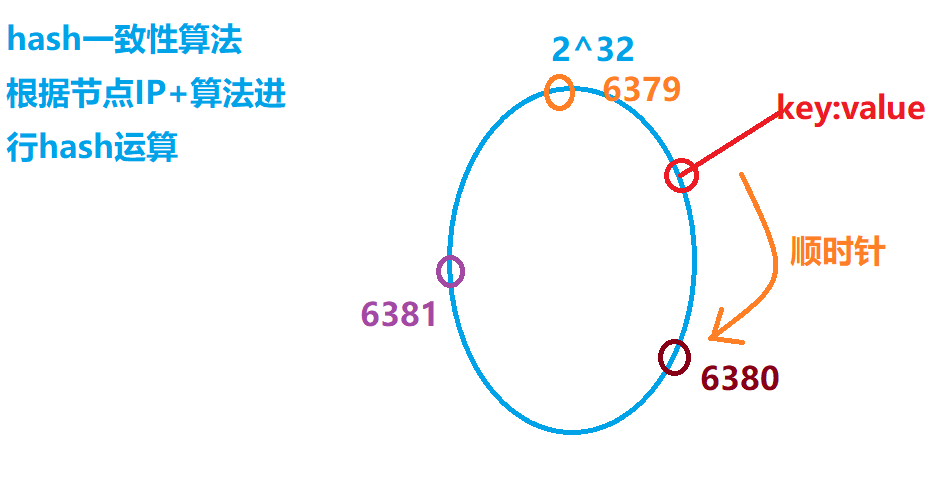
# Redis高级(一)

## Hash一致性问题

### 问题

说明:采用redis分片数据是如何存储的

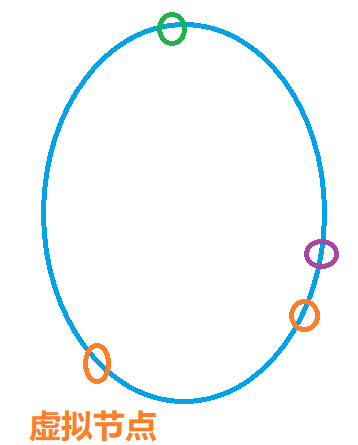
### 理解hash一致性



说明:根据节点IP进行计算.算出唯一的地址.之后保存数据时,key经过Hash计算顺时针找到最近的节点进行set操作.

### 均衡性

说明:让节点尽可能的保存的数据均匀.

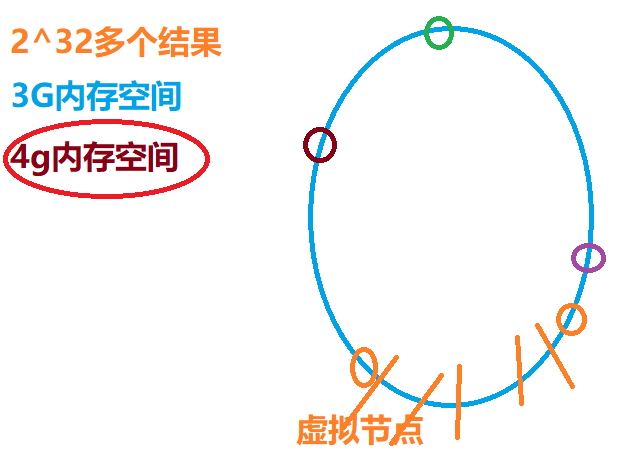


虚拟节点:如果出现负载不均的现象.则引入虚拟节点.可以平衡数据.

### 单调性

核心:当节点**添加**时,数据可以自动的实现迁移.

原则:尽可能保证原有数据不变,只变化迁移的数据.



分散性/负载:

核心思想:由于分布式的项目的引用,导致使用redis时**不能使用全部的内存空间**.导致一些不友好的现象.

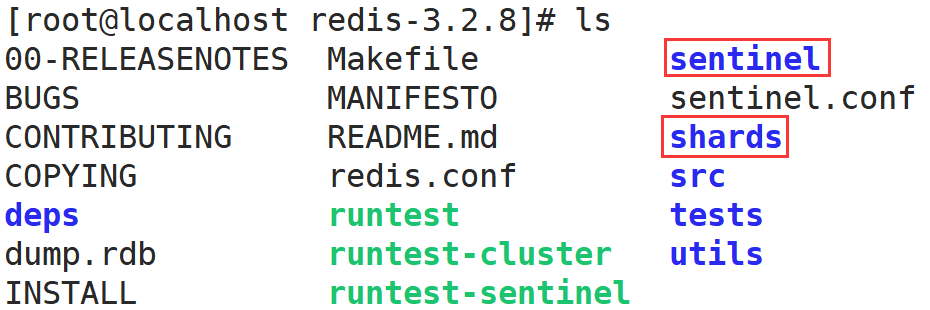
## Redis哨兵实现高可用

### 实现高可用的前提

说明:如果需要让redis实现高可用,则redis中的节点数据必须同步!!!

### 实现redis主从

1. cp -r shards sentinel



2.启动3台redis

[root@localhost sentinel]# redis-server redis-6379.conf

[root@localhost sentinel]# redis-server redis-6380.conf

[root@localhost sentinel]# redis-server redis-6381.conf

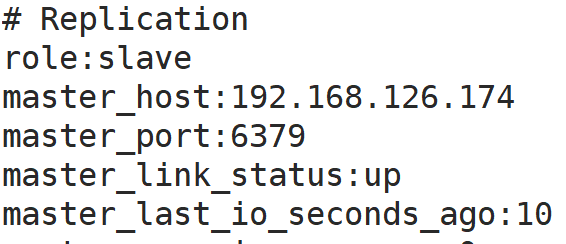
3.实现redis主从结构

命令:在从机中执行 slaveof IP 端口

127.0.0.1:6380> SLAVEOF 192.168.126.174 6379

4.检查主从的状态

命令: info replication



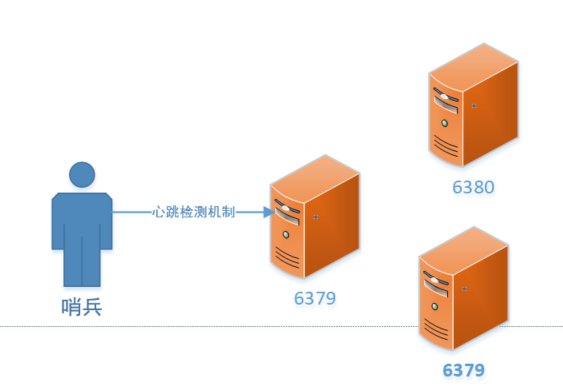
自己实现6381挂载到6379上

挂载说明:

回顾:数据库挂载时,只有从库知道主库的地址,但是主库不能获取从库的信息.

**现在redis主从挂载.主和从之间实现了相互通信!!!,是实现高可用的重要的配置.**

### 哨兵的工作原理

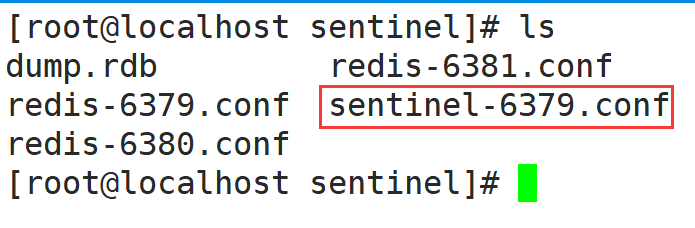


1. 通过哨兵,利用心跳检测机制监控redis主机(节点),同时记录redis主节点中的从机的信息.
2. 当哨兵发现6379宕机了,则哨兵通过内部的推选机制,选择其中一台从机代替6379继续为用户提供服务.
3. 当哨兵推选了新的主机后,其余的全部的redis节点,都会动态修改主从关系.修改**配置文件**标识当前的主机.
4. 一般注意:进行推选时,投票者一般都是单数.

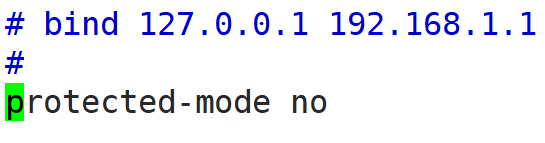
脑裂现象:一般在实现高可用时,推选的者一般都是奇数个.防止出现脑裂现象.

### Redis哨兵的实现

1. 复制哨兵的配置文件



1. 关闭保护模式

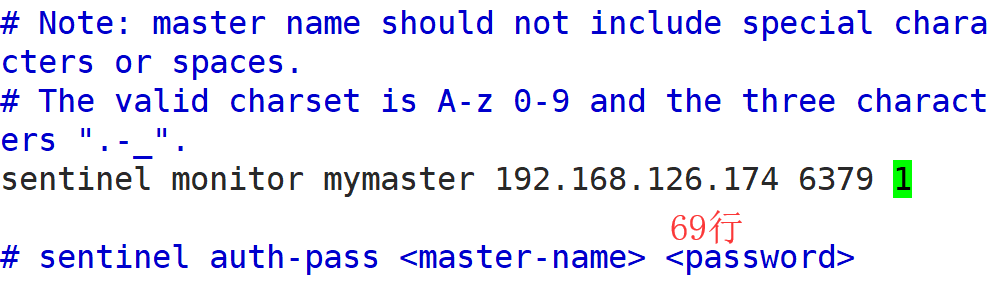


1. 修改哨兵的监听的redis主机及端口

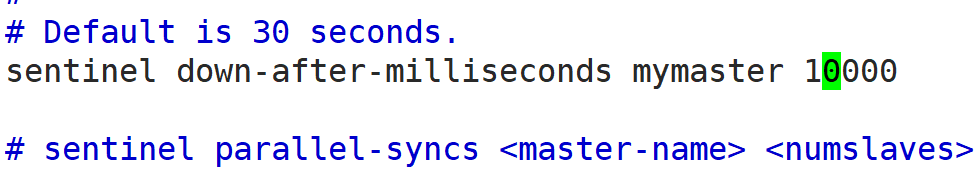
sentinel monitor **mymaster** 127.0.0.1 6379 1

mymaster:代表当前主机的IP和端口

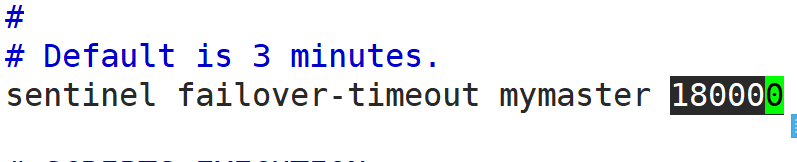
1: 哨兵的投票生效的票数.



1. 修改推选时间



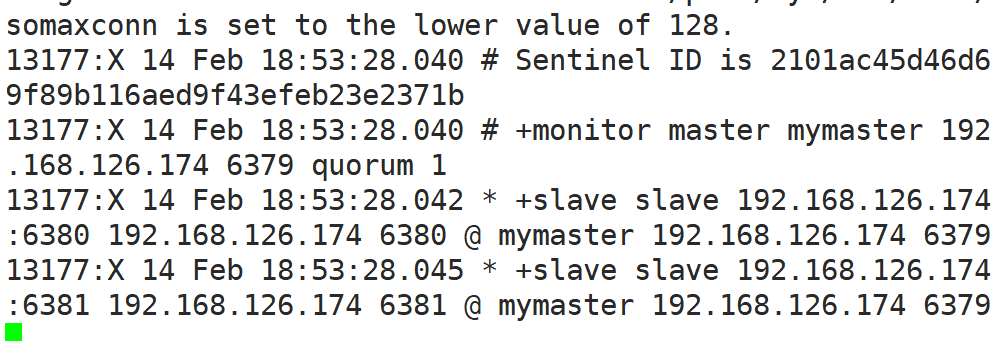
5.修改推选失败的超时时间



### Redis哨兵测试

1. 启动哨兵

redis-sentinel sentinel-6379.conf



1. 关闭redis的主机



检测redis哨兵是否正常推选.

## Spring整合哨兵

### 入门案例

@Test

**public** **void** testSentinel() {

Set<String> sentinels = **new** HashSet<>();

//sentinels.add(new HostAndPort(host, port).toString());

sentinels.add("192.168.126.174:26379");

JedisSentinelPool pool =

**new** JedisSentinelPool("mymaster", sentinels);

//获取redis链接

Jedis jedis = pool.getResource();

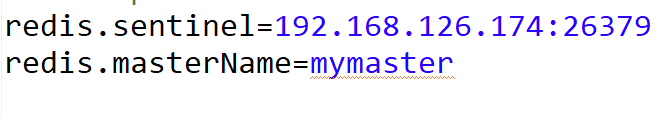
jedis.set("aa", "redis哨兵配置");

System.***out***.println(jedis.get("aa"));

jedis.close(); //将链接关闭

}

### 编辑pro文件



### 编辑spring配置文件

<!--配置redis哨兵 -->

<bean id=*"jedisSentinelPool"* class=*"redis.clients.jedis.JedisSentinelPool"*>

<constructor-arg name=*"masterName"*

value=*"${redis.masterName}"*/>

<constructor-arg name=*"sentinels"*>

<set>

<value>${redis.sentinel}</value>

</set>

</constructor-arg>

</bean>

### 编辑工具类

@Service

**public** **class** RedisService {

//表示spring容器在启动时,必须注入对象

//required=false 当使用对象时才会校验

@Autowired(required=**false**)

**private** JedisSentinelPool jedisSentinelPool;

**public** **void** set(String key,String value) {

Jedis jedis =

jedisSentinelPool.getResource();

jedis.set(key, value);

jedis.close();

}

**public** String get(String key) {

Jedis jedis =

jedisSentinelPool.getResource();

String value = jedis.get(key);

jedis.close();

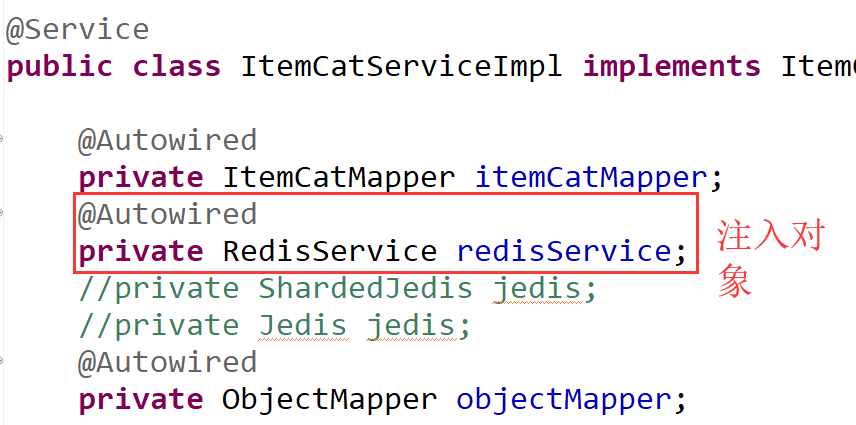
**return** value;

}

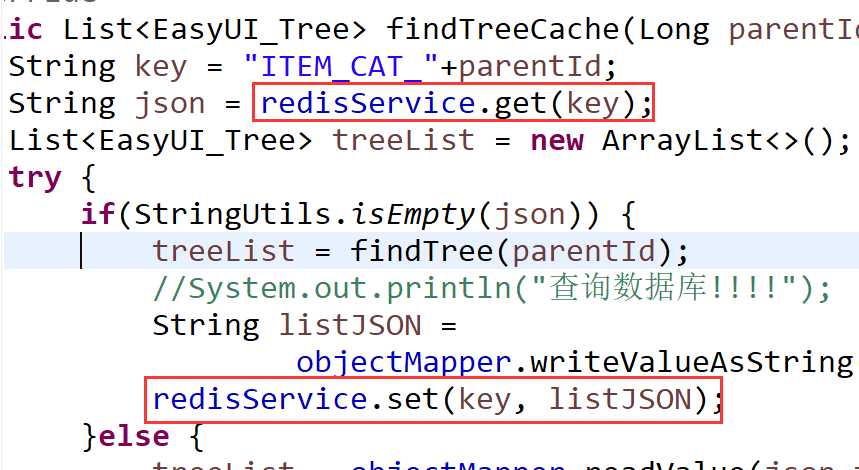
}

修改工具类后,将项目打包.

### 编辑Service实现



业务操作:



### Redis分片和哨兵说明

1. redis分片

特点:实现了内存的动态扩容.

根据hash一致性算法,实现了数据的存储.

缺点:分片中只要有一个节点发生故障,则整合分片将不能正常的使用.

1. redis哨兵

特点:实现了redis的高可用.

缺点:哨兵无法实现内存的扩容.哨兵如果宕机,则影响redis整个服务.

## Redis数据如何持久化

### Redis持久化思想

说明:redis操作时,首先操作的都是内存中的数据.redis根据自身的配置选择不同的持久化方式.定期将内存中的数据保存到本地磁盘中. 当redis重启服务时,首先会根据配置文件中指定的持久化文件恢复内存数据.

### RDB模式

特点:

1. 定期实现数据备份,但是可能丢失数据.
2. 该操作的执行的效率最高.
3. 该持久化方式是redis的默认策略.
4. RDB模式做的是**内存的快照,能够有效的节省磁盘空间,控制持久化文件的大小!!!**
5. RDB模式持久化文件是加密的

### AOF模式

特点:

1. AOF模式默认是关闭的.
2. AOF模式可以实现数据的实时备份.
3. AOF的执行的效率相对RDB模式低.
4. **AOF模式做持久化操作时,对原有的持久化文件做追加操作.**

**记录的是用户的全部执行过程!!!**

1. AOF的持久化文件,内容明文保存

### RDB模式持久化文件配置

1. 备份指令

Save: 表示立即执行持久化操作.

Bgsave:执行持久化操作在后台运行.

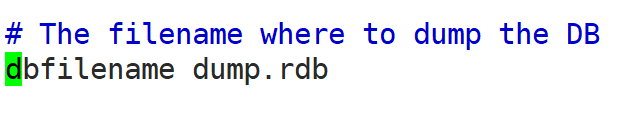
1. RDB持久化策略

save 900 1 900秒内执行了1次set操作,则持久化一次

save 300 10 300秒内执行了10次set操作,则持久化一次

save 60 10000 60秒内执行了10000次set操作,则持久化一次.

1. 需要RDB文件名称 237行

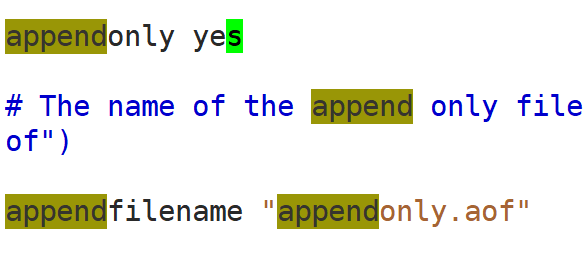


1. 持久化文件保存的路径

dir ./ 表示当前路径.

### AOF模式持久化文件配置

1. 开启AOF模式 593行



1. AOF模式的持久化策略

# appendfsync always 主要执行set操作,就持久化一次

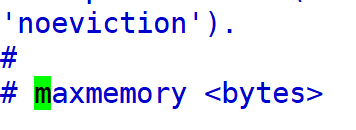
appendfsync everysec 每秒持久化一次

# appendfsync no 不做操作.

## Redis中内存策略

### 内存设定

537行,设定内存的大小.



### Redis中内存策略

1.**volatile-lru** -> 在设定了超时时间的数据中采用lru算法删除数据.

2.**allkeys-lru** -> 所有的数据采用LRU算法删除数据.

3.volatile-random ->在设定了超时时间的数据中 随机删除数据

4.allkeys-random -> 所有的数据随机删除

5.**volatile-ttl** -> 在设定了超时时间的数据中,将马上要过期的数据提前删除.

6.noeviction -> 不做操作,用户访问时,返回错误数据.

### 内存策略修改

560行,修改内存策略.修改为上述策略.

