Redis教程

1. **概述**

Remote Dictionary Server(Redis) 是一个由Salvatore Sanfilippo写的key-value存储系统。

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

它通常被称为数据结构服务器，因为值（value）可以是 字符串(String), 哈希(Map), 列表(list), 集合(sets) 和 有序集合(sorted sets)等类型。

特点：

* Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
* Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
* Redis支持数据的备份，即master-slave模式的数据备份。

优势：

 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。

 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

 原子 – Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。

 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性

优点：

1 读写性能优异

2 支持数据持久化，支持AOF和RDB两种持久化方式

3 支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。

4 数据结构丰富：除了支持string类型的value外还支持string、hash、set、sortedset、list等数据结构。

5 易扩展，nosql数据库种类繁多，但是有一个共同特点都是去掉了关系数据库的关系特性，数据之间无关系，这样就非常容易扩展，也无形之间，在架构的层面上带来了可扩展的能力。

缺点：

1 Redis不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复。

2 主机宕机，宕机前有部分数据未能及时同步到从机，切换IP后还会引入数据不一致的问题，降低了系统的可用性。

3 Redis的主从复制采用全量复制，复制过程中主机会fork出一个子进程对内存做一份快照，并将子进程的内存快照保存为文件发送给从机，这一过程需要确保主机有足够多的空余内存。若快照文件较大，对集群的服务能力会产生较大的影响，而且复制过程是在从机新加入集群或者从机和主机网络断开重连时都会进行，也就是网络波动都会造成主机和从机间的一次全量的数据复制，这对实际的系统运营造成了不小的麻烦。

4 Redis较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

**redis内存占用情况：**

测试情况：

100万个键值对（键是0到999999值是字符串“hello world”）在32位操作系统的笔记本上 用了100MB

使用64位的操作系统的话，相对来说占用的内存会多一点，这是因为64位的系统里指针占用了8个字节，但是64位系统也能支持更大的内存，所以运行大型的redis服务还是建议使用64位服务器

**Redis实例最多存keys数**

理论上Redis可以处理多达2的32次方的keys，并且在实际中进行了测试，每个实例至少存放了2亿5千万的keys

也可以说Redis的存储极限是系统中的可用内存值。

想了解更多可以访问地址：

http://redis.cn/commands.html

**Redis为什么这么快**

1）、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；

2）、数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；

3）、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4）、使用多路I/O复用模型，非阻塞IO；

5）、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

**为什么Redis是单线程**

官方回答：因为Redis是基于内存的操作，CPU(单核)不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了。

但是，我们使用单线程的方式是无法发挥多核CPU 性能，不过我们可以通过在单机开多个Redis 实例来完善！

1. **IO多路复用**

为什么 Redis 中要使用 I/O 多路复用这种技术呢？

首先，**Redis 是跑在单线程**中的，所有的操作都是按照顺序线性执行的， 但是由于读写操作等待用户输入或输出都是阻塞的，所以 I/O 操作在一般情况下往往不能直接返回，这会导致某一文件的 I/O 阻塞导致整个进程无法对其它客户提供服务，而 I/O 多路复用就是为了解决这个问题而出现的。

阻塞式的 I/O 模型并不能满足这里的需求，我们需要一种效率更高的 I/O 模型来支撑 Redis 的多个客户（redis-cli），这里涉及的就是 I/O 多路复用模型了。

redis的io模型主要是基于epoll实现的，不过它也提供了 select和kqueue的实现，默认采用epoll。

那么epoll到底是个什么东西呢？ 其实只是众多i/o多路复用技术当中的一种而已，但是相比其他io多路复用技术(select, poll等等)，epoll有诸多优点：

1. epoll 没有最大并发连接的限制，上限是最大可以打开文件的数目，这个数字一般远大于 2048, 一般来说这个数目和系统内存关系很大 ，具体数目可以 cat /proc/sys/fs/file-max 察看。

2. 效率提升， Epoll 最大的优点就在于它只管你“活跃”的连接 ，而跟连接总数无关，因此在实际的网络环境中， Epoll 的效率就会远远高于 select 和 poll 。

3. 内存拷贝， Epoll 在这点上使用了“共享内存 ”，这个内存拷贝也省略了。

1. **什么是BSD协议**

BSD开源协议是一个给于使用者很大自由的协议。可以自由的使用，修改源代码，也可以将修改后的代码作为开源或者专有软件再发布。当你发布使用了BSD协议的代码，或者以BSD协议代码为基础做二次开发自己的产品时，需要满足三个条件：

* 如果再发布的产品中包含源代码，则在源代码中必须带有原来代码中的BSD协议。
* 如果再发布的只是二进制类库/软件，则需要在类库/软件的文档和版权声明中包含原来代码中的BSD协议。
* 不可以用开源代码的作者/机构名字和原来产品的名字做市场推广。

BSD代码鼓励代码共享，但需要尊重代码作者的著作权。BSD由于允许使用者修改和重新发布代码，也允许使用或在BSD代码上开发商业软件发布和销售，因此是对商业集成很友好的协议。很多的公司企业在选用开源产品的时候都首选BSD协议，因为可以完全控制这些第三方的代码，在必要的时候可以修改或者二次开发。

1. **下载安装**

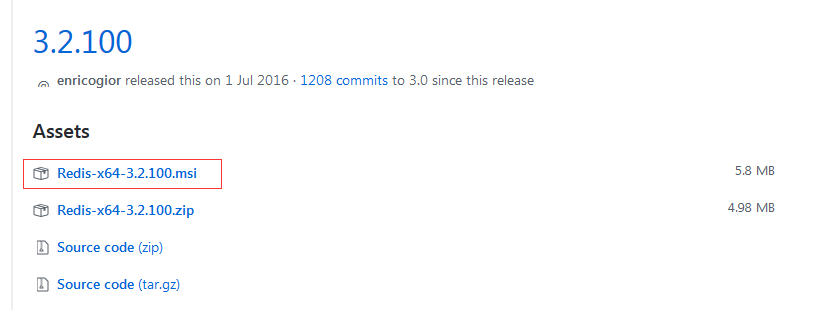
**下载地址：**官网上可以下载的windows版本的，现在官网以及没有下载地址，只能在github上下载，官网只提供linux版本的下载

官网下载地址：http://redis.io/download

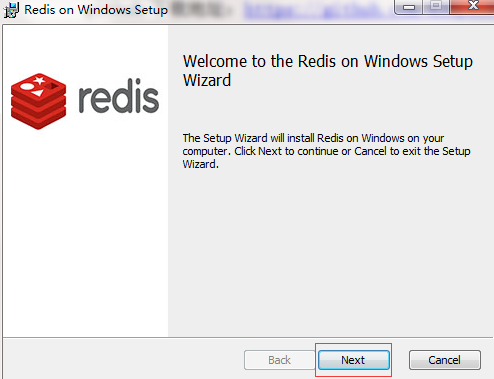
github下载地址：<https://github.com/MSOpenTech/redis/tags>

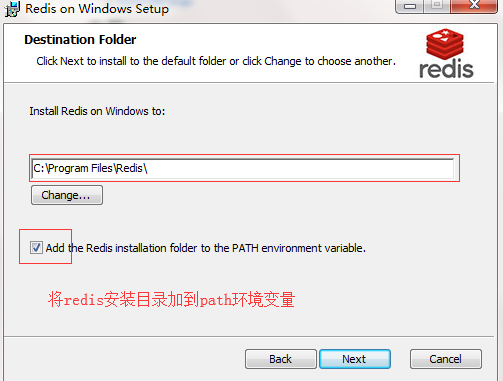
**windows安装：**

安装版本如下：



**安装：**

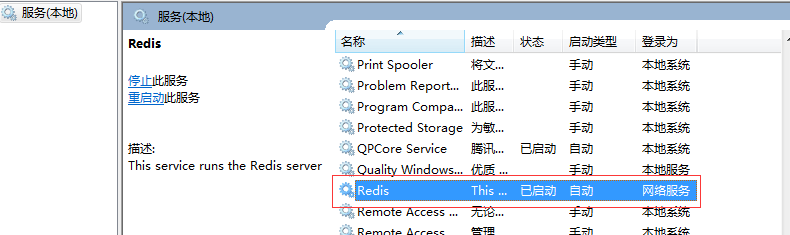








安装成功后，可以查看Windows服务里面能看到redis这时已经启动服务



测试安装成功后的redis是否运行正常

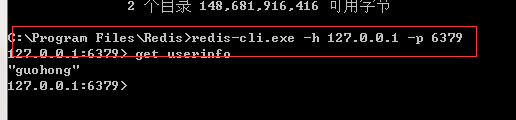
打开dos命令，进入到redis的安装目录，然后输入如下命令

redis-cli.exe –h 127.0.0.1 –p 6379

如果能连接上redis说明安装成功

set userinfo guohong 设置key为userinfo键值为guohong的key-val

get userinfo 查看键为userinfo的值

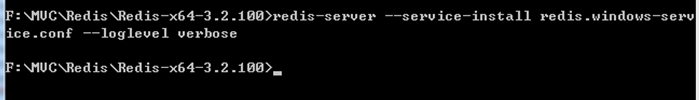


通过Windows命令启动服务，该命令在redis的安装目录下执行

redis-server redis.windows.conf

如果我们安装后，在Windows的服务里面没有redis的服务，则需要我们手动加入到服务中

redis-server --service-install redis.windows-service.conf --loglevel verbose



常用服务命令：

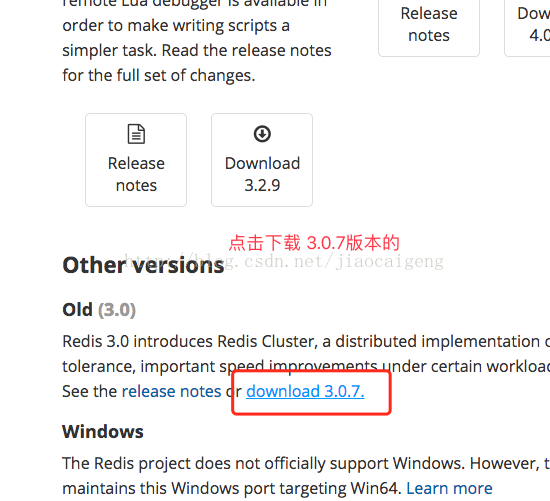
卸载服务：redis-server --service-uninstall

开启服务：redis-server --service-start

停止服务：redis-server --service-stop

**mac安装：**

百度 redis下载，进官网【https://redis.io/】下载。目前已3.2.9版本选择稳定一点的版本3.0.7



将下载的文件解压到某处。然后进行安装。

打开终端，cd到redis的解压目录下，输入sudo make install命令

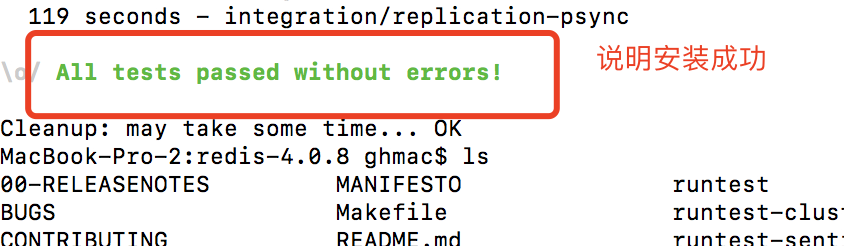


如果make失败，重新再次make之前，最好make distclean把之前的一些东西清理掉，类似于mvn clean。

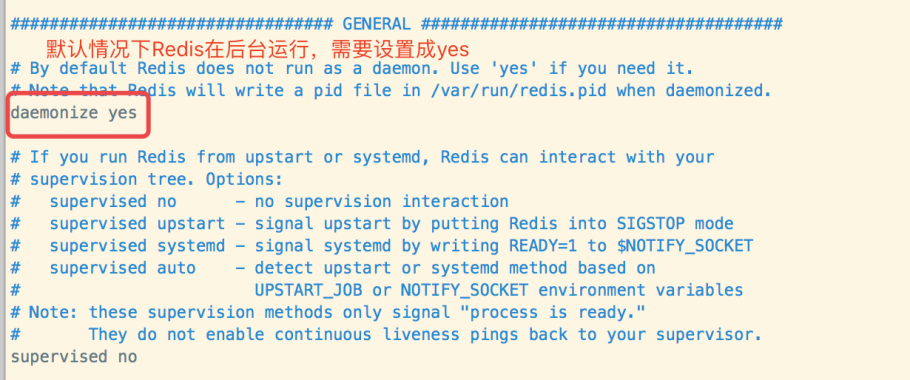
当看到 It is a good idea to run 'make test' 即表示安装成功！（既然是个好的想法 就do it），期间会不断要求输入电脑密码，填写即可。

输入make test



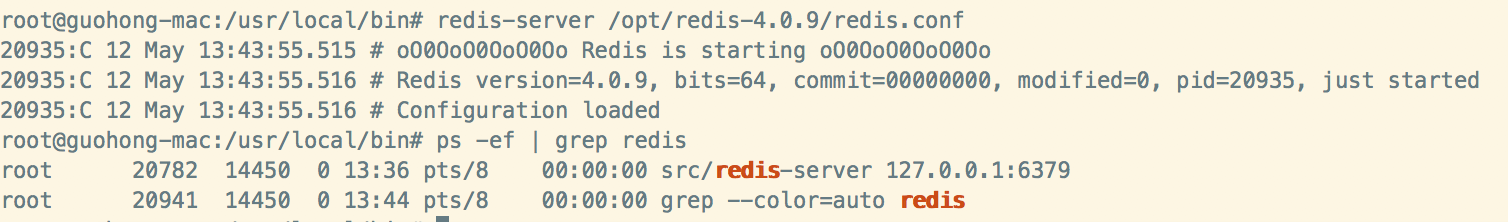


修改配置文件：一般情况下，新建文件夹myredis，把redis.conf放到下面，我们修改配置文件就修改这个配置文件的内容，然后运行redis的时候，以这个配置运行：redis-server ./myredis/redis.conf(daemon守护进程的意思)



启动服务：

cd src或者在/usr/local/bin目录下也有redis-server命令，在此运行也行。



./redis-server 不在默认配置下运行

./redis-server redis.conf 在默认配置下运行(redis.conf在redis安装目录下，需要切换到redis.conf目录下输入上述命令，redis的命令配置到了环境变量)

./redis-server & ：输入该命令，ctrl+c时，退出运行界面redis也能保持运行，不加&时，退出的时候，redis就会关闭。

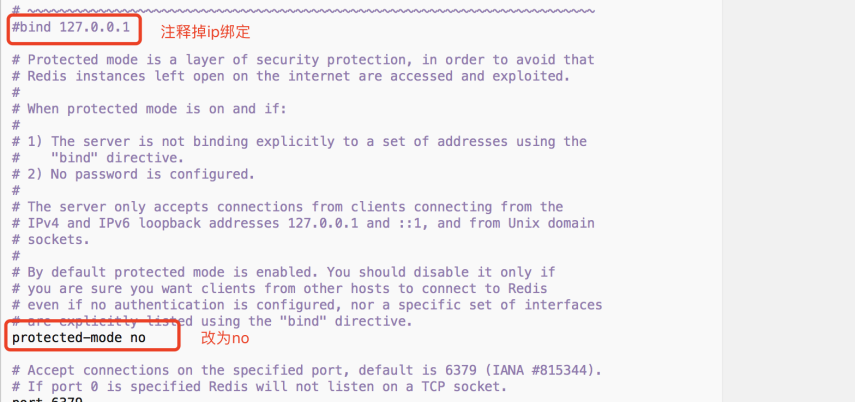
启动客户端进行访问

./redis-cli –h 127.0.0.1 –p 6379

set admin guohong 设置用户名：guohong

get admin

在mac上以默认配置文件的方式启动后，让局域网的其他机器访问本机redis服务。需要修改redis.conf配置文件



关闭redis:先redis-cli连上redis服务，然后输入SHUTDOWN关闭

**Ubuntu系统：**

官网是有教程的

$ wget http://download.redis.io/releases/redis-4.0.9.tar.gz

$ tar xzf redis-4.0.9.tar.gz

$ cd redis-4.0.9

$ make

$ src/redis-server

基本上安装和mac版是一样的

**centos安装：**

前期步骤跟Ubuntu系统是一样的。

因为Redis是C实现的，需要gcc来进行编译，所以我们要确保系统安装了如下程序，如果没有安装，则顺序安装：

yum install cpp

yum install binutils

yum install glibc

yum install glibc-kernheaders

yum install glibc-common

yum install glibc-devel

yum install gcc

yum install make

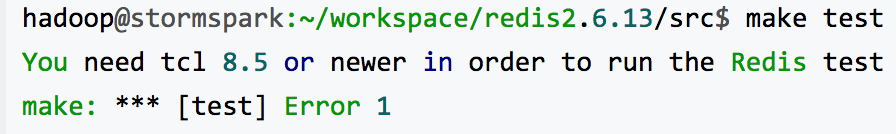
如上操作完成后，make命令仍然出错



我们可以输入如下命令：

make MALLOC=libc

安装成功后，make test可能会出错：



此时需要更新升级tcl

wget http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz

sudo tar xzvf tcl8.6.1-src.tar.gz -C /usr/local/

cd /usr/local/tcl8.6.1/unix/

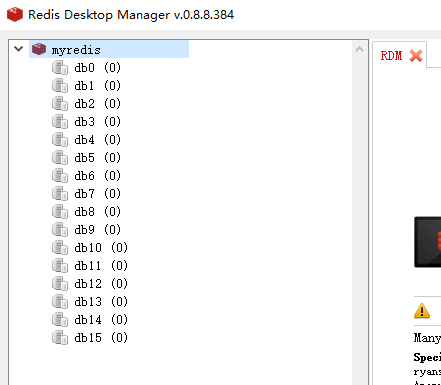
sudo ./configure

sudo make

sudo make install

**客户端**

Redis Desktop Manager

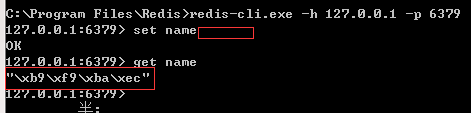


1. **查看redis的值为中文的情况**

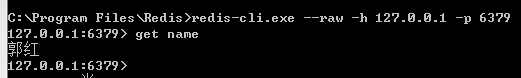
使用如下命令，连接redis的时候加上—raw

redis-cli.exe --raw -h 127.0.0.1 -p 6379

没加--raw的效果如下，设置name的值为郭红，这里可能是dos的问题，中文值没显示出来，在get的时候拿到的就不是中文

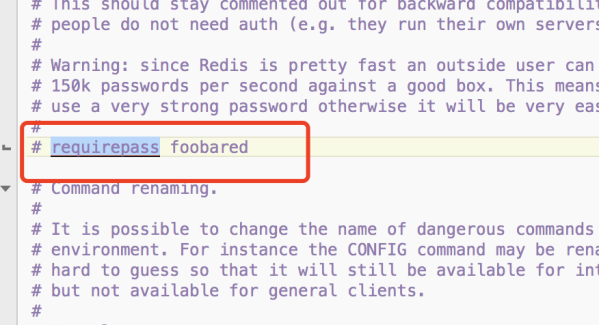


加上后的效果



1. **安全设置-设置密码连接**

**方法**1：修改redis.conf配置文件



打开注释，重新设置密码，但是为了让密码生效，启动的时候必须读取默认配置文件，也就是以如下方式启动

./redis-server redis.conf

**方法2：**

以下都是用的命令操作

步骤1：连接redis

redis-cli.exe -h 127.0.0.1 -p 6379

步骤2：

CONFIG GET requirepass 查看是否设置了密码



步骤3：设置密码，密码为guohong

CONFIG SET requirepass "guohong"



这个时候再进行操作会提示你没有权限，需要验证密码



步骤4：验证密码

AUTH guohong



然后再查看密码，此时已经有了密码



1. **Redis的java demo**

我的是maven工程

所以在pom.xml文件中还需要配置如下：

<dependencies>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.9.0</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.9</version>

<scope>test</scope>

</dependency>

</dependencies>

Redis连接类：

package com.gh;

import redis.clients.jedis.Jedis;

import redis.clients.jedis.JedisPool;

import redis.clients.jedis.JedisPoolConfig;

public final class RedisPool {

// Redis服务器IP

private static String ADDR = "127.0.0.1";

// Redis的端口号

private static Integer PORT = 6379;

// 访问密码

private static String AUTH = "guohong";

// 可用连接实例的最大数目，默认为8；

// 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(耗尽)

private static Integer MAX\_TOTAL = 1024;

// 控制一个pool最多有多少个状态为idle(空闲)的jedis实例，默认值是8

private static Integer MAX\_IDLE = 200;

// 等待可用连接的最大时间，单位是毫秒，默认值为-1，表示永不超时。

// 如果超过等待时间，则直接抛出JedisConnectionException

private static Integer MAX\_WAIT\_MILLIS = 10000;

private static Integer TIMEOUT = 10000;

// 在borrow(用)一个jedis实例时，是否提前进行validate(验证)操作；

// 如果为true，则得到的jedis实例均是可用的

private static Boolean TEST\_ON\_BORROW = true;

private static JedisPool jedisPool = null;

/\*\*

\* 静态块，初始化Redis连接池

\*/

static {

try {

JedisPoolConfig config = new JedisPoolConfig();

/\*

\* 注意： 在高版本的jedis

\* jar包，比如本版本2.9.0，JedisPoolConfig没有setMaxActive和setMaxWait属性了

\* 这是因为高版本中官方废弃了此方法，用以下两个属性替换。 maxActive ==> maxTotal maxWait==> maxWaitMillis

\*/

config.setMaxTotal(MAX\_TOTAL);

config.setMaxIdle(MAX\_IDLE);

config.setMaxWaitMillis(MAX\_WAIT\_MILLIS);

config.setTestOnBorrow(TEST\_ON\_BORROW);

jedisPool = new JedisPool(config,ADDR,PORT,TIMEOUT,AUTH);

//无密码的连接

//jedisPool = new JedisPool(config, ADDR, PORT, TIMEOUT);

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*

\* 获取Jedis实例

\*

\* @return

\*/

public synchronized static Jedis getJedis() {

try {

if (jedisPool != null) {

Jedis jedis = jedisPool.getResource();

return jedis;

} else {

return null;

}

} catch (Exception e) {

e.printStackTrace();

return null;

}

}

public static void returnResource(final Jedis jedis) {

// 方法参数被声明为final，表示它是只读的。

if (jedis != null) {

jedisPool.returnResource(jedis);

// jedis.close()取代jedisPool.returnResource(jedis)方法将3.0版本开始

// jedis.close();

}

}

}

测试用例：

package com.gh;

import java.util.HashMap;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import org.junit.Before;

import org.junit.Test;

import redis.clients.jedis.Jedis;

public class TestRedis {

private Jedis jedis;

@Before

public void setJedis() {

//连接redis服务器(在这里是连接本地的)

jedis = new Jedis("127.0.0.1", 6379);

//权限认证

jedis.auth("guohong");

System.out.println("连接服务成功");

}

/\*\*

\* Redis操作字符串

\*/

@Test

public void testString() {

//添加数据

jedis.set("name", "chx"); //key为name放入value值为chx

System.out.println("拼接前:" + jedis.get("name"));//读取key为name的值

//向key为name的值后面加上数据 ---拼接

jedis.append("name", " is my name;");

System.out.println("拼接后:" + jedis.get("name"));

//删除某个键值对

jedis.del("name");

System.out.println("删除后:" + jedis.get("name"));

//s设置多个键值对

jedis.mset("name", "chenhaoxiang", "age", "20", "email", "chxpostbox@outlook.com");

jedis.incr("age");//用于将键的整数值递增1。如果键不存在，则在执行操作之前将其设置为0。 如果键包含错误类型的值或包含无法表示为整数的字符串，则会返回错误。此操作限于64位有符号整数。

System.out.println(jedis.get("name") + " " + jedis.get("age") + " " + jedis.get("email"));

}

@Test

public void testMap() {

//添加数据

Map<String, String> map = new HashMap<String, String>();

map.put("name", "chx");

map.put("age", "100");

map.put("email", "\*\*\*@outlook.com");

jedis.hmset("user", map);

//取出user中的name，结果是一个泛型的List

//第一个参数是存入redis中map对象的key，后面跟的是放入map中的对象的key，后面的key是可变参数

List<String> list = jedis.hmget("user", "name", "age", "email");

System.out.println(list);

//删除map中的某个键值

jedis.hdel("user", "age");

System.out.println("age:" + jedis.hmget("user", "age")); //因为删除了，所以返回的是null

System.out.println("user的键中存放的值的个数:" + jedis.hlen("user")); //返回key为user的键中存放的值的个数2

System.out.println("是否存在key为user的记录:" + jedis.exists("user"));//是否存在key为user的记录 返回true

System.out.println("user对象中的所有key:" + jedis.hkeys("user"));//返回user对象中的所有key

System.out.println("user对象中的所有value:" + jedis.hvals("user"));//返回map对象中的所有value

//拿到key，再通过迭代器得到值

Iterator<String> iterator = jedis.hkeys("user").iterator();

while (iterator.hasNext()) {

String key = iterator.next();

System.out.println(key + ":" + jedis.hmget("user", key));

}

jedis.del("user");

System.out.println("删除后是否存在key为user的记录:" + jedis.exists("user"));//是否存在key为user的记录

}

/\*\*

\* jedis操作List

\*/

@Test

public void testList(){

//移除javaFramwork所所有内容

jedis.del("javaFramwork");

//存放数据

jedis.lpush("javaFramework","spring");

jedis.lpush("javaFramework","springMVC");

jedis.lpush("javaFramework","mybatis");

//取出所有数据,jedis.lrange是按范围取出

//第一个是key，第二个是起始位置，第三个是结束位置

System.out.println("长度:"+jedis.llen("javaFramework"));

//jedis.llen获取长度，-1表示取得所有

System.out.println("javaFramework:"+jedis.lrange("javaFramework",0,-1));

jedis.del("javaFramework");

System.out.println("删除后长度:"+jedis.llen("javaFramework"));

System.out.println(jedis.lrange("javaFramework",0,-1));

}

/\*\*

\* jedis操作Set

\*/

@Test

public void testSet(){

//添加

jedis.sadd("user","chenhaoxiang");

jedis.sadd("user","hu");

jedis.sadd("user","chen");

jedis.sadd("user","xiyu");

jedis.sadd("user","chx");

jedis.sadd("user","are");

//移除user集合中的元素are

jedis.srem("user","are");

System.out.println("user中的value:"+jedis.smembers("user"));//获取所有加入user的value

System.out.println("chx是否是user中的元素:"+jedis.sismember("user","chx"));//判断chx是否是user集合中的元素

System.out.println("集合中的一个随机元素:"+jedis.srandmember("user"));//返回集合中的一个随机元素

System.out.println("user中元素的个数:"+jedis.scard("user"));

}

/\*\*

\* 排序

\*/

@Test

public void test(){

jedis.del("number");//先删除数据，再进行测试

jedis.rpush("number","4");//将一个或多个值插入到列表的尾部(最右边)

jedis.rpush("number","5");

jedis.rpush("number","3");

jedis.lpush("number","9");//将一个或多个值插入到列表头部

jedis.lpush("number","1");

jedis.lpush("number","2");

System.out.println(jedis.lrange("number",0,jedis.llen("number")));

System.out.println("排序:"+jedis.sort("number"));

System.out.println(jedis.lrange("number",0,-1));//不改变原来的排序

jedis.del("number");//测试完删除数据

}

}

1. **Redis高级特性及应用场景**

**键的生存时间（expire）:**

redis中可以使用expire命令设置一个键的生存时间，到时间后redis会自动删除它。

1. 过期时间可以设置为秒或者毫秒精度。
2. 过期时间分辨率总是 1 毫秒。
3. 过期信息被复制和持久化到磁盘，当 Redis 停止时时间仍然在计算 (也就是说 Redis 保存了过期时间)。

expire 设置生存时间（单位/秒）: expire key seconds(秒)

ttl 查看键的剩余生存时间: ttl key

persist 取消生存时间: persist key

expireat [key] unix时间戳1351858600:

EXPIREAT cache 1355292000 # 这个 key 将在 2012.12.12 过期

**应用场景：**

1. 限时的优惠活动信息
2. 网站数据缓存（对于一些需要定时更新的数据，例如：积分排行榜）
3. 手机验证码
4. 限制网站访客访问频率（例如：1分钟最多访问10次）

**Redis事务：**

Redis事务解决并发问题。

redis中的事务是一组命令的集合。事务同命令一样都是redis的最小执行单元。一组事务中的命令要么都执行，要么都不执行。(例如：转账)

Redis保证一个事务中的所有命令要么都执行，要么都不执行。如果在发送EXEC命令前客户端断线了，则Redis会清空事务队列（命令放在队列里），事务中的所有命令都不会执行。而一旦客户端发送了EXEC命令，所有的命令就都会被执行，即使此后客户端断线也没关系，因为Redis中已经记录了所有要执行的命令。

除此之外，Redis的事务还能保证一个事务内的命令依次执行而不被其他命令插入。试想客户端A需要执行几条命令，同时客户端B发送了一条命令，如果不使用事务，则客户端B的命令可能会插入到客户端A的几条命令中执行。如果不希望发生这种情况，也可以使用事务。

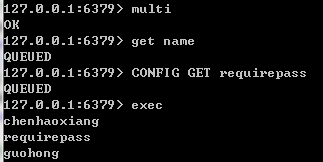
**原理：**先将属于一个事务的命令发送给redis进行缓存，最后再让redis依次执行这些命令。

**应用场景：**

1. 一组命令必须同时都执行，或者都不执行。

2．我们想要保证一组命令在执行的过程之中不被其它命令插入。

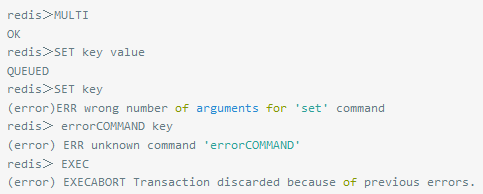
**示例：**



discard：取消事务

**错误处理：**

1：语法错误：致命的错误，事务中的所有命令都不会执行。



语法错误指命令不存在或者命令参数的个数不对。跟在MULTI命令后执行了3个命令：一个是正确的命令，成功地加入事务队列；其余两个命令都有语法错误。而只要有一个命令有语法错误，执行EXEC命令后Redis就会直接返回错误，连语法正确的命令也不会执行。

这里需要注意一点：

Redis 2.6.5之前的版本会忽略有语法错误的命令，然后执行事务中其他语法正确的命令。就此例而言，SET key value会被执行，EXEC命令会返回一个结果：1) OK。

2：运行错误：不会影响事务中其他命令的执行（包括出错命令之后的命令）

Redis 不支持回滚（roll back）

为此开发者必须在事务执行出错后自己收拾剩下的摊子（将数据库复原回事务执行前的状态等,这里我们一般采取日志记录然后业务补偿的方式来处理，但是一般情况下，在redis做的操作不应该有这种强一致性要求的需求，我们认为这种需求为不合理的设计）。

正因为redis不支持回滚功能，才使得redis在事务上可以保持简洁和快速。

注：和传统的mysql事务不同的事，即使我们的加钱操作失败,我们也无法在这一组命令中让整个状态回滚到操作之前

**Redis管道：**

redis的pipeline(管道)功能在命令行中没有，但是redis是支持管道的，在java的客户端(jedis)中是可以使用的。

使用管道保存数据比不使用管道快很多

不使用管道方式，插入1000条数据耗时328毫秒

// 测试不使用管道

public static void testInsert() {

long currentTimeMillis = System.currentTimeMillis();

Jedis jedis = new Jedis("192.168.33.130", 6379);

for (int i = 0; i < 1000; i++) {

jedis.set("test" + i, "test" + i);

}

long endTimeMillis = System.currentTimeMillis();

System.out.println(endTimeMillis - currentTimeMillis);

}

使用管道方式，插入1000条数据耗时37毫秒

// 测试管道

public static void testPip() {

long currentTimeMillis = System.currentTimeMillis();

Jedis jedis = new Jedis("192.168.33.130", 6379);

Pipeline pipelined = jedis.pipelined();

for (int i = 0; i < 1000; i++) {

pipelined.set("bb" + i, i + "bb");

}

pipelined.sync();

long endTimeMillis = System.currentTimeMillis();

System.out.println(endTimeMillis - currentTimeMillis);

}

在插入更多数据的时候，管道的优势更加明显：测试10万条数据的时候，不使用管道要40秒，实用管道378毫秒。

**发布与订阅:**

在更新中保持用户对数据的映射是系统中的一个普遍任务。Redis的pub/sub功能使用了SUBSCRIBE、UNSUBSCRIBE和PUBLISH命令，让这个变得更加容易。

// 订阅频道数据

@Test

**public** **void** testSubscribe() {

//连接Redis数据库

Jedis jedis = **new** Jedis("192.168.33.130", 6379);

JedisPubSub jedisPubSub = **new** JedisPubSub() {

// 当向监听的频道发送数据时，这个方法会被触发

@Override

**public** **void** onMessage(String channel, String message) {

System.***out***.println("收到消息" + message);

//当收到 "unsubscribe" 消息时，调用取消订阅方法

**if** ("unsubscribe".equals(message)) {

**this**.unsubscribe();

}

}

// 当取消订阅指定频道的时候，这个方法会被触发

@Override

**public** **void** onUnsubscribe(String channel, **int** subscribedChannels) {

System.***out***.println("取消订阅频道" + channel);

}

};

// 订阅之后，当前进程一致处于监听状态，当被取消订阅之后，当前进程会结束

jedis.subscribe(jedisPubSub, "ch1");

}

// 发布频道数据

@Test

**public** **void** testPubSub() **throws** Exception {

//链接Redis数据库

Jedis jedis = **new** Jedis("192.168.33.130", 6379);

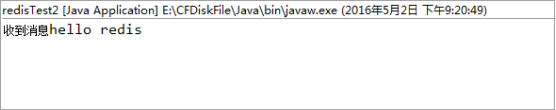
//发布频道 "ch1" 和消息 "hello redis"

jedis.publish("ch1", "hello redis");

//关闭连接

jedis.close();

}



**限制网站访客访问频率**

**package** com.gh;

**import** org.junit.Test;

**import** redis.clients.jedis.Jedis;

**public** **class** VisitorLimit {

// 指定Redis数据库连接的IP和端口

String host = "127.0.0.1";

**int** port = 6379;

Jedis jedis = **new** Jedis(host, port);

/\*\*

\* 限制网站访客访问频率 一分钟之内最多访问10次

\*

\* **@throws** Exception

\*/

@Test

**public** **void** test3() **throws** Exception {

// 模拟用户的频繁请求

**for** (**int** i = 0; i < 20; i++) {

**boolean** result = testLogin("192.168.1.100");

**if** (result) {

System.***out***.println("正常访问");

} **else** {

System.***err***.println("访问受限");

}

}

}

/\*\*

\* 判断用户是否可以访问网站

\*

\* **@param** ip

\* **@return**

\*/

**public** **boolean** testLogin(String ip) {

String value = jedis.get(ip);

**if** (value == **null**) {

// 初始化时设置IP访问次数为1

jedis.set(ip, "1");

// 设置IP的生存时间为60秒，60秒内IP的访问次数由程序控制

jedis.expire(ip, 60);

} **else** {

**int** parseInt = Integer.*parseInt*(value);

// 如果60秒内IP的访问次数超过10，返回false,实现了超过10次禁止分的功能

**if** (parseInt > 10) {

**return** **false**;

} **else** {

// 如果没有10次，可以自增

jedis.incr(ip);

}

}

**return** **true**;

}

}

**用作缓存代替memcached**

缓存内容示例：（商品列表，评论列表，@提示列表，etc）

相对memcached 简单的key-value存储来说，redis众多的数据结构（list,set,sorted set,hash, etc）可以更方便cache各种业务数据，性能也不亚于memcached。

**消息队列**

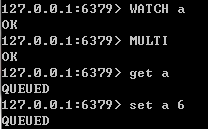
当在集群环境时候，java ConcurrentLinkedQueue 就无法满足我们需求，此时可以采用Redis的List数据结构实现分布式的消息队列。

**watch命令**

WATCH命令可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行。监控一直持续到EXEC命令（事务中的命令是在EXEC之后才执行的，所以在MULTI命令后可以修改WATCH监控的键值）.相当于乐观锁的概念

dos开两个命令窗口，其中一个执行watch和multi命令，另外一个修改键值。

a窗口执行watch和multi命令



在b窗口没有修改a的值时，a窗口不要执行exec命令

b窗口修改a的值



然后a窗口执行exec命令



此时a窗口的事务提交失败

Java代码实现：此处有些问题，我在dos下修改a值，并不起作用

/\*\*

\* 监控变量a在一段时间内是否被修改，若没有，则执行事务，若被修改，则事务不执行

\*

\* **@throws** Exception

\*/

@Test

**public** **void** test4() **throws** Exception {

// 监控变量a，在事务执行后watch功能也结束

jedis.watch("a");

// 需要数据库中先有a，并且a的值为字符串数字

String value = jedis.get("a");

**int** parseInt = Integer.*parseInt*(value);

parseInt++;

System.***out***.println("线程开始休息。。。"+value);

Thread.*sleep*(10000);

// 开启事务

Transaction transaction = jedis.multi();

transaction.set("a", parseInt + "");

// 执行事务

List<Object> exec = transaction.exec();

**if** (exec == **null**) {

System.***out***.println("事务没有执行.....");

} **else** {

System.***out***.println("正常执行......");

}

}

**排行榜/计数器**

Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的10个用户–我们称之为“user\_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE user\_scores 0 10 WITHSCORES

Agora Games就是一个很好的例子，用Ruby实现的，它的排行榜就是使用Redis来存储数据的，你可以在这里看到。

1. **Redis持久化RDB和AOF**

redis支持两种方式的持久化，可以单独使用或者结合起来使用。

第一种：RDB方式（redis默认的持久化方式）

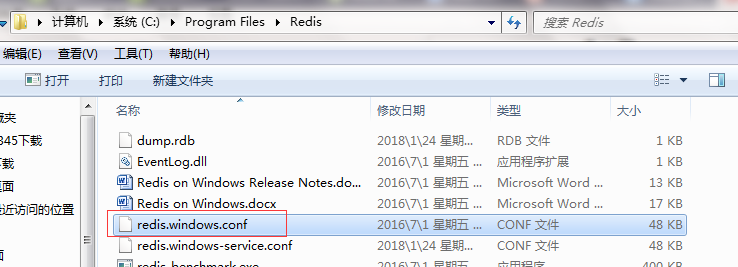
第二种：AOF方式

**redis持久化之RDB:**

rdb（redis database）方式的持久化是通过快照完成的，当符合一定条件时redis会自动将内存中的所有数据执行快照操作并存储到硬盘上。默认存储在dump.rdb文件中。(文件名在配置文件中dbfilename)。



配置文件在安装目录下的红圈文件



Redis自动实现快照的过程：

1、redis使用fork函数复制一份当前进程的副本(子进程)

2、父进程继续接收并处理客户端发来的命令，而子进程开始将内存中的数据写入硬盘中的临时文件

3、当子进程写入完所有数据后会用该临时文件替换旧的RDB文件，至此，一次快照操作完成。

注意：redis在进行快照的过程中不会修改RDB文件，只有快照结束后才会将旧的文件替换成新的，也就是说任何时候RDB文件都是完整的。这就使得我们可以通过定时备份RDB文件来实现redis数据库的备份。

RDB文件是经过压缩的二进制文件，占用的空间会小于内存中的数据，更加利于传输。

手动执行save或者bgsave命令让redis执行快照。

两个命令的区别在于，save是由主进程进行快照操作，会阻塞其它请求。bgsave是由redis执行fork函数复制出一个子进程来进行快照操作。

文件修复：redis-check-dump

文件恢复：当redis宕机后，重启的时候，redis会默认加载dump.rdb文件，进行数据恢复。

**rdb的优缺点：**

优点：由于存储的有数据快照文件，恢复数据很方便。

缺点：会丢失最后一次快照以后更改的所有数据。

**redis持久化之AOF：**

AOF (Append only file)则以协议文本的方式，将所有对数据库进行过写入的命令（及其参数）记录到 AOF 文件，以此达到记录数据库状态的目的。aof将“操作 + 数据”以格式化指令的方式追加到操作日志文件的尾部，在append操作返回后(已经写入到文件或者即将写入)，才进行实际的数据变更，“日志文件”保存了历史所有的操作过程；当server需要数据恢复时，可以直接replay此日志文件，即可还原所有的操作过程。AOF相对可靠.。AOF文件内容是字符串，非常容易阅读和解析。

同步命令到 AOF 文件的整个过程可以分为三个阶段：

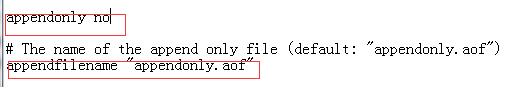
1. 命令传播：Redis 将执行完的命令、命令的参数、命令的参数个数等信息发送到 AOF 程序中。
2. 缓存追加：AOF 程序根据接收到的命令数据，将命令转换为网络通讯协议的格式，然后将协议内容追加到服务器的 AOF 缓存中。
3. 文件写入和保存：AOF 缓存中的内容被写入到 AOF 文件末尾，如果设定的 AOF 保存条件被满足的话，fsync 函数或者fdatasync函数会被调用，将写入的内容真正地保存到磁盘中。

aof方式的持久化是通过日志文件的方式。默认情况下redis没有开启aof，可以通过参数appendonly参数开启。

命令：appendonly yes

aof文件的保存位置和rdb文件的位置相同，都是dir参数设置的，默认的文件名是appendonly.aof，可以通过appendfilename参数修改

appendfilename appendonly.aof



redis写命令同步的时机

appendfsync always 每执行一个命令保存一次。

appendfsync everysec 默认 每秒执行一次同步操作（推荐，默认）

appendfsync no不主动进行同步保存，由操作系统来做，30秒一次

aof日志文件重写

auto-aof-rewrite-percentage 100(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时会再次进行重写，如果之前没有重写，则以启动时的aof文件大小为依据)

auto-aof-rewrite-min-size 64mb（一般生产都是3Gb起）

手动执行bgrewriteaof进行重写

重写的过程只和内存中的数据有关，和之前的aof文件无关。

所谓的“重写”其实是一个有歧义的词语， 实际上， AOF 重写并不需要对原有的 AOF 文件进行任何写入和读取， 它针对的是数据库中键的当前值。

为什么要重写？

答：AOF 文件通过同步 Redis 服务器所执行的命令， 从而实现了数据库状态的记录， 但是， 这种同步方式会造成一个问题： 随着运行时间的流逝， AOF 文件会变得越来越大，对Redis甚至整个系统的造成影响。

为了解决以上的问题， Redis 需要对 AOF 文件进行重写（rewrite）： 创建一个新的 AOF 文件来代替原有的 AOF 文件， 新 AOF 文件和原有 AOF 文件保存的数据库状态完全一样， 但新 AOF 文件的体积小于等于原有 AOF 文件的体积。（比如key为name的值有上百次修改，aof没必要执行每个更新操作，只需要执行最终状态的那次更新，比如执行100次更新的最终结果是100，那我们直接update set 为100就行了。）

文件修复：redis-check-aof；redis-check-aof --fix appendonly.aof

动态切换redis持久方式，从 RDB 切换到 AOF（支持Redis 2.2及以上）

CONFIG SET appendonly yes

CONFIG SET save ""（可选）

优点：可以保持更高的数据完整性，如果设置追加file的时间是1s，如果redis发生故障，最多会丢失1s的数据；且如果日志写入不完整，支持redis-check-aof来进行日志修复；AOF文件没被rewrite之前（文件过大时会对命令进行合并重写），可以删除其中的某些命令（比如误操作的flushall）。

缺点：AOF文件比RDB文件大，且恢复速度慢。

注意：

1、当redis启动时，如果rdb持久化和aof持久化都打开了，那么程序会优先使用aof方式来恢复数据集，因为aof方式所保存的数据通常是最完整的。如果aof文件丢失了，则启动之后数据库内容为空。最好不要只用aof做持久化，因为rdb更适合备份数据库，aof在不断变化，不好备份，快速重启，aof可能存在潜在的bug

2、如果想把正在运行的redis数据库，从RDB切换到AOF，建议先使用动态切换方式，再修改配置文件，重启数据库。(不能自己修改配置文件，重启数据库，否则数据库中数据就为空了。) 

1. **Redis优化**

**精简键名和键值**

键名：尽量精简，但是也不能单纯为了节约空间而使用不易理解的键名。

键值：对于键值的数量固定的话可以使用0和1这样的数字来表示，（例如：male/female、right/wrong）

当业务场景不需要数据持久化时，关闭所有的持久化方式可以获得最佳的性能。

**关闭Transparent Huge Pages(THP)**

THP会造成内存锁影响redis性能，建议关闭

Transparent HugePages ：用来提高内存管理的性能

Transparent Huge Pages在32位的RHEL 6中是不支持的

1. **常用命令**

启动Redis：redis-server 默认启动

redis-server redis.conf 按redis.conf配置文件启动

关闭：redis-cli -p 6379 shutdown 6379是端口号

或者先redis-cli连上redis服务，然后输入SHUTDOWN关闭

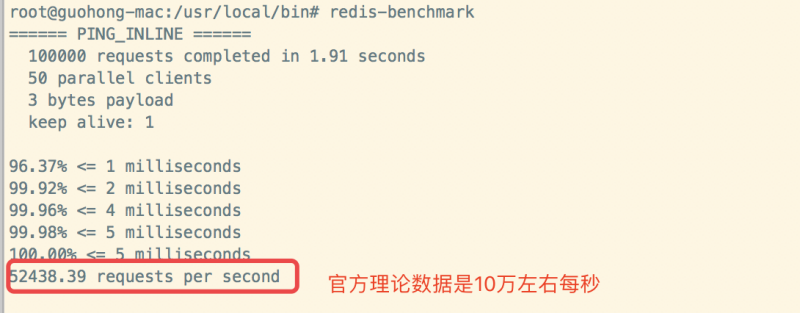
config get dir：查看当前redis命令路径

config get requirepass：查看密码

config set requirepass gh0316:设置密码

**redis一些基本特性：**

redis-benchmark命令可以测试机器redis的读写性能



redis默认的数据库个数是16，在配置文件里面有配置

切换库：select 7切换到8号库（计数从0开始）,第一次进来的时候，默认的是0号库

DBSIZE:查看key的数量

keys fo：查看包含fo字符串的key

keys \*:支持通配符这些

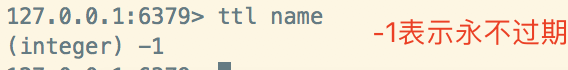
FLUSHDB:清库，如果先切换了库，则只会清掉选择的库，FLUSHALL清掉所有库。

**String类型：**

EXISTS name:是否存在键为name的值

move name 1:将键为name的数据移到1号库

ttl name:(time to live还能活到久)查询某键值的存活时间



-1:表示永不过期

-2：表示已过期，过期会被删除，redis会自动去掉过期的数据

EXPIRE name 10：设置key为name的过期时间，单位是秒

type name：查看key为name的数据类型

set name gh:设置key为name的值为gh，如果存在则覆盖，不存在则新增

get name：获得key为name的值

del name:删除key为name的值

**list类型：**

LPUSH list1 1 2 3 4:设置list1的数据，如果list1已经存在则从栈顶追加到list1里

RPUSH list1 -1 -2: 设置list1的数据，如果list1已经存在则从栈底追加到list1里

LRANGE list1 0 2：查看list1中的前3个数据

LRANGE list1 0 -1：查看所有的数据

LPOP list1：返回栈顶的数值，并且删除，栈顶的数据就是最新增到list1的数据

RPOP list1: 返回栈底的数值，并且删除

LINDEX list1 2:根据角标位置2获取该位置数据

LLEN list1:查看list1的长度

LTRIM list1 2 4:截取角标2-4的值，重新赋给list1

LREM list1 0 3:删除值为3的数据

LINSERT list1 before 5 34:在值为5的后面插入34

**Set类型：**

其他命令可以进官网查看

地址：<https://redis.io/topics/data-types-intro>

**Zset类型：**

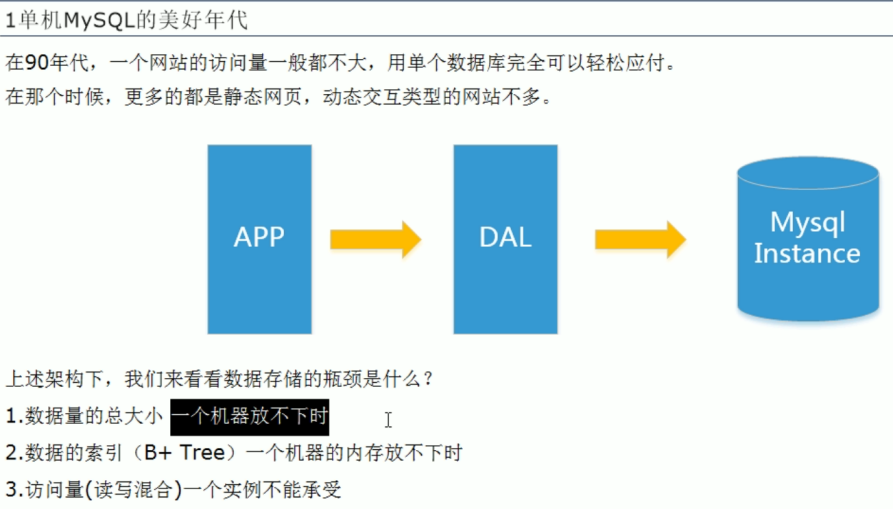
ZADD customer nx 10 v4：如果v4不存在则新增，如果存在则什么都不做。

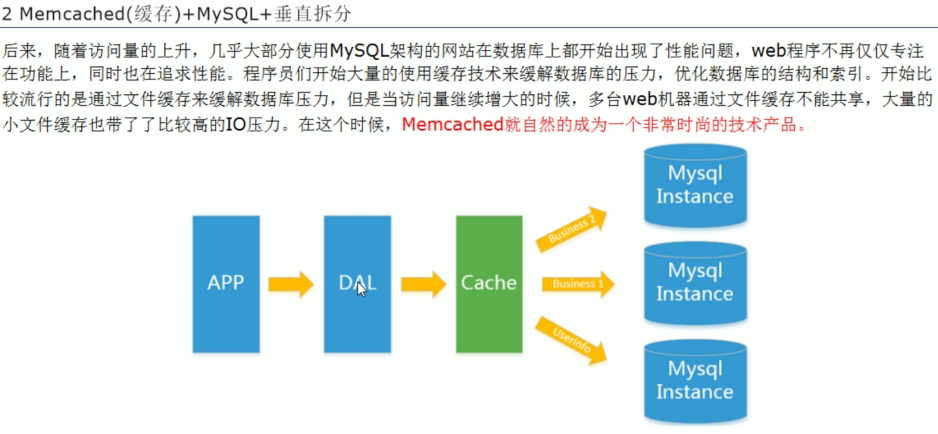
参数：nx-不存在则新增，存在则什么都不做

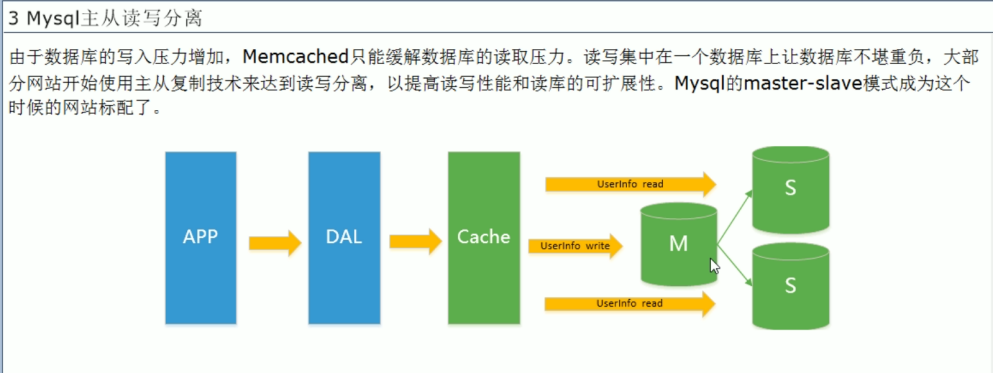
xx-不存在则新增，存在则更新

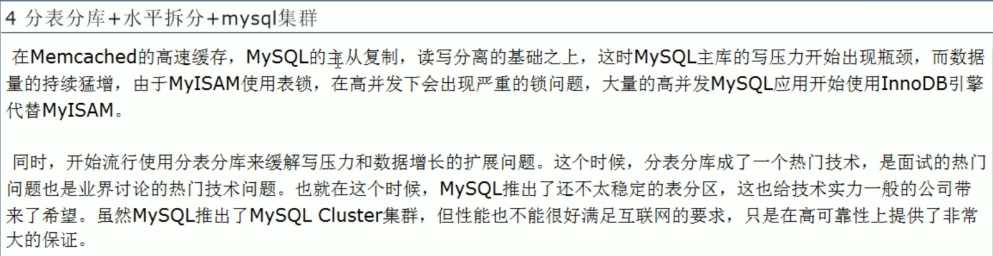
incr-不存在则新增，存在则累加更新

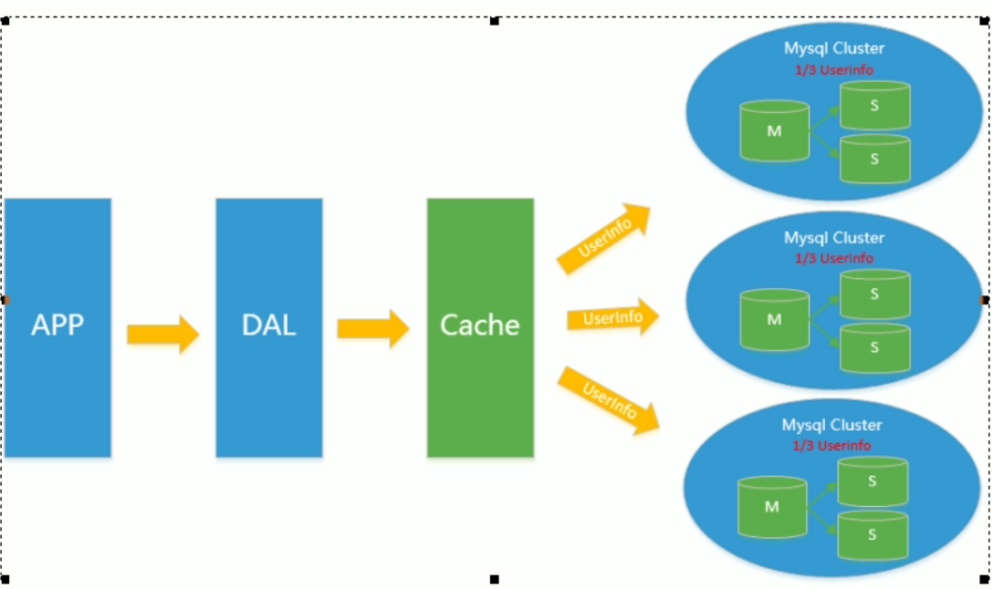
1. **数据库使用的演化**

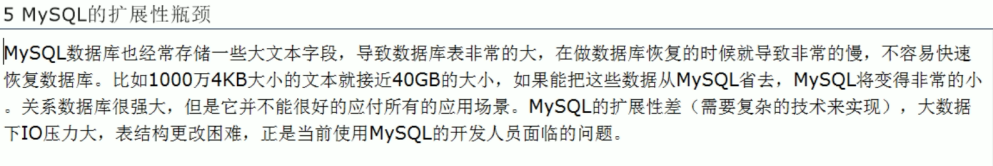


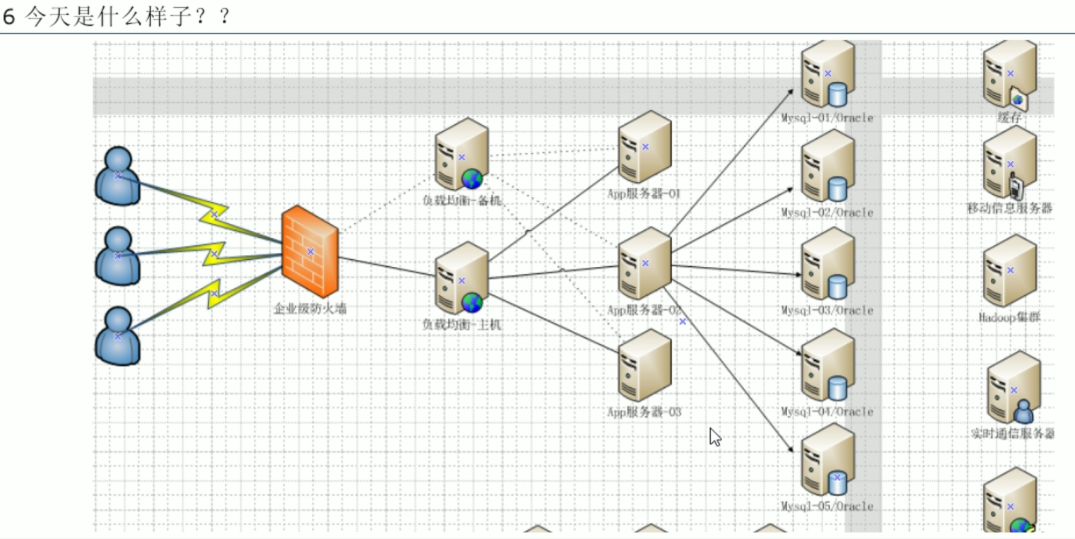


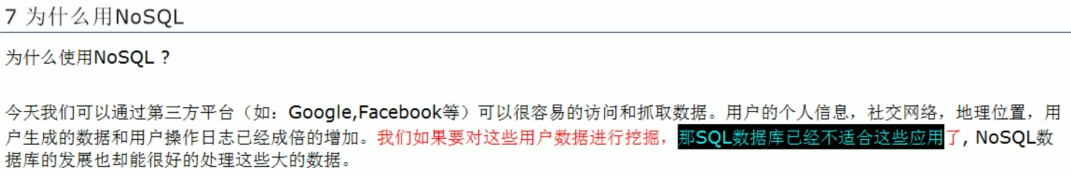












1. **nosql和传统关系型数据库的比较**

nosql优势;

(1)大数据量高性能，nosql数据库都具有非常高的读写性能，尤其在大数据下，同样表现优秀，这得益于它的无关系性，数据库的结构简单。一般MySQL使用query cache，每次表的更新cache就失效，是一种大粒度的cache，在针对web2.0的交互频繁的应用，cache性能不高，而nosql的cache是记录级的，是一种细粒度的cache，所以nosql在这个层面上来说就要性能高很多。

(2) 易扩展，nosql数据库种类繁多，但是有一个共同特点都是去掉了关系数据库的关系特性，数据之间无关系，这样就非常容易扩展，也无形之间，在架构的层面上带来了可扩展的能力。

(3)多样灵活的数据模型，nosql无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式，而在关系数据库里，增删字段是一件非常麻烦的事情，如果是非常大数据量的表，增加字段简直就是一场噩梦。

**RDBMS关系型数据库特点：**

高度组织化结构化数据

结构化查询语言sql

数据和关系都存储在单独的表中

数据操纵语言，数据定义语言

严格的一致性

基础事务

**nosql的特点：**

代表着不仅仅是sql

没有声明性查询语言

没有预定义的模式

键-值对存储，列存储，文档存储，图形数据库

最终一致性，而非ACID属性

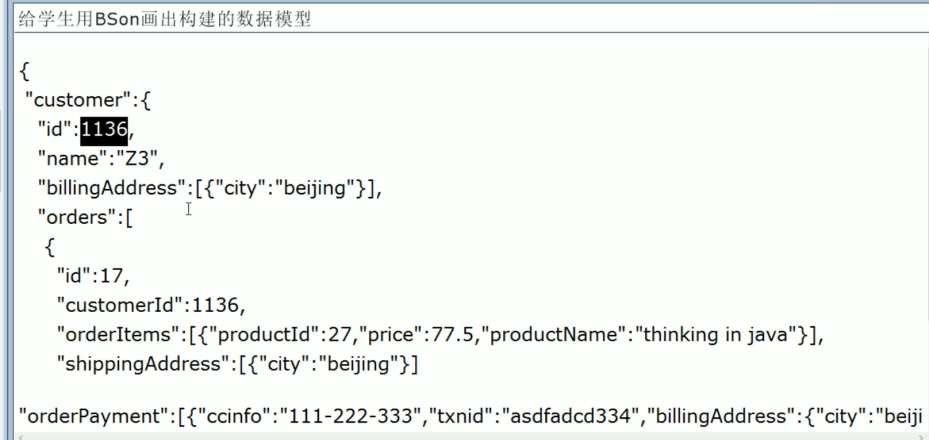
非结构化和不可预知的数据

CAP定理：Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性）

高性能，高可用性和可伸缩性

1. **nosql数据模型**

聚合模型：KV键值、Bson（类似于json）、列族、图形，前两个最重要



因为数据都放在一个json里面，根据用户id就能把所有的数据拿出来，所以不存在传统关系型数据库那种join操作，查询很快

1. **nosql四大分类**

KV键值：redis

文档型数据库：MongoDB

列存储数据库：Cassandra,HBase

图关系数据库：放的社交关系，广告推荐，社交网络，推荐系统，专注于构建关系图谱，Neo4j,infoGrid



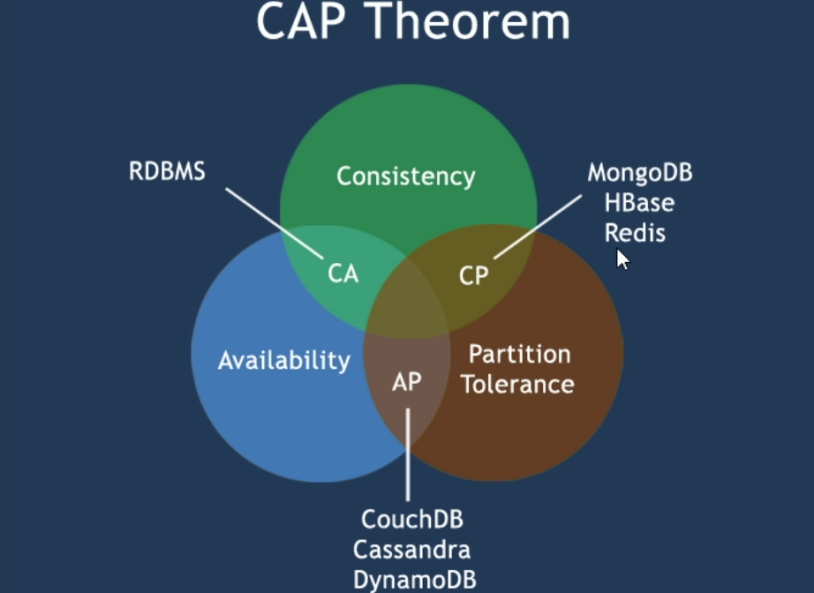
1. **CAP和BASE原理**

CAP:Consistency、Availability、Partition tolerance。CAP的核心理论是一个分布式系统不可能同时很好的满足一致性、可用性和分区容错性三个需求，最多只能同时较好的满足两个。因此根据cap原理将nosql数据库分成了满足CA原则，满足CP原则和满足AP原则三大类。

C:强一致性

A:可用性

P:分区容错性（nosql必须实现的）



CA:单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大

CP:满足一致性，分区容忍性的系统，通常性能不是特别高

AP:满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。（大多数网站都是这种）

注：数据库一般都要一致性，分区容错性，因为数据必须一致，所以牺牲了可用性，比如redis，但是我们做系统或者服务器的时候最好是AP。

数据库事务一致性需求：

很多web实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写的一致性要求并不高，允许实现最终一致性。

数据库的写实时性和读实时性需求：

对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比方说发一条消息之后，过几秒乃至十几秒之后，我的订阅者才看到这条消息是完全可以接受的。

对复杂的sql查询，特别是多表关联查询的需求：

任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的报表查询，特别是SNS类型的网站，从需求以及产品设计角度，就避免了这种情况的产生，往往更多的只是单表的主键查询，以及单表的简单条件分页查询，sql的功能被极大的弱化了。

BASE：为了解决关系数据库强一致性引起的问题而引起可用性降低而提出的解决方案。

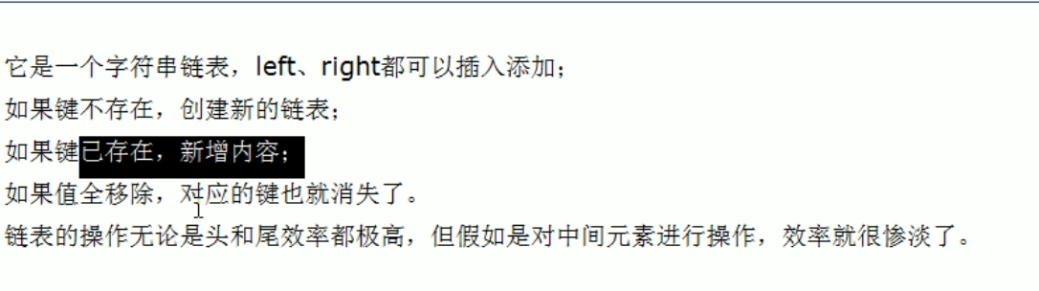
BASE：基本可用、软状态、最终一致性。

它的思想是通过让系统放松对某一时刻数据一致性的要求来换取系统整体伸缩性和性能上改观，为什么这么说呢，缘由就在于大型系统往往由于地域分布和极高性能的要求，不可能采用分布式事务来完成这些指标，要想获得这些指标，我们必须采用另外一种方式来完成，这里BASE就是解决这个问题的办法。

1. **数据类型**
2. String类型，String是Redis最基本的类型，你可以理解与memcached一模一样的类型，一个key对应一个value，String类型是二进制安全的，意思是Redis的String可以包含任何数据，比如jpg图片或者序列化对象。一个Redis中字符串value最多可以是512M。

(2)Hash哈希，类似于Java的Map,redis hash是一个键值对集合，redis hash是一个String类型的field和value的映射表，hash特别适合用于存储对象。

(3)list,redis列表是简单的字符串列表，按照插入顺序排序，你可以添加一个元素到列表的头部或者尾部，它的底层实际是个链表。

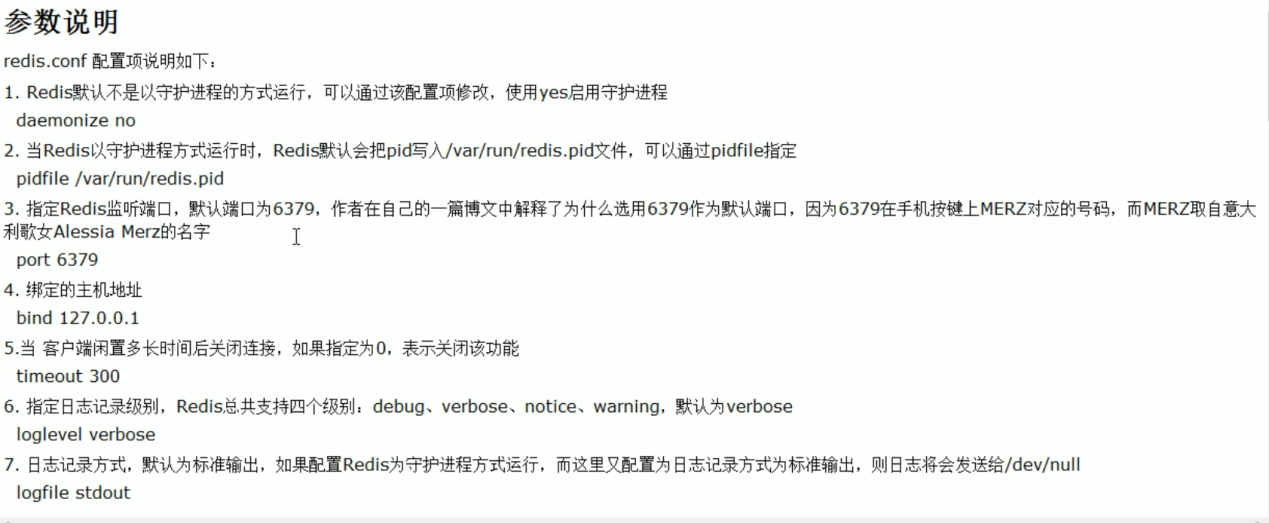


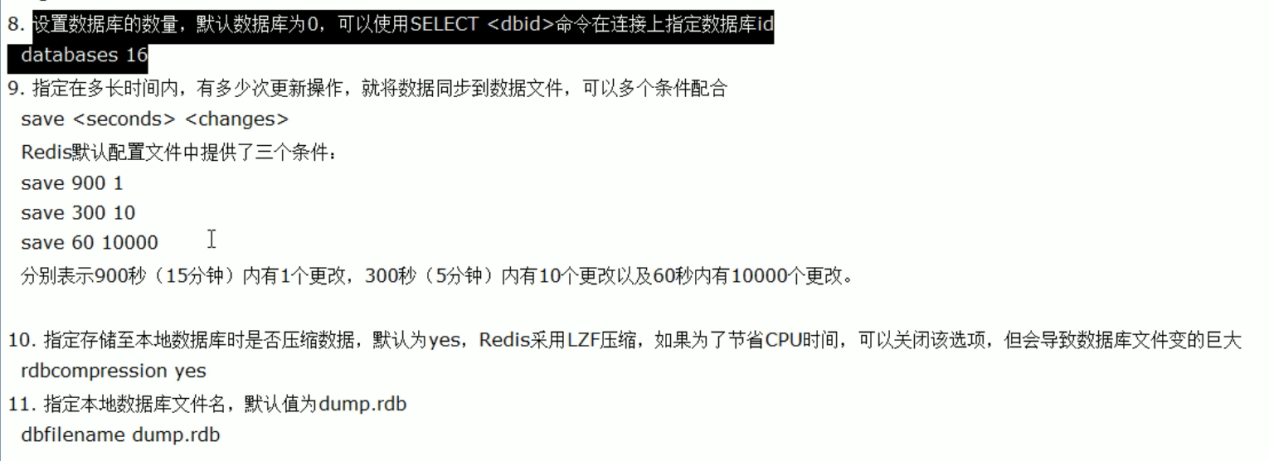
(4)set，reids的set是String类型的无序集合，它是通过hashtable实现的。

(5)Zset:sorted set有序集合，redis zset和set一样也是String类型元素的集合，且不允许重复的成员，不同的是每个元素都会关联一个double类型的分数，redis正是通过分数来为集合中的成员进行从小到大的排序，zset的成员是唯一的，但分数却可以重复。（比如游戏中是通过积分来进行排名的）

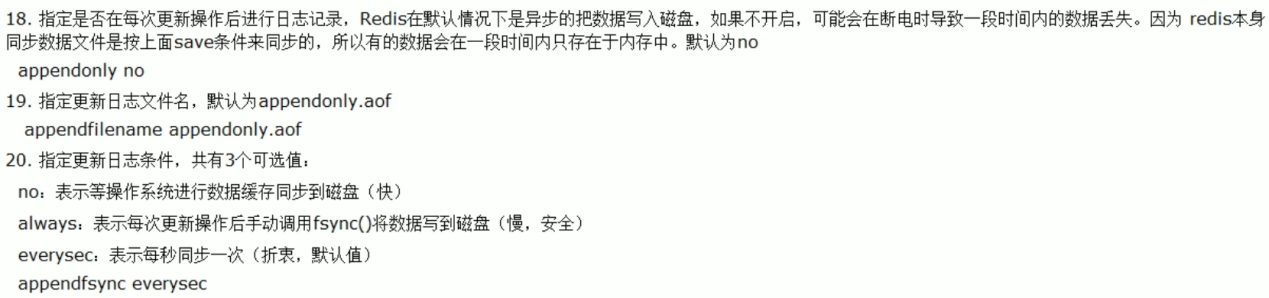
1. **配置文件解析**

redis.conf的内容解析

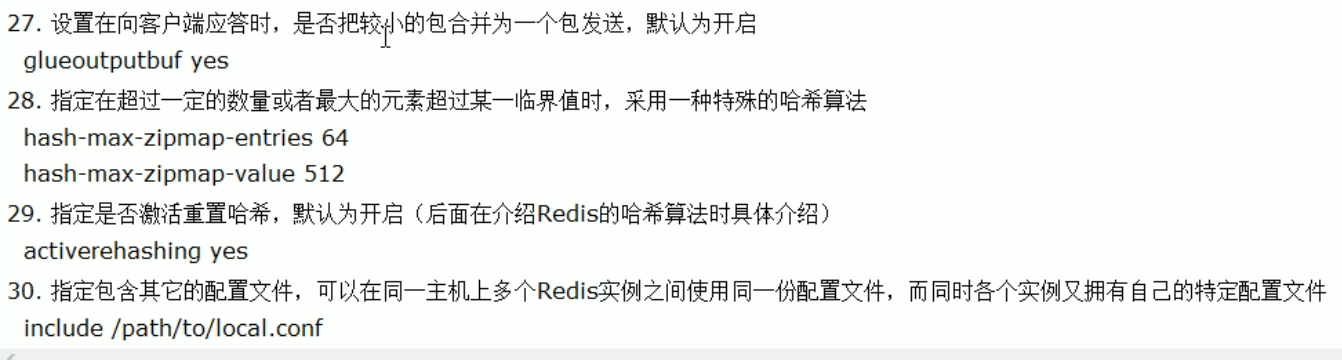












**General：通用标准配置**

port:配置端口号

pidfile:

tcp-backlog:设置tcp的backlog其实是一个连接队列，backlog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。

timeout:关闭客户端连接的时间，设置50，那么客户端有50秒是空闲的则关闭，0表示一直连接，不关闭

tcp-keepalive:单位是秒，如果设置为0，则不会进行keepalive检测，建议设置为60秒（心跳检测），检测tcp连接是否是通畅的

loglevel：日志级别（notice，debug，warning，verbose）

logfile:日志文件路径

syslog-enabled:系统日志，默认是关闭

syslog-ident:设置系统日志的开头为redis，相当于是redis.log

syslog-facility:输出日志的设备

databases:设置redis数据库个数，默认是16个，默认是0号库

**SNAPSHOTTING:快照配置**

与RDB和AOF相关

dbfilename dump.rdb:redis如果宕机，重启启动的时候，redis会默认去加载dump.rdb文件，dump.rdb文件是数据持久化的文件

stop-writes-on-bgsave-error yes:fork进程在进行快照存储的时候报错要停止主进程的写操作。设置成no，一般情况是数据一致要求比较低的情况

rdbcompression yes：是否压缩，一般可以设置为不压缩，减小cpu压力

rdbchecksum：在存储快照后，还可以让Redis使用CRC64算法进行数据校验，但是这样做会增加大约10%的性能消耗，如果希望获取到最大的性能提升，可以关闭此功能

**SECURITY安全设置**

**LIMITS限制：**

maxclients:最大连接数

maxmemory:最大内存

maxmemory-policy:当达到最大内存的时候，就会根据这个配置的策略清理内存的数据。



maxmemory-samples:设置样本数量

1. **主从复制**

**复制原理：**

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

1）从服务器连接主服务器，发送SYNC命令；

2）主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；

3）主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；

4）从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；

5）主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；

6）从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；



增量复制：Redis增量复制是指Slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。 增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。

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

**本机多个实例复制：**

一个服务器上开启多个redis实例，一个做master，其他做slave

一主一备：新增配置文件，redis\_slave.conf（修改日志文件名称，端口号为6380，pid等）

主机启动：redis-server redis.conf

从机启动：redis-server redis\_slave.conf

登录从机：redis-cli –p 6380

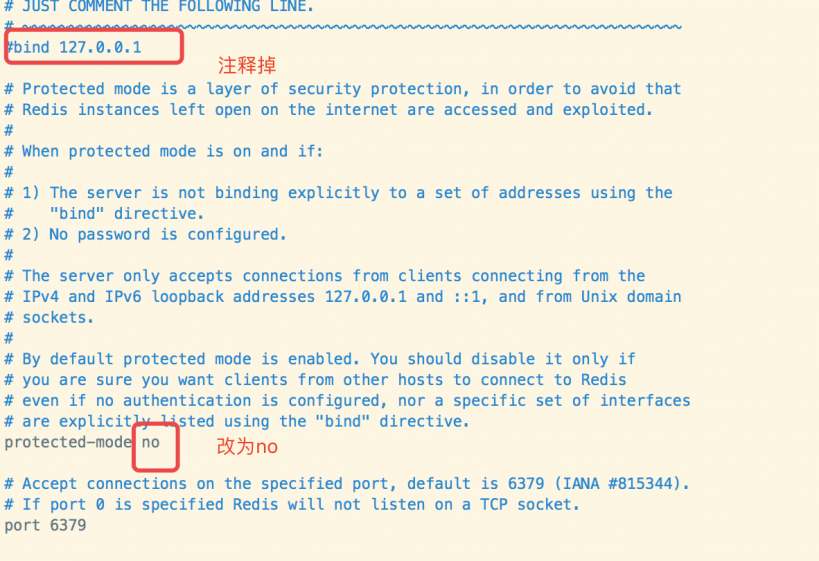
输入命令：SLAVEOF 127.0.0.1 6379

此时可以查看从机状态：info replication



**跨服务器主从复制：**

主机修改redis.conf配置文件：



主机开放6379端口：

Ubuntu系统：ufw allow 6379

ufw status查看状态

1. redis主从复制，比如主机已经有数据，后面开启的从机，从机会复制之前的所有数据，而不只是复制开启后的数据。也就是第一次复制全量数据，后面是复制增量的数据
2. redis主机如果挂了，从机不会自动提升为主机，而是等待。直到主机重新启动，恢复正常，会自动连接到主机
3. 从机如果挂了，如果是上面这种配置，从机恢复后不能再继续复制主机数据，如果想从机启动后还能继续复制主机数据，则需要在配置文件redis.conf里面配置
4. 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
5. 中途变更转向，会清除之前的数据，重新建立拷贝最新的slaveof新主库的数据

**从机变主机：**

主机挂掉，我们提升从机A变主机，从机B跟随从机A

从机A命令：

slaveof no one

此时从机A就变主机了，然后从机B设置跟随主机A

slaveof 172.16.25.80 6379

**哨兵模式：**

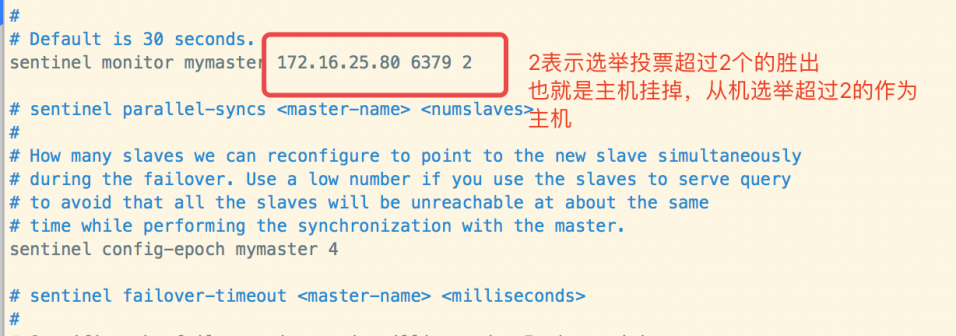
Sentinel(哨兵)进程是用于监控redis集群中Master主服务器工作的状态，在Master主服务器发生故障的时候，可以实现Master和Slave服务器的切换，保证系统的高可用，其已经被集成在redis2.6+的版本中，Redis的哨兵模式到了2.8版本之后就稳定了下来。一般在生产环境也建议使用Redis的2.8版本的以后版本。哨兵(Sentinel) 是一个分布式系统，你可以在一个架构中运行多个哨兵(sentinel) 进程，这些进程使用流言协议(gossipprotocols)来接收关于Master主服务器是否下线的信息，并使用投票协议(Agreement Protocols)来决定是否执行自动故障迁移,以及选择哪个Slave作为新的Master。每个哨兵(Sentinel)进程会向其它哨兵(Sentinel)、Master、Slave定时发送消息，以确认对方是否”活”着，如果发现对方在指定配置时间(可配置的)内未得到回应，则暂时认为对方已掉线，也就是所谓的”主观认为宕机” ，英文名称：Subjective Down，简称SDOWN。有主观宕机，肯定就有客观宕机。当“哨兵群”中的多数Sentinel进程在对Master主服务器做出 SDOWN 的判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的Master Server下线判断，这种方式就是“客观宕机”，英文名称是：Objectively Down， 简称 ODOWN。通过一定的vote算法，从剩下的slave从服务器节点中，选一台提升为Master服务器节点，然后自动修改相关配置，并开启故障转移（failover）。

单机配置：

一个服务器上开启3个redis示例，主机A，从机B、C,启动一个哨兵。

开启哨兵模式

修改哨兵配置文件：sentinel.conf



注：我们在多重启几次哨兵，会发现sentinel.conf配置文件尾部会多出一些可能不太准确的信息，此时要是哨兵不起作用，需要去手动删除掉一些无用的信息，再重启哨兵。

主机宕机，会马上选出一个新的主机

集群模式：

主：172.16.25.80，从：172.16.25.188和172.16.25.1，哨兵单台：部署在:172.16.25.80

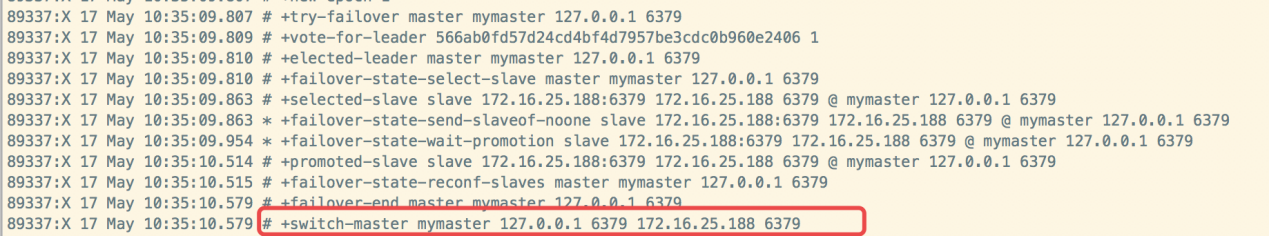
从机redis配置：在redis.conf文件中新增slaveof 172.16.25.80或者客户端登录redis输入命令slaveof 172.16.25.80，建议使用命令形式，如果在配置文件中配死，之前我遇到一些莫名其妙的问题（主机80宕机，哨兵选出188作为主机，80重启作为从机，但是当188宕机的时候，80选为主机，此时从机跟随80主机没问题，80主机的信息显示也是跟随80主机，他应该是作为主机而存在，并不应该跟随自己）

修改哨兵配置文件：sentinel.conf

sentinel monitor mymaster 172.16.25.80 6379 1

启动主从redis服务器，然后启动哨兵

此时主机可以修改内容，看看从机能不能正常同步到数据，然后我们人为造成主redis服务器宕机，看看从机能不能自动提升为主机。



如上所示表示切换成功

此时我们再重新启动宕机的主redis服务器，发现主redis服务器会成为新的slave，然后跟随新主机，而不是自己成为主机。

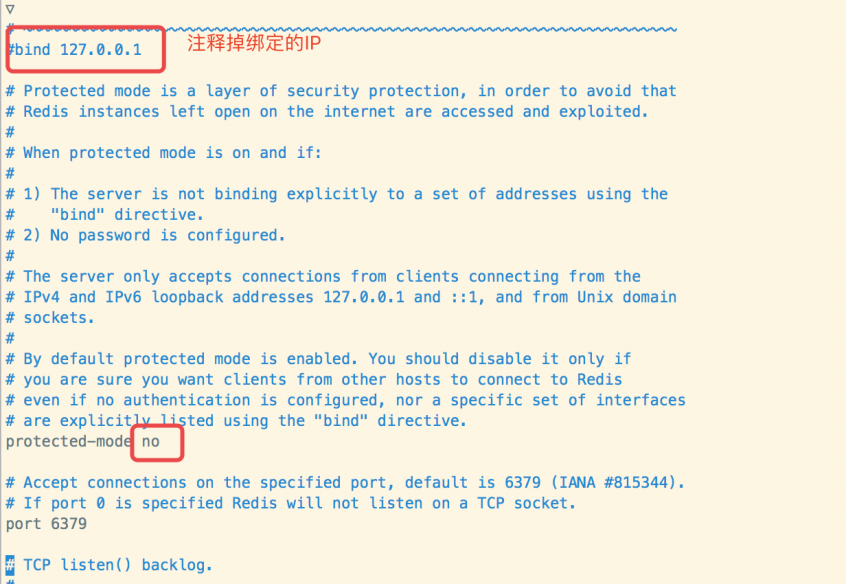
多哨兵：

1. **redis开启远程访问**

描述：如果想远程访问某个机器上的redis服务，可以通过下面的步骤实现

第一步：防火墙开放端口6379

第二步：Redis自身开启远程访问，此时需要修改redis.conf配置文件实现，修改完配置文件，重启服务



1. **Redis中数据穿透和雪崩**

一般情况下，不止Redis才会出现数据穿透和雪崩，只要是拿来做缓存的框架，都会存在这个问题。因此，不管缓存是Redis还是其他的，解决方法都是近似的。

**数据穿透/缓存穿透：**

**概念**：一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

如何避免？

（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。**布隆过滤器**

**数据雪崩/缓存雪崩：**

如果缓存集中在一段时间内失效（比如重启服务），发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。

如何避免？

（1）在缓存失效后，通过加锁或者队列来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）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。

（3）做二级缓存，或者双缓存策略。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。也就是说，一个缓存失效，另外一个缓存还有效，这样就避免了直接访问数据库。

**缓存数据的淘汰更新：**

也就是说某些缓存数据我们需要去更新到数据库最新的，需要把之前的数据删除，换成刚从数据库查询到的数据。

缓存淘汰的策略有两种：

（1）定时去清理过期的缓存。

（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂，具体用哪种方案，大家可以根据自己的应用场景来权衡。

1. **Redis解决高并发问题**

redis真的是一个很好的技术，它可以很好的在一定程度上解决网站一瞬间的并发量，例如商品抢购秒杀等活动。

redis之所以能解决高并发的原因是它可以直接访问内存，而以往我们用的是数据库(硬盘)。Redis提高了访问效率,解决了数据库服务器压力。

**抢购秒杀：**

用redis解决瞬间秒杀活动

如果20w人一瞬间涌入这个页面进行秒杀,能够秒杀成功的只有500人,我们把先进来的用户放入redis队列中,当队列中的用户达到500时，后来用户就转到秒杀结束页面。

**并发问题：**

问题描述：假如有某个key = "price"， value值为10，现在想把value值进行+10操作。正常逻辑下，就是先把数据key为price的值读回来，加上10，再把值给设置回去。如果只有一个连接的情况下，这种方式没有问题，可以工作得很好，但如果有两个连接时，两个连接同时想对还price进行+10操作，就可能会出现问题了。

例如：两个连接同时对price进行写操作，同时加10，最终结果我们知道，应该为30才是正确。

T1时刻，连接1将price读出，目标设置的数据为10+10 = 20。

T2时刻，连接2也将数据读出，也是为10，目标设置为20。

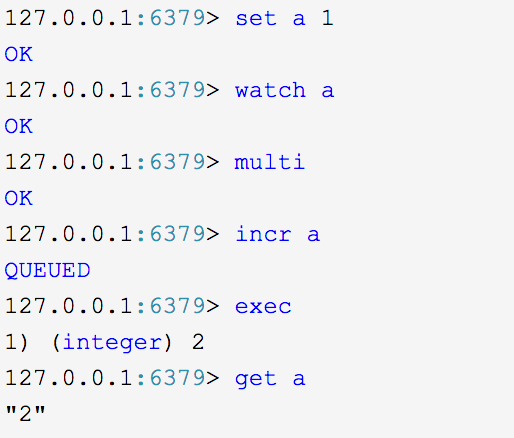
T3时刻，连接1将price设置为20。

T4时刻，连接2也将price设置为20，则最终结果是一个错误值20。

解决方法：

（1）redis有一个watch命令，这个命令可以解决这个问题。

看下面的例子，对一个键执行watch，然后执行事务，由于watch的存在，他会监测键a，当a被修该之后，后面的事务就会执行失败，这就确保了多个连接同时来了，都监测着a，只有一个能执行成功，其他都返回失败。



（2）redis中命令是满足原子性的，因此在值为阿拉伯数字的时候，我可以将get和set命令修改为incr或者incrby来解决这个问题。

1. **Redis为什么采用单线程**

注：单线程指的是网络请求模块使用了一个线程（所以不需考虑并发安全性），即一个线程处理所有网络请求，其他模块仍用了多个线程。

以前一直有个误区，以为：高性能服务器一定是多线程来实现的

原因很简单因为误区二导致的： 多线程一定比单线程效率高。其实不然（比如在单核处理器上）。

**单线程优点：**

代码更清晰，处理逻辑更简单

不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

不存在多进程或者多线程导致的切换而消耗CPU

**单线程缺点**：

无法发挥多核CPU性能，不过可以通过在单机开多个Redis实例来完善；

回到正题：为什么redis采用单线程？

redis 核心就是如果我的数据全都在内存里，我单线程的去操作 就是效率最高的，为什么呢，因为多线程的本质就是 CPU 模拟出来多个线程的情况，这种模拟出来的情况就有一个代价，就是上下文的切换，对于一个内存的系统来说，它没有上下文的切换就是效率最高的。redis 用 单个CPU 绑定一块内存的数据，然后针对这块内存的数据进行多次读写的时候，都是在一个CPU上完成的，所以它是单线程处理这个事。在内存的情况下，这个方案就是最佳方案。

总结：

（1）纯内存操作；

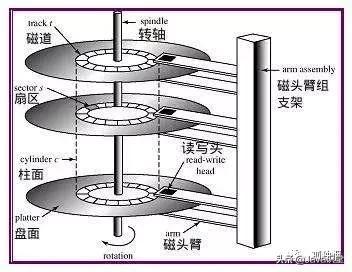
（2）多线程仍然会有上下文切换的损耗，虽然比进程切换损耗小；

（3）采用了非阻塞I/O多路复用机制

**什么情况用多线程？**

在I/O操作的时候，例如磁盘I/O，网络I/O等！为什么一般是在I/O操作的时候，要用多线程呢？因为I/O操作一般可以分为两个阶段:即等待I/O准备就绪和真正操作I/O资源！"

"以磁盘操作为例，磁盘的结构如下"



在磁盘上数据是分磁道、分簇存储的，而数据往往并不是连续排列在同一磁道上，所以磁头在读取数据时往往需要在磁道之间反复移动，因此这里就有一个寻道耗时！另外，盘面旋转将请求数据所在扇区移至读写头下方也是需要时间，这里还存在一个旋转耗时！

那么，在这一时间段（即"I/O等待"）内，线程是在“阻塞”着等待磁盘，此时操作系统可以将那个空闲的CPU核心用于服务其他线程。因此在I/O操作的情况下，使用多线程，效率会更高！。

Redis读写数据有涉及到I/O操作么？

Redis是内存数据库，不涉及I/O操作，因此设计为单线程是效率最高的！那么，既然你知道既然Redis的性能和CPU无关，那你知道Redis的性能瓶颈在哪么？

一般在两个地方

其一是机器内存大小，内存大小关系到Redis存储的数据量

其二是网络带宽

Redis客户端执行一条命令分为四个过程：发送命令、命令排队、命令执行、返回结果。

而其中发送命令+返回结果这一过程被称为Round Trip Time（RTT，往返时间）。

Redis的客户端和服务端可能部署在不同的机器上。例如客户端在北京，Redis服务端在上海，两地直线距离约为1300公里，那么1次RTT时间=1300×2/（300000×2/3）=13毫秒（光在真空中传输速度为每秒30万公里，这里假设光纤为光速的2/3），那么客户端在1秒内大约只能执行80次左右的命令，这就和Redis的高并发高吞吐特性背道而驰啦！所以一般情况下，都是就近部署！

1. **Redis为什么快**

1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；

2、数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；

3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4、使用多路I/O复用模型，非阻塞IO；

5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

**多路 I/O 复用模型（网络）**

多路I/O复用模型是利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。

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

1. **Redis跳跃表**

跳跃表原理

1. **Bigkey问题**

什么是bigkey?

* 一个STRING类型的Key，它的值为5MB（数据过大）
* 一个LIST类型的Key，它的列表数量为20000个（列表数量过多）
* 一个ZSET类型的Key，它的成员数量为10000个（成员数量过多）
* 一个HASH格式的Key，它的成员数量虽然只有1000个但这些成员的value总大小为100MB（成员体积过大）

Bigkey的危害:

* Redis Cluster中的某个node内存远超其余node，但因Redis Cluster的数据迁移最小粒度为Key而无法将node上的内存均衡化
* 超时阻塞:由于Redis单线程的特性，操作bigkey的通常比较耗时，也就意味着阻塞Redis可能性越大，这样会造成客户端阻塞或者引起故障切换，它们通常出现在慢查询中。
* Client发现Redis变慢
* Redis内存不断变大引发OOM，或达到maxmemory设置值引发写阻塞或重要Key被逐出
* 大Key上的读请求使Redis占用服务器全部带宽，自身变慢的同时影响到该服务器上的其它服务
* 删除一个大Key造成主库较长时间的阻塞并引发同步中断或主从切换

解决方法:

* 拆分
* 对大Key进行清理
* 时刻监控Redis的内存水位
* 对失效数据进行定期清理

1. **HotKey问题**

什么是hotkey?

在某个Key接收到的访问次数、显著高于其它Key时，我们可以将其称之为热Key，常见的热Key如：

* 某Redis实例的每秒总访问量为10000，而其中一个Key的每秒访问量达到了7000（访问次数显著高于其它Key）
* 对一个拥有上千个成员且总大小为1MB的HASH Key每秒发送大量的HGETALL（带宽占用显著高于其它Key）
* 对一个拥有数万个成员的ZSET Key每秒发送大量的ZRANGE（CPU时间占用显著高于其它Key）

Hotkey的危害?

* 热Key占用大量的Redis CPU时间使其性能变差并影响其它请求
* Redis Cluster中各node流量不均衡造成Redis Cluster的分布式优势无法被Client利用，一个分片负载很高而其它分片十分空闲从而产生读/写热点问题
* 在抢购、秒杀活动中，由于商品对应库存Key的请求量过大超出Redis处理能力造成超卖
* 热Key的请求压力数量超出Redis的承受能力造成缓存击穿，此时大量强求将直接指向后端存储将其打挂并影响到其它业务

解决方法:

* 在Redis Cluster结构中对热Key进行复制:在Redis Cluster中，热Key由于迁移粒度问题造成请求无法打散使单一node的压力无法下降。此时可以将对应热Key进行复制并迁移至其他node.该方案的缺点在于代码需要联动修改，同时，Key一变多带来了数据一致性挑战：由更新一个Key演变为需要同时更新多个Key，在很多时候，该方案仅建议用来临时解决当前的棘手问题。
* 使用读写分离架构:如果热Key的产生来自于读请求，那么读写分离是一个很好的解决方案。在使用读写分离架构时可以通过不断的增加从节点来降低每个Redis实例中的读请求压力。读写分离架构同样存在缺点，在请求量极大的场景下，读写分离架构会产生不可避免的延迟，此时会有读取到脏数据的问题，因此，在读写压力都较大写对数据一致性要求很高的场景下，读写分离架构并不合适。

1. **啊啊啊**