安装 Redis，那么也可以进入到 Docker 容器中去操作 Redis：

```
docker exec -it javaboy-redis redis-cli -a 123
```

3.3 直接安装

CentOS:

```
yum install redis
```

Ubuntu:

```
apt-get install redis
```

Mac:

```
brew install redis
```

3.4 在线体验

- <http://try.redis.io/>

4. Redis 五种基本数据类型

4.1 Redis 启动

首先，修改 redis.conf 配置文件：

```
# Redis default starting with Redis 3.2.1.
tcp-keepalive 300

##### GENERAL #####

# By default Redis does not run as a daemon. Use 'yes' if you need it.
# Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
daemonize yes

# If you run Redis from upstart or systemd, Redis can interact with your
# supervision tree. Options:
# supervised no      - no supervision interaction
# supervised upstart - signal upstart by putting Redis into SIGSTOP mode
# supervised systemd - signal systemd by writing READY=1 to $NOTIFY_SOCKET
# supervised auto    - detect upstart or systemd method based on
#                     UPSTART_JOB or NOTIFY_SOCKET environment variables
# Note: these supervision methods only signal "process is ready."
```

表示以守护进程的方式启动 Redis

配置完成后，保存退出，再次通过 `redis-server redis.conf` 命令启动 Redis，此时，就是在后台启动了。

```
[root@localhost redis-5.0.7]# redis-server redis.conf
12742:C 06 Mar 2020 15:36:56.936 # oO0oO0oOoO0o Redis is starting oO0oO0oOoO0o
12742:C 06 Mar 2020 15:36:56.936 # Redis version=5.0.7, bits=64, commit=00000000, modified=0, pid=12742, just started
12742:C 06 Mar 2020 15:36:56.936 # Configuration loaded
[root@localhost redis-5.0.7]#
```

4.2 String

String 是 Redis 里边最简单的一种数据结构。在 Redis 中，所有的 key 都是字符串，但是，不同的 key 对应的 value 则具备不同的数据结构，我们所说的五种不同的数据类型，主要是指 value 的数据类型不同。

Redis 中的字符串是动态字符串，内部是可以修改的，像 Java 中的 StringBuffer，它采用分配冗余空间的方式来减少内存的频繁分配。在 Redis 内部结构中，一般实际分配的内存会大于需要的内存，当字符串小于 1M 的时候，扩容都是在现有的空间基础上加倍，扩容每次扩 1M 空间，最大 512M。

- set

set 就是给一个 key 赋值的。

- append

使用 append 命令时，如果 key 已经存在，则直接在对应的 value 后追加值，否则就创建新的键值对。

- decr

可以实现对 value 的减 1 操作（前提是 value 是一个数字），如果 value 不是数字，会报错，如果 value 不存在，则会给一个默认值为 0，在默认值的基础上减一。

- decrby

和 decr 类似，但是可以自己设置步长，该命令第二个参数就是步长。

- get

get 用来获取一个 key 的 value。

- getrange

getrange 可以用来返回 key 对应的 value 的子串，这有点类似于 Java 里边的 substring。这个命令第二个和第三个参数就是截取的起始和终止位置，其中，-1 表示最后一个字符串，-2 表示倒数第二个字符串，以此类推...

- getset

获取并更新某一个 key。

- incr

给某一个 key 的 value 自增。

- incrby

给某一个 key 的 value 自增，同时还可以设置步长。

- incrbyfloat

和 incrby 类似，但是自增的步长可以设置为浮点数。

- mget 和 mset

批量获取和批量存储

- ttl

查看 key 的有效期

- setex

在给 key 设置 value 的同时，还设置过期时间。

- psetex

和 setex 类似，只不过这里的时间单位是毫秒。

- setnx

默认情况下，set 命令会覆盖已经存在的 key，setnx 则不会。

- msetnx

批量设置。

- setrange

覆盖一个已经存在的 key 的 value。

- strlen

查看字符串长度

4.2.1 BIT 命令

在 Redis 中，字符串都是以二进制的方式来存储的。例如 set k1 a，a 对应的 ASCII 码是 97，97 转为二进制是 01100001，BIT 相关的命令就是对二进制进行操作的。

- getbit

key 对应的 value 在 offset 处的 bit 值。

- setbit

修改 key 对应的 value 在 offset 处的 bit 值

- bitcount

统计二进制数据中 1 的个数。

4.3 List

- lpush

将所有指定的值插入到存于 key 的列表的头部。如果 key 不存在，那么在进行 push 操作前会创建一个空列表。如果 key 对应的值不是一个 list 的话，那么会返回一个错误。

- lrange

返回列表指定区间内的元素。

- rpush

向存于 key 的列表的尾部插入所有指定的值。

- rpop

移除并返回列表的尾元素。

- lpop

移除并返回列表的头元素。

- lindex

返回列表中，下标为 index 的元素。

- ltrim

ltrim 可以对一个列表进行修剪。

- blpop

阻塞式的弹出，相当于 lpop 的阻塞版。

4.4 Set

- sadd

添加元素到一个 key 中

- smembers

获取一个 key 下的所有元素

- srem

移除指定的元素

- sismember

返回某一个成员是否在集合中

- scard

返回集合的数量

- srandmember

随机返回一个元素

- spop

随机返回并且出栈一个元素。

- smove

把一个元素从一个集合移到另一个集合中去。

- sdiff

返回两个集合的差集。

- sinter

返回两个集合的交集。

- sdiffstore

这个类似于 sdiff，不同的是，计算出来的结果会保存在一个新的集合中。

- sinterstore

类似于 sinter，只是将计算出来的交集保存到一个新的集合中。

- sunion

求并集。

- sunionstore

求并集并且将结果保存到新的集合中。

4.5 Hash

在 hash 结构中，key 是一个字符串，value 则是一个 key/value 键值对。

- hset

添加值。

- hget

获取值

- hmset

批量设置

- hmget

批量获取

- hdel

删除一个指定的 field

- hsetnx

默认情况下，如果 key 和 field 相同，会覆盖掉已有的 value，hsetnx 则不会。

- hvals

获取所有的 value

- hkeys

获取所有的 key

- hgetall

同时获取所有的 key 和 value

- hexists

返回 field 是否存在

- hincrby

给指定的 value 自增

- hincrbyfloat

可以自增一个浮点数

- hlen

返回 某一个 key 中 value 的数量

- hstrlen

返回某一个 key 中的某一个 field 的字符串长度

4.6 ZSet

- zadd

将指定的元素添加到有序集合中。

- zscore

返回 member 的 score 值

- zrange

返回集合中的一组元素。

- zrevrange

返回一组元素，但是是倒序。

- zcard

返回元素个数

- zcount

返回 score 在某一个区间内的元素。

- zrangebyscore

按照 score 的范围返回元素。

- zrank

返回元素的排名（从小到大）

- zrevrank

返回元素排名（从大到小）

- zincrby

score 自增

- zinterstore

给两个集合求交集。

- zrem

弹出一个元素

- zlexcount

计算有序集合中成员数量

- zrangebylex

返回指定区间内的成员。

4.7 key

- del

删除一个 key/value

- dump

序列化给定的 key

- exists

判断一个 key 是否存在

- ttl

查看一个 key 的有效期

- expire

给一个 key 设置有效期，如果 key 在过期之前被重新 set 了，则过期时间会失效。

- persist

移除一个 key 的过期时间

- keys *

查看所有的 key

- pttl

和 ttl 一样，只不过这里返回的是毫秒

4.8 补充

1. 四种数据类型 (list/set/zset/hash) ， 在第一次使用时，如果容器不存在，就自动创建一个
2. 四种数据类型 (list/set/zset/hash) ， 如果里边没有元素了，那么立即删除容器，释放内存。

5.Redis 的 Java 客户端

5.1 开启远程连接

Redis 默认是不支持远程连接的，需要手动开启。

一共修改两个地方：

1. 注释掉 bind: 127.0.0.1
2. 开启密码校验，去掉 requirepass 的注释

改完之后，保存退出，启动 Redis。

5.2 Jedis

5.2.1 基本使用

Jedis 的 GitHub 地址：<https://github.com/xetorthio/jedis>

首先创建一个普通的 Maven 项目。

项目创建成功后，添加 Jedis 依赖：

```
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>3.2.0</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
```

然后创建一个测试方法。

```
public class MyJedis {
    public static void main(String[] args) {
        //1. 构造一个 Jedis 对象，因为这里使用的默认端口 6379，所以不用配置端口
        Jedis jedis = new Jedis("192.168.91.128");
        //2. 密码认证
        jedis.auth("javaboy");
        //3. 测试是否连接成功
        String ping = jedis.ping();
        //4. 返回 pong 表示连接成功
        System.out.println(ping);
    }
}
```

对于 Jedis 而言，一旦连接上 Redis 服务端，剩下的操作都很容易了。

在 Jedis 中，由于方法的 API 和 Redis 的命令高度一致，所以，Jedis 中的方法见名知意，直接使用即可。

5.2.2 连接池

在实际应用中，Jedis 实例我们一般都是通过连接池来获取，由于 Jedis 对象不是线程安全的，所以，当我们使用 Jedis 对象时，从连接池获取 Jedis，使用完成之后，再还给连接池。

```
public class JedisPoolTest {
    public static void main(String[] args) {
        //1. 构造一个 Jedis 连接池
        JedisPool pool = new JedisPool("192.168.91.128", 6379);
        //2. 从连接池中获取一个 Jedis 连接
        Jedis jedis = pool.getResource();
        //3. Jedis 操作
        String ping = jedis.ping();
        System.out.println(ping);
        //4. 归还连接
        jedis.close();
    }
}
```

如果第三步抛出异常的话，会导致第四步无法执行，所以，我们要对代码进行改进，确保第四步能够执行。

```
public class JedisPoolTest {
    public static void main(String[] args) {
        Jedis jedis = null;
        //1. 构造一个 Jedis 连接池
        JedisPool pool = new JedisPool("192.168.91.128", 6379);
```

```

//2. 从连接池中获取一个 Jedis 连接
jedis = pool.getResource();
jedis.auth("javaboy");
try {
    //3. Jedis 操作
    String ping = jedis.ping();
    System.out.println(ping);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    //4. 归还连接
    if (jedis != null) {
        jedis.close();
    }
}
}
}

```

通过 finally 我们可以确保 jedis 一定被关闭。

利用 JDK1.7 中的 try-with-resource 特性，可以对上面的代码进行改造：

```

public class JedisPoolTest {
    public static void main(String[] args) {
        JedisPool pool = new JedisPool("192.168.91.128");
        try(Jedis jedis = pool.getResource()) {
            jedis.auth("javaboy");
            String ping = jedis.ping();
            System.out.println(ping);
        }
    }
}

```

这段代码的作用和上面的是一致的。

但是，上面这段代码无法实现强约束。我们可以做进一步的改进：

```

public interface CallWithJedis {
    void call(Jedis jedis);
}

public class Redis {
    private JedisPool pool;

    public Redis() {
        GenericObjectPoolConfig config = new GenericObjectPoolConfig();
        //连接池最大空闲数
        config.setMaxIdle(300);
        //最大连接数
        config.setMaxTotal(1000);
        //连接最大等待时间，如果是 -1 表示没有限制
        config.setMaxWaitMillis(30000);
        //在空闲时检查有效性
        config.setTestOnBorrow(true);
        /**
         * 1. Redis 地址
         * 2. Redis 端口
         * 3. 连接超时时间

```

```

    * 4. 密码
    */
    pool = new JedisPool(config, "192.168.91.128", 6379, 30000, "javaboy");
}

public void execute(CallWithJedis callWithJedis) {
    try (Jedis jedis = pool.getResource()) {
        callWithJedis.call(jedis);
    }
}

}

Redis redis = new Redis();
redis.execute(jedis -> {
    System.out.println(jedis.ping());
});

```

5.3 Lettuce

GitHub: <https://github.com/lettuce-io/lettuce-core>

Lettuce 和 Jedis 的一个比较:

1. Jedis 在实现的过程中是直接连接 Redis 的, 在多个线程之间共享一个 Jedis 实例, 这是线程不安全的, 如果想在多线程场景下使用 Jedis, 就得使用连接池, 这样, 每个线程都有自己的 Jedis 实例。
2. Lettuce 基于目前很火的 Netty NIO 框架来构建, 所以克服了 Jedis 中线程不安全的问题, Lettuce 支持同步、异步 以及 响应式调用, 多个线程可以共享一个连接实例。

使用 Lettuce, 首先创建一个普通的 Maven 项目, 添加 Lettuce 依赖:

```

<dependency>
  <groupId>io.lettuce</groupId>
  <artifactId>lettuce-core</artifactId>
  <version>5.2.2.RELEASE</version>
</dependency>

```

然后来一个简单的测试案例:

```

public class LettuceTest {
    public static void main(String[] args) {
        RedisClient redisClient =
        RedisClient.create("redis://javaboy@192.168.91.128");
        StatefulRedisConnection<String, String> connect = redisClient.connect();
        RedisCommands<String, String> sync = connect.sync();
        sync.set("name", "javaboy");
        String name = sync.get("name");
        System.out.println(name);
    }
}

```

注意这里的密码传递方式, 密码直接写在连接地址里边。

6.Redis 做分布式锁

分布式锁也算是 Redis 比较常见的使用场景。

问题场景：

例如一个简单的用户操作，一个线程去修改用户的状态，首先从数据库中读出用户的状态，然后在内存中进行修改，修改完成后，再存回去。在单线程中，这个操作没有问题，但是在多线程中，由于读取、修改、存 这是三个操作，不是原子操作，所以在多线程中，这样会出问题。

对于这种问题，我们可以使用分布式锁来限制程序的并发执行。

6.1 基本用法

分布式锁实现的思路很简单，就是进来一个线程先占位，当别的线程进来操作时，发现已经有人占位了，就会放弃或者稍后再试。

在 Redis 中，占位一般使用 setnx 指令，先进来的线程先占位，线程的操作执行完成后，再调用 del 指令释放位子。

根据上面的思路，我们写出的代码如下：

```
public class LockTest {
    public static void main(String[] args) {
        Redis redis = new Redis();
        redis.execute(jedis->{
            Long setnx = jedis.setnx("k1", "v1");
            if (setnx == 1) {
                //没人占位
                jedis.set("name", "javaboy");
                String name = jedis.get("name");
                System.out.println(name);
                jedis.del("k1");//释放资源
            }else{
                //有人占位，停止/暂缓 操作
            }
        });
    }
}
```

上面的代码存在一个小小问题：如果代码业务执行的过程中抛异常或者挂了，这样会导致 del 指令没有被调用，这样，k1 无法释放，后面来的请求全部堵塞在这里，锁也永远得不到释放。

要解决这个问题，我们可以给锁添加一个过期时间，确保锁在一定的时间之后，能够得到释放。改进后的代码如下：

```
public class LockTest {
    public static void main(String[] args) {
        Redis redis = new Redis();
        redis.execute(jedis->{
            Long setnx = jedis.setnx("k1", "v1");
            if (setnx == 1) {
                //给锁添加一个过期时间，防止应用在运行过程中抛出异常导致锁无法及时得到释放
                jedis.expire("k1", 5);
                //没人占位
                jedis.set("name", "javaboy");
                String name = jedis.get("name");
                System.out.println(name);
                jedis.del("k1");//释放资源
            }
        });
    }
}
```

```

        }else{
            //有人占位，停止/暂缓 操作
        }
    });
}
}

```

这样改造之后，还有一个问题，就是在获取锁和设置过期时间之间如果服务器突然挂掉了，这个时候锁被占用，无法及时得到释放，也会造成死锁，因为获取锁和设置过期时间是两个操作，不具备原子性。

为了解决这个问题，从 Redis2.8 开始，setnx 和 expire 可以通过一个命令一起来执行了，我们对上述代码再做改进：

```

public class LockTest {
    public static void main(String[] args) {
        Redis redis = new Redis();
        redis.execute(jedis->{
            String set = jedis.set("k1", "v1", new SetParams().nx().ex(5));
            if (set !=null && "OK".equals(set)) {
                //给锁添加一个过期时间，防止应用在运行过程中抛出异常导致锁无法及时得到释放
                jedis.expire("k1", 5);
                //没人占位
                jedis.set("name", "javaboy");
                String name = jedis.get("name");
                System.out.println(name);
                jedis.del("k1");//释放资源
            }else{
                //有人占位，停止/暂缓 操作
            }
        });
    }
}

```

6.2 解决超时问题

为了防止业务代码在执行的时候抛出异常，我们给每一个锁添加了一个超时时间，超时之后，锁会被自动释放，但是这也带来了一个新的问题：如果要执行的业务非常耗时，可能会出现紊乱。举个例子：第一个线程首先获取到锁，然后开始执行业务代码，但是业务代码比较耗时，执行了 8 秒，这样，会在第一个线程的任务还未执行成功锁就会被释放了，此时第二个线程会获取到锁开始执行，在第二个线程刚执行了 3 秒，第一个线程也执行完了，此时第一个线程会释放锁，但是注意，它释放的第二个线程的锁，释放之后，第三个线程进来。

对于这个问题，我们可以从两个角度入手：

- 尽量避免在获取锁之后，执行耗时操作。
- 可以在锁上面做文章，将锁的 value 设置为一个随机字符串，每次释放锁的时候，都去比较随机字符串是否一致，如果一致，再去释放，否则，不释放。

对于第二种方案，由于释放锁的时候，要去查看锁的 value，第二个比较 value 的值是否正确，第三步释放锁，有三个步骤，很明显三个步骤不具备原子性，为了解决这个问题，我们得引入 Lua 脚本。

Lua 脚本的优势：

- 使用方便，Redis 中内置了对 Lua 脚本的支持。
- Lua 脚本可以在 Redis 服务端原子的执行多个 Redis 命令。

- 由于网络在很大程度上会影响到 Redis 性能，而使用 Lua 脚本可以让多个命令一次执行，可以有效解决网络给 Redis 带来的性能问题。

在 Redis 中，使用 Lua 脚本，大致上两种思路：

1. 提前在 Redis 服务端写好 Lua 脚本，然后在 Java 客户端去调用脚本（推荐）。
2. 可以直接在 Java 端去写 Lua 脚本，写好之后，需要执行时，每次将脚本发送到 Redis 上去执行。

首先在 Redis 服务端创建 Lua 脚本，内容如下：

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

接下来，可以给 Lua 脚本求一个 SHA1 和，命令如下：

```
cat lua/releasewherevalueequal.lua | redis-cli -a javaboy script load --pipe
```

```
[root@localhost redis-5.0.7]# cat lua/releasewherevalueequal.lua | redis-cli -a javaboy script load --pipe
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
"b8059ba43af6ffe8bed3db65bac35d452f8115d8"
[root@localhost redis-5.0.7]#
```

script load 这个命令会在 Redis 服务器中缓存 Lua 脚本，并返回脚本内容的 SHA1 校验和，然后在 Java 端调用时，传入 SHA1 校验和作为参数，这样 Redis 服务端就知道执行哪个脚本了。

接下来，在 Java 端调用这个脚本。

```
public class LuaTest {
    public static void main(String[] args) {
        Redis redis = new Redis();
        for (int i = 0; i < 2; i++) {
            redis.execute(jedis -> {
                //1.先获取一个随机字符串
                String value = UUID.randomUUID().toString();
                //2.获取锁
                String k1 = jedis.set("k1", value, new SetParams().nx().ex(5));
                //3.判断是否成功拿到锁
                if (k1 != null && "OK".equals(k1)) {
                    //4.具体的业务操作
                    jedis.set("site", "www.javaboy.org");
                    String site = jedis.get("site");
                    System.out.println(site);
                    //5.释放锁
                    jedis.evalsha("b8059ba43af6ffe8bed3db65bac35d452f8115d8",
                        Arrays.asList("k1"), Arrays.asList(value));
                } else {
                    System.out.println("没拿到锁");
                }
            });
        }
    }
}
```


7.Redis 做消息队列

我们平时说到消息队列，一般都是指 RabbitMQ、RocketMQ、ActiveMQ 以及大数据里边的 Kafka，这些是我们比较常见的消息中间件，也是非常专业的消息中间件，作为专业的中间件，它里边提供了许多功能。

但是，当我们需要使用消息中间件的时候，并非每次都需要非常专业的消息中间件，假如我们只有一个消息队列，只有一个消费者，那就没有必要去使用上面这些专业的消息中间件，这种情况我们可以直接使用 Redis 来做消息队列。

Redis 的消息队列不是特别专业，他没有很多高级特性，适用简单的场景，如果对于消息可靠性有着极高的追求，那么不适合使用 Redis 做消息队列。

7.1 消息队列

Redis 做消息队列，使用它里边的 List 数据结构就可以实现，我们可以使用 lpush/rpush 操作来实现入队，然后使用 lpop/rpop 来实现出队。

回顾一下：

```
127.0.0.1:6379> lpush javaboy-queue java php python vue js
(integer) 5
127.0.0.1:6379> LLEN javaboy-queue
(integer) 5
127.0.0.1:6379> lpop javaboy-queue
"js"
127.0.0.1:6379> rpop javaboy-queue
"java"
127.0.0.1:6379> LLEN javaboy-queue
(integer) 3
127.0.0.1:6379> █
```

在客户端（例如 Java 端），我们会维护一个死循环来不停的从队列中读取消息，并处理，如果队列中有消息，则直接获取到，如果没有消息，就会陷入死循环，直到下一次有消息进入，这种死循环会造成大量的资源浪费，这个时候，我们可以使用之前讲的 blpop/brpop。

7.2 延迟消息队列

延迟队列可以通过 zset 来实现，因为 zset 中有一个 score，我们可以把时间作为 score，将 value 存到 redis 中，然后通过轮询的方式，去不断的读取消息出来。

首先，如果消息是一个字符串，直接发送即可，如果是一个对象，则需要对对象进行序列化，这里我们使用 JSON 来实现序列化和反序列化。

所以，首先在项目中，添加 JSON 依赖：

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.10.3</version>
</dependency>
```

接下来，构造一个消息对象：

```

public class JavaboyMessage {
    private String id;
    private Object data;

    @Override
    public String toString() {
        return "JavaboyMessage{" +
            "id='" + id + '\'' +
            ", data=" + data +
            '}';
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Object getData() {
        return data;
    }

    public void setData(Object data) {
        this.data = data;
    }
}

```

接下来封装一个消息队列：

```

public class DelayMsgQueue {
    private Jedis jedis;
    private String queue;

    public DelayMsgQueue(Jedis jedis, String queue) {
        this.jedis = jedis;
        this.queue = queue;
    }

    /**
     * 消息入队
     *
     * @param data 要发送的消息
     */
    public void queue(Object data) {
        //构造一个 JavaboyMessage
        JavaboyMessage msg = new JavaboyMessage();
        msg.setId(UUID.randomUUID().toString());
        msg.setData(data);
        //序列化
        try {
            String s = new ObjectMapper().writeValueAsString(msg);
            System.out.println("msg publish:" + new Date());
            //消息发送, score 延迟 5 秒
            jedis.zadd(queue, System.currentTimeMillis() + 5000, s);
        } catch (JsonProcessingException e) {

```

```

        e.printStackTrace();
    }
}

/**
 * 消息消费
 */
public void loop() {
    while (!Thread.interrupted()) {
        //读取 score 在 0 到当前时间戳之间的消息
        Set<String> zrange = jedis.zrangeByScore(queue, 0,
System.currentTimeMillis(), 0, 1);
        if (zrange.isEmpty()) {
            //如果消息是空的，则休息 500 毫秒然后继续
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                break;
            }
            continue;
        }
        //如果读取到了消息，则直接读取消息出来
        String next = zrange.iterator().next();
        if (jedis.zrem(queue, next) > 0) {
            //抢到了，接下来处理业务
            try {
                JavaboyMessage msg = new ObjectMapper().readValue(next,
JavaboyMessage.class);
                System.out.println("receive msg:" + msg);
            } catch (JsonProcessingException e) {
                e.printStackTrace();
            }
        }
    }
}
}
}

```

测试:

```

public class DelayMsgTest {
    public static void main(String[] args) {
        Redis redis = new Redis();
        redis.execute(jedis -> {
            //构造一个消息队列
            DelayMsgQueue queue = new DelayMsgQueue(jedis, "javaboy-delay-
queue");

            //构造消息生产者
            Thread producer = new Thread(){
                @Override
                public void run() {
                    for (int i = 0; i < 5; i++) {
                        queue.queue("www.javaboy.org>>>>" + i);
                    }
                }
            };

            //构造一个消息消费者
            Thread consumer = new Thread(){

```

```
        @Override
        public void run() {
            queue.loop();
        }
    };
    //启动
    producer.start();
    consumer.start();
    //休息 7 秒后，停止程序
    try {
        Thread.sleep(7000);
        consumer.interrupt();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});
}
```

8.再谈 Bit 操作

8.1 基本介绍

用户一年的签到记录，如果你用 string 类型来存储，那你需要 365 个 key/value，操作起来麻烦。通过位图可以有效的简化这个操作。

它的统计很简单：

01111000111

每天的记录占一个位，365 天就是 365 个位，大概 46 个字节，这样可以有效的节省存储空间，如果有一天想要统计用户一共签到了多少天，统计 1 的个数即可。

对于位图的操作，可以直接操作对应的字符串（get/set），可以直接操作位（getbit/setbit）。

8.2 基本操作

Redis 的基本操作可以归为两大类：

8.2.1 零存整取

例如存储一个 Java 字符串：

字符	ASCII	二进制
J	74	01001010
a	97	01100001
v	118	01110110

接下来去存储：

```
127.0.0.1:6379> get name
"Ja"
127.0.0.1:6379> SETBIT name 17 1
(integer) 0
127.0.0.1:6379> SETBIT name 18 1
(integer) 0
127.0.0.1:6379> SETBIT name 19 1
(integer) 0
127.0.0.1:6379> SETBIT name 21 1
(integer) 0
127.0.0.1:6379> SETBIT name 22 1
(integer) 0
127.0.0.1:6379> get name
"Jav"
```

8.2.1 整存零取

存一个字符串进去，但是通过位操作获取字符串。

8.3 统计

例如签到记录：

01111000111

1 表示签到的天，0 表示没签到，统计总的签到天数：

可以使用 bitcount。

```
127.0.0.1:6379> BITCOUNT name
(integer) 14
```

bitcount 中，可以统计的起始位置，但是注意，这个起始位置是指字符的起始位置而不是 bit 的起始位置。

除了 bitcount 之外，还有一个 bitpos。bitpos 可以用来统计在指定范围内出现的第一个 1 或者 0 的位置，这个命令中的起始和结束位置都是字符索引，不是 bit 索引，一定要注意。

8.4 Bit 批处理

在 Redis 3.2 之后，新加了一个功能叫做 bitfield，可以对 bit 进行批量操作。

例如：

BITFIELD name get u4 0

表示获取 name 中的位，从 0 开始获取，获取 4 个位，返回一个无符号数字。

- u 表示无符号数字
- i 表示有符号数字，有符号的话，第一个符号就表示符号位，1 表示是一个负数。

bitfield 也可以一次执行多个操作。

GET:

```
127.0.0.1:6379> BITFIELD name get u4 1 get i4 1 get u4 0 get i4 0
1) (integer) 9
2) (integer) -7
3) (integer) 4
4) (integer) 4
127.0.0.1:6379>
```

SET:

用无符号的 98 转成的 8 位二进制数字，代替从第 8 位开始接下来的 8 位数字。

```
127.0.0.1:6379> BITFIELD name set u8 8 98  
1) (integer) 97
```

INCRBY:

对置顶范围进行自增操作，自增操作可能会出现溢出，既可能是向上溢出，也可能是向下溢出。Redis 中对于溢出的处理方案是折返。8 位无符号数 255 加 1 溢出变为 0；8 位有符号数 127，加 1 变为 -128。

也可以修改默认的溢出策略，可以改为 fail，表示执行失败。

```
BITFIELD name overflow fail incrby u2 6 1
```

sat 表示留在在最大/最小值。

```
BITFIELD name overflow sat incrby u2 6 1
```

9.HyperLogLog

一般我们评估一个网站的访问量，有几个主要的参数：

- pv, Page View, 网页的浏览量
- uv, User View, 访问的用户

一般来说，pv 或者 uv 的统计，可以自己来做，也可以借助一些第三方的工具，比如 cnzz，友盟 等。

如果自己实现，pv 比较简单，可以直接通过 Redis 计数器就能实现。但是 uv 就不一样，uv 涉及到另外一个问题，去重。

我们首先需要在前端给每一个用户生成一个唯一 id，无论是登录用户还是未登录用户，都要有一个唯一 id，这个 id 伴随着请求一起到达后端，在后端我们通过 set 集合中的 sadd 命令来存储这个 id，最后通过 scard 统计集合大小，进而得出 uv 数据。

如果是千万级别的 UV，需要的存储空间就非常惊人。而且，像 UV 统计这种，一般也不需要特别精确，800w 的 uv 和 803w 的 uv，其实差别不大。所以，我们要介绍今天的主角---HyperLogLog

Redis 中提供的 HyperLogLog 就是专门用来解决这个问题的，HyperLogLog 提供了一套不怎么精确但是够用的去重方案，会有误差，官方给出的误差数据是 0.81%，这个精确度，统计 UV 够用了。

HyperLogLog 主要提供了两个命令：pfadd 和 pfcount。

pfadd 用来添加记录，类似于 sadd，添加过程中，重复的记录会自动去重。

pfcount 则用来统计数据。

数据量少的时候看不出来误差。

在 Java 中，我们多添加几个元素：

```
public class HyperLogLog {
    public static void main(String[] args) {
        Redis redis = new Redis();
        redis.execute(jedis -> {
            for (int i = 0; i < 1000; i++) {
                jedis.pfadd("uv", "u" + i, "u" + (i + 1));
            }
            long uv = jedis.pfcount("uv");
            System.out.println(uv); //理论值是 1001
        });
    }
}
```

理论值是 1001，实际打印出来 994，有误差，但是在可以接受的范围内。

除了 pfadd 和 pfcount 之外，还有一个命令 pfmerge，合并多个统计结果，在合并的过程中，会自动去重多个集合中重复的元素。

10. 布隆过滤器

10.1 场景重现

我们用 HyperLogLog 来估计一个数，有偏差但是也够用。HyperLogLog 主要提供两个方法：

- pfadd
- pfcount

但是 HyperLogLog 没有判断是否包含的方法，例如 pfexists、pfcontains 等。没有这样的方法存在，但是我们有这样的业务需求。

例如刷今日头条，推送的内容有相似的，但是没有重复的。这就涉及到如何在推送的时候去重？

解决方案很多，例如将用户的浏览历史记录下来，然后每次推送时去比较该条消息是否已经给用户推送了。但是这种方式效率极低，不推荐。

解决这个问题，就要靠我们今天要说的布隆过滤器。

10.2 Bloom Filter 介绍

Bloom Filter 专门用来解决我们上面所说的去重问题的，使用 Bloom Filter 不会像使用缓存那么浪费空间。当然，他也存在一个小小问题，就是不太精确。

Bloom Filter 相当于是一个不太精确的 set 集合，我们可以利用它里边的 contains 方法去判断某一个对象是否存在，但是需要注意，这个判断不是特别精确。一般来说，通过 contains 判断某个值不存在，那就一定不存在，但是判断某个值存在的话，则他可能不存在。

以今日头条为例，假设我们将用户的浏览记录用 B 表示，A 表示用户没有浏览的新闻，现在要给用户推送消息，先去 B 里边判断这条消息是否已经推送过，如果判断结果说没推送过（B 里边没有这条记录），那就一定没有推送过。如果判断结果说有推送过（B 里边也有可能没有这条消息），这个时候该条消息就不会推送给用户，导致用户错过该条消息，当然这是概率极低的。

10.3 Bloom Filter 原理

每一个布隆过滤器，在 Redis 中都对应了一个**大型的位数组**以及几个不同的 hash 函数。

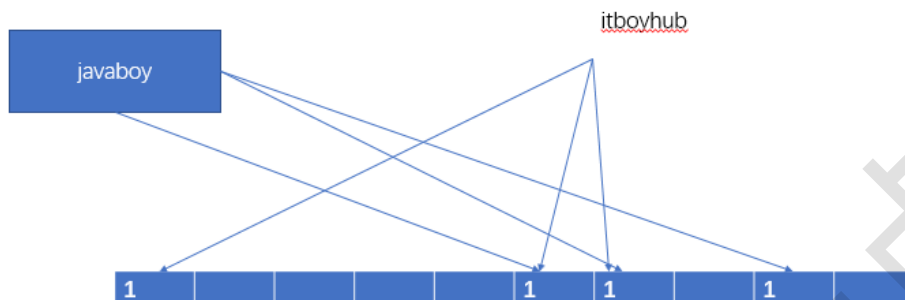
所谓的 add 操作是这样的：

首先根据几个不同的 hash 函数给元素进行 hash 运算一个整数索引值，拿到这个索引值之后，对位数组的长度进行取模运算，得到一个位置，每一个 hash 函数都会得到一个位置，将位数组中对应的位置设置位 1，这样就完成了添加操作。

存一个 javaboy

假设用三个 hash 给 javaboy 求一个 hash 值出来，分别是 5、6、8，用 5、6、8 分别和数组的长度取模，结果还是 5、6、8，将数组中这几个位置设置为 1

itboyhub 20\25\36 取模结果 0 5 6



当判断元素是否粗存在时，依然先对元素进行 hash 运算，将运算的结果和位数组取模，然后去对应的位置查看是否有相应的数据，如果有，表示元素可能存在（因为这个有数据的地方也可能是其他元素存进来的），如果没有表示元素一定不存在。

Bloom Filter 中，误判的概率和位数组的大小有很大关系，位数组越大，误判概率越小，当然占用的存储空间越大；位数组越小，误判概率越大，当然占用的存储空间就小。

10.4 Bloom Filter 安装

- <https://oss.redislabs.com/redisbloom/Quick Start/>

这里给大家介绍两种安装方式：

1. Docker：

```
docker run -p 6379:6379 --name redis-redisbloom redislabs/rebloom:latest
```

2. 自己编译安装：

```
cd redis-5.0.7
git clone https://github.com/RedisBloom/RedisBloom.git
cd RedisBloom/
make
cd ..
redis-server redis.conf --loadmodule ./RedisBloom/redisbloom.so
```

安装完成后，执行 bf.add 命令，测试安装是否成功。

每次启动时都输入 `redis-server redis.conf --loadmodule ./RedisBloom/redisbloom.so` 比较麻烦，我们可以将要加载的模块在 redis.conf 中提前配置好。


```
##### MODULES #####

# Load modules at startup. If the server is not able to load modules
# it will abort. It is possible to use multiple loadmodule directives.
#
# loadmodule /path/to/my_module.so
# loadmodule /path/to/other_module.so
loadmodule /root/redis-5.0.7/RedisBloom/redisbloom.so
```

最下面这一句，配置完成后，以后只需要 `redis-server redis.conf` 来启动 Redis 即可。

10.5 基本用法

主要是两类命令，添加和判断是否存在。

- `bf.add\bf.madd` 添加和批量添加
- `bf.exists\bf.mexists` 判断是否存在和批量判断

使用 Jedis 操作布隆过滤器，首先添加依赖：

```
<dependency>
  <groupId>com.redislabs</groupId>
  <artifactId>jredbloom</artifactId>
  <version>1.2.0</version>
</dependency>
```

然后进行测试：

```
public class BloomFilter {
    public static void main(String[] args) {
        GenericObjectPoolConfig config = new GenericObjectPoolConfig();
        config.setMaxIdle(300);
        config.setMaxTotal(1000);
        config.setMaxWaitMillis(30000);
        config.setTestOnBorrow(true);
        JedisPool pool = new JedisPool(config, "192.168.91.128", 6379, 30000,
"javaboy");
        Client client = new Client(pool);
        //存入数据
        for (int i = 0; i < 100000; i++) {
            client.add("name", "javaboy-" + i);
        }
        //检查数据是否存在
        boolean exists = client.exists("name", "javaboy-999999");
        System.out.println(exists);
    }
}
```

默认情况下，我们使用的布隆过滤器它的错误率是 0.01，默认的元素大小是 100。但是这两个参数也是可以配置的。

我们可以调用 `bf.reserve` 方法进行配置。

```
BF.RESERVE k1 0.0001 1000000
```

第一个参数是 key，第二个参数是错误率，错误率越低，占用的空间越大，第三个参数预计存储的数量，当实际数量超出预计数量时，错误率会上升。

10.6 典型场景

前面所说的新闻推送过滤算是一个应用场景。

解决 Redis 穿透或者又叫缓存击穿问题。

假设我有 1 亿条用户数据，现在查询用户要去数据库中查，效率低而且数据库压力大，所以我们会把请求首先在 Redis 中处理（活跃用户存在 Redis 中），Redis 中没有的用户，再去数据库中查询。

现在可能会存在一种恶意请求，这个请求携带上了很多不存在的用户，这个时候 Redis 无法拦截下来请求，所以请求会直接跑到数据库里去。这个时候，这些恶意请求会击穿我们的缓存，甚至数据库，进而引起“雪崩效应”。

为了解决这个问题，我们就可以使用布隆过滤器。将 1 亿条用户数据存在 Redis 中不现实，但是可以存在布隆过滤器中，请求来了，首先去判断数据是否存在，如果存在，再去数据库中查询，否则就不去数据库中查询。

11.Redis 限流

11.1 预备知识

Pipeline（管道）本质上是由客户端提供的一种操作。Pipeline 通过调整指令列表的读写顺序，可以大幅度的节省 IO 时间，提高效率。

11.2 简单限流

```
public class RateLimiter {
    private Jedis jedis;

    public RateLimiter(Jedis jedis) {
        this.jedis = jedis;
    }

    /**
     * 限流方法
     * @param user 操作的用户，相当于是限流的对象
     * @param action 具体的操作
     * @param period 时间窗，限流的周期
     * @param maxCount 限流的次数
     * @return
     */
    public boolean isAllowed(String user, String action, int period, int maxCount) {
        //1.数据用 zset 保存，首先生成一个 key
        String key = user + "-" + action;
        //2.获取当前时间戳
        long nowTime = System.currentTimeMillis();
        //3.建立管道
        Pipeline pipelined = jedis.pipelined();
        pipelined.multi();
```

```

//4.将当前的操作先存储下来
pipelined.zadd(key, nowTime, String.valueOf(nowTime));
//5.移除时间窗之外的数据
pipelined.zremrangeByScore(key, 0, nowTime - period * 1000);
//6.统计剩下的 key
Response<Long> response = pipelined.zcard(key);
//7.将当前 key 设置一个过期时间, 过期时间就是时间窗
pipelined.expire(key, period + 1);
//关闭管道
pipelined.exec();
pipelined.close();
//8.比较时间窗内的操作数
return response.get() <= maxCount;
}

public static void main(String[] args) {
    Redis redis = new Redis();
    redis.execute(j -> {
        RateLimiter rateLimiter = new RateLimiter(j);
        for (int i = 0; i < 20; i++) {
            System.out.println(rateLimiter.isAllowed("javaboy", "publish",
5, 3));
        }
    });
}
}

```

11.3 深入限流操作

Redis4.0 开始提供了一个 Redis-Cell 模块, 这个模块使用**漏斗算法**, 提供了一个非常好用的限流指令。

漏斗算法就像名字一样, 是一个漏斗, 请求从漏斗的大口进, 然后从小口出进入到系统中, 这样, 无论是多大的访问量, 最终进入到系统中的请求, 都是固定的。

使用漏斗算法, 需要我们首先安装 Redis-Cell 模块:

- <https://github.com/brandur/redis-cell>

安装步骤:

```

wget https://github.com/brandur/redis-cell/releases/download/v0.2.4/redis-cell-
v0.2.4-x86_64-unknown-linux-gnu.tar.gz
tar -zxvf redis-cell-v0.2.4-x86_64-unknown-linux-gnu.tar.gz
mkdir redis-cell
mv libredis_cell.d ./redis-cell
mv libredis_cell.so ./redis-cell

```

接下来修改 redis.conf 文件, 加载额外的模块:

```
loadmodule /root/redis-5.0.7/redis-cell/libredis_cell.so
```

然后, 启动 Redis:

```
redis-server redis.conf
```

redis 启动成功后, 如果存在 CL.THROTTLE 命令, 说明 redis-cell 已经安装成功了。

CL.THROTTLE 命令一共有五个参数

1. 第一个参数是 key
2. 第二个参数是漏斗的容量
3. 时间窗内可以操作的次数
4. 时间窗
5. 每次漏出数量

执行完成后，返回值也有五个：

1. 第一个 0 表示允许，1 表示拒绝
2. 第二个参数是漏斗的容量
3. 第三个参数是漏斗的剩余空间
4. 如果拒绝了，多长时间后，可以再试
5. 多长时间后，漏斗会完全空出来

11.4 Lettuce扩展

首先定义一个命令接口：

```
public interface RedisCommandInterface extends Commands {
    @Command("CL.THROTTLE ?0 ?1 ?2 ?3 ?4")
    List<Object> throttle(String key, Long init, Long count, Long period, Long
quota);
}
```

定义完成后，接下来，直接调用即可：

```
public class ThrottleTest {
    public static void main(String[] args) {
        RedisClient redisClient =
RedisClient.create("redis://javaboy@192.168.91.128");
        StatefulRedisConnection<String, String> connect = redisClient.connect();
        RedisCommandFactory factory = new RedisCommandFactory(connect);
        RedisCommandInterface commands =
factory.getCommands(RedisCommandInterface.class);
        List<Object> list = commands.throttle("javaboy-publish", 10L, 10L, 60L,
1L);
        System.out.println(list);
    }
}
```

12.Redis 之 Geo

Redis3.2 开始提供了 GEO 模块。该模块也使用了 GeoHash 算法。

1.GeoHash

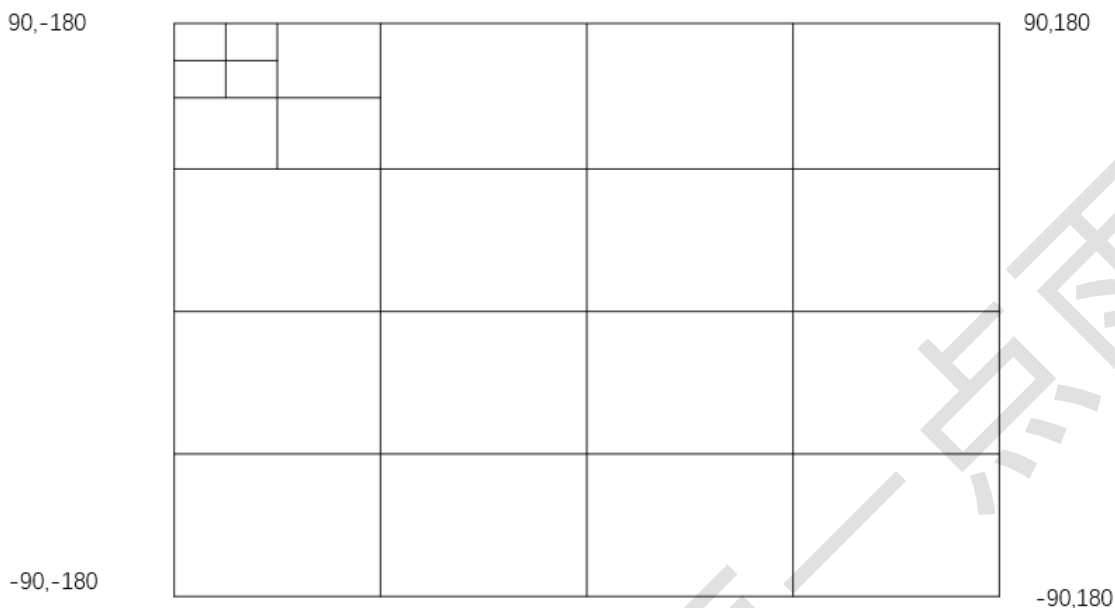
核心思想：GeoHash 是一种地址编码方法，使用这种方式，能够将二维的空间经纬度数据编码成一个一维字符串。

地球上经纬度的划分：

以经过伦敦格林尼治天文台旧址的经线为 0 度经线，向东就是东经，向西就是西经。如果我们将西经定义负，经度的范围就是 $[-180, 180]$ 。

纬度北纬 90 度到南纬 90 度，如果我们将南纬定义负，则纬度的范围就是 $[-90, 90]$ 。

接下来，以本初子午线和赤道为界，我们可以将地球上的点分配到一个二维坐标中：



GeoHash 算法就是基于这样的思想，划分的次数越多，区域越多，每个区域中的面积就更小了，精确度就会提高。

GeoHash 具体算法：

以北京天安门广场为例 ($39.9053908600, 116.3980007200$)：

1. 纬度的范围在 $(-90, 90)$ 之间，中间值为 0，对于 39.9053908600 值落在 $(0, 90)$ ，因此得到的值为 1
2. $(0, 90)$ 的中间值为 45， 39.9053908600 落在 $(0, 45)$ 之间，因此得到一个 0
3. $(0, 45)$ 的中间值为 22.5， 39.9053908600 落在 $(22.5, 45)$ 之间，因此得到一个 1
4.

这样，我们得到的纬度二进制是 101

按照同样的步骤，我们可以算出来经度的二进制是 110

接下来将经纬度合并（经度占偶数位，纬度占奇数位）：

111001

按照 Base32 (0-9, a-z, 去掉 a i l o) 对合并后的二进制数据进行编码，编码的时候，先将二进制转换为十进制，然后进行编码。

将编码得到的字符串，可以拿去 geohash.org 网站上解析。

GeoHash 有哪些特点：

1. 用一个字符串表示经纬度
2. GeoHash 表示的是一个区域，而不是一个点。
3. 编码格式有规律，例如一个地址编码之后的格式是 123，另一个地址编码之后的格式是 123456，从字符串上就可以看出来，123456 处于 123 之中。

2. Redis 中使用

添加地址：

```
GEOADD city 116.3980007200 39.9053908600 beijing
GEOADD city 114.0592002900 22.5536230800 shenzhen
```

查看两个地址之间的距离：

```
127.0.0.1:6379> GEODIST city beijing shenzhen km
"1942.5435"
```

获取元素的位置：

```
127.0.0.1:6379> GEOPOS city beijing
1) 1) "116.39800339937210083"
   2) "39.90539144357683909"
```

获取元素 hash 值：

```
127.0.0.1:6379> GEOHASH city beijing
1) "wx4g08w3y00"
```

通过 hash 值可以查看定位。 <http://geohash.org/wx4g08w3y00>

查看附近的人：

```
127.0.0.1:6379> GEORADIUSBYMEMBER city beijing 200 km count 3 asc
1) "beijing"
```

以北京为中心，方圆 200km 以内的城市找出来 3 个，按照远近顺序排列，这个命令不会排除 北京。

当然，也可以根据经纬度来查询（将 member 换成对应的经纬度）：

```
127.0.0.1:6379> GEORADIUS city 116.3980007200 39.9053908600 2000 km withdist
withhash withcoord count 4 desc
```

- <http://www.gpspg.com/maps.htm>

13.Redis 之 Scan

1. 简单介绍

scan 实际上是 keys 的一个升级版。

可以用 keys 来查询 key，在查询的过程中，可以使用通配符。keys 虽然用着还算方便，但是没有分页功能。同时因为 Redis 是单线程，所以 key 的执行会比较消耗时间，特别是当数据量大的时候，影响整个程序的运行。

为了解决 keys 存在的问题，从 Redis2.8 中开始，引入了 scan。

scan 具备 keys 的功能，但是不会阻塞线程，而且可以控制每次返回的结果数。

2. 基本用法

首先准备 10000 条测试数据：

```
public class ScanTest {
    public static void main(String[] args) {
        Redis redis = new Redis();
        redis.execute(jedis -> {
            for (int i = 0; i < 10000; i++) {
                jedis.set("k" + i, "v" + i);
            }
        });
    }
}
```

scan 命令一共提供了三个参数，第一个 cursor，第二个参数是 key，第三个参数是 limit。

cursor 实际上是指一维数组的位置索引，limit 则是遍历的一维数组个数（所以每次返回的数据大小可能不确定）。

```
scan 0 match k8* count 1000
```

3.原理

SCAN 的遍历顺序。

假设目有三条数据：

```
127.0.0.1:6379> keys *
1) "key1"
2) "db_number"
3) "myKey"
127.0.0.1:6379> scan 0 match * count 1
1) "2"
2) 1) "key1"
127.0.0.1:6379> scan 2 match * count 1
1) "1"
2) 1) "myKey"
127.0.0.1:6379> scan 1 match * count 1
1) "3"
2) 1) "db_number"
127.0.0.1:6379> scan 3 match * count 1
1) "0"
2) (empty list or set)
127.0.0.1:6379>
```

在遍历的过程中，大家发现游标的顺序是 0 2 1 3，从十进制来看好像没有规律，但是从转为二进制，则是有规律的：

```
00->10->01->11
```

这种规律就是高位进1，传统的二进制加法，是从右往左加，这里是从左往右加。

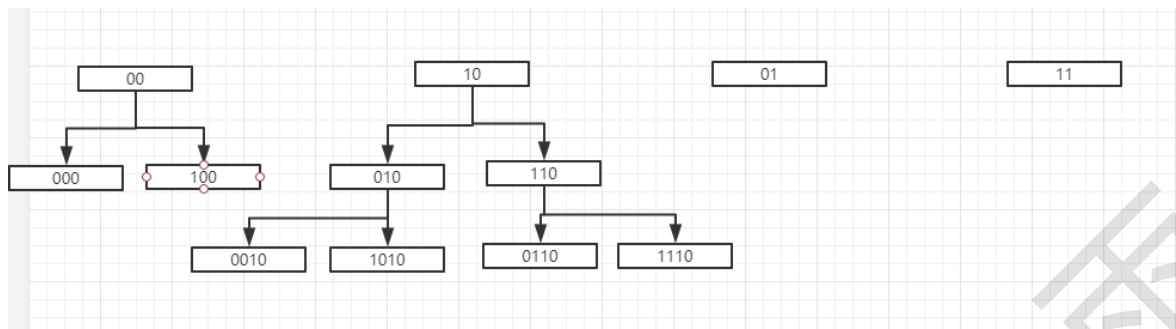
实际上，在 Redis 中，它的具体计算流程给是这样：

1. 将要计算的数字反转
2. 给反转后的数字加 1

3. 再反转

那么为什么不是按照 0、1、2、3、4...这样的顺序遍历呢？因为主要考虑到两个问题：

1. 字典扩容
2. 字典缩容



假如我们将要访问 110 时，发生了扩容，此时 scan 就会从 0110 开始遍历，之前已经被遍历过的元素就不会被重复遍历了。

假如我们将要访问 110 时，发生缩容，此时 scan 就会从 10 开始遍历，这个时候，也会遍历到 010，但是 010 之前的不会再被遍历了。所以，在发生缩容的时候，可能返回重复的元素。

4.其他用法

scan 是一系列的指令，除了遍历所有的 key 之外，也可以遍历某一个类型的 key，对应的命令有：

```
zscan-->zset  
hscan-->hash  
sscan-->set
```

14.Redis 单线程如何处理高并发

1.阻塞 IO 与非阻塞 IO

Java 在 JDK1.4 中引入 NIO，但是也有很多人在使用阻塞 IO，这两种 IO 有什么区别？

在阻塞模式下，如果你从数据流中读取不到指定大小的数据，IO 就会阻塞。比如已知会有 10 个字节发送过来，但是我目前只收到 4 个，还剩六个，此时就会发生阻塞。如果是非阻塞模式，虽然此时只收到 4 个字节，但是读到 4 个字节就会立即返回，不会傻傻等着，等另外 6 个字节来的时候，再去继续读取。

所以阻塞 IO 性能低于 非阻塞 IO。

如果有一个 Web 服务器，使用阻塞 IO 来处理请求，那么每一个请求都需要开启一个新的线程；但是如果使用了非阻塞 IO，基本上一个小小线程池就够用了，因为不会发生阻塞，每一个线程都能够高效利用。

2.Redis 的线程模型

首先一点，Redis 是单线程。单线程如何解决高并发问题的？

实际上，能够处理高并发的单线程应用不仅仅是 Redis，除了 Redis 之外，还有 NodeJS、Nginx 等等也是单线程。

Redis 虽然是单线程，但是运行很快，主要有如下几方面原因：

1. Redis 中的所有数据都是基于内存的，所有的计算也都是内存级别的计算，所以快。
2. Redis 是单线程的，所以有一些时间复杂度高的指令，可能会导致 Redis 卡顿，例如 keys。
3. Redis 在处理并发的客户端连接时，使用了非阻塞 IO。

在使用非阻塞 IO 时，有一个问题，就是线程如何知道剩下的数据来了？

这里就涉及到一个新的概念叫做**多路复用**，本质上就是一个事件轮询 API。

4. Redis 会给每一个客户端指令通过队列来排队进行顺序处理。
5. Redis 做出响应时，也会有一个响应的队列。

15.Redis 通信协议

Redis 通信使用了文本协议，文本协议比较费流量，但是 Redis 作者认为数据库的瓶颈不在于网络流量，而在于内部逻辑，所以采用了这样一个费流量的文本协议。

这个文本协议叫做 Redis Serialization Protocol，简称 RESP。

Redis 协议将传输的数据结构分为 5 种最小单元，单元结束时，加上回车换行符 `\r\n`。

1. 单行字符串以 `+` 开始，例如 `+javaboy.org\r\n`
2. 多行字符串以 `$` 开始，后面加上字符串长度，例如 `$11\r\njavaboy.org\r\n`
3. 整数值以 `:` 开始，例如 `:1024\r\n`
4. 错误消息以 `-` 开始
5. 数组以 `*` 开始，后面加上数组长度。

需要注意的是，如果是客户端连接服务端，只能使用第 5 种。

1.准备工作

做两件事情：

为了方便客户端连接 Redis，我们关闭 Redis 种的保护模式(在 `redis.conf` 文件中)

`protected no`

同时关闭密码：

```
# requirepass xxxx
```

配置完成后，重启 Redis。

2. 实战

接下来，我们通过 Socket+RESP 来定义两个最最常见的命令 `set` 和 `get`。

```
public class JavaboyRedisClient {
    private Socket socket;
    public JavaboyRedisClient() {
        try {
            socket = new Socket("192.168.91.128", 6379);
        } catch (IOException e) {
            e.printStackTrace();
            System.out.println("Redis 连接失败");
        }
    }
}
```

```

}

/**
 * 执行 Redis 中的 set 命令 [set,key,value]
 * @param key
 * @param value
 * @return
 */
public String set(String key, String value) throws IOException {
    StringBuilder sb = new StringBuilder();
    sb.append("*3")
        .append("\r\n")
        .append("$")
        .append("set".length())
        .append("\r\n")
        .append("set")
        .append("\r\n")
        .append("$")
        .append(key.getBytes().length)
        .append("\r\n")
        .append(key)
        .append("\r\n")
        .append("$")
        .append(value.getBytes().length)
        .append("\r\n")
        .append(value)
        .append("\r\n");
    System.out.println(sb.toString());
    socket.getOutputStream().write(sb.toString().getBytes());
    byte[] buf = new byte[1024];
    socket.getInputStream().read(buf);
    return new String(buf);
}

/**
 * 执行 Redis 中的 get 命令 [get,key]
 * @param key
 * @return
 */
public String get(String key) throws IOException {
    StringBuilder sb = new StringBuilder();
    sb.append("*2")
        .append("\r\n")
        .append("$")
        .append("get".length())
        .append("\r\n")
        .append("get")
        .append("\r\n")
        .append("$")
        .append(key.getBytes().length)
        .append("\r\n")
        .append(key)
        .append("\r\n");
    socket.getOutputStream().write(sb.toString().getBytes());
    byte[] buf = new byte[1024];
    socket.getInputStream().read(buf);
    return new String(buf);
}

```

```
public static void main(String[] args) throws IOException {  
    String set = new JavaboyRedisClient().set("k1", "江南一点雨");  
    System.out.println(set);  
    String k1 = new JavaboyRedisClient().get("k1");  
    System.out.println(k1);  
}  
}
```

16.Redis 持久化

Redis 是一个缓存工具，也叫做 NoSQL 数据库，既然是数据库，必然支持数据的持久化操作。在 Redis 中，数据库持久化一共有两种方案：

1. 快照方式
2. AOF 日志

1.快照

1.1 原理

Redis 使用操作系统的多进程机制来实现快照持久化：Redis 在持久化时，会调用 glibc 函数 fork 一个子进程，然后将快照持久化操作完全交给子进程去处理，而父进程则继续处理客户端请求。在这个过程中，子进程能够看到的内存中的数据在子进程产生的一瞬间就固定下来了，再也不会改变，也就是为什么 Redis 持久化叫做 快照。

1.2 具体配置

在 Redis 中，默认情况下，快照持久化的方式就是开启的。

默认情况下会产生一个 dump.rdb 文件，这个文件就是备份下来的文件。当 Redis 启动时，会自动的去加载这个 rdb 文件，从该文件中恢复数据。

具体的配置，在 redis.conf 中：

```
# 表示快照的频率，第一个表示 900 秒内如果有一个键被修改，则进行快照  
save 900 1  
save 300 10  
save 60 10000  
# 快照执行出错后，是否继续处理客户端的写命令  
stop-writes-on-bgsave-error yes  
# 是否对快照文件进行压缩  
rdbcompression yes  
# 表示生成的快照文件名  
dbfilename dump.rdb  
# 表示生成的快照文件位置  
dir ./
```

1.3 备份流程

1. 在 Redis 运行过程中，我们可以向 Redis 发送一条 save 命令来创建一个快照。但是需要注意，save 是一个阻塞命令，Redis 在收到 save 命令开始处理备份操作之后，在处理完成之前，将不再处理其他的请求。其他命令会被挂起，所以 save 使用的并不多。

2. 我们一般可以使用 bgsave, bgsave 会 fork 一个子进程去处理备份的事情, 不影响父进程处理客户端请求。
3. 我们定义的备份规则, 如果有规则满足, 也会自动触发 bgsave。
4. 另外, 当我们执行 shutdown 命令时, 也会触发 save 命令, 备份工作完成后, Redis 才会关闭。
5. 用 Redis 搭建主从复制时, 在从机连上主机之后, 会自动发送一条 sync 同步命令, 主机收到命令之后, 首先执行 bgsave 对数据进行快照, 然后才会给从机发送快照数据进行同步。

2.AOF

与快照持久化不同, AOF 持久化是将被执行的命令追加到 aof 文件末尾, 在恢复时, 只需要把记录下来的命令从头到尾执行一遍即可。

默认情况下, AOF 是没有开启的。我们需要手动开启:

```
# 开启 aof 配置
appendonly yes
# AOF 文件名
appendfilename "appendonly.aof"
# 备份的时机, 下面的配置表示每秒钟备份一次
appendfsync everysec
# 表示 aof 文件在压缩时, 是否还继续进行同步操作
no-appendfsync-on-rewrite no
# 表示当目前 aof 文件大小超过上一次重写时的 aof 文件大小的百分之多少的时候, 再次进行重写
auto-aof-rewrite-percentage 100
# 如果之前没有重写过, 则以启动时的 aof 大小为依据, 同时要求 aof 文件至少要大于 64M
auto-aof-rewrite-min-size 64mb
```

同时为了避免快照备份的影响, 记得将快照备份关闭:

```
save ""

#save 900 1
#save 300 10
#save 60 10000
```

17.Redis 事务

正常来说, 一个可以商用的数据库往往都有比较完善的事务支持, Redis 当然也不例外。相对于关系型数据库中的事务模型, Redis 中的事务要简单很多。因为简单, 所以 Redis 中的事务模型不太严格, 所以我们不能像使用关系型数据库中的事务那样来使用 Redis。

在关系型数据库中, 和事务相关的三个指令分别是:

- begin
- commit
- rollback

在 Redis 中, 当然也有对应的指令:

- multi
- exec
- discard

1.原子性

注意，Redis 中的事务并不能算作原子性。它仅仅具备隔离性，也就是说当前的事务可以不被其他事务打断。

由于每一次事务操作涉及到的指令还是比较多的，为了提高执行效率，我们在使用客户端的时候，可以通过 pipeline 来优化指令的执行。

Redis 中还有一个 watch 指令，watch 可以用来监控一个 key，通过这种监控，我们可以确保在 exec 之前，watch 的键的没有被修改过。

2.Java代码实现

```
public class TransactionTest {
    public static void main(String[] args) {
        new Redis().execute(jedis -> {
            new TransactionTest().saveMoney(jedis, "javaboy", 1000);
        });
    }

    public Integer saveMoney(Jedis jedis, String userId, Integer money) {
        while (true) {
            jedis.watch(userId);
            int v = Integer.parseInt(jedis.get(userId)) + money;
            Transaction tx = jedis.multi();
            tx.set(userId, String.valueOf(v));
            List<Object> exec = tx.exec();
            if (exec != null) {
                break;
            }
        }
        return Integer.parseInt(jedis.get(userId));
    }
}
```

18.Redis 主从同步

1.CAP

在分布式环境下，CAP 原理是一个非常基础的东西，所有的分布式存储系统，都只能在 CAP 中选择两项实现。

c: consistent 一致性

a: availability 可用性

p: partition tolerance 分布式容忍性

在一个分布式系统中，这三个只能满足两个：在一个分布式系统中，P 肯定是要实现的，c 和 a 只能选择其中一个。大部分情况下，大多数网站架构选择了 ap。

在 Redis 中，实际上就是保证最终一致。

Redis 中，当搭建了主从服务之后，如果主从之间的连接断开了，Redis 依然是可以操作的，相当于它满足可用性，但是此时主从两个节点中的数据会有差异，相当于牺牲了一致性。但是 Redis 保证最终一致，就是说当网络恢复的时候，从机会追赶主机，尽量保持数据一致。

2.主从复制

主从复制可以在一定程度上扩展 redis 性能，redis 的主从复制和关系型数据库的主从复制类似，从机能够精确的复制主机上的内容。实现了主从复制之后，一方面能够实现数据的读写分离，降低master的压力，另一方面也能实现数据的备份。

2.1 配置方式

假设我有三个redis实例，地址分别如下：

```
192.168.91.128:6379
192.168.91.128:6380
192.168.91.128:6381
```

即同一台服务器上三个实例，配置方式如下：

1. 将 redis.conf 文件更名为 redis6379.conf，方便我们区分，然后把 redis6379.conf 再复制两份，分别为 redis6380.conf 和 redis6381.conf。如下：

```
-rw-rw-r--. 1 root root 150927 2月 3 00:39 00-RELEASENOTES
-rw-r--r--. 1 root root 81 5月 12 22:44 appendonly.aof
-rw-rw-r--. 1 root root 53 2月 3 00:39 BUGS
-rw-rw-r--. 1 root root 1815 2月 3 00:39 CONTRIBUTING
-rw-rw-r--. 1 root root 1487 2月 3 00:39 COPYING
drwxrwxr-x. 6 root root 192 5月 12 14:17 deps
-rw-rw-r--. 1 root root 11 2月 3 00:39 INSTALL
-rw-rw-r--. 1 root root 151 2月 3 00:39 Makefile
-rw-rw-r--. 1 root root 4223 2月 3 00:39 MANIFESTO
-rw-rw-r--. 1 root root 20543 2月 3 00:39 README.md
-rw-rw-r--. 1 root root 58354 5月 12 22:01 redis6379.conf
-rw-r--r--. 1 root root 58354 5月 13 14:16 redis6380.conf
-rw-r--r--. 1 root root 58354 5月 13 14:16 redis6381.conf
-rwxrwxr-x. 1 root root 271 2月 3 00:39 runtest
-rwxrwxr-x. 1 root root 280 2月 3 00:39 runtest-cluster
-rwxrwxr-x. 1 root root 281 2月 3 00:39 runtest-sentinel
-rw-rw-r--. 1 root root 7606 2月 3 00:39 sentinel.conf
drwxrwxr-x. 3 root root 8192 5月 12 14:18 src
drwxrwxr-x. 10 root root 167 2月 3 00:39 tests
drwxrwxr-x. 8 root root 4096 2月 3 00:39 utils
[root@localhost redis-4.0.8]#
```

2. 打开 redis6379.conf，将如下配置均加上 6379,(默认是6379的不用修改)，如下：

```
port 6379
pidfile /var/run/redis_6379.pid
logfile "6379.log"
dbfilename dump6379.rdb
appendfilename "appendonly6379.aof"
```

3. 同理，分别打开 redis6380.conf 和 redis6381.conf 两个配置文件，将第二步涉及到 6379 的分别改为 6380 和 6381。
4. 输入如下命令，启动三个redis实例：

```
[root@localhost redis-4.0.8]# redis-server redis6379.conf
[root@localhost redis-4.0.8]# redis-server redis6380.conf
[root@localhost redis-4.0.8]# redis-server redis6381.conf
```

5. 输入如下命令，分别进入三个实例的控制台：

```
[root@localhost redis-4.0.8]# redis-cli -p 6379
[root@localhost redis-4.0.8]# redis-cli -p 6380
[root@localhost redis-4.0.8]# redis-cli -p 6381
```

此时我就成功配置了三个redis实例了。

6. 假设在这三个实例中，6379 是主机，即 master，6380 和 6381 是从机，即 slave，那么如何配置这种实例关系呢，很简单，分别在 6380 和 6381 上执行如下命令：

```
127.0.0.1:6381> SLAVEOF 127.0.0.1 6379
OK
```

这一步也可以通过在两个从机的 redis.conf 中添加如下配置来解决：

```
slaveof 127.0.0.1 6379
```

OK，主从关系搭建好后，我们可以通过如下命令可以查看每个实例当前的状态，如下：

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=56,lag=1
slave1:ip=127.0.0.1,port=6381,state=online,offset=56,lag=0
master_replid:26ca818360d6510b717e471f3f0a6f5985b6225d
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:56
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:56
```

我们可以看到 6379 是一个主机，上面挂了两个从机，两个从机的地址、端口等信息都展现出来了。如果我们在 6380 上执行 INFO replication，显示信息如下：

```
127.0.0.1:6380> INFO replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:6
master_sync_in_progress:0
slave_repl_offset:630
slave_priority:100
slave_read_only:1
connected_slaves:0
```



```
master_replid:26ca818360d6510b717e471f3f0a6f5985b6225d
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:630
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:630
```

我们可以看到 6380 是一个从机，从机的信息以及它的主机的信息都展示出来了。

7. 此时，我们在主机中存储一条数据，在从机中就可以 get 到这条数据了。

2.2 主从复制注意点

1. 如果主机已经运行了一段时间了，并且已经存储了一些数据了，此时从机连上来，那么从机会将主机上所有的数据进行备份，而不是从连接的那个时间点开始备份。
2. 配置了主从复制之后，主机上可读可写，但是从机只能读取不能写入（可以通过修改redis.conf 中 slave-read-only 的值让从机也可以执行写操作）。
3. 在整个主从结构运行过程中，如果主机不幸挂掉，重启之后，他依然是主机，主从复制操作也能够继续进行。

2.3 复制原理

每一个 master 都有一个 replication ID，这是一个较大的伪随机字符串，标记了一个给定的数据集。每个 master 也持有一个偏移量，master 将自己产生的复制流发送给 slave 时，发送多少字节的数据，自身的偏移量就会增加多少，目的是当有新的操作修改自己的数据集时，它可以以此更新 slave 的状态。复制偏移量即使在没有一个 slave 连接到 master 时，也会自增，所以基本上每一对给定的 Replication ID, offset 都会标识一个 master 数据集的确切版本。当 slave 连接到 master 时，它们使用 PSYNC 命令来发送它们记录的旧的 master replication ID 和它们至今为止处理的偏移量。通过这种方式，master 能够仅发送 slave 所需的增量部分。但是如果 master 的缓冲区中没有足够的命令积压缓冲记录，或者如果 slave 引用了不再知道的历史记录（replication ID），则会转而进行一个全量重同步：在这种情况下，slave 会得到一个完整的数据集副本，从头开始(参考redis官网)。

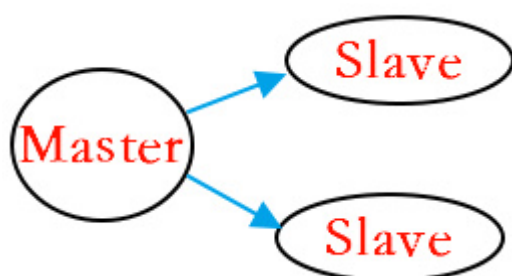
简单来说，就是以下几个步骤：

1. slave 启动成功连接到 master 后会发送一个 sync 命令。
2. Master 接到命令启动后台的存盘进程，同时收集所有接收到的用于修改数据集命令。
3. 在后台进程执行完毕之后，master 将传送整个数据文件到 slave,以完成一次完全同步。
4. 全量复制：而 slave 服务在接收到数据库文件数据后，将其存盘并加载到内存中。
5. 增量复制：Master 继续将新的所有收集到的修改命令依次传给 slave,完成同步。
6. 但是只要是重新连接 master,一次完全同步（全量复制）将被自动执行。

本文接上文，所用三个 redis 实例和上文一致，这里就不再赘述三个实例搭建方式。

2.4 一场接力赛

在上篇文章中，我们搭建的主从复制模式是下面这样的：



实际上，一主二仆的主从复制，我们可以搭建成下面这种结构：



搭建方式很简单，在前文基础上，我们只需要修改 6381 的 master 即可，在 6381 实例上执行如下命令，让 6381 从 6380 实例上复制数据，如下：

```
127.0.0.1:6381> SLAVEOF 127.0.0.1 6380
OK
```

此时，我们再看 6379 的 slave，如下：

```
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=0,lag=1
master_replid:4a38bbfa37586c29139b4ca1e04e8a9c88793651
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:0
```

只有一个 slave，就是 6380，我们再看 6380 的信息，如下：

```
127.0.0.1:6380> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_repl_offset:70
slave_priority:100
slave_read_only:1
connected_slaves:1
slave0:ip=127.0.0.1,port=6381,state=online,offset=70,lag=0
master_replid:4a38bbfa37586c29139b4ca1e04e8a9c88793651
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:70
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:70
```

6380 此时的角色是一个从机，它的主机是 6379，但是 6380 自己也有一个从机，那就是 6381。此时我们的主从结构如下图：



2.5 哨兵模式

结合上篇文章，我们一共介绍了两种主从模式了，但是这两种，不管是哪一种，都会存在这样一个问题，那就是当主机宕机时，就会发生群龙无首的情况，如果在主机宕机时，能够从从机中选出一个来充当主机，那么就不用我们每次去手动重启主机了，这就涉及到一个新的话题，那就是哨兵模式。

所谓的哨兵模式，其实并不复杂，我们还是在前面基础上来搭建哨兵模式。假设现在我的 master 是 6379，两个从机分别是 6380 和 6381，两个从机都是从 6379 上复制数据。先按照上文的步骤，我们配置好一主二仆，然后在 redis 目录下打开 sentinel.conf 文件，做如下配置：

```
sentinel monitor mymaster 127.0.0.1 6379 1
```

其中 mymaster 是给要监控的主机取的名字，随意取，后面是主机地址，最后面的 1 表示有多少个 sentinel 认为主机挂掉了，就进行切换（我这里只有一个，因此设置为 1）。好了，配置完成后，输入如下命令启动哨兵：

```
redis-sentinel sentinel.conf
```

然后启动我们的一主二仆架构，启动成功后，关闭 master，观察哨兵窗口输出的日志，如下：

```
5160:X 13 May 20:30:05.345 # +sdown master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.345 # +odown master mymaster 127.0.0.1 6379 #quorum 1/1
5160:X 13 May 20:30:05.345 # +new-epoch 1
5160:X 13 May 20:30:05.345 # +try-failover master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.350 # +vote-for-leader a1ab5ac4dc8252f368edc9e9c537d7ca700ba85e 1
5160:X 13 May 20:30:05.350 # +elected-leader master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.350 # +failover-state-select-slave master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.454 # +selected-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.454 # +failover-state-send-slaveof-noone slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.521 # +failover-state-wait-promotion slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.895 # +promoted-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.895 # +failover-state-reconf-slaves master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:05.906 # +slave-reconf-sent slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:06.924 # +slave-reconf-inprog slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:06.924 # +slave-reconf-done slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
5160:X 13 May 20:30:06.978 # +failover-end master mymaster 127.0.0.1 6379
5160:X 13 May 20:30:06.978 # +switch-master mymaster 127.0.0.1 6379 127.0.0.1 6380
5160:X 13 May 20:30:06.979 # +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6380
5160:X 13 May 20:30:06.979 # +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6380
5160:X 13 May 20:30:37.021 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6380
```

小伙伴们可以看到，6379 挂掉之后，redis 内部重新举行了选举，6380 重新上位。此时，如果 6379 重启，也不再是扛把子了，只能屈身做一个 slave 了

2.6 注意事项

由于所有的写操作都是先在 Master 上操作，然后同步更新到 Slave 上，所以从 Master 同步到 Slave 机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，Slave 机器数量的增加也会使这个问题更加严重。因此我们还需要集群来进一步提升 redis 性能，这个问题我们将在后面说到。

3. Jedis 操作哨兵模式

准备工作：

1. 所有的实例均配置 masterauth（在 redis.conf 配置文件中）
2. 所有实例均需要配置绑定地址：bind 192.168.91.128

另外，哨兵配置的时候，监控的 master 也不要直接写 127.0.0.1，按如下方式写：

```
sentinel monitor mymaster 192.168.91.128 6380 1
```

做好准备工作，然后启动三个 redis 实例，同时启动哨兵。

```
public class Sentinel {
    public static void main(String[] args) {
        JedisPoolConfig config = new JedisPoolConfig();
        config.setMaxTotal(10);
        config.setMaxWaitMillis(1000);
        String master = "mymaster";
        Set<String> sentinels = new HashSet<>();
        sentinels.add("192.168.91.128:26379");
        JedisSentinelPool sentinelPool = new JedisSentinelPool(master,
            sentinels, config, "javaboy");
        Jedis jedis = null;
        while (true) {
            try {
                jedis = sentinelPool.getResource();
                String k1 = jedis.get("k1");
                System.out.println(k1);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (jedis != null) {
                    jedis.close();
                }
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

4.SpringBoot

Spring Boot 操作哨兵模式

添加 Spring Boot 依赖：

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

配置 Redis 连接：

```
spring:
  redis:
    password: javaboy
    timeout: 5000
    sentinel:
      master: mymaster
      nodes: 192.168.91.128:26379
```

测试代码:

```
@SpringBootTest
class SentinelApplicationTests {

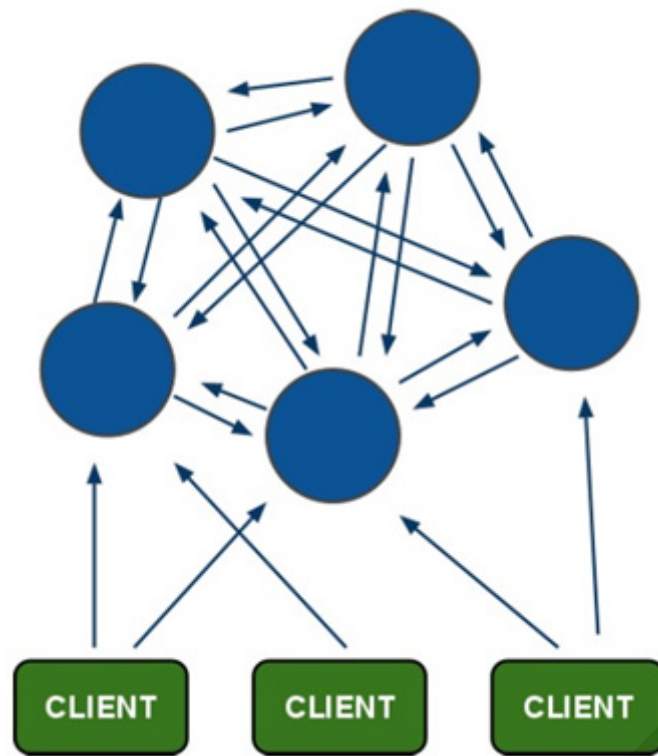
    @Autowired
    StringRedisTemplate redisTemplate;

    @Test
    void contextLoads() {
        while (true) {
            try {
                String k1 = redisTemplate.opsForValue().get("k1");
                System.out.println(k1);
            } catch (Exception e) {
            } finally {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

19.Redis 集群

集群原理

Redis 集群架构如下图:



Redis 集群运行原理如下：

1. 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
2. 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
3. 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
4. Redis-cluster 把所有的物理节点映射到 [0-16383]slot 上,cluster (簇)负责维护 `node<->slot<->value`。Redis 集群中内置了 16384 个哈希槽, 当需要在 Redis 集群中放置一个key-value 时, Redis 先对 key 使用 crc16 算法算出一个结果, 然后把结果对 16384 求余数, 这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽, Redis 会根据节点数量大致均等的将哈希槽映射到不同的节点

怎么样投票

投票过程是集群中所有 master 参与,如果半数以上 master 节点与 master 节点通信超过 `cluster-node-timeout` 设置的时间,认为当前 master 节点挂掉。

怎么样判定节点不可用

1. 如果集群任意 master 挂掉,且当前 master 没有 slave.集群进入 fail 状态,也可以理解成集群的 slot 映射 [0-16383] 不完整时进入 fail 状态。
2. 如果集群超过半数以上 master 挂掉,无论是否有 slave,集群进入 fail 状态,当集群不可用时,所有对集群的操作做都不可用, 收到((error) CLUSTERDOWN The cluster is down)错误。

ruby 环境

Redis 集群管理工具 redis-trib.rb 依赖 ruby 环境, 首先需要安装 ruby 环境:

安装 ruby:

```
yum install ruby
yum install rubygems
```


但是这种安装方式装好的 ruby 版本可能不适用，如果安装失败，可以参考这篇文章解决[redis requires Ruby version >= 2.2.2](#)。

集群搭建

首先我们对集群做一个简单规划，假设我的集群中一共有三个节点，每个节点一个主机一个从机，这样我一共需要 6 个 Redis 实例。首先创建 redis-cluster 文件夹，在该文件夹下分别创建 7001、7002、7003、7004、7005、7006 文件夹，用来存放我的 Redis 配置文件，如下：

```
[root@localhost redis-cluster]# ls
7001 7002 7003 7004 7005 7006 redis-4.0.9
```

将 Redis 也在 redis-cluster 目录下安装一份，然后将 redis.conf 文件向 7001-7006 这 6 个文件夹中分别拷贝一份，拷贝完成后，分别修改如下参数：

```
port 7001
#bind 127.0.0.1
cluster-enabled yes
cluster-config-xxxxx7001.conf
protected no
daemonize yes
```

这是 7001 目录下的配置，其他的文件夹将 7001 改为对应的数字即可。修改完成后，进入到 redis 安装目录中，分别启动各个 redis，使用刚刚修改过的配置文件，如下：

```
[root@localhost redis-cluster]# cd redis-4.0.9/
[root@localhost redis-4.0.9]# redis-server ../7001/redis.conf
7688:C 01 Jun 11:15:24.737 # 0000000000000000 Redis is starting 0000000000000000
7688:C 01 Jun 11:15:24.737 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7688, just started
7688:C 01 Jun 11:15:24.737 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7002/redis.conf
7693:C 01 Jun 11:15:30.039 # 0000000000000000 Redis is starting 0000000000000000
7693:C 01 Jun 11:15:30.039 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7693, just started
7693:C 01 Jun 11:15:30.039 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7003/redis.conf
7698:C 01 Jun 11:15:35.890 # 0000000000000000 Redis is starting 0000000000000000
7698:C 01 Jun 11:15:35.890 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7698, just started
7698:C 01 Jun 11:15:35.890 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7004/redis.conf
7703:C 01 Jun 11:15:39.731 # 0000000000000000 Redis is starting 0000000000000000
7703:C 01 Jun 11:15:39.731 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7703, just started
7703:C 01 Jun 11:15:39.731 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7005/redis.conf
7708:C 01 Jun 11:15:44.856 # 0000000000000000 Redis is starting 0000000000000000
7708:C 01 Jun 11:15:44.856 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7708, just started
7708:C 01 Jun 11:15:44.857 # Configuration loaded
[root@localhost redis-4.0.9]# redis-server ../7006/redis.conf
7713:C 01 Jun 11:15:52.596 # 0000000000000000 Redis is starting 0000000000000000
7713:C 01 Jun 11:15:52.596 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=7713, just started
7713:C 01 Jun 11:15:52.596 # Configuration loaded
```

启动成功后，我们可以查看 redis 进程，如下：

```
[root@localhost redis-4.0.9]# ps -aux|grep redis
root      4001  0.0  0.2 16264 5448 pts/1    S+   10:13   0:00 redis-cli -p 6380
root      4003  0.1  0.5 147300 9792 ?        Ssl  10:13   0:07 redis-server *:6381
root      4007  0.0  0.2 16264 5444 pts/2    S+   10:13   0:00 redis-cli -p 6381
root      4017  0.0  0.2 16264 5424 pts/0    S+   10:19   0:00 redis-cli
root      7689  0.1  0.5 151400 9680 ?        Ssl  11:15   0:00 redis-server *:7001 [cluster]
root      7694  0.1  0.5 147304 9680 ?        Ssl  11:15   0:00 redis-server *:7002 [cluster]
root      7699  0.1  0.5 147304 9680 ?        Ssl  11:15   0:00 redis-server *:7003 [cluster]
root      7704  0.2  0.5 147304 9680 ?        Ssl  11:15   0:00 redis-server *:7004 [cluster]
root      7709  0.2  0.5 147304 9676 ?        Ssl  11:15   0:00 redis-server *:7005 [cluster]
root      7714  0.4  0.5 147304 9680 ?        Ssl  11:15   0:00 redis-server *:7006 [cluster]
root      7719  0.0  0.0 112668 972 pts/3    R+   11:16   0:00 grep --color=auto redis
[root@localhost redis-4.0.9]#
```

这个表示各个节点都启动成功了。接下来我们就可以进行集群的创建了，首先将 redis/src 目录下的 redis-trib.rb 文件拷贝到 redis-cluster 目录下，然后在 redis-cluster 目录下执行如下命令：

```
./redis-trib.rb create --replicas 1 192.168.248.128:7001 192.168.248.128:7002  
192.168.248.128:7003 192.168.248.128:7004 192.168.248.128:7005  
192.168.248.128:7006
```

注意，replicas 后面的 1 表示每个主机都带有 1 个从机，执行过程如下：

```
[root@localhost redis-cluster]# ./redis-trib.rb create --replicas 1 192.168.248.128:7001 192.168.248.128:7002 192.168.248.128:7003 192.168.248.128:7004 192.168.248.128:7005 192.168.248.128:7006  
>>> Creating cluster  
>>> Performing hash slots allocation on 6 nodes...  
Using 3 masters:  
192.168.248.128:7001  
192.168.248.128:7002  
192.168.248.128:7003  
Adding replica 192.168.248.128:7005 to 192.168.248.128:7001  
Adding replica 192.168.248.128:7006 to 192.168.248.128:7002  
Adding replica 192.168.248.128:7004 to 192.168.248.128:7003  
>>> Trying to optimize slaves allocation for anti-affinity  
[WARNING] Some slaves are in the same host as their master  
M: 96aff797e48a9c832354a0f26f71bcd9c803184 192.168.248.128:7001  
slots:0-5460 (5461 slots) master  
M: 2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468 192.168.248.128:7002  
slots:5461-10922 (5462 slots) master  
M: 5e6bd470a1daf742207165b7d5e042f8c01f3081 192.168.248.128:7003  
slots:10923-16383 (5461 slots) master  
S: 269ef58fd4104d535b3e4ad781f3c4fa88ff8f49 192.168.248.128:7004  
replicas: 96aff797e48a9c832354a0f26f71bcd9c803184  
S: 40502040f08494c8dc19dd20b55a5a307edc8fc1 192.168.248.128:7005  
replicas: 2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468  
S: 2d09b81328910c5a525253ceeb1831cd8d9bf74 192.168.248.128:7006  
replicas: 5e6bd470a1daf742207165b7d5e042f8c01f3081  
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 192.168.248.128:7001)  
M: 96aff797e48a9c832354a0f26f71bcd9c803184 192.168.248.128:7001  
slots:0-5460 (5461 slots) master  
1 additional replica(s)  
M: 2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468 192.168.248.128:7002  
slots:5461-10922 (5462 slots) master  
1 additional replica(s)  
S: 269ef58fd4104d535b3e4ad781f3c4fa88ff8f49 192.168.248.128:7004  
slots: (0 slots) slave  
replicas: 96aff797e48a9c832354a0f26f71bcd9c803184  
S: 2d09b81328910c5a525253ceeb1831cd8d9bf74 192.168.248.128:7006  
replicas: 5e6bd470a1daf742207165b7d5e042f8c01f3081
```

注意创建过程的日志，每个redis都获得了一个编号，同时日志也说明了哪些实例做主机，哪些实例做从机，每个从机的主机是谁，每个主机所分配到的hash槽范围等等。

查询集群信息

集群创建成功后，我们可以登录到 Redis 控制台查看集群信息，注意登录时要添加 `-c` 参数，表示以集群方式连接，如下：

```
[root@localhost redis-4.0.9]# redis-cli -p 7001 -c  
127.0.0.1:7001> cluster info  
cluster_state:ok  
cluster_slots_assigned:16384  
cluster_slots_ok:16384  
cluster_slots_pfail:0  
cluster_slots_fail:0  
cluster_known_nodes:6  
cluster_size:3  
cluster_current_epoch:6  
cluster_my_epoch:1  
cluster_stats_messages_ping_sent:265  
cluster_stats_messages_pong_sent:256  
cluster_stats_messages_sent:521  
cluster_stats_messages_ping_received:251  
cluster_stats_messages_pong_received:265  
cluster_stats_messages_meet_received:5  
cluster_stats_messages_received:521  
127.0.0.1:7001>  
  
127.0.0.1:7001> cluster nodes  
2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468 192.168.248.128:7002 master - 0 1527823610000 2 connected 5461-10922  
269ef58fd4104d535b3e4ad781f3c4fa88ff8f49 192.168.248.128:7004 slave 96aff797e48a9c832354a0f26f71bcd9c803184 0 1527823611000 4 connected  
2d09b81328910c5a525253ceeb1831cd8d9bf74 192.168.248.128:7006 slave 5e6bd470a1daf742207165b7d5e042f8c01f3081 0 1527823612603 6 connected  
40502040f08494c8dc19dd20b55a5a307edc8fc1 192.168.248.128:7005 slave 2ccae3f43e8142529f8ad9a8a8ab28d9aff2a468 0 1527823611644 5 connected  
5e6bd470a1daf742207165b7d5e042f8c01f3081 192.168.248.128:7003 master - 0 1527823610000 3 connected 10923-16383  
96aff797e48a9c832354a0f26f71bcd9c803184 192.168.248.128:7001 myself,master - 0 1527823611000 1 connected 0-5460  
127.0.0.1:7001>
```

添加主节点

首先我们准备一个端口为 7007 的主节点并启动，准备方式和前面步骤一样，启动成功后，通过如下命令添加主节点：

```
./redis-trib.rb add-node 127.0.0.1:7007 127.0.0.1:7001
```

主节点添加之后，我们可以通过 cluster nodes 命令查看主节点是否添加成功，此时我们发现新添加的节点没有分配到 slot，如下：

```
127.0.0.1:7007> cluster nodes
6d4ae8e64a340837acefb4d85c7eff5a0b85a7a5 192.168.248.128:7006@17006 slave 91b6e0a668b21f3cad35c2588fd959839b8ce6db 0 15278368
ted
c5e409f65d0bd292088f0500283ebb5db7fa08e2 192.168.248.128:7004@17004 slave 80fd8e9e5f981e63d9852dc29c273e703d653afa 0 15278368
ted
91b6e0a668b21f3cad35c2588fd959839b8ce6db 192.168.248.128:7003@17003 master - 0 1527836867038 3 connected 11256-16383
d9db9c4fd4b22f794694c85f90d6f3d066c85ed 192.168.248.128:7005@17005 slave f1bea8b2cb0fde9c3b1806f42a410526c34cb2a2 0 15278368
ted
80fd8e9e5f981e63d9852dc29c273e703d653afa 127.0.0.1:7001@17001 master - 0 1527836862000 9 connected 167-5794 10923-11255
f1bea8b2cb0fde9c3b1806f42a410526c34cb2a2 192.168.248.128:7002@17002 master - 0 1527836864000 8 connected 0-166 5795-10922
ed1cc96d20909a503b40f6d69f8a23ac0d6ba845 127.0.0.1:7007@17007 myself,master - 0 1527836865000 0 connected
```

没有分配到 slot 将不能存储数据，此时我们需要手动分配 slot，分配命令如下：

```
./redis-trib.rb reshard 127.0.0.1:7001
```

后面的地址为任意一个节点地址，在分配的过程中，我们一共要输入如下几个参数：

1. 一共要划分多少个 hash 槽出来？就是我们总共要给新添加的节点分多少 hash 槽，这个参数依实际情况而定，如下：

```
How many slots do you want to move (from 1 to 16384)?
```

2. 这些划分出来的槽要给谁，这里输入 7007 节点的编号，如下：

```
What is the receiving node ID? ed1cc96d20909a503b40f6d69f8a23ac0d6ba845
```

3. 要让谁出血？因为 hash 槽目前已经全部分配完毕，要重新从已经分好的节点中拿出来一部分给 7007，必然要让另外三个节点把吃进去的吐出来，这里我们可以输入多个节点的编号，每次输完一个点击回车，输完所有的输入 done 表示输入完成，这样就让这几个节点让出部分 slot，如果要让所有具有 slot 的节点都参与到此次 slot 重新分配的活动中，那么这里直接输入 all 即可，如下：

```
What is the receiving node ID? ed1cc96d20909a503b40f6d69f8a23ac0d6ba845
Please enter all the source node IDs.
Type 'all' to use all the nodes as source nodes for the hash slots.
Type 'done' once you entered all the source nodes IDs.
Source node #1:all
```

OK，主要就是这几个参数，输完之后进入到 slot 重新分配环节，分配完成后，通过 cluster nodes 命令，我们可以发现 7007 已经具有 slot 了，如下：

```
127.0.0.1:7001> cluster nodes
80fd8e9e5f981e63d9852dc29c273e703d653afa 127.0.0.1:7001@17001 myself,master - 0 1527838117000 9 connected 531-5794 10923-11255
d9db9c4fd4b22f794694c85f90d6f3d066c85ed 192.168.248.128:7005@17005 slave f1bea8b2cb0fde9c3b1806f42a410526c34cb2a2 0 1527838117767 8 con
ted
6d4ae8e64a340837acefb4d85c7eff5a0b85a7a5 192.168.248.128:7006@17006 slave 91b6e0a668b21f3cad35c2588fd959839b8ce6db 0 1527838117000 6 con
ted
ed1cc96d20909a503b40f6d69f8a23ac0d6ba845 127.0.0.1:7007@17007 master - 0 1527838115735 10 connected 0-530 5795-5950 11256-11567
91b6e0a668b21f3cad35c2588fd959839b8ce6db 192.168.248.128:7003@17003 master - 0 1527838118783 3 connected 11568-16383
c5e409f65d0bd292088f0500283ebb5db7fa08e2 192.168.248.128:7004@17004 slave 80fd8e9e5f981e63d9852dc29c273e703d653afa 0 1527838115000 9 con
ted
f1bea8b2cb0fde9c3b1806f42a410526c34cb2a2 192.168.248.128:7002@17002 master - 0 1527838117000 8 connected 5951-10922
127.0.0.1:7001>
```

OK，刚刚我们是添加主节点，我们也可以添加从节点，比如我要把 7008 作为 7007 的从节点，添加方式如下：


```
./redis-trib.rb add-node --slave --master-id  
79bbb30bba66b4997b9360dd09849c67d2d02bb9 192.168.31.135:7008  
192.168.31.135:7007
```

其中 79bbb30bba66b4997b9360dd09849c67d2d02bb9 是 7007 的编号。

删除节点

删除节点也比较简单，如下：

```
./redis-trib.rb del-node 127.0.0.1:7005 4b45eb75c8b428fbd77ab979b85080146a9bc017
```

注意 4b45eb75c8b428fbd77ab979b85080146a9bc017 是要删除节点的编号。

再注意：删除已经占有 hash 槽的结点会失败，报错如下：

```
[ERR] Node 127.0.0.1:7005 is not empty! Reshard data away and try again.
```

需要将该结点占用的 hash 槽分配出去（分配方式与上文一致，不赘述）。

Jedis 操作 RedisCluster

```
public class RedisCluster {  
    public static void main(String[] args) {  
        Set<HostAndPort> nodes = new HashSet<>();  
        nodes.add(new HostAndPort("192.168.91.128", 7001));  
        nodes.add(new HostAndPort("192.168.91.128", 7002));  
        nodes.add(new HostAndPort("192.168.91.128", 7003));  
        nodes.add(new HostAndPort("192.168.91.128", 7004));  
        nodes.add(new HostAndPort("192.168.91.128", 7005));  
        nodes.add(new HostAndPort("192.168.91.128", 7006));  
        nodes.add(new HostAndPort("192.168.91.128", 7007));  
        JedisPoolConfig config = new JedisPoolConfig();  
        //连接池最大空闲数  
        config.setMaxIdle(300);  
        //最大连接数  
        config.setMaxTotal(1000);  
        //连接最大等待时间，如果是 -1 表示没有限制  
        config.setMaxWaitMillis(30000);  
        //在空闲时检查有效性  
        config.setTestOnBorrow(true);  
        JedisCluster cluster = new JedisCluster(nodes, 15000, 15000, 5,  
        "javaboy", config);  
        String set = cluster.set("k1", "v1");  
        System.out.println(set);  
        String k1 = cluster.get("k1");  
        System.out.println(k1);  
    }  
}
```