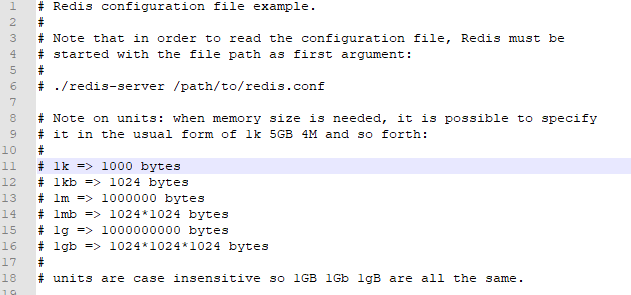
# 【熟悉】解析配置文件redis.conf

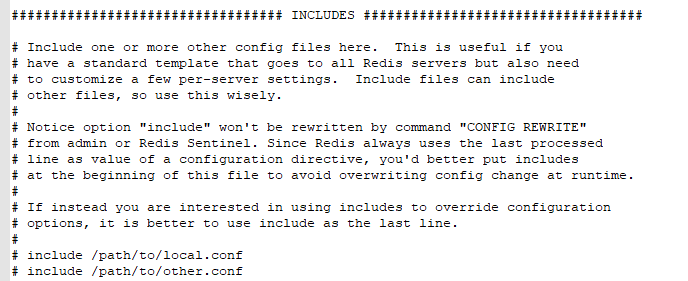
## Units单位



1 配置大小单位,开头定义了一些基本的度量单位，只支持bytes，不支持bit

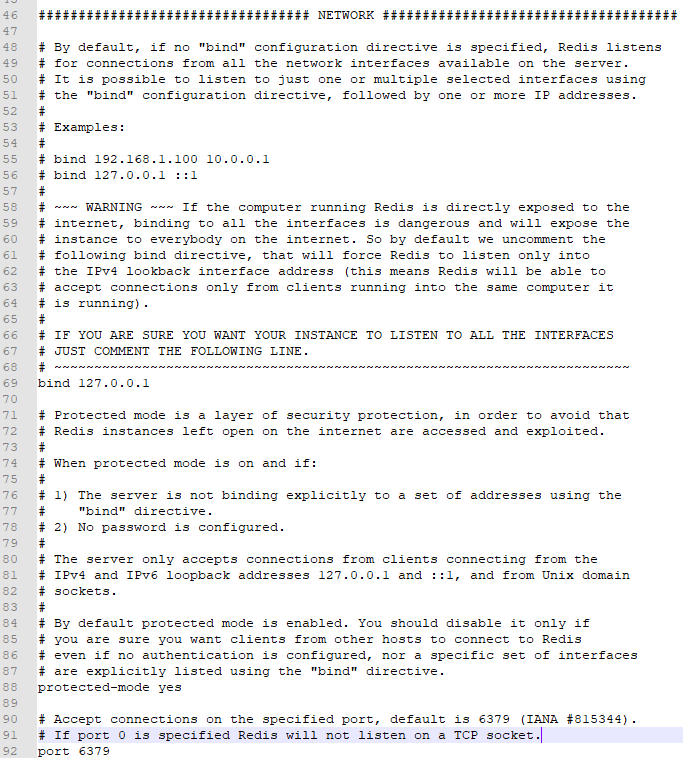
2 对大小写不敏感

## INCLUDES包含配置文件



   和spring配置文件类似，可以通过includes包含其他配置文件

## NETWORK通用



指定一个或多个"bind"配置，后面跟着一个或多个IP地址。只有这些能够连接redis

1，bind

默认情况下，redis在server上所有有效的网络接口上监听客户端连接。如果只想让它在一个或多个网络接口上监听，那你就绑定一个IP或者多个IP。多个ip空格分隔。



2，prot运行端口



3，Tcp-backlog

backlog连接队列，队列总和=未完成三次握手队列 + 已经完成三次握手队列。

在高并发环境下你需要一个高backlog值来避免慢客户端连接问题。注意Linux内核会将这个值减小到/proc/sys/net/core/somaxconn的值，所以需要确认增大somaxconn和tcp\_max\_syn\_backlog两个值

来达到想要的效果

4，timeout



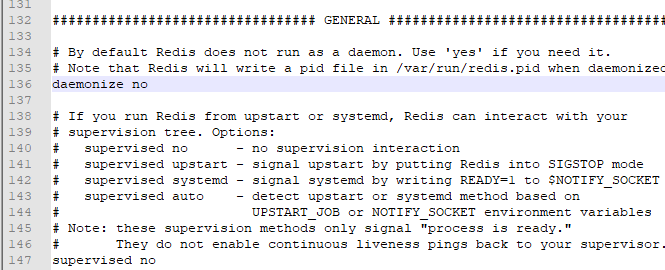
当客户端闲置多少秒后关闭连接， 0表示关闭该功能。

5，tcp-keepalive 300

单位是秒，周期性的使用SO\_KEEPALIVE检测客户端是否还处于健康状态，避免服务器一直阻塞，官方给出的建议值是300S

 建议设置成60

## GRNERAL通用



**daemonize no**是否以守护模式启动，默认为no，配置为yes时以守护模式启动，这时redis instance会将进程号pid写入默认文件/var/run/redis.pid。

**supervised no** 可以通过upstart和systemd管理Redis守护进程，这个参数是和具体的操作系统相关的。

**pidfile /var/run/redis\_6379.pid** 配置pid文件路径。当redis以守护模式启动时，如果没有配置pidfile，pidfile默认值是/var/run/redis.pid 。

**loglevel notice** 日志级别。可选项有：debug（记录大量日志信息，适用于开发、测试）； verbose（较多日志信息）； notice（适量日志信息，使用于生产环境）；warning（仅有部分重要、关键信息才会被记录）。

**logfile ""**日志文件的位置，当指定为空字符串时，为标准输出，如果redis已守护进程模式运行，那么日志将会输出到 /dev/null 。

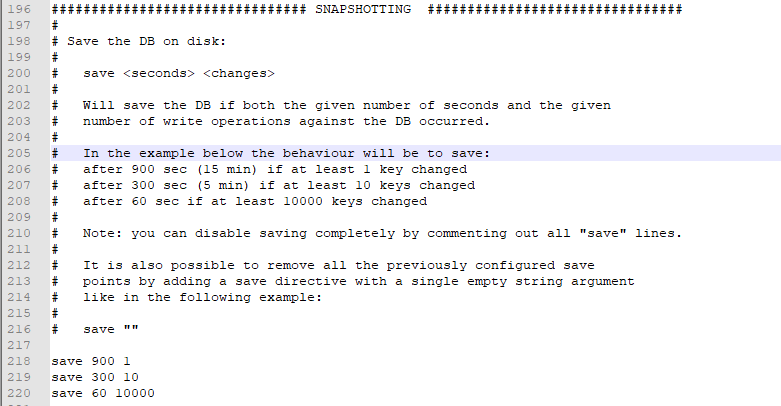
**syslog-enabled no**  是否把日志记录到系统日志。

**syslog-ident**  设置系统日志的id     如  syslog-ident redis

**databases 16**  设置数据库的数目。默认的数据库是DB 0 ，可以在每个连接上使用select <dbid> 命令选择一个不同的数据库，dbid是一个介于0到databases - 1 之间的数值。

**always-show-logo yes** 是否一直显示日志

## SNAPSHOTTING快照



save <seconds> <changes> second秒内,changes个keys发生改变则保存一次。

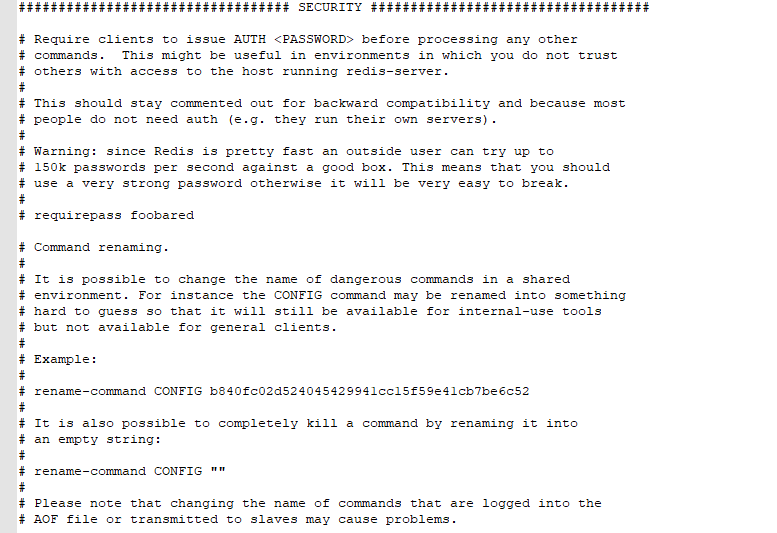
**stop-writes-on-bgsave-error**yes  默认情况下，如果 redis 最后一次的后台保存失败，redis将停止接受写操作，以强硬的方式让用户知道数据不能正确持久化。如果后台保存进程重新启动工作了,redis也将自动的允许写操作。然而你要是安装了靠谱的监控，你可能不希望 redis 这样做，那你就改成 no 好了。

**rdbcompression**yes  是否在dump .rdb数据库的时候压缩字符串，默认设置为yes。如果你想节约一些cpu资源的话，可以把它设置为no，这样的话数据集就可能会比较大。

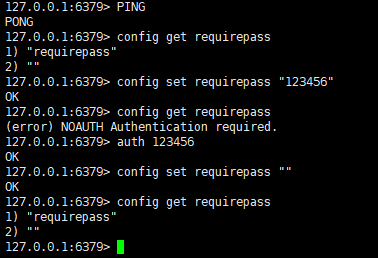
**rdbchecksum**yes  是否CRC64校验rdb文件，会有一定的性能损失（大概10%）

**dbfilename** dump.rdb   rdb文件的名字。

## SECURITY安全



主要用于访问密码和查看，设置和取消



**LIMITS限制**

**maxclients**10000

redis同时可以与10000个客户端进行连接。

**maxmemory**

到达内存上限，redis会试图移除内部数据，移除规则可以通过maxmemory-policy来指定。如果redis无法根据移除规则来移除内存中的数据，或者设置“不允许移除”，那么redis则会针对那些需要申请内存的指令返回错误信息，比如SET、LPUSH等

 但是对于无内存申请的指令，仍然会正常响应，如GET。如果你的redis是主redis（说明你的redis有从redis），那么在设置内存使用上限时，需要在系统中留出一些内存空间给同步队列缓存，只有在你设置的是“不移除”的情况下，才不用考虑这个因素

**maxmemory-policy**

（1）volatile-lru：LRU 过期

（2）allkeys-lru： LRU 最近最少原则

（5）volatile-ttl：TTL值最小,即最近要过期

一般不用

（3）volatile-random：过期移除随机key

（4）allkeys-random：移除随机的key

（6）noeviction：不进行移除。针对写操作，只返回错误信息 默认,需修改

**maxmemory-samples**

设置样本数量，LRU算法和最小TTL算法都并非是精确的算法，而是估算值，所以你可以设置样本的大小，

redis默认会检查这么多个key并选择其中LRU的那个

bin目录的文件说明

　　redis-benchmark：redis性能测试工具

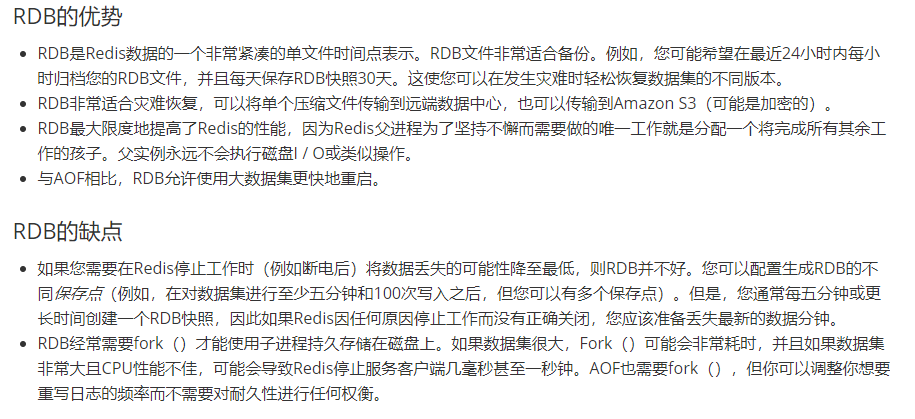
　　redis-check-aof：检查aof日志的工具

　　redis-check-dump：检查rdb日志的工具

　　redis-cli：连接用的客户端

　　redis-server：redis服务进程

## RDB【Redis DataBase】

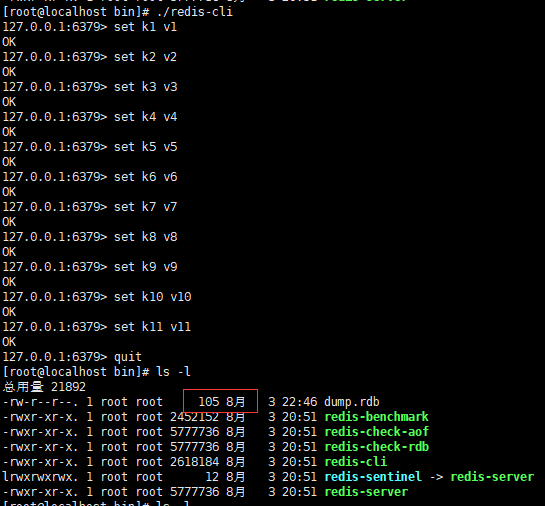


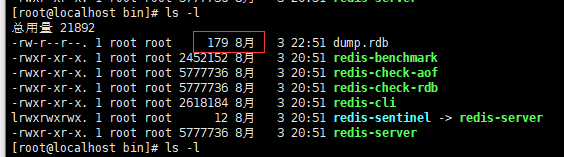
rdb在进行恢复时是将Snapshot快照文件直接读到内存里

Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能.如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB要比AOF更加高效。RDB的缺点是最后一次持久化后的数据可能丢失。

Fork的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程

### 如何触发RDB快照



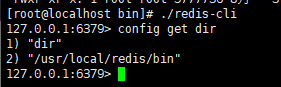


执行flushall命令，也会产生dump.rdb文件，但里面是空的，无意义

### 恢复数据

将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可

CONFIG GET dir获取目录



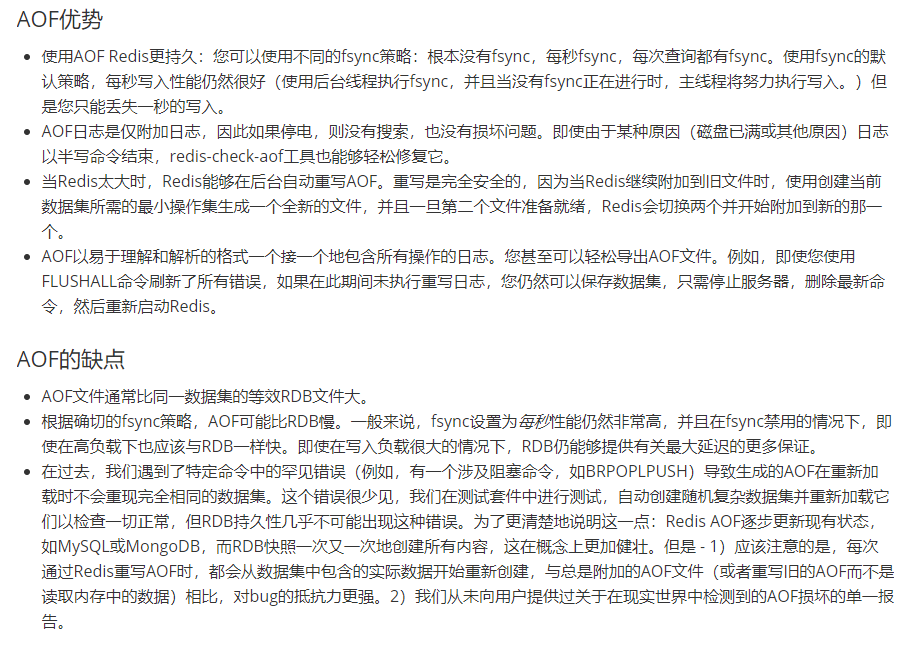
### 优缺点

适合大规模的数据恢复,但是不保证数据完整性和一致性

备份时redis意外down掉的话，就会丢失最后一次快照后的所有修改

Fork的时候，内存中的数据被克隆了一份，大致2倍的膨胀性需要考虑

## AOF



以日志的形式来记录每个写操作，将Redis执行过的所有写指令记录下来，只许追加文件但不可以改写文件，redis启动之初会读取该文件重新构建数据，换言之，redis重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作

### AOF启动/修复/恢复

**正常恢复**

  启动：设置Yes   修改默认的appendonly no，改为yes

  将有数据的aof文件复制一份保存到对应目录(config get dir)

  恢复：重启redis然后重新加载

**异常恢复**

  启动：设置Yes   修改默认的appendonly no，改为yes

  备份被写坏的AOF文件

  修复：  Redis-check-aof --fix进行修复

  恢复：重启redis然后重新加载

### 优缺点

  每修改同步：appendfsync always 同步持久化 数据变更会被立即记录到磁盘,性能较差但数据完整性比较好

  每秒同步：appendfsync everysec 异步操作，每秒记录 如果一秒内宕机，有数据丢失

  不同步：appendfsync no 从不同步

  相同数据集的数据而言aof文件要远大于rdb文件，恢复速度慢于rdb

  Aof运行效率要慢于rdb,每秒同步策略效率较好，不同步效率和rdb相同

总结

RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储

AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF命令以redis协议追加保存每次写的操作到文件末尾. Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大

**只做缓存**：如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.

**同时开启两种持久化方式**

  在这种情况下,当redis重启的时候会优先载入AOF文件来恢复原始的数据, 因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整.

  RDB的数据不实时，同时使用两者时服务器重启也只会找AOF文件。那要不要只使用AOF呢？ 作者建议不要，因为RDB更适合用于备份数据库(AOF在不断变化不好备份)， 快速重启，而且不会有AOF可能潜在的bug，留着作为一个万一的手段。

**性能建议**

因为RDB文件只用作后备用途，建议只在Slave上持久化RDB文件，而且只要15分钟备份一次就够了，只保留save 900 1这条规则。

如果Enalbe AOF，好处是在最恶劣情况下也只会丢失不超过两秒数据，启动脚本较简单只load自己的AOF文件就可以了。代价一是带来了持续的IO，二是AOF rewrite的最后将rewrite过程中产生的新数据写到新文件造成的阻塞几乎是不可避免的。只要硬盘许可，应该尽量减少AOF rewrite的频率，AOF重写的基础大小默认值64M太小了，可以设到5G以上。默认超过原大小100%大小时重写可以改到适当的数值。

如果不Enable AOF ，仅靠Master-Slave Replication 实现高可用性也可以。能省掉一大笔IO也减少了rewrite时带来的系统波动。代价是如果Master/Slave同时down，会丢失十几分钟的数据，启动脚本也要比较两个Master/Slave中的RDB文件，载入较新的那个。新浪微博就选用了这种架构

# 【了解】Redis事务

Redis的单线程决定了它并不需要事务

事务可以一次执行多个命令，本质是一组命令的集合。一个事务中的所有命令都会序列化，按顺序地串行化执行而不会被其它命令插入，不许加塞

<https://redis.io/topics/transactions>

  一个队列中，一次性、顺序性、排它性的执行一系列命令

常用命令

Discard 取消事务,放弃事务块中存储的所有指令

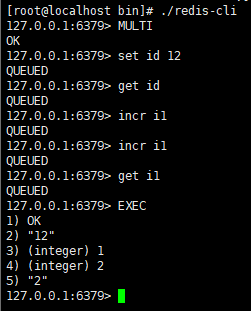
Exec 执行事务块中所有指令

Multi 标记事务块的开始

Watch key 监视若干个key,如果在事务前key发生修改,打断事务

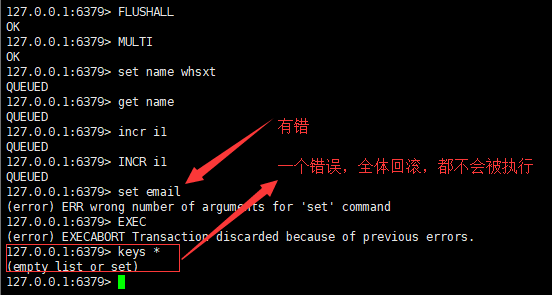
Unwatch 取消对key的监视

情况1：正常执行



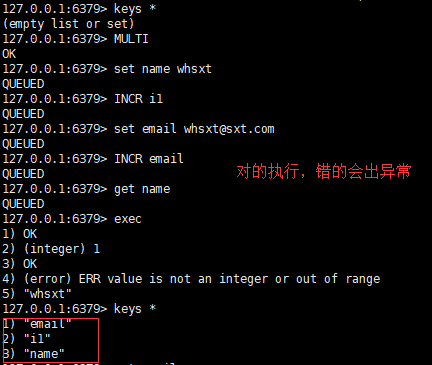
情况2：全体连坐

输入事务时直接提示的错误(语法错误),将导致整个事务块回滚

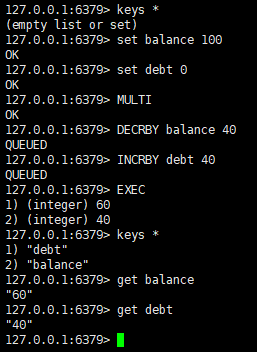


情况3：冤头债主

运行时错误,将导致正确语句执行,错误语句跳过

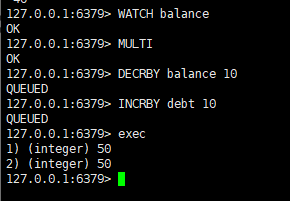


**3.6.2，初始化信用卡可用余额和欠额**



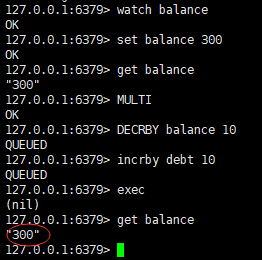
**3.6.2，无加塞篡改**

先监控再开启multi， 保证两笔金额变动在同一个事务内

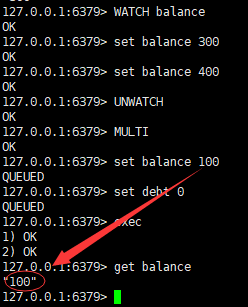


**3.6.2，有加塞篡改**

监控了key，如果key被修改了，后面一个事务的执行失效



**3.6.3，unwatch**



    一旦执行了exec之前加的监控锁都会被取消掉了

**3.6.4，小结**

      Watch指令类似乐观锁，事务提交时，如果Key的值已被别的客户端改变，比如某个list已被别的客户端push/pop过了,这个key的版本号就高于提交的版本号,被乐观锁锁住，整个事务队列都不会被执行

      通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化， EXEC命令执行的事务都将被放弃，同时返回Nullmulti-bulk应答以通知调用者事务执行失败

3大特性

**单独的隔离操作：**事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

**没有隔离级别的概念：**队列中的命令没有提交之前都不会实际的被执行，因为事务提交前任何指令都不会被实际执行， 也就不存在”事务内的查要看到事务里的更新，在事务外查询不能看到”这个让人万分头痛的问题

**不保证原子性：**redis同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

# Redis的复制(Master/Slave)

**主从复制**，主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制，Master以写为主，Slave以读为主

实现了读写分离/容灾恢复

**一主二仆(推荐)**

SLAVEOF 主库IP 主库端口

每次与master断开之后，都需要重新连接，除非你配置进redis.conf文件

Info replication查看当前的主从关系

①拷贝多个redis.conf文件,文件名加上端口号区分

②开启daemonize yes

③修改从机端口

④修改从机Log文件名(可与主机同一文件)

⑤修从机Dump.rdb文件名

1 slave1、slave2是从头开始复制还是从切入点开始复制? 从头复制

2 从机是否可以写？ 主机可写,从机只读

3 主机shutdown后情况如何？从机是上位还是原地待命 原地待命

4 主机又回来了后，主机新增记录，从机还能否顺利复制？ 能

5 其中一台从机down后情况如何？依照原有它能跟上大部队吗？ 能

薪火相传

上一个Slave可以是下一个slave的Master，Slave同样可以接收其他 slaves的连接和同步请求，那么该slave作为了链条中下一个的master, 可以有效减轻master的写压力

中途变更转向:会清除之前的数据，重新建立拷贝最新的

翻身做主人

SLAVEOF no one

  使当前数据库停止与其他数据库的同步，转成主数据库(并不是把之前的master取代了)

## 复制的原理

Slave启动成功连接到master后会发送一个sync命令

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， 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

全量复制：第一次链接成功,全部都复制给从机

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

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

# 哨兵模式

上述的主从配置中,主机down会导致从机原地待命,这时候无法进行写操作,不能满足需求.合理的设计是主机down马上会有从机顶替主机,使得写操作不会中断

    哨兵模式能够后台监控主机是否故障，如果故障了根据投票数自动将从库转换为主库

配置哨兵

    /myredis目录下新建sentinel.conf文件

     sentinel monitor 自定义主机名 127.0.0.1 6379 1 数字1表示主机挂掉后salve投票,得票1就能当主机.数字设置的越大,投票消耗的时间越长

sentinel monitor port6379 127.0.0.1 6379 1

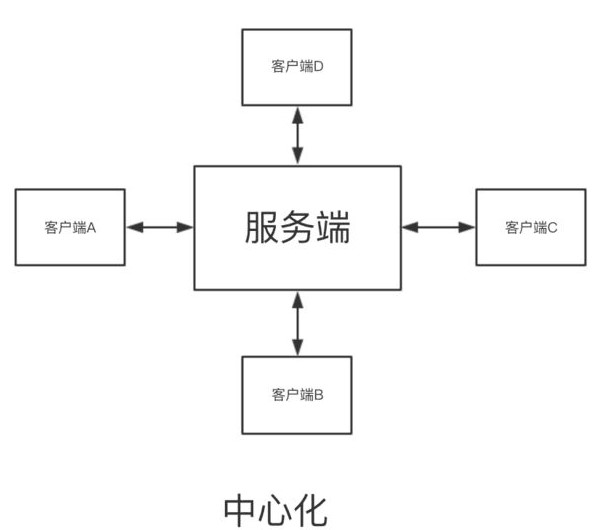
启动哨兵

/usr/local/redis/bin/redis-sentinel /root/myredis/sentinel.conf

同一哨兵监视多个主机,只需要在配置文件另起一行

哨兵配置集群:同一份配置文件复制多分,启动

# 集群配置

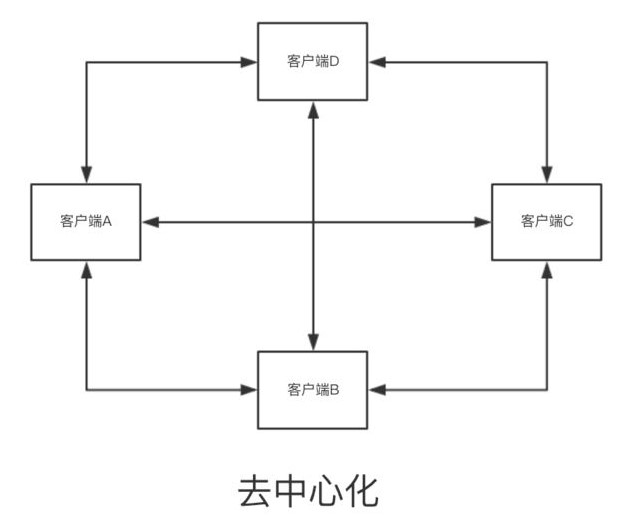


中心化:所有的节点都要有一个主节点

缺点：中心挂了，服务就挂了.中心处理数据的能力有限，不能把节点性能发挥到最大

特点：就是一个路由作用

在之前的配置中,只有主机负责写入,从机负责读取,很容易造成性能的瓶颈



去中心化:去掉路由，我自己来路由

## 集群搭建(去中心化)

|  |  |  |
| --- | --- | --- |
| 机器编号 | ip | port |
| 1 | 192.168.186.128 | 7002 |
| 2 | 192.168.186.128 | 7003 |
| 3 | 192.168.186.128 | 7004 |
| 4 | 192.168.186.128 | 7005 |
| 5 | 192.168.186.128 | 7006 |
| 6 | 192.168.186.128 | 7007 |

Redis集群中内置了16384个哈希槽

在进行set时,会先对key进行crc16算法算出应该插入到哪个槽中.

在进行get时,也会进行crc16算法算出key在哪个插槽中

可以将哈希槽理解为分表

### 搭建过程

新建redis集群文件夹

mkdir redis-cluster

复制一个server

cp /usr/local/redis/bin/redis-server ./

配置第一个redis

mkdir redis1

复制一个配置文件

cp /usr/local/src/redis/redis.conf redis1

修改配置

vim redis1/redis.conf

bind 0.0.0.0 69行

port 7001 92行

daemonize yes 136行

# 打开aof 持久化

appendonly yes 672行

# 开启集群

cluster-enabled yes 814行

# 集群的配置文件,该文件自动生成

cluster-config-file nodes-7001.conf 822行

# 集群的超时时间

cluster-node-timeout 15000 828行

将这个redis1复制6分,每份替换端口号

%s/7000/7001/g

安装完成后同时启动所有的redis

#### 使用docker 下载redis-trib的镜像运行

安装Docker

yum install docker

启动docker

systemctl start docker

A: 下载镜像

docker pull inem0o/redis-trib

# 集群启动

docker run -it --net host inem0o/redis-trib create --replicas 1 192.168.186.128:7002 192.168.186.128:7003 192.168.186.128:7004 192.168.186.128:7005 192.168.186.128:7006 192.168.186.128:7007

docker run -it --net host inem0o/redis-trib create --replicas 1 127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 127.0.0.1:7006 127.0.0.1:7007

-it是为了可以输入

--net host 是为了上docker容器能连接上本地的宿主机

集群后客户端连接需要加上 -c

# 【掌握】java连接redis

## Jedis所需要的jar包依赖

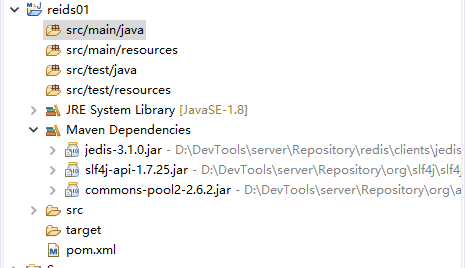
<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>3.1.0</version>

</dependency>



### 测试连通性

public class TestRedisPing {

public static void main(String[] args) {

//创建连接

Jedis jedis=new Jedis("192.168.186.128", 6379);

//设置密码 如果没有密码可以不设置

jedis.auth("123456");

//调用 ping方法

String ping = jedis.ping();

//输出PONG

System.out.println(ping);

jedis.close();

}

}

### 主从复制

public class TestReidsMS {

public static void main(String[] args) throws InterruptedException {

// 创建连接

Jedis jedis\_M = new Jedis("192.168.186.128", 6379);

Jedis jedis\_S = new Jedis("192.168.186.128", 6380);

jedis\_S.slaveof("127.0.0.1", 6379);

jedis\_M.set("k6", "v6");

Thread.sleep(500);

System.out.println(jedis\_S.get("k6"));

jedis\_M.close();

jedis\_S.close();

}

}

## JedisPool

  1，获取Jedis实例需要从JedisPool中获取

  2，用完Jedis实例需要返还给JedisPool

  3，如果Jedis在使用过程中出错，则也需要还给JedisPool

### 案例见代码

    JedisPoolUtil

public class JedisPoolUtil {

private static volatile JedisPool jedisPool = null;// 被volatile修饰的变量不会被本地线程缓存，对该变量的读写都是直接操作共享内存。

private JedisPoolUtil() {

}

public static JedisPool getJedisPoolInstance() {

if (null == jedisPool) {

synchronized (JedisPoolUtil.class) {

if (null == jedisPool) {

JedisPoolConfig poolConfig = new JedisPoolConfig();

poolConfig.setMaxTotal(1000);

poolConfig.setMaxIdle(32);

poolConfig.setMaxWaitMillis(100 \* 1000);

poolConfig.setTestOnBorrow(true);

jedisPool = new JedisPool(poolConfig, "192.168.186.128", 6379);

}

}

}

return jedisPool;

}

public static void release(JedisPool jedisPool, Jedis jedis) {

if (null != jedis) {

Jedis jedis2 = null;

try {

jedis2 = jedisPool.getResource();

} finally {

jedis2.close();

}

}

}

​

}

使用

public class TestJedisPool {

public static void main(String[] args) {

JedisPool jedisPool = JedisPoolUtil.getJedisPoolInstance();

Jedis jedis = null;

​

try {

jedis = jedisPool.getResource();

jedis.set("k18", "v183");

​

} catch (Exception e) {

e.printStackTrace();

} finally {

JedisPoolUtil.release(jedisPool, jedis);

}

}

​

}

​

## 配置总结all

JedisPool的配置参数大部分是由JedisPoolConfig的对应项来赋值的。

maxTotal/maxActive：控制一个pool可分配多少个jedis实例，通过pool.getResource()来获取；如果赋值为-1，则表示不限制；如果pool已经分配了maxActive个jedis实例，则此时pool的状态为exhausted。

maxIdle：控制一个pool最多有多少个状态为idle(空闲)的jedis实例；

whenExhaustedAction：表示当pool中的jedis实例都被allocated完时，pool要采取的操作；默认有三种。

WHEN\_EXHAUSTED\_FAIL --> 表示无jedis实例时，直接抛出NoSuchElementException；

WHEN\_EXHAUSTED\_BLOCK --> 则表示阻塞住，或者达到maxWait时抛出JedisConnectionException；

WHEN\_EXHAUSTED\_GROW --> 则表示新建一个jedis实例，也就说设置的maxActive无用；

setMaxWaitMillis/maxWait：表示当borrow一个jedis实例时，最大的等待时间，如果超过等待时间，则直接抛JedisConnectionException；

testOnBorrow：获得一个jedis实例的时候是否检查连接可用性（ping()）；如果为true，则得到的jedis实例均是可用的；

testOnReturn：return 一个jedis实例给pool时，是否检查连接可用性（ping()）；

testWhileIdle：如果为true，表示有一个idle object evitor线程对idle object进行扫描，如果validate失败，此object会被从pool中drop掉；这一项只有在timeBetweenEvictionRunsMillis大于0时才有意义；

timeBetweenEvictionRunsMillis：表示idle object evitor两次扫描之间要sleep的毫秒数；

numTestsPerEvictionRun：表示idle object evitor每次扫描的最多的对象数；

minEvictableIdleTimeMillis：表示一个对象至少停留在idle状态的最短时间，然后才能被idle object evitor扫描并驱逐；这一项只有在timeBetweenEvictionRunsMillis大于0时才有意义；

softMinEvictableIdleTimeMillis：在minEvictableIdleTimeMillis基础上，加入了至少minIdle个对象已经在pool里面了。如果为-1，evicted不会根据idle time驱逐任何对象。如果minEvictableIdleTimeMillis>0，则此项设置无意义，且只有在timeBetweenEvictionRunsMillis大于0时才有意义；

lifo：borrowObject返回对象时，是采用DEFAULT\_LIFO（last in first out，即类似cache的最频繁使用队列），如果为False，则表示FIFO队列；

==================================================================================================================

其中JedisPoolConfig对一些参数的默认设置如下：

testWhileIdle=true

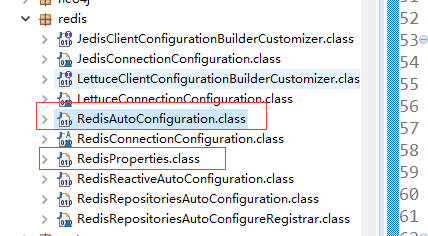
minEvictableIdleTimeMills=60000

timeBetweenEvictionRunsMillis=30000

numTestsPerEvictionRun=-1

# 【掌握】springboot中使用redis

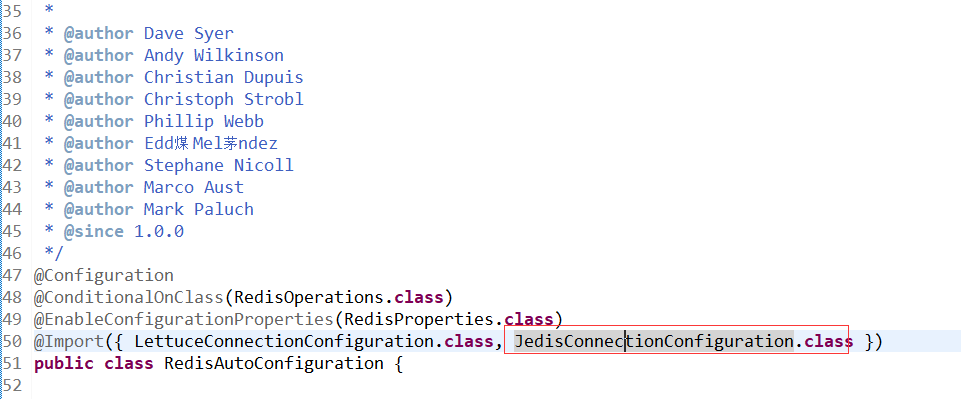
## spring boot 如何加载默认的对象

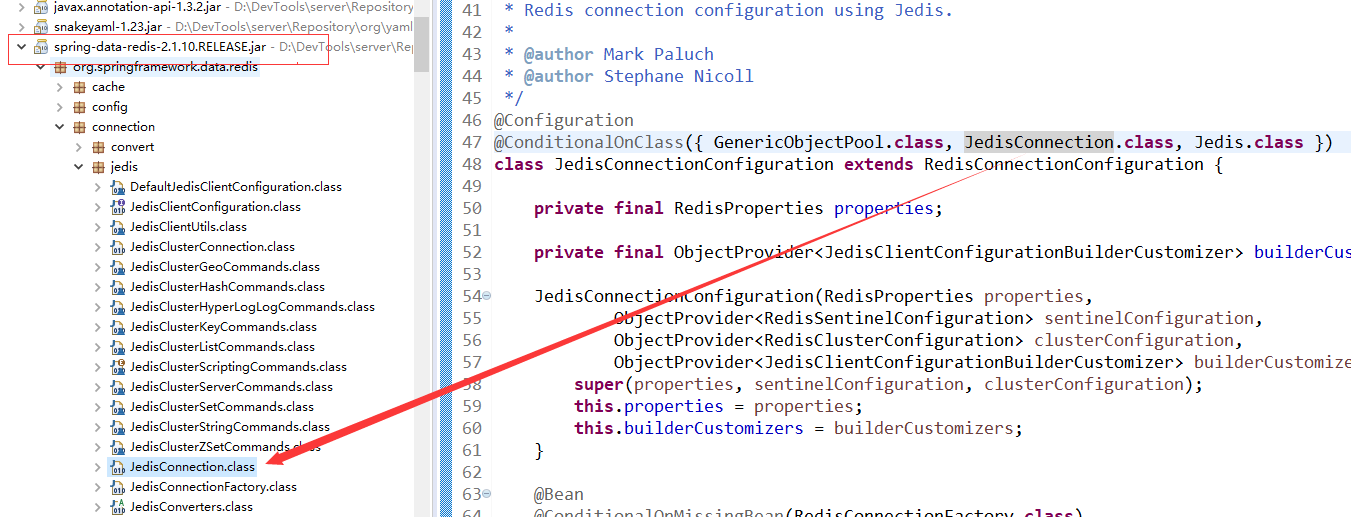


RedisAutoCongiguration创建对象

RedisProperties读取配置文件

为什么会配置成功呢？





可以看出是由springdata提供的访问对象

## 修改配置文件yml

#redis的配置

spring:

redis:

host: 192.168.120.130

port: 6379

password: 123456

jedis:

pool:

max-idle: 20

max-active: 25

min-idle: 10

## StringRedisTemplate操作Redis

extends RedisTemplate<String,String>

|  |
| --- |
| @RunWith(SpringRunner.class)  @SpringBootTest  public class SpringbootRedisApplicationTests {  @Autowired  private StringRedisTemplate redisTemplate;  /\*\*  \* redis数据类型为String的操作  \*/  @Test  public void testString() {  // 操作String类型  ValueOperations<String, String> opsValue = redisTemplate.opsForValue();  // 给redis 里面set 一个key  opsValue.set("boot", "spring-boot"); // k -v 都是String  // 从redis 里面获取key  String value = opsValue.get("boot");  System.out.println(value);  // 从redis 里面或多个key  List<String> asList = Arrays.asList("boot", "alll-menu-data");  List<String> mulitValues = opsValue.multiGet(asList);  System.out.println(mulitValues);  // redis的自动增长  Long increment = opsValue.increment("boot-incr", 2);// delta 可以+ 任意的数（步长）  System.out.println(increment);  }  @Test  public void testHash() {  HashOperations<String, Object, Object> opsForHash = redisTemplate.opsForHash();  // hset  opsForHash.put("object-1", "name", "sxt"); // 后面的2 个参数都是object,但是只支持String 类型  opsForHash.put("object-1", "age", "27"); // 后面的2 个参数都是object,但是只支持String 类型  opsForHash.put("object-1", "sex", "man"); // 后面的2 个参数都是object,但是只支持String 类型  Object value = opsForHash.get("object-1", "sex");  System.out.println(value);  // 取多个值  List<Object> multiGet = opsForHash.multiGet("object-1", Arrays.asList("name", "sex"));  System.out.println(multiGet);  }  @Test  public void testZset() {  ZSetOperations<String, String> opsForZSet = redisTemplate.opsForZSet();  // 放到zset集合里面  opsForZSet.add("lol", "sxt", 2500);  opsForZSet.add("lol", "lz", 0);  opsForZSet.add("lol", "ln", 1400);  opsForZSet.add("lol", "ll", -10);  opsForZSet.add("lol", "lt", 2700);  Set<String> rangeAsc = opsForZSet.range("lol", 0, 2); // 通过排序取值 ll lz ln  System.out.println(rangeAsc);  Set<String> reverseRange = opsForZSet.reverseRange("lol", 0, 2);// lt lz ln  System.out.println(reverseRange);  Set<TypedTuple<String>> tuples = new HashSet<ZSetOperations.TypedTuple<String>>();  tuples.add(new DefaultTypedTuple<String>("sxt", 1000.00));  tuples.add(new DefaultTypedTuple<String>("lv", 1200.00));  tuples.add(new DefaultTypedTuple<String>("lz", 2900.00));  tuples.add(new DefaultTypedTuple<String>("lt", 100.00));  // 若redis 存在该key ，则需要数据类型相同，不然报错  opsForZSet.add("dnf", tuples);  }  } |

## RedisTemplate<Object,Object>（扩展String类型）操作Redis

若不设置序列化规则，它将使用JDK自动的序列化将对象转换为字节，存到Redis 里面

它可以存在对象到redis里面

如果对象没有序列化，那么默认使用的JDK的序列化方式



@RunWith(SpringRunner.class)

@SpringBootTest

public class SpringbootRedisTemplateTests {

@Autowired

private RedisTemplate<Object, Object> redisTemplate ; // 因为创建RedisTemplate 没有使用泛型信息来创建，泛型 本质还是Object，只不过泛型能自动推断并强转

@Test

public void testString() {

redisTemplate.setKeySerializer(new StringRedisSerializer()); // key的序列化使用String 类型来完成 因为key 很多时候都是一个字符串

redisTemplate.setValueSerializer(new GenericJackson2JsonRedisSerializer()); // 优先没有泛型的，因为有泛型后，它支持的数据类型就定了

ValueOperations<Object, Object> valueOperations = redisTemplate.opsForValue();

// valueOperations.set("boot-redis", "boot-value"); //对象->字符串 json

User user = new User(1, "laolei", "xx.jpg", "78414842@qq.com");

// KEY : com.sxt.domain.User:1

// com.fasterxml.jackson.databind.JsonSerializer 没有依赖jackson 之前大家可能使用spring-boot-web，这里面会自动依赖

valueOperations.set(User.class.getName()+":"+user.getId(), user);

// 若该对象的强转转换，则redis 内部会使用JackSon 的工具将字符串-> 转换为java 对象 ，那jackson 转换为对象时，需要一个对象的类型 ，其实它已经自动对象的类型了"@class": "com.sxt.domain.User",

User object = (User)valueOperations.get(User.class.getName()+":"+user.getId());

System.out.println(object.getName()+":"+object.getIcon());

}

/\*\*

\* hash

\*/

@Test

public void testHash() {

redisTemplate.setKeySerializer(new StringRedisSerializer()); // key的序列化使用String 类型来完成 因为key 很多时候都是一个字符串

redisTemplate.setHashKeySerializer(new StringRedisSerializer());

redisTemplate.setHashValueSerializer(new StringRedisSerializer()); // 若都是string 则和StringRedisTempalte一样了

HashOperations<Object, Object, Object> opsForHash = redisTemplate.opsForHash();

opsForHash.put("redis-hash", "prop1", "value");

}

}

## 集群的额外操作

@Test

public void testCluster() {

ClusterOperations<Object, Object> opsForCluster = redisTemplate.opsForCluster();

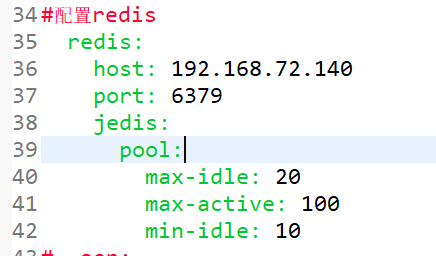
//关闭集群的7000端口的主机

opsForCluster.shutdown(new RedisClusterNode("192.168.120.130", 7000));

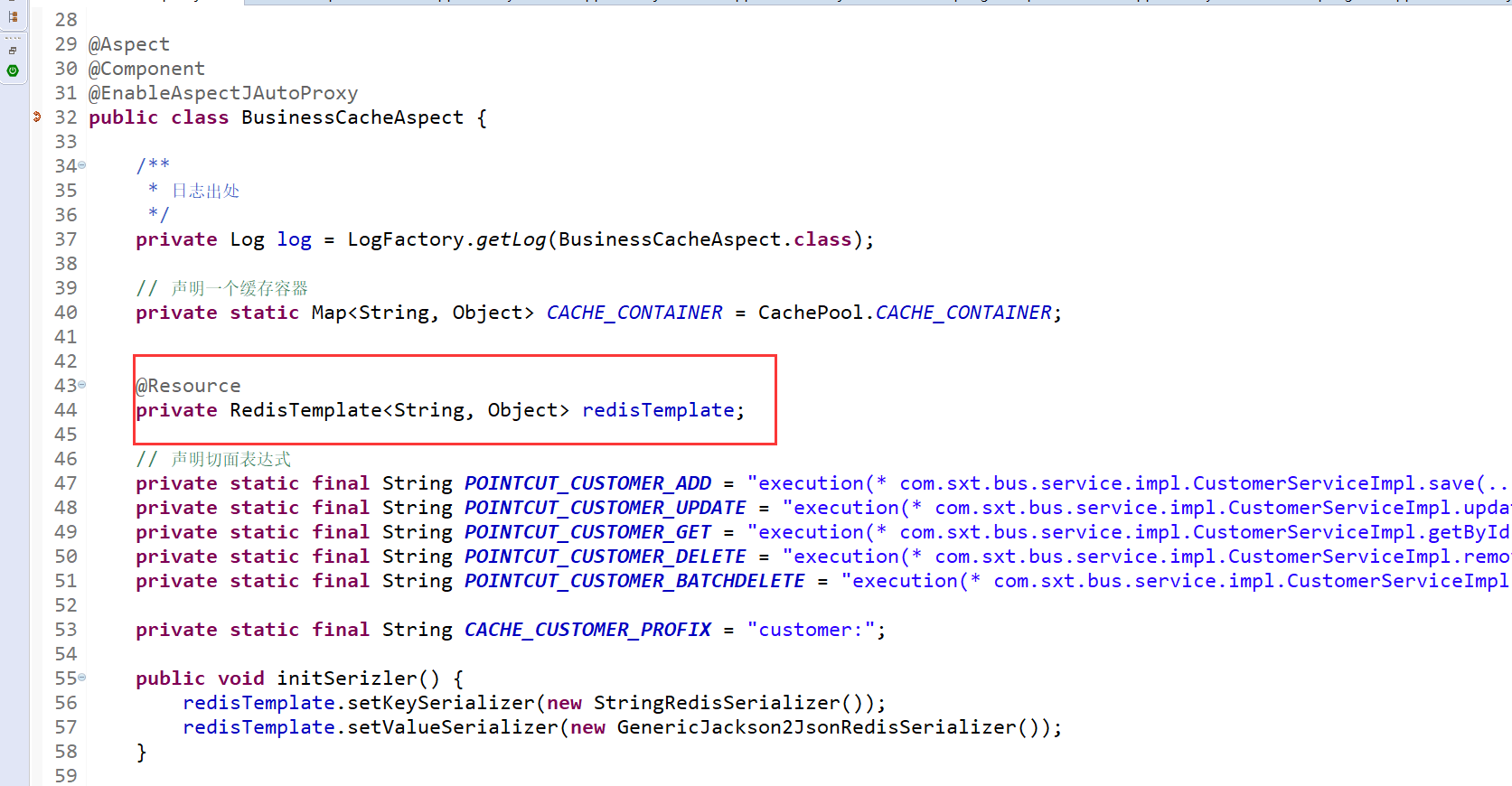
}

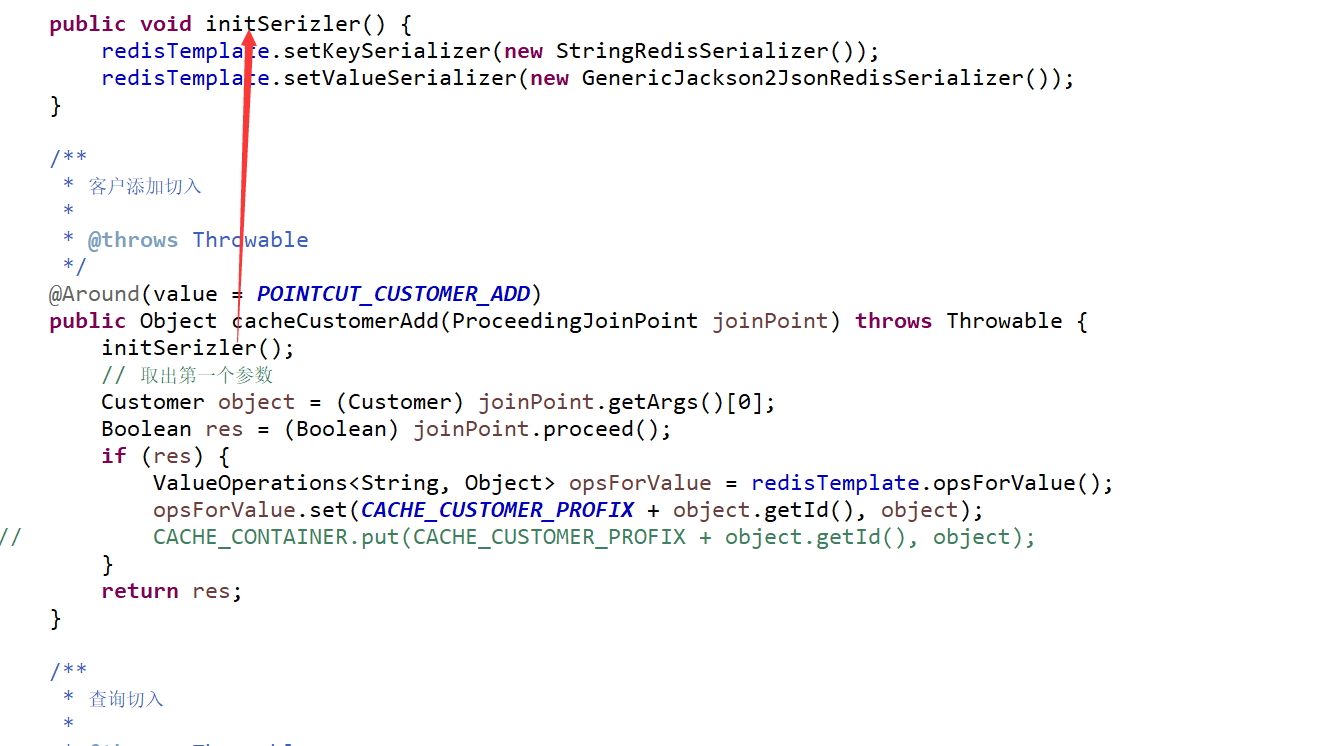
# 【掌握】springboot使用redis做缓存

### 配置yml



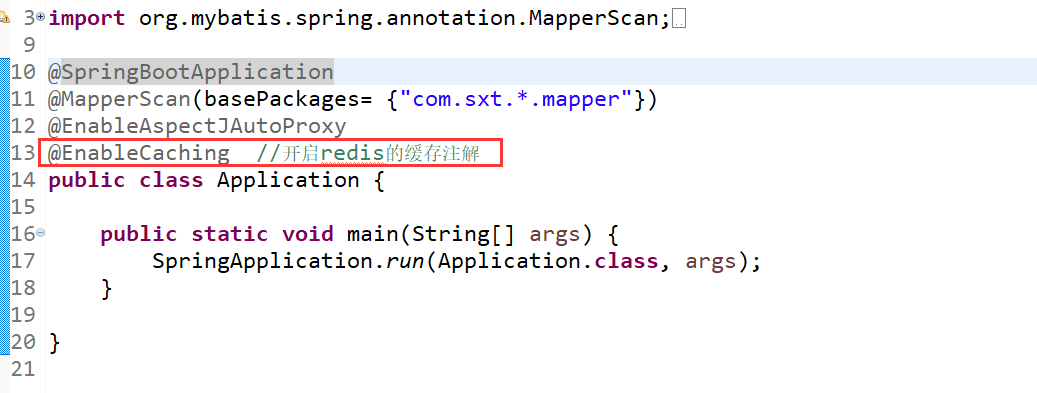
### 修改切面类



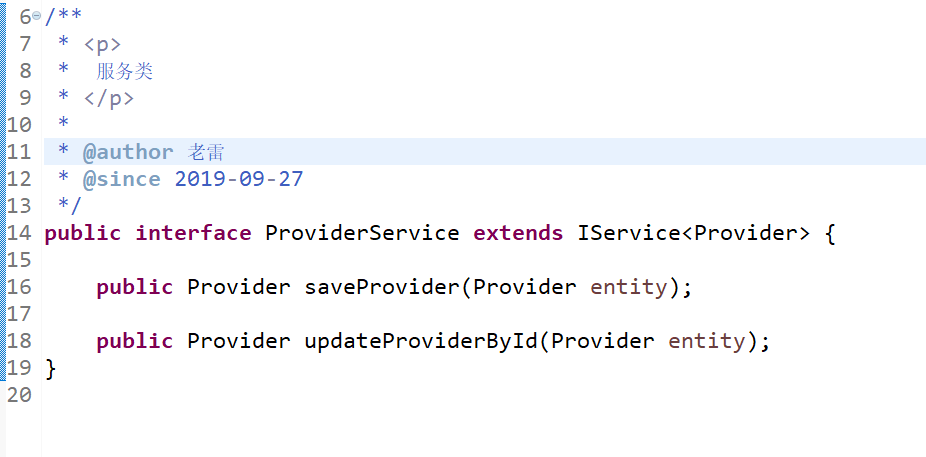


## 使用spring提供的注解方式来操作redis

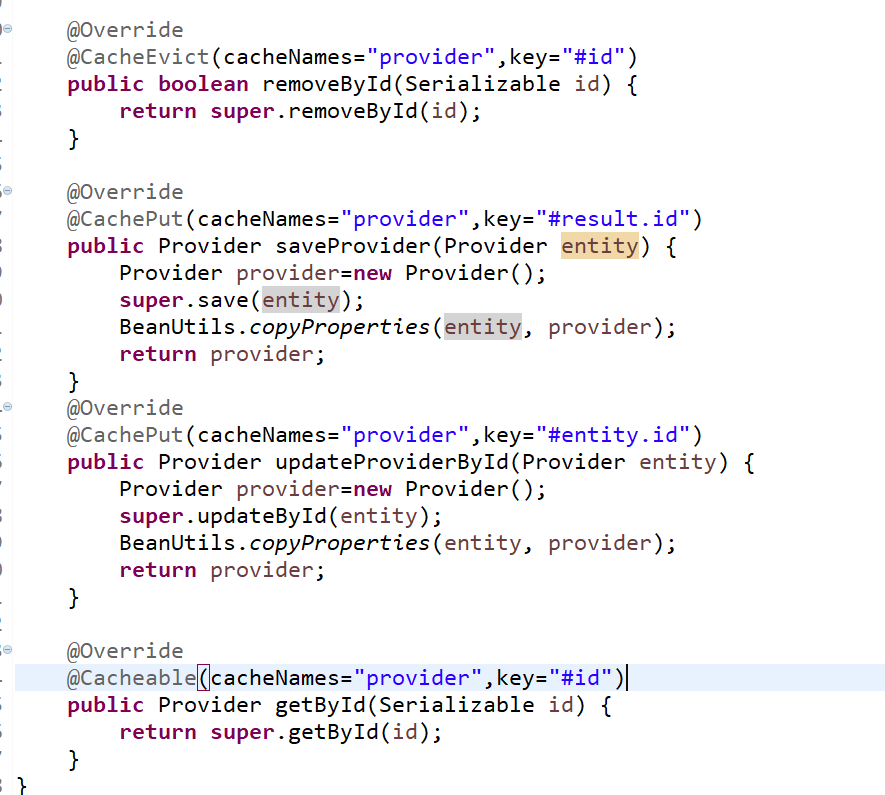
### 开启缓存注解



### 修改ProviderService



### 修改ProviderServiceImpl



### 修改ProviderContrller

### 更改序列化的方式[创建配置类去解决]

|  |
| --- |
| @Configuration  **public** **class** RedisConfig {  @Bean  **public** RedisCacheConfiguration redisCacheConfiguration(CacheProperties cacheProperties) {  CacheProperties.Redis redisProperties = cacheProperties.getRedis();  RedisCacheConfiguration config = RedisCacheConfiguration  .*defaultCacheConfig*();  config = config.serializeValuesWith(RedisSerializationContext.SerializationPair  //把默认的jdk的序列化方式变成jackson  .*fromSerializer*(**new** GenericJackson2JsonRedisSerializer()));  **if** (redisProperties.getTimeToLive() != **null**) {  config = config.entryTtl(redisProperties.getTimeToLive());  }  **if** (redisProperties.getKeyPrefix() != **null**) {  config = config.prefixKeysWith(redisProperties.getKeyPrefix());  }  **if** (!redisProperties.isCacheNullValues()) {  config = config.disableCachingNullValues();  }  **if** (!redisProperties.isUseKeyPrefix()) {  config = config.disableKeyPrefix();  }  **return** config;    }  } |

### 注意点

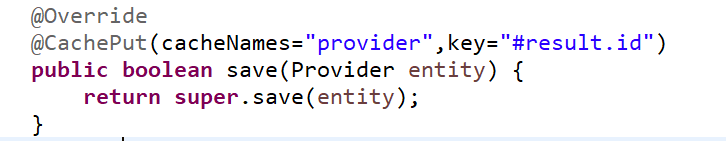
注解缓存缓存的是当前注解所在方法的返回值

cacheNames 缓存的前缀

key 代表:内容

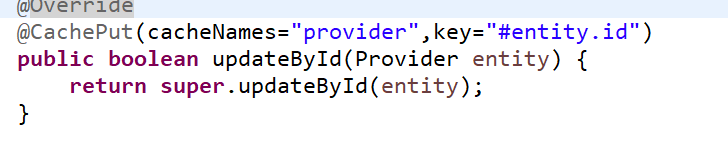
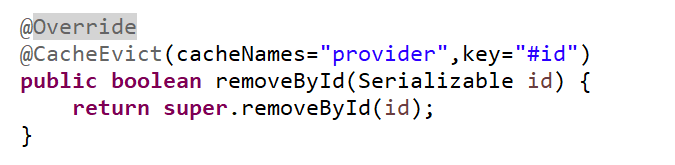
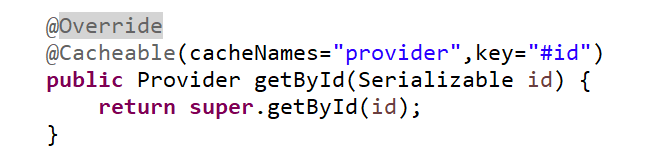
CachePut ===jedis.set方法

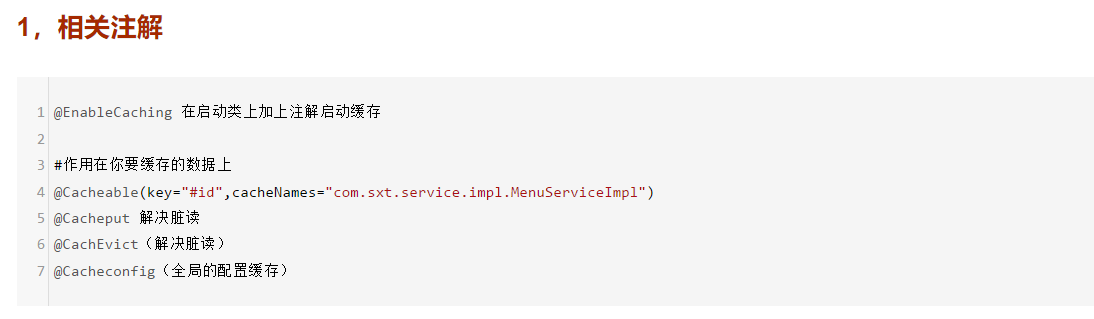
#result代表当前方法执行完成之后的返回值对象



entity.id

entity指当前方法的形式参数的名字 从里面可以取出id做为key



# 【掌握】Redis键和值的设计原则

## redis value 值格式

在Java常规开发中，我们需要有面向对象的思想，相对于对象来说，比较常用且能快速转换的格式就是JSON了；比较常用的Java处理JSON数据有三个比较流行的类库FastJSON、Gson和Jackson

上面提到了JSON，这是因为在Redis的存储中，我们使用它来存储value值，为什么要这样做呢？主要是因为json格式有如下几种好处：

标准，主流数据交换格式

简单，结构清晰，相对于XML来说更加的轻量级，易于解析

语言无关

类型安全，值是有类型的，比如整数、字符串、布尔等

代码中redis value在存储前我们对其做了一次转换，转换为json对象后存储.也就是一个key对应一个json串

## redis key 键格式

上面讲了简单的key存储，如 xdd的存储，此时普通的需求可以满足；然而在实际业务中，往往key键的存储会非常的复杂，比如我们现在有一个需求：

需求：根据基础数据系统中的数据字典类型查询对应的字典集合

这时，我们需要关注的业务就变得复杂了，就不能使用常规的key键存储方式，上面的需求大致可以拆分为：

系统：基础数据系统 user:sex:1=男 user:sex:0= 女

模块：数据字典 sys:available:1=true sys:available:0=false

方法：根据数据字典类型查询

参数：字典类型

为什么要这样拆分呢？为了可读性；也为了抽象出key存储规则；因为业务复杂情况下，我们定义的key键太多时就不便于管理，也不便于查找，以 系统-模块-方法-参数 这样的规则定义，我们可以很清晰的了解redis key存储的值是做了什么事情

common:sys:sex:1 男

common:sys:sex:0 女

common:sys:flag:1 是

common:sys:flag:0 否

common:page:title 欢迎使用XX管理系统

user:1 {id:1.name:小明}

user:2 {id:2.name:习大大}

这个在使用工具去查看的时候就可以看出层级关系啦

# 【掌握】面试

## Redis持久化方式？优缺点是什么

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

　　（Redis 数据都放在内存中。如果机器挂掉，内存的数据就不存在。所以需要做持久化，将内存中的数据保存在磁盘，下一次启动的时候就可以恢复数据到内存中。）

　　Redis提供了两种持久化方式:RDB（默认）和AOF 。

RDB （快照）：

　　Redis可以通过快照来获得存储在内存里面的数据在某个时间点上的副本。Redis创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis主从结构，主要用来提高Redis性能），还可以将快照留在原地以便重启服务器的时候使用。

AOF（只追加文件）：

　　与快照持久化相比，AOF持久化的实时性更好，因此已成为主流的持久化方案。默认情况下Redis没有开启AOF（append only file）方式的持久化，可以通过appendonly参数开启：appendonly yes

　　开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是appendonly.aof。

　　在Redis的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

appendfsync always #每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度

appendfsync everysec #每秒钟同步一次

appendfsync no #让操作系统决定何时同步

　　为了兼顾数据和写入性能，用户可以考虑appendfsync everysec选项 ，让Redis每秒同步一次AOF文件，Redis性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis还会优雅的放慢自己的速度以便适应硬盘的最大写入速度

RDB （快照）：快照形式 ，定期将当前时刻的数据保存磁盘中。会产生一个dump.rdb文件

特点：性能较好，数据备份。但可能会存在数据丢失。

AOF（只追加文件）：append only file (所有对redis的操作命令记录在aof文件中)，恢复数据，重新执行一遍即可。

特点：每秒保存，数据比较完整。但耗费性能。

【注】如果两个都配了优先加载AOF。（同时开启两个持久化方案，则按照 AOF的持久化放案恢复数据。）

## Redis架构模式

主从模式（redis2.8版本之前的模式）、哨兵sentinel模式（redis2.8及之后的模式）、redis cluster模式（redis3.0版本之后）

## 缓存穿透

缓存和数据库中都没有的数据被恶意请求,对后端系统造成压力。

解决方法

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

2：对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该bitmap过滤

3：布隆过滤器bloom filter

## 缓存雪崩

缓存服务器重启或缓存集体失效

如何避免？

1：互斥锁排队 在缓存失效后，通过锁或者队列控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

2：二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期

3：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

4.缓存预热

## 缓存预热

系统上线后,将相关的缓存数据直接加载到缓存系统

解决方案

数据量小,工程启动时进行加载缓存

数据量大时,设定定时脚本,进行缓存刷新

数据量过大,优先保证热点数据提前加载到缓存

## 缓存降级

缓存失效或缓存服务器宕机时,也不去访问数据库,而访问内存部分内存部分和数据缓存或直接返回默认数据.