# 概述

远程字典服务器Redis（**Remote Dictionary Server**）是一个完全开源免费的，用C语言编写的，遵守BSD协议的高性能（key/value）分布式内存数据库，基于内存运行并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之一，也被称为数据结构服务器。

Redis本质上是Key-Value类型的内存数据库。

它可以用作数据库、缓存和消息中间件，支持多种类型的数据结构，如strings、hashes、lists、sets、和sorted sets与范围查询、bitmaps、hyperloglogs 和地理空间（geospatial）索引半径查询。Redis内置了复制（replication），LUA脚本（Lua scripting），LRU驱动事件（LRU eviction），事务（transactions） 和不同级别的磁盘持久化（persistence），并通过Redis哨兵（Sentinel）和自动分区（Cluster）提供高可用性（high availability）。

Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。在内存越来越便宜的今天，redis将会越来越受欢迎。 如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

## NoSQL

NoSQL(**Not only SQL**)是传统关系型数据库的补充而非替代，是非关系型数据库，数据存储不需要固定的模式，无需多余操作就可以进行横向扩展。在整个NoSQL技术栈中MemCache、Redis、MongoDB被称为NoSQL三剑客。NoSQL数据库的最大优势体现为：高性能、高可用性和可伸缩性。

|  | **关系型数据库** | **NoSQL数据库** |
| --- | --- | --- |
| 数据存储位置 | 硬盘 | 内存 |
| 数据结构 | 高度组织化结构化数据 | 没有预定义的模式 |
| 数据操作方式 | SQL | 所有数据都是键值对，没有声明性查询语言 |
| 事务控制 | 严格的基础事务ACID原则 | CAP定理 |

修改频率低的冷数据用SQL，修改频率高的数据用NoSQL。

### NoSql数据库分类

NoSQL可以大体上分为4个种类：**Key-value、Document-Oriented、Column-Family Databases以及 Graph-Oriented Databases。**下面就一览这些类型的特性：

**一、 键值（Key-Value）数据库**

键值数据库就像在传统语言中使用的哈希表。你可以通过key来添加、查询或者删除数据，鉴于使用主键访问，所以会获得不错的性能及扩展性。

**产品：Riak、Redis、Memcached、Amazon’s Dynamo、Project Voldemort**

**有谁在使用：**GitHub （Riak）、BestBuy （Riak）、Twitter （Redis和Memcached）、StackOverFlow （Redis）、 Instagram （Redis）、Youtube （Memcached）、Wikipedia（Memcached）

**适用的场景**

储存用户信息，比如会话、配置文件、参数、购物车等等。这些信息一般都和ID（键）挂钩，这种情景下键值数据库是个很好的选择。

**不适用场景**

**1.**取代通过键查询，而是通过值来查询。Key-Value数据库中根本没有通过值查询的途径。

**2.**需要储存数据之间的关系。在Key-Value数据库中不能通过两个或以上的键来关联数据。

**3.**事务的支持。在Key-Value数据库中故障产生时不可以进行回滚。

**二、 面向文档（Document-Oriented）数据库**

面向文档数据库会将数据以文档的形式储存。每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个名称与对应的值，值既可以是简单的数据类型，如字符串、数字和日期等；也可以是复杂的类型，如有序列表和关联对象。数据存储的最小单位是文档，同一个表中存储的文档属性可以是不同的，数据可以使用XML、JSON或者JSONB等多种形式存储。

**产品：**MongoDB、CouchDB、RavenDB

**有谁在使用：**SAP （MongoDB）、Codecademy （MongoDB）、Foursquare （MongoDB）、NBC News （RavenDB）

**适用的场景**

**1.** 日志。企业环境下，每个应用程序都有不同的日志信息。Document-Oriented数据库并没有固定的模式，所以我们可以使用它储存不同的信息。

**2.**分析。鉴于它的弱模式结构，不改变模式下就可以储存不同的度量方法及添加新的度量。

**不适用场景**

在不同的文档上添加事务。Document-Oriented数据库并不支持文档间的事务，如果对这方面有需求则不应该选用这个解决方案。

**三、 列存储（Wide Column Store/Column-Family）数据库**

列存储数据库将数据储存在列族（column family）中，一个列族存储经常被一起查询的相关数据。举个例子，如果我们有一个Person类，我们通常会一起查询他们的姓名和年龄而不是薪资。这种情况下，姓名和年龄就会被放入一个列族中，而薪资则在另一个列族中。

**产品：**Cassandra、HBase

**有谁在使用：**Ebay （Cassandra）、Instagram （Cassandra）、NASA （Cassandra）、Twitter （Cassandra and HBase）、Facebook （HBase）、Yahoo!（HBase）

**适用的场景**

**1.**日志。因为我们可以将数据储存在不同的列中，每个应用程序可以将信息写入自己的列族中。

**2.**博客平台。我们储存每个信息到不同的列族中。举个例子，标签可以储存在一个，类别可以在一个，而文章则在另一个。

**不适用场景**

**1.**如果我们需要ACID事务。Vassandra就不支持事务。

**2.** 原型设计。如果我们分析Cassandra的数据结构，我们就会发现结构是基于我们期望的数据查询方式而定。在模型设计之初，我们根本不可能去预测它的查询方式，而一旦查询方式改变，我们就必须重新设计列族。

**四、 图（Graph-Oriented）数据库**

图数据库允许我们将数据以图的方式储存。实体会被作为顶点，而实体之间的关系则会被作为边。比如我们有三个实体，Steve Jobs、Apple和Next，则会有两个“Founded by”的边将Apple和Next连接到Steve Jobs。

**产品：**Neo4J、Infinite Graph、OrientDB

**有谁在使用：**Adobe （Neo4J）、Cisco （Neo4J）、T-Mobile （Neo4J）

**适用的场景**

**1.**在一些关系性强的数据中

**2.**推荐引擎。如果我们将数据以图的形式表现，那么将会非常有益于推荐的制定

**不适用场景**

不适合的数据模型。图数据库的适用范围很小，因为很少有操作涉及到整个图。

### CAP和BASE

#### ACID

原子性（**aotmicity**）、一致性（**consistency**）、隔离性（**isolation**）和持久性（**durability**）。

#### CAP

CAP指的是强一致性（**consistency**）、可用性（**availability**）和分区容错性（**partition tolerance**）。

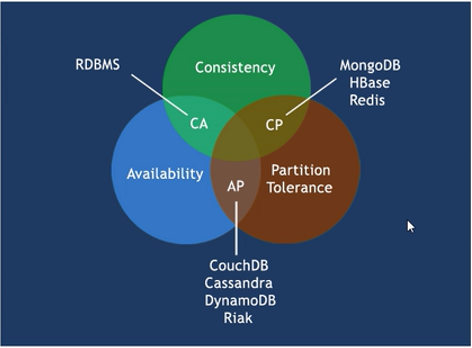
CAP理论的核心是：一个分布式系统不可能同时很好的满足强一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。

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：

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，分区容错性的系统，性能不是特别高。

AP-满足可用性，分区容错性的系统，通常可能对一致性要求低一些。



由于当前网络硬件肯定会出现延迟丢包等问题，所以分区容错性是必须需要实现的。所以只能在一致性（c）和可用性（a）之间权衡，没有NoSql系统能同时保证这三点。

**数据库事务一致性的需求**

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行需求**

对于关系数据库来说，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动态是完全可以接受的。（读己知所写之，自己先看到，订阅者再慢慢看到）

**对于复杂的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

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

BASE其实是下面三个属于的缩写：

基本可用：**Basically Available**

软状态：**Soft state**

最终一致：**Eventually consistent**

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区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，我们必须采用另一种方式来完成，这里BASE就是解决这些问题的办法。

**分布式系统**

分布式系统（**distributed system**）是由于多台计算机和通信的软件组件通过计算机网络连接（本地网络或广域网）组成。分布式系统是建立在网络之上的软件系统。正是因为软件的特性，所以分布式系统具有高度的内聚性和透明性。因此，网络和分布式系统之间的区别更多的在于高层软件（特鄙视操作系统），而不是硬件。分布式系统可以应用在不同平台上如PC，工作站，局域网，和广域网上等。

1分布式：不同的多台服务器上面部署不同的服务模块（工程），他们之间通过Rpc、Rmi之间通信和调用，对外提供服务和组内协作。

2集群：不同的多台服务器上面部署相同的服务模块，通过分布式跳读软件进行统一的调度，对外提供服务和访问。

## Linux系统下安装Redis

1、上传Redis的tar包并解压

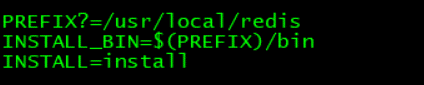
[root@xmm opt]# tar -zxvf redis-4.0.2.tar.gz

2、执行yum install -y gcc-c++安装Redis需要的环境，此过程全部自动化且需要联网

[root@xmm opt]# yum install -y gcc-c++

3、使用vim编辑器打开Redis解压目录下的src目录中的Makefile，将PREFIX修改为/usr/local/redis。修改的目的是为了让redis的运行程序不要和其他文件混杂在一起。

[root@xmm src]# vim Makefile



4、进入redis解压目录执行make install命令进行redis的编译安装

[root@xmm redis-4.0.2]# make INSTALL

5、将redis解压目录下的redis.conf配置文件复制到/usr/local/redis/目录下

[root@xmm redis-4.0.2]# cp /opt/redis-4.0.2/redis.conf /usr/local/redis

6、使用vim编辑器打开redis.conf配置文件修改其中的daemonize、logfile、dir配置

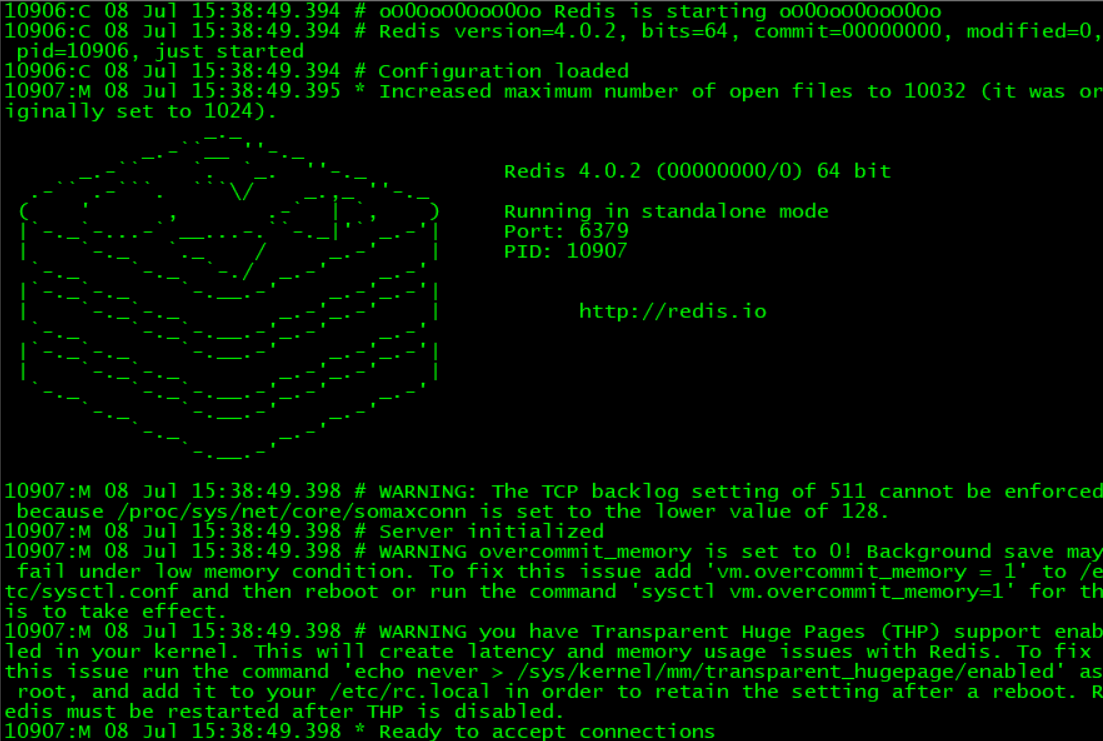
[root@xmm redis-4.0.2]# vim /usr/local/redis/redis.conf

| **配置项名称** | **作用** | **取值** |
| --- | --- | --- |
| daemonize | 控制是否以守护进程形式运行Redis服务器 | yes |
| logfile | 指定日志文件位置 | "/var/logs/redis.log" |
| dir | Redis工作目录 | /usr/local/redis |

/var/logs目录需要提前创建好。

7、让redis服务器根据指定的配置文件启动

[root@xmm redis-4.0.2]# /usr/local/redis/bin/redis-server /usr/local/redis/redis.conf



停止redis服务器

/usr/local/redis/bin/redis-cli shutdown

登陆redis客户端，redis的默认端口是6379

/usr/local/redis/bin/redis-cli [-h 192.168.235.23] [-p 6379]

### 配置redis开机自启

1、复制redis解压目录下的utils目录中的redis\_init\_script脚本到/etc/init.d目录下，并重命名为redis。

|  |
| --- |
| [root@xmm opt]# cd redis-4.0.2/  [root@xmm redis-4.0.2]# ls  00-RELEASENOTES deps README.md runtest-sentinel utils  BUGS INSTALL redis.conf sentinel.conf  CONTRIBUTING Makefile runtest src  COPYING MANIFESTO runtest-cluster tests  [root@xmm redis-4.0.2]# cd utils/  [root@xmm utils]# ls  build-static-symbols.tcl hashtable redis\_init\_script.tpl  cluster\_fail\_time.tcl hyperloglog redis-sha1.rb  corrupt\_rdb.c install\_server.sh releasetools  create-cluster lru speed-regression.tcl  generate-command-help.rb redis-copy.rb whatisdoing.sh  graphs redis\_init\_script  [root@xmm utils]# cp redis\_init\_script /etc/init.d/redis |

2、在/etc/init.d目录下修改刚刚复制过来的redis文件，通过vim编辑器打开该文件进行修改。

|  |
| --- |
| [root@xmm init.d]# vim redis  #!/bin/sh  #  # Simple Redis init.d script conceived to work on Linux systems  # as it does use of the /proc filesystem.  #!/bin/sh  #  # Simple Redis init.d script conceived to work on Linux systems  # as it does use of the /proc filesystem.  #chkconfig: 2345 80 05  REDISPORT=6379  EXEC=/usr/local/redis/bin/redis-server  CLIEXEC=/usr/local/redis/bin/redis-cli  PIDFILE=/var/run/redis\_${REDISPORT}.pid  CONF="/usr/local/redis/redis.conf"  case "$1" in  start)  if [ -f $PIDFILE ]  then  echo "$PIDFILE exists, process is already running or crashed"  else  echo "Starting Redis server..."  $EXEC $CONF  fi  ;;  stop)  if [ ! -f $PIDFILE ]  then  echo "$PIDFILE does not exist, process is not running"  else  PID=$(cat $PIDFILE)  echo "Stopping ..."  $CLIEXEC -p $REDISPORT shutdown  while [ -x /proc/${PID} ]  do  echo "Waiting for Redis to shutdown ..."  sleep 1  done  echo "Redis stopped"  fi  ;;  \*)  echo "Please use start or stop as first argument"  ;;  esac |

3、修改redis的权限并注册到服务中。

|  |
| --- |
| [root@xmm init.d]# chmod 777 redis  [root@xmm init.d]# chkconfig redis on |

### Redis安全机制

redis默认情况下会绑定在0.0.0.0:6379，这样就会将redis的服务暴露到公网上，如果在没有开启认证的情况下，可以导致任意用户在访问目标服务器的情况下未授权访问redis以及读取redis的数据，攻击者就可以在未授权访问redis的情况下可以利用redis的相关方法，成功在redis服务器上写入公钥，进而可以直接使用私钥进行直接登录目标主机，使用FLUSHALL方法，将整个redis数据库清空。

**解决方案如下所示：**

1、禁止一些高危命令修改redis.conf文件，用来禁止远程修改DB文件地址，比如 rename-command FLUSHALL "" 、rename-command CONFIG"" 、rename-command EVAL""等；

2、以低权限运行redis服务，为redis服务创建单独的用户和根目录，并且配置禁止登录；

3、为redis添加密码验证，修改redis.conf文件，添加requirepass mypassword；

设置密码：config set requirepass 123456

授权密码：auth 123456

4、禁止外网访问redis，修改redis.conf文件，添加或修改 bind 127.0.0.1，使得redis服务只在当前主机使用；

5、做log监控，及时发现攻击。

## 常见缓存问题

### 缓存穿透

缓存查询一般都是通过key去查找value，如果不存在对应的value，就要去数据库中查找。如果这个key对应的value在数据库中也不存在，并且对该key并发请求很大，就会对数据库产生很大的压力，这就叫缓存穿透。

解决方案：如果一个查询返回的结果为空，仍然对空结果进行缓存。

### 缓存雪崩

当缓存服务器重启或者大量缓存集中在某个时间失效，这样在失效的瞬间对数据库的访问压力就比较大。

解决方案：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一些。

## 其他缓存技术

### Memcache和redis的区别

memcache是一套分布式的高速缓存系统，用以提升网站的访问速度，尤其对于一些大型的、需要频繁访问数据库的网站访问速度提升效果十分显著。

**存储方式：**

redis不仅仅支持简单的k/v类型的数据，同时还支持list、set、zset、hash等数据结构的存储；memcache只支持简单的k/v类型的数据，key和value都是string类型

**可靠性：**

memcache不支持数据持久化，断电或重启后数据消失，但其稳定性是有保证的；redis支持数据持久化和数据恢复，允许单点故障，但是同时也会付出性能的代价

**性能上：**

对于存储大数据，memcache的性能要高于redis

**应用场景：**

Memcache：适合多读少写，大数据量的情况（一些官网的文章信息等）

Redis：适用于对读写效率要求高、数据处理业务复杂、安全性要求较高的系统

# 基本操作及数据结构

## 基本操作

Redis默认有16个数据库，类似于数组下标从零开始，初始默认使用零号库。可以使用select index选择数据库，使用dbsize查看数据库数据的个数。

|  |
| --- |
| 127.0.0.1:6379> select -1  (error) ERR DB index is out of range  127.0.0.1:6379> select 0  OK  127.0.0.1:6379> select 1  OK  127.0.0.1:6379[1]> select 2  OK  127.0.0.1:6379[2]> select 15  OK  127.0.0.1:6379[15]> select 16  (error) ERR DB index is out of range  127.0.0.1:6379[15]> DBSIZE  (integer) 0  127.0.0.1:6379[15]> FLUSHDB//清空数据库  127.0.0.1:6379[15]> FLUSHALL//通杀数据库 |

## key操作

|  |  |
| --- | --- |
| **命令** | **描述** |
| [**del key**](https://www.runoob.com/redis/keys-del.html) | 该命令用于在 key 存在时删除 key。 |
| [**dump key**](https://www.runoob.com/redis/keys-dump.html) | 序列化给定 key ，并返回被序列化的值。 |
| [**exists key**](https://www.runoob.com/redis/keys-exists.html) | 检查给定 key 是否存在。 |
| [**expire key**](https://www.runoob.com/redis/keys-expire.html)**seconds** | 为给定 key 设置过期时间，以秒计。 |
| [**expireat key timestamp**](https://www.runoob.com/redis/keys-expireat.html) | EXPIREAT 的作用和 EXPIRE 类似，都用于为 key 设置过期时间。 不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳(unix timestamp)。 |
| [**pexpire key milliseconds**](https://www.runoob.com/redis/keys-pexpire.html) | 设置 key 的过期时间以毫秒计。 |
| [**pexpireat key milliseconds-timestamp**](https://www.runoob.com/redis/keys-pexpireat.html) | 设置 key 过期时间的时间戳(unix timestamp) 以毫秒计 |
| [**keys pattern**](https://www.runoob.com/redis/keys-keys.html) | 查找所有符合给定模式( pattern)的 key 。 |
| [**move key db**](https://www.runoob.com/redis/keys-move.html) | 将当前数据库的 key 移动到给定的数据库 db 当中。 |
| [**persist key**](https://www.runoob.com/redis/keys-persist.html) | 移除 key 的过期时间，key 将持久保持。 |
| [**pttl key**](https://www.runoob.com/redis/keys-pttl.html) | 以毫秒为单位返回 key 的剩余的过期时间。 |
| [**ttl key**](https://www.runoob.com/redis/keys-ttl.html) | 以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)。 |
| [**randomkey**](https://www.runoob.com/redis/keys-randomkey.html) | 从当前数据库中随机返回一个 key 。 |
| [**rename key newkey**](https://www.runoob.com/redis/keys-rename.html) | 修改 key 的名称 |
| [**renamenx key newkey**](https://www.runoob.com/redis/keys-renamenx.html) | 仅当 newkey 不存在时，将 key 改名为 newkey 。 |
| [**type key**](https://www.runoob.com/redis/keys-type.html) | 返回 key 所储存的值的类型。 |

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，实际应用方式如下：

1、设置限制的优惠活动的信息；

2、一些及时需要更新的数据，积分排行榜；

3、手机验证码的时间；

4、限制网站访客访问频率；

Redis中的数据，总体上是键值对，不同数据类型指的是键值对中值的类型。Redis中的数据类型有五种，分别为string、list、set、hash和zset（sorted set）。

## string

Redis中最基本的类型，它是key对应的一个单一值。二进制安全，不必担心由于编码等问题导致二进制数据变化。所以redis的string可以包含任何数据，比如jpg图片或者序列化的对象。Redis中一个字符串值的最大容量是512M。

单点登陆中的token就是用string类型来存储用户登陆信息。

|  |  |
| --- | --- |
| **命令** | **描述** |
| [**set key value**](https://www.runoob.com/redis/strings-set.html) | 设置指定 key 的值 |
| [**get key**](https://www.runoob.com/redis/strings-get.html) | 获取指定 key 的值。 |
| [**getrange key start end**](https://www.runoob.com/redis/strings-getrange.html) | 返回 key 中字符串值的子字符 |
| [**getset key value**](https://www.runoob.com/redis/strings-getset.html) | 将给定 key 的值设为 value ，并返回 key 的旧值(old value)。 |
| [**getbit key offset**](https://www.runoob.com/redis/strings-getbit.html) | 对 key 所储存的字符串值，获取指定偏移量上的位(bit)。 |
| [**mget key1 [key2..]**](https://www.runoob.com/redis/strings-mget.html) | 获取所有(一个或多个)给定 key 的值。 |
| [**setbit key offset value**](https://www.runoob.com/redis/strings-setbit.html) | 对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)。 |
| [**setex key seconds value**](https://www.runoob.com/redis/strings-setex.html) | 将值 value 关联到 key ，并将 key 的过期时间设为 seconds (以秒为单位)。 |
| [**setnx key value**](https://www.runoob.com/redis/strings-setnx.html) | 只有在 key 不存在时设置 key 的值。 |
| [**setrange key offset value**](https://www.runoob.com/redis/strings-setrange.html) | 用 value 参数覆写给定 key 所储存的字符串值，从偏移量 offset 开始。 |
| [**strlen key**](https://www.runoob.com/redis/strings-strlen.html) | 返回 key 所储存的字符串值的长度。 |
| [**mset key value [key value ...]**](https://www.runoob.com/redis/strings-mset.html) | 同时设置一个或多个 key-value 对。 |
| [**msetnx key value [key value ...]**](https://www.runoob.com/redis/strings-msetnx.html) | 同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在。 |
| [**psetex key milliseconds value**](https://www.runoob.com/redis/strings-psetex.html) | 这个命令和 SETEX 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 SETEX 命令那样，以秒为单位。 |
| [**incr key**](https://www.runoob.com/redis/strings-incr.html) | 将 key 中储存的数字值增一。 |
| [**incrby key increment**](https://www.runoob.com/redis/strings-incrby.html) | 将 key 所储存的值加上给定的增量值（increment） 。 |
| [**incrbyfloat key increment**](https://www.runoob.com/redis/strings-incrbyfloat.html) | 将 key 所储存的值加上给定的浮点增量值（increment） 。 |
| [**decr key**](https://www.runoob.com/redis/strings-decr.html) | 将 key 中储存的数字值减一。 |
| [**decrby key decrement**](https://www.runoob.com/redis/strings-decrby.html) | key 所储存的值减去给定的减量值（decrement） 。 |
| [**append key value**](https://www.runoob.com/redis/strings-append.html) | 如果 key 已经存在并且是一个字符串， APPEND 命令将指定的 value 追加到该 key 原来值（value）的末尾。 |

## 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（左边）或者尾部（右边）。说明它的底层是基于链表实现的，所以它操作时头尾效率高，中间效率低。

index -5 -4 -3 -2 -1

item1 item2 item3 item4 item5

index 0 1 2 3 4

左 右

lpush rpush

lpeek rpeek

lpop rpop

一般可以用于存储商品评论列表，key是该商品的ID，value是该商品的评论信息列表。一个商品会被不同的用户评论，并且商品的评论会按照时间顺序进行排序：

Key ：items:comment:1001

Value:’{“id”:1,”name”:”商品很好！”,”date”:1432079}’

|  |  |
| --- | --- |
| **命令** | **描述** |
| [**blpop key1 [key2 ] timeout**](https://www.runoob.com/redis/lists-blpop.html) | 移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。 |
| [**brpop key1 [key2 ] timeout**](https://www.runoob.com/redis/lists-brpop.html) | 移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。 |
| [**brpoplpush source destination timeout**](https://www.runoob.com/redis/lists-brpoplpush.html) | 从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。 |
| [**lindex key index**](https://www.runoob.com/redis/lists-lindex.html) | 通过索引获取列表中的元素 |
| [**linsert key before|after pivot value**](https://www.runoob.com/redis/lists-linsert.html) | 在列表的元素前或者后插入元素 |
| [**llen key**](https://www.runoob.com/redis/lists-llen.html) | 获取列表长度 |
| [**lpop key**](https://www.runoob.com/redis/lists-lpop.html) | 移出并获取列表的第一个元素 |
| [**lpush key value1 [value2]**](https://www.runoob.com/redis/lists-lpush.html) | 将一个或多个值插入到列表头部 |
| [**lpushx key value**](https://www.runoob.com/redis/lists-lpushx.html) | 将一个值插入到已存在的列表头部 |
| [**lrange key start stop**](https://www.runoob.com/redis/lists-lrange.html) | 获取列表指定范围内的元素 |
| [**lrem key count value**](https://www.runoob.com/redis/lists-lrem.html) | 移除列表元素 |
| [**lset key index value**](https://www.runoob.com/redis/lists-lset.html) | 通过索引设置列表元素的值 |
| [**ltrim key start stop**](https://www.runoob.com/redis/lists-ltrim.html) | 对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。 |
| [**rpop key**](https://www.runoob.com/redis/lists-rpop.html) | 移除列表的最后一个元素，返回值为移除的元素。 |
| [**rpoplpush source destination**](https://www.runoob.com/redis/lists-rpoplpush.html) | 移除列表的最后一个元素，并将该元素添加到另一个列表并返回 |
| [**rpush key value1 [value2]**](https://www.runoob.com/redis/lists-rpush.html) | 在列表中添加一个或多个值 |
| [**rpushx key value**](https://www.runoob.com/redis/lists-rpushx.html) | 为已存在的列表添加值 |

## hash

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，可以当做Java中的Map<String,Object>对待。

hash数据类型对应的value内部实际是一个hashmap，底层有两种实现方式：

（1）hash的成员比较少时，Redis为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的hashmap结构，value是redisObject的zipmap;

（2）当成员数量比较多的时候，会自动转化成真正的hashmap

一般可以用于存储购物车信息。

比如我们要存储一个用户信息对象数据，包含以下信息：

用户ID，为查找的key，存储的value用户对象包含姓名name，年龄age，生日birthday 等信息，如果用普通的key/value结构来存储，主要有以下2种存储方式：

第一种方式将用户ID作为查找key,把其他信息封装成一个对象以序列化的方式存储，

如：set u001 "李三,18,20010101"

这种方式的缺点是，增加了序列化/反序列化的开销，并且在需要修改其中一项信息时，需要把整个对象取回，并且修改操作需要对并发进行保护。

第二种方法是这个用户信息对象有多少成员就存成多少个key-value对儿，用用户ID+对应属性的名称作为唯一标识来取得对应属性的值，

如：mset user:001:name "李三 "user:001:age18 user:001:birthday "20010101"

虽然省去了序列化开销和并发问题，但是用户ID为重复存储，如果存在大量这样的数据，内存浪费还是非常可观的。

那么Redis提供的Hash很好的解决了这个问题，Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口，

如：hmset user:001 name "李三" age 18 birthday "20010101" ,也就是说，Key仍然是用户ID,value是一个Map，这个Map的key是成员的属性名，value是属性值，这样对数据的修改和存取都可以直接通过其内部Map的Key(Redis里称内部Map的key为field), 也就是通过 key(用户ID) + field(属性标签) 操作对应属性数据了，既不需要重复存储数据，也不会带来序列化和并发修改控制的问题。很好的解决了问题。

这里同时需要注意，Redis提供了接(hgetall)可以直接取到全部的属性数据,但是如果内部Map的成员很多，那么涉及到遍历整个内部Map的操作，由于Redis单线程模型的缘故，这个遍历操作可能会比较耗时，而另其它客户端的请求完全不响应，这点需要格外注意

|  |  |
| --- | --- |
| **命令** | **描述** |
| [**hdel key field1 [field2]**](https://www.runoob.com/redis/hashes-hdel.html) | 删除一个或多个哈希表字段 |
| [**hexists key field**](https://www.runoob.com/redis/hashes-hexists.html) | 查看哈希表 key 中，指定的字段是否存在。 |
| [**hget key field**](https://www.runoob.com/redis/hashes-hget.html) | 获取存储在哈希表中指定字段的值。 |
| [**hgetall key**](https://www.runoob.com/redis/hashes-hgetall.html) | 获取在哈希表中指定 key 的所有字段和值 |
| [**hincrby key field increment**](https://www.runoob.com/redis/hashes-hincrby.html) | 为哈希表 key 中的指定字段的整数值加上增量 increment 。 |
| [**hincrbyfloat key field increment**](https://www.runoob.com/redis/hashes-hincrbyfloat.html) | 为哈希表 key 中的指定字段的浮点数值加上增量 increment 。 |
| [**hkeys key**](https://www.runoob.com/redis/hashes-hkeys.html) | 获取所有哈希表中的字段 |
| [**hlen key**](https://www.runoob.com/redis/hashes-hlen.html) | 获取哈希表中字段的数量 |
| [**hmget key field1 [field2]**](https://www.runoob.com/redis/hashes-hmget.html) | 获取所有给定字段的值 |
| [**hmset key field1 value1 [field2 value2 ]**](https://www.runoob.com/redis/hashes-hmset.html) | 同时将多个 field-value (域-值)对设置到哈希表 key 中。 |
| [**hset key field value**](https://www.runoob.com/redis/hashes-hset.html) | 将哈希表 key 中的字段 field 的值设为 value 。 |
| [**hsetnx key field value**](https://www.runoob.com/redis/hashes-hsetnx.html) | 只有在字段 field 不存在时，设置哈希表字段的值。 |
| [**hvals key**](https://www.runoob.com/redis/hashes-hvals.html) | 获取哈希表中所有值 |
| **hscan key cursor [match pattern] [count count]** | 迭代哈希表中的键值对。 |

## set

Redis的set是string类型的无序集合，成员是唯一的，没有重复数据。它是基于一个value永远为null的hashmap来实现的。

set数据类型之所以不能存储重复的数据，就是通过计算hash值的方式来进行快速排重的。

这个应用场景一般存储一个列表数据，但列表里面又不希望出现重复数据，比如微博应用中，可以将一个用户所有关注的对象放在一个集合中，将其所有粉丝存在一个集合，这样我们就可以实现两个人的共同好友、共同关注等需求。

这种数据类型可以用来求两个集合中的交集、并集或差集等。

|  |  |
| --- | --- |
| **命令** | **描述** |
| [**sadd key member1 [member2]**](https://www.runoob.com/redis/sets-sadd.html) | 向集合添加一个或多个成员 |
| [**scard key**](https://www.runoob.com/redis/sets-scard.html) | 获取集合的成员数 |
| [**sdiff key1 [key2]**](https://www.runoob.com/redis/sets-sdiff.html) | 返回给定所有集合的差集 |
| [**sdiffstore destination key1 [key2]**](https://www.runoob.com/redis/sets-sdiffstore.html) | 返回给定所有集合的差集并存储在 destination 中 |
| [**sinter key1 [key2]**](https://www.runoob.com/redis/sets-sinter.html) | 返回给定所有集合的交集 |
| [**sinterstore destination key1 [key2]**](https://www.runoob.com/redis/sets-sinterstore.html) | 返回给定所有集合的交集并存储在 destination 中 |
| [**sismember key member**](https://www.runoob.com/redis/sets-sismember.html) | 判断 member 元素是否是集合 key 的成员 |
| [**smembers key**](https://www.runoob.com/redis/sets-smembers.html) | 返回集合中的所有成员 |
| [**smove source destination member**](https://www.runoob.com/redis/sets-smove.html) | 将 member 元素从 source 集合移动到 destination 集合 |
| [**spop key**](https://www.runoob.com/redis/sets-spop.html) | 移除并返回集合中的一个随机元素 |
| [**srandmember key [count]**](https://www.runoob.com/redis/sets-srandmember.html) | 返回集合中一个或多个随机数 |
| [**srem key member1 [member2]**](https://www.runoob.com/redis/sets-srem.html) | 移除集合中一个或多个成员 |
| [**sunion key1 [key2]**](https://www.runoob.com/redis/sets-sunion.html) | 返回所有给定集合的并集 |
| [**sunionstore destination key1 [key2]**](https://www.runoob.com/redis/sets-sunionstore.html) | 所有给定集合的并集存储在 destination 集合中 |
| [**sscan key cursor [match pattern] [count count]**](https://www.runoob.com/redis/sets-sscan.html) | 迭代集合中的元素 |

## zset

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(score)却可以重复。

这种数据类型如果使用的话主要用来统计商品的销售排行榜。

|  |  |  |
| --- | --- | --- |
| **命令** | **描述** | |
| [**zadd key score1 member1 [score2 member2]**](https://www.runoob.com/redis/sorted-sets-zadd.html) | | 向有序集合添加一个或多个成员，或者更新已存在成员的分数 |
| [**zcard key**](https://www.runoob.com/redis/sorted-sets-zcard.html) | | 获取有序集合的成员数 |
| [**zcount key min max**](https://www.runoob.com/redis/sorted-sets-zcount.html) | | 计算在有序集合中指定区间分数的成员数 |
| [**zincrby key increment member**](https://www.runoob.com/redis/sorted-sets-zincrby.html) | | 有序集合中对指定成员的分数加上增量 increment |
| [**zinterstore destination numkeys key [key ...]**](https://www.runoob.com/redis/sorted-sets-zinterstore.html) | | 计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中 |
| [**zlexcount key min max**](https://www.runoob.com/redis/sorted-sets-zlexcount.html) | | 在有序集合中计算指定字典区间内成员数量 |
| [**zrange key start stop [withscores]**](https://www.runoob.com/redis/sorted-sets-zrange.html) | | 通过索引区间返回有序集合成指定区间内的成员 |
| [**zrangebylex key min max [limit offset count]**](https://www.runoob.com/redis/sorted-sets-zrangebylex.html) | | 通过字典区间返回有序集合的成员 |
| [**zrangebyscore key min max [withscores] [limit]**](https://www.runoob.com/redis/sorted-sets-zrangebyscore.html) | | 通过分数返回有序集合指定区间内的成员 |
| [**zrank key member**](https://www.runoob.com/redis/sorted-sets-zrank.html) | | 返回有序集合中指定成员的索引 |
| [**zrem key member [member ...]**](https://www.runoob.com/redis/sorted-sets-zrem.html) | | 移除有序集合中的一个或多个成员 |
| [**zremrangebylex key min max**](https://www.runoob.com/redis/sorted-sets-zremrangebylex.html) | | 移除有序集合中给定的字典区间的所有成员 |
| [**zremrangebyrank key start stop**](https://www.runoob.com/redis/sorted-sets-zremrangebyrank.html) | | 移除有序集合中给定的排名区间的所有成员 |
| [**zremrangebyscore key min max**](https://www.runoob.com/redis/sorted-sets-zremrangebyscore.html) | | 移除有序集合中给定的分数区间的所有成员 |
| [**zrevrange key start stop [withscores]**](https://www.runoob.com/redis/sorted-sets-zrevrange.html) | | 返回有序集中指定区间内的成员，通过索引，分数从高到底 |
| [**zrevrangebyscore key max min [withscores]**](https://www.runoob.com/redis/sorted-sets-zrevrangebyscore.html) | | 返回有序集中指定分数区间内的成员，分数从高到低排序 |
| [**zrevrank key member**](https://www.runoob.com/redis/sorted-sets-zrevrank.html) | | 返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序 |
| [**zscore key member**](https://www.runoob.com/redis/sorted-sets-zscore.html) | | 返回有序集中，成员的分数值 |
| [**zunionstore destination numkeys key [key ...]**](https://www.runoob.com/redis/sorted-sets-zunionstore.html) | | 计算给定的一个或多个有序集的并集，并存储在新的 key 中 |
| [**zscan key cursor [match pattern] [count count]**](https://www.runoob.com/redis/sorted-sets-zscan.html) | | 迭代有序集合中的元素（包括元素成员和元素分值） |

|  |
| --- |
| redis> ZADD myzset 0 a 0 b 0 c 0 d 0 e  (integer) 5  redis> ZADD myzset 0 f 0 g  (integer) 2  redis> ZLEXCOUNT myzset - +  (integer) 7  redis> ZLEXCOUNT myzset [b [f  (integer) 5 |

按照字母顺序在区间内返回member

min和max使用“[a”表示闭区间，使用“(a”表示开区间

-表示负无穷

+表示正无穷

# 配置文件

Redis配置文件位于/usr/local/redis目录下。

## 常用配置

1.Redis默认不是以守护进程的方式运行的，可以通过该配置项修改，是有yes启用守护进程

daemonize no

2.当Redis以守护线程的方式运行时，Redis默认会把pid写入到/var/run/redis.pid文件中（或者）redis\_6379.pid文件中，可以通过pidfile指定

pidfile /var/run/redis.pid

3.指定Redis监听端口，默认端口为6379，作者在自己的一篇文章中结识了为什么选用6379作为默认端口，因为6379在手机按键上MERZ对应的号码，而MERZ取自意大利歌女Alessia Merz的名字

port 6379

4.绑定的主机地址

bind 127.0.0.1

5.当客户端闲置多长时间后关闭连接；如果指定为0，表示关闭该功能

timeout 300

6.指定日志记录级别。Redis总共支持四个级别：debug，verbose，notice，warning，默认verbose

loglevel verbose

7.日志记录方式，默认为标准输出，如果配置Redis为守护进程运行方式，而这里又配置为日志记录方式为标准输出，则日志将会发送给/dev/null

logfile stdout

8.设置数据库的数量，默认数据库为0号数据库，可以使用select <dbid>命令在连接上指定数据库id

databases 16

9.指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合

save <seconds> <changes>

Redis默认配置文件中提供了三个条件：

save 900 1 900秒（15分钟）内有1个更改

save 300 10 300秒（5分钟）内有10个更改

save 60 10000 60秒（1分钟）内有10000个更改

save "" 禁用

10.指定存储至本地数据库是是否压缩数据，默认为yes，Redis采用LZF压缩，如果为了节省CPU时间，可以关闭该选项，但会导致数据库文件变得巨大

rdbcompression yes

11.指定本地数据库文件名，默认值为dump.rdb

dbfilename dump.rdb

12.指定本地数据库存放目录；默认存放执行入口文件夹（即打开命令台的文件夹）

dir ./

13.设置当本机为slav服务时，设置master服务的IP地址几端口号，在Redis启动时，它会自动从master进行数据同步

slaveof <masterip> <masterport>

14.当master服务设置了密码保护时，slav服务链接master的密码

masterauth <master-password>

15.设置Redis连接密码，如果配置了连接密码，

客户端在连接Redis时需要通过AUTH <password>命令提供密码，默认关闭

requirepass foobared

16.设置同一时间最大客户端连接数，默认无限制，Redis可以同时打开的客户端连接数为Redis进程可以打开的最大文件描述符数，如果设置maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis会关闭新的连接并想客户端返回max number of clients reached错误信息

maxclients 128

17.指定Redis最大内存限制，Redis在启动时会把数据加载到内存中，达到最大内存后，Redis会先尝试清楚已到期或即将到期的Key，当此方法处理后，仍然到达最大内存设置，将无法在进行写入操作，但仍然可以进行读取操作。Redis新的vm机制，会把Key存放内存，Value存放在swap区

maxmemory <bytes>

18.指定是否在每次更新操作后进行日志记录。Redis在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为Redis本身同步数据文件是按上面save条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为no

appendonly no

19.指定更新日志文件名，默认为appendonly.aof

appendfilename appendonly.aof

20.指定更新日志条件，共有3个可选值：

no：表示操作系统进行数据缓存同步到磁盘（快）

always：表示每次更新操作后手动调用fsync()将数据写到磁盘（慢，安全）

everysec：表示美妙同步一次（折中，默认值）

appendfsync everysec

21. 指定是否启用虚拟内存机制，默认值为no，简单的介绍一下，vm机制将数据分页存放，由Redis将访问量较少的页即冷数据swap到磁盘上，访问多的页面由磁盘自动换出到内存中

vm-enabled no

22.虚拟内存文件路径，默认值为/tmp/redis.swap，不可多个Redis实例共享

vm-swap-file- /tmp/redis.swap

23.将所有大于vm-max-memory的数据存入虚拟内存，无论vm-max-memory设置多小，所有索引数据都是内存存储的（Redis的索引数据 就是keys），也就是说，当vm-max-memory设置为0的时候，其实是所有value都存在于磁盘。默认值为0

vm-max-memory 0

24.Redis swap文件分成了很多的page，一个对象可以保存在多个page上面，但一个page上不能呢被多个对象共享，m-page-size是要根据存储的数据大小来设定的，作者建议如果存储很多小对象，page大小最好设置为32或者64bytes；如果存储很大的对象，则可以使用更大的page，如果不确定，就使用默认值

vm-page-size 32

25.设置swap文件中的page数量，由于页表（一种表示页面空闲或使用的bitmap）是存放在内存中的，在磁盘上每8个pages将消耗1byte的内存。

vm-pages 134217728

26.设置访问swap文件的线程数，最好不要超过机器的核数，如果设置为0，那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4

vm-max-threads 4

27.设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

glueoutputbuf yes

28.指定在超过一定的数量或者最大的元素超过某一临界值，采用一种特殊的哈希值算法

hash-max-zipmap-entires 64

hash-max-zipmap-value 512

29.指定是否激活重置哈希，默认为开启

activerehashing yes

30.指定包含其他的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，而同时哥哥实例又拥有自己的特定配置文件

include /path/to/local.conf

# 持久化

Redis是内存型数据库，同时它也可以持久化到硬盘中，redis的持久化方式有RDB和AOF两种。

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。

## RDB半持久化方式

RDB（**redis database**）是指定期将内存中数据通过异步方式、快照形式保存到硬盘上的二进制文件dump.rdb中。redis启动后读取dump.rdb快照文件，将数据从硬盘载入到内存中，需要花费的时间由数据量大小、结构、服务器的性能决定，通常将一千万个字符串类型键、大小为1GB的快照文件载入到内存中需要花费20-30秒。redis默认开启RDB机制。

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，待持久化过程结束后，再用这个临时文件替换上次持久化好的文件。使用写时拷贝技术（**copy-on-write**），主进程不进行任何IO操作，确保高性能。

fork的作用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。

redis server

child process

rdb

临时文件

rdb

最终文件

覆盖

dump到

fork子进程

定时执行

**RDB优点**

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方式要比AOF方式更加的高效。

只包含一个文件，便于文件备份和灾难恢复，可以将一个单独文件转移到其他存储媒介上；性能最大化，这种方式使用的是写时拷贝技术，可以极大的避免服务进程执行IO操作；相对于AOF来说，如果数据集很大，RDB的启动效率就会很高。

**RDB缺点**

如果想保证数据的高可用（最大限度的包装数据丢失），那么RDB这种半持久化方式不是一个很好的选择，因为系统一旦在持久化策略之前出现宕机现象，此前没有来得及持久化的数据将会产生丢失；RDB通过fork进程来协助完成持久化的，内存中的数据被克隆了一份，大致2倍的膨胀性需要考虑，当数据集较大时，就需要等待服务器停止几百毫秒甚至一秒。

### 触发时机

o在配置文件中的SNAPSHOTTING快照配置中，默认配置了三种存储策略（save <seconds> <changes>）。当一定到达一定时间（<seconds>）且修改次数（<changes>）达到配置信息则进行数据持久化。

当redis启动的时候会读取配置文件中的SNAPSHOTTING快照配置中的dbfilename 所配置的文件名称对启动目录进行扫描加载，将对应的文件（dump.rdb）快速加载到内存中。

| **配置** | **含义** |
| --- | --- |
| save 900 1 | 900秒内至少有一次修改则触发保存操作 |
| save 300 10 | 300秒内至少有10次修改则触发保存操作 |
| save 60 10000 | 60秒内至少有1万次修改则触发保存操作 |

o在任意时刻手动输入保存命令：save或bgsave，save只管保存，其它不管，全部阻塞；bgsave：Redis会在后台异步进行快照操作，快照同时还可以响应客户端请求。可以通过lastsave命令获取最后一次成功执行快照的时间。

o使用flushall命令，会产生dump.rdb文件，但是该文件为空文件，无意义

o执行shutdown关闭redis服务器时，redis会执行一次持久化保存

### 恢复dump.rdb

拷贝dump.rdb文件到redis安装目录启动服务即可, config get dir获取目录。

### 停止RDB

动态所有停止RDB保存规则的方法：redis-cli config set save ""

### 相关配置

| **配置项** | **取值** | **作用** |
| --- | --- | --- |
| save | "" | 禁用RDB机制 |
| dbfilename | 文件名，例如：dump.rdb | 设置RDB机制下，数据存储文件的文件名 |
| dir | Redis工作目录路径 | 指定存放持久化文件的目录的路径。注意：这里指定的必须是目录不能是文件名 |

值得注意的是，使用RDB机制也是不能够保证数据的绝对安全的。

## AOF全持久化方式

AOF(**Append-only file**) 以日志的形式来记录每个写操作，根据配置文件中指定的策略，把生成数据的命令通过write()方法保存到硬盘上的appendonly.aof文件中。事实上，AOF不会立即将命令写入硬盘文件中，而是写入硬盘缓存，可以配置策略，配置多久从硬盘缓存写入到硬盘文件中。

将Redis执行过的所有写指令记录下来（读操作不记录），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重新启动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。

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。

**AOF优点**

AOF文件是一个只进行追加的日志文件

Redis可以在AOF文件体积变得过大时，自动地在后台对AOF进行重写

AOF文件有序地保存了对数据库执行的所有写入操作，这些以Redis协议的格式保存，因此AOF文件的内容非常容易被人读懂，对文件进行分析也很轻松

**AOF缺点**

对于相同的数据集来说，AOF文件的体积通常大于RDB文件的体积

根据所使用的fsync策略，AOF的速度会慢于RDB（短时间内多次提交）

### AOF基本配置

redis默认是不支持AOF全持久化方式的，需要将appendonly的配置从no改为yes。

| 配置项 | 取值 | 作用 |
| --- | --- | --- |
| appendonly | yes | 启用AOF持久化机制 |
|  | no | 禁用AOF持久化机制[默认值] |
| appendfilename | "文件名" | AOF持久化文件名 |
| dir | Redis工作目录路径 | 指定存放持久化文件的目录的路径。注意：这里指定的必须是目录不能是文件名 |
| appendfsync | always | 每一次数据修改后都将执行文件写入操作，缓慢但是最安全。 |
|  | everysec | 每秒执行一次写入操作，效率折中且有可能丢失一秒内的数据。 |
|  | no | 不自动同步到硬盘上，由操作系统在适当的时候执行写入操作，最快，但是安全性较差。 |

### AOF重写

AOF采用文件追加方式，文件会越来越大为避免出现此种情况，新增了重写机制，当AOF文件的大小超过所设定的阈值时，redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集，可以使用命令bgrewriteaof。

**重写原理**

AOF文件持续增长而过大时，会fork出一条新进程来将文件重写（也是先写临时文件最后再rename），遍历新进程的内存中数据，每条记录有一条的set语句。重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件，这点和快照有点类似。

redis会记录上次重写时的AOF大小，默认配置是当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发。

redis需要持久化的时候，fork出一个子进程，子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令；父进程会继续处理客户端的请求，除了把写命令写到原来的aof中，同时把收到的写命令缓存起来，这样包装如果子进程重写失败的话不会出问题；当子进程把快照内容以命令方式写入临时文件中后，子进程会发送信号给父进程，父进程会把缓存的写命令写入到临时文件中；接下来父进程可以使用临时的aof文件替换原来的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。下面的图为最简单的方式，其实也是利用写时复制原则。

对比下面两组命令：

| AOF重写前 | AOF重写后 |
| --- | --- |
| set count 1 incr count incr count incr count | set count 4 |

两组命令执行后对于count来说最终的值是一致的，但是进行AOF重写后省略了中间过程，可以让AOF文件体积更小。而Redis会根据AOF文件的体积来决定是否进行AOF重写。参考的配置项如下：

| **配置项** | **含义** |
| --- | --- |
| auto-aof-rewrite-percentage 100 | 文件体积增大100%时执行AOF重写 |
| auto-aof-rewrite-min-size 64mb | 文件体积增长到64mb时执行AOF重写 |

实际工作中不要进行频繁的AOF重写，因为CPU资源和硬盘资源二者之间肯定是CPU资源更加宝贵，所以不应该过多耗费CPU性能去节省硬盘空间。

#### 优点

数据安全性高

该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机问题，也不会破坏日志文件中已经存在的内容；

如果本次操作只是写入了一半数据就出现系统崩溃问题，可以在redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

#### 缺点

对于数量相同的数据集来说，aof文件通常要比rdb文件大，因此rdb在恢复大数据集时的速度大于AOF；

根据同步策略的不同，AOF在运行效率上往往慢于RDB，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效；

针对以上两种不同的持久化方式，如果缓存数据安全性要求比较高的话，用aof这种持久化方式（比如项目中的购物车）；如果对于大数据集要求效率高的话，就可以使用默认的。而且这两种持久化方式可以同时使用。

### 持久化文件损坏修复

Redis服务器启动时如果读取了损坏的持久化文件会导致启动失败，此时为了让Redis服务器能够正常启动，需要对损坏的持久化文件进行修复。这里以AOF文件为例介绍修复操作的步骤。

第一步：备份要修复的appendonly.aof文件

第二步：执行修复程序

/usr/local/redis/bin/redis-check-aof /usr/local/redis/appendonly.aof检查文件是否损坏

/usr/local/redis/bin/redis-check-aof --fix /usr/local/redis/appendonly.aof修复损坏文件

第三步：重启Redis

注意：所谓修复持久化文件仅仅是把损坏的部分去掉，而没法把受损的数据找回。

## 持久化方式的选择

### RDB

[1]优势：适合大规模的数据恢复，速度较快

[2]劣势：会丢失最后一次快照后的所有修改，不能绝对保证数据的高度一致性和完整性。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，但上述成立有条件，Linux也有优化手段

### AOF

[1]优势：选择appendfsync always方式运行时理论上能够做到数据完整一致，但此时性能又不好。文件内容具备一定可读性，能够用来分析Redis工作情况。

[2]劣势：持久化相同的数据，文件体积比RDB大，恢复速度比RDB慢。效率在同步写入时低于RDB，不同步写入时与RDB相同。

### RDB和AOF并存

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。

### 使用建议

如果想达到足以媲美PostgreSQL的数据安全性， 应同时使用两种持久化功能。在这种情况下，当redis重启的时候会优先载入AOF文件，因为通常情况下AOF文件保存的数据集要RDB文件保存的数据集完整。RDB的数据不实时，同时使用两者服务器重启也只会找AOF文件。

如果不是特别在意数据的完整性，可以考虑简单地使用RDB。

不要只使用AOF持久化，因为定时生成RDB快照非常便于进行数据库备份， 并且 RDB 恢复数据的速度也要比AOF恢复的速度要快。而且假如AOF持久化出错的时候，RDB持久化还是能够进行。

如果Redis仅仅作为缓存可以不使用任何持久化方式。

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。

如果不开启AOF，仅靠Master-Slave Replication 实现高可用性能也不错。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同时倒掉，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。

# 事务控制

Redis事务是一个单独的隔离操作：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。

Redis事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

|  |  |
| --- | --- |
| **命令名** | **作用** |
| MULTI | 事务的开始，表示开始收集命令，后面所有命令都不是马上执行，而是加入到一个队列中。 |
| EXEC | 事务的结束，执行MULTI后面命令队列中的所有命令。 |
| DISCARD | 放弃执行队列中的命令。 |
| WATCH | “观察“、”监控“一个KEY，在当前队列外的其他命令操作这个KEY时，放弃执行自己队列的命令，相当于MySQL中的锁 |
| UNWATCH | 放弃监控一个KEY |

如果一个事务中的B命令依赖上一个命令A，此时可以通过watch命令来监控一个或多个键，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，EXEC命令执行完之后被监控的键会自动被UNWATCH）。

## 事务执行三个阶段

开启：以multi开始一个事务。

入队：将多个命令入队到事务中，接到这些命令并不会立即执行，而是放到等待执行的事务队列里面。

执行：由EXEC命令触发事务。

|  |
| --- |
| 192.168.235.23:6379> MULTI  OK  192.168.235.23:6379> set key1 value1  QUEUED  192.168.235.23:6379> set key2 value2  QUEUED  192.168.235.23:6379> set key3 value3  QUEUED  192.168.235.23:6379> EXEC  1) OK  2) OK  3) OK |

当输入MULTI命令后，服务器返回OK表示事务开始成功，然后依次输入需要在本次事务中执行的所有命令，每次输入一个命令服务器并不会马上执行，而是返回”QUEUED”，这表示命令已经被服务器接受并且暂时保存起来，最后输入EXEC命令后，本次事务中的所有命令才会被依次执行，可以看到最后服务器一次性返回了三个OK，这里返回的结果与发送的命令是按顺序一一对应的，这说明这次事务中的命令全都执行成功了。

Redis的事务除了保证所有命令要不全部执行，要不全部不执行外，还能保证一个事务中的命令依次执行而不被其他命令插入。同时，redis的事务是不支持回滚操作的。

## Redis事务的特性

**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。事务在执行过程中，不会被其它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“事务内的查询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条命令执行失败，其后的命令仍然会被执行，没有回滚。

## 命令队列执行失败的两种情况

### 加入队列时失败

|  |
| --- |
| 127.0.0.1:6379> multi  OK  127.0.0.1:6379> set age 20  QUEUED  127.0.0.1:6379> incr age  QUEUED  127.0.0.1:6379> incr age www  (error) ERR wrong number of arguments for 'incr' command  127.0.0.1:6379> exec  (error) EXECABORT Transaction discarded because of previous errors. |

遇到了入队时即可检测到的错误，整个队列都不会执行。

### 执行队列时失败

|  |
| --- |
| 127.0.0.1:6379> multi  OK  127.0.0.1:6379> set age 30  QUEUED  127.0.0.1:6379> incrby age 5  QUEUED  127.0.0.1:6379> incrby age 5  QUEUED  127.0.0.1:6379> incrby age ww  QUEUED  127.0.0.1:6379> incrby age 5  QUEUED  127.0.0.1:6379> EXEC  1) OK  2) (integer) 35  3) (integer) 40  4) (error) ERR value is not an integer or out of range  5) (integer) 45  127.0.0.1:6379> get age  "45" |

错误在入队时检测不出来，整个队列执行时有错的命令执行失败，但是其他命令并没有回滚。

## Redis回滚

redis不支持回滚的原因有如下两个：

1.Redis 命令只会因为错误的语法而失败（并且这些问题不能在入队时发现），或是命令用在了错误类型的键上面：这也就是说，从实用性的角度来说，失败的命令是由编程错误造成的，而这些错误应该在开发的过程中被发现，而不应该出现在生产环境中。

2.因为不需要对回滚进行支持，所以 Redis 的内部可以保持简单且快速。 有种观点认为 Redis 处理事务的做法会产生 bug ， 然而需要注意的是， 在通常情况下， 回滚并不能解决编程错误带来的问题。

## 悲观锁和乐观锁

在使用WATCH命令监控一个KEY后，当前队列中的命令会由于外部命令的执行而放弃，这是乐观锁的体现。

**悲观锁**

认为当前环境非常容易发生碰撞，所以执行操作前需要把数据锁定，操作完成后释放锁，其他操作才可以继续操作。

**乐观锁**

认为当前环境不容易发生碰撞，所以执行操作前不锁定数据，万一碰撞真的发生了，那么放弃自己的操作。

# 集群

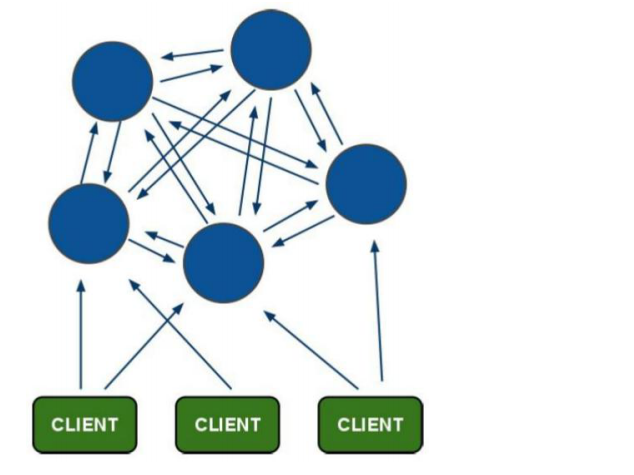
Redis的数据是存放在内存中的，这就意味着redis不适合存储大数据，大数据存储一般公司常用hadoop中的Hbase或者MogoDB。因此redis主要用来处理高并发的，项目如果并发大的话，一台单独的redis不能支持足够的并发，这就需要扩展多台设备协同合作，即用到集群。

## Redis集群方案

1.twemproxy，大概概念是，它类似于一个代理方式，使用方法和普通redis无任何区别，设置好它下属的多个redis实例后，使用时在本需要连接redis的地方改为连接twemproxy，它会以一个代理的身份接收请求并使用一致性hash算法，将请求转接到具体redis，将结果再返回twemproxy。使用方式简便(相对redis只需修改连接端口)，对旧项目扩展的首选。 问题：twemproxy自身单端口实例的压力，使用一致性hash后，对redis节点数量改变时候的计算值的改变，数据无法自动移动到新的节点。

2.codis，目前用的最多的集群方案，基本和twemproxy一致的效果，但它支持在 节点数量改变情况下，旧节点数据可恢复到新hash节点。

3.redis cluster3.0自带的集群，特点在于他的分布式算法不是一致性hash，而是hash槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。这种方式采用的是无中心结构，每个节点保存数据和整个集群的状态，每个节点都和其他所有节点连接。



4.在业务代码层实现，起几个毫无关联的redis实例，在代码层，对key 进行hash计算，然后去对应的redis实例操作数据。 这种方式对hash层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控，等等。

## Redis集群特点

1、所有的节点都通过PING-PONG机制彼此互相连接；

2、每个节点的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集群连接，只需要连接集群中的任何一个节点即可；

4、Redis-cluster把所有的物理节点映射到【0-16383】slot上，负责维护。这条会涉及到redis-cluster的原理问题：Redis-cluster中内置了16384个哈希槽，当需要set一个值的时候，redis先会对key进行crc16算法算出一个结果，然后把这个结果对16384取余，这样的话每个key就会对应一个0-16283之间的哈希槽，redis会根据节点的数量进行均等的分配哈希槽映射到不同的节点上。比如：有三个节点 A1、A2、A3

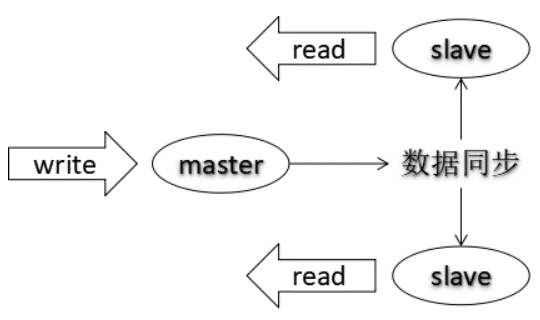
A1：【0-5000】

A2：【5001-10000】

A3：【10001-16383】

Redis集群没有使用一致性hash,而是引入了哈希槽的概念，Redis集群有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。

## Redis集群读写分离



**读写分离好处**

性能优化：主服务器专注于写操作，可以用更适合写入数据的模式工作；同样，从服务器专注于读操作，可以用更适合读取数据的模式工作。

强化数据安全，避免单点故障：由于数据同步机制的存在，各个服务器之间数据保持一致，所以其中某个服务器宕机不会导致数据丢失或无法访问。从这个角度说参与主从复制的Redis服务器构成了一个集群。

## 搭建Redis集群

Redis集群在运行时使用的是同一个可执行文件，只是对应的配置文件不同。

### 配置文件参数

每个配置文件中相同的参数是

daemonize yes

dir /usr/local/cluster-redis

每个配置文件中不同的参数是

| **配置项名称** | **作用** | **取值** |
| --- | --- | --- |
| port | Redis服务器启动后监听的端口号 | 6000、7000、8000 |
| dbfilename | RDB文件存储位置 | dump6000.rdb、dump7000.rdb、dump8000.rdb |
| logfile | 日志文件位置 | /var/logs/redis6000.log、/var/logs/redis7000.log /var/logs/redis8000.log |
| pidfile | pid文件位置 | /var/run/redis6000.pid、/var/run/redis7000.pid /var/run/redis8000.pid |

### 步骤

第一步：创建/usr/local/cluster-redis目录

第二步：把原始未经修改的redis.conf复制到/usr/local/cluster-redis目录

第三步：把/usr/local/cluster-redis目录下的redis.conf复制为redis6000.conf

第四步：按照既定计划修改redis6000.conf中的相关配置项：daemonize yes、dir、port、dbfilename、logfile、pidfile

第五步：复制redis6000.conf为redis7000.conf

第六步：修改redis7000.conf中的相关配置项：port、dbfilename、logfile、pidfile

第七步：复制redis6000.conf为redis8000.conf

第八步：修改redis8000.conf中的相关配置项：port、dbfilename、logfile、pidfile

### 启动Redis主从复制集群

/usr/local/redis/bin/redis-server /usr/local/cluster-redis/redis6000.conf

/usr/local/redis/bin/redis-server /usr/local/cluster-redis/redis7000.conf

/usr/local/redis/bin/redis-server /usr/local/cluster-redis/redis8000.conf

使用redis-cli停止指定服务器的命令格式如下：

/usr/local/bin/redis-cli -h IP地址 -p 端口号shutdown

## 主从复制机制

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有N-1个复制品。

### 主从关系

**查看主从关系**

127.0.0.1:6000> info replication

# Replication

role:master

connected\_slaves:0

刚刚启动的集群服务器中每一个节点服务器都认为自己是主服务器。需要建立主从关系。

**设定主从关系**

在从机上指定主机位置即可：SLAVEOF 127.0.0.1 6000

**取消主从关系**

在从机上执行命令：SLAVEOF NO ONE

### 初步测试

测试1：在主机写入数据，在从机查看

测试2：在从机写入数据报错。配置文件中的依据是：slave-read-only yes

测试3：主机执行SHUTDOWN看从机状态

测试4：主机恢复启动，看从机状态

测试5：从机SHUTDOWN，此时主机写入数据，从机恢复启动查看状态。重新设定主从关系后看新写入的数据是否同步。

### 哨兵模式

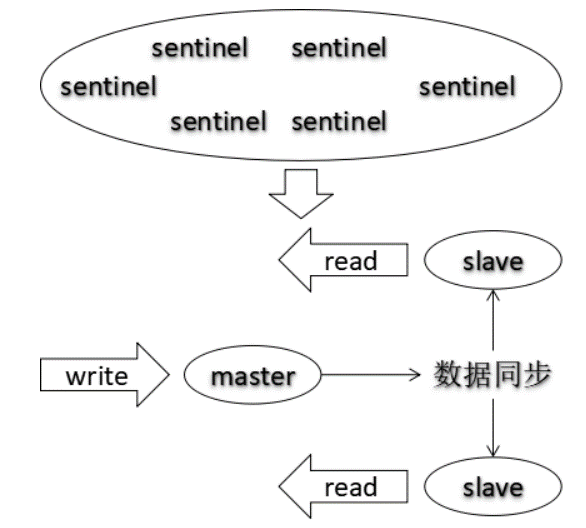
通过哨兵服务器监控master/slave实现主从复制集群的自动管理。哨兵服务器主要功能如下：

1、监控：监控主数据库和从数据库是否正常运行；

2、提醒：当被监控的某个redis出现问题的时候，哨兵可以通过API向管理员或者其他应用程序发送通知

3、自动故障迁移：主数据库出现故障时，可以自动将从数据库转化为主数据库，实现自动切换；

如果master主服务器设置了密码，哨兵的配置文件（sentinel.conf）需要配置访问密码。



**主观下线和客观下线**

主观下线：1台哨兵检测到某节点服务器下线。

客观下线：认为某个节点服务器下线的哨兵服务器达到指定数量。这个数量后面在哨兵的启动配置文件中指定。

**配置哨兵**

vim /usr/local/cluster-redis/sentinel.conf

| **格式** | sentinel monitor 为主机命名 主机IP 主机端口号 将主机判定为下线时需要Sentinel同意的数量 |
| --- | --- |
| **例子** | sentinel monitor mymaster 127.0.0.1 6000 1 |

**启动哨兵**

/usr/local/redis/bin/redis-server /usr/local/cluster-redis/sentinel.conf –sentinel

|  |
| --- |
| +sdown master mymaster 127.0.0.1 6379 【主观下线】  +odown master mymaster 127.0.0.1 6379 #quorum 1/1【客观下线】  ……  +vote-for-leader 17818eb9240c8a625d2c8a13ae9d99ae3a70f9d2 1【选举leader】  ……  +failover-state-send-slaveof-noone slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379【把一个从机设置为主机】  -------------挂掉的主机又重新启动---------------------  -sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6381【离开主观下线状态】  +convert-to-slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6381【转换为从机】 |

### 容错（投票）机制

选举过程是集群中的所有master都参与，如果半数以上master节点与故障节点连接超过时间，则认为该节点故障，自动会触发故障转移操作。

集群不可用？

a:如果集群任意master挂掉，并且当前的master没有slave，集群就会fail;

b:如果集群超过半数以上master挂掉，无论是否有slave，整个集群都会fail;

# 发布订阅

分布订阅是指进程间的一种消息通信模式，发送者（pub）发送消息，订阅者（sub）接收消息。redis发布订阅功能可以用于消息队列，但是其功能不如其他消息队列，因此很少使用。

## 订阅频道

127.0.0.1:6379> SUBSCRIBE cctv

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "cctv"

3) (integer) 1

## 发布信息

127.0.0.1:6379> PUBLISH cctv hai

(integer) 1

127.0.0.1:6379> SUBSCRIBE cctv

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "cctv"

3) (integer) 1

1) "message"

2) "cctv"

3) "hai"

# 客户端

Redisson、Jedis、lettuce等等，官方推荐使用Redisson。

## Redisson

Redisson是一个高级的分布式协调Redis客服端，能帮助用户在分布式环境中轻松实现一些Java的对象 (Bloom filter, BitSet, Set, SetMultimap, ScoredSortedSet, SortedSet, Map, ConcurrentMap, List, ListMultimap, Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock, ReadWriteLock, AtomicLong, CountDownLatch, Publish / Subscribe, HyperLogLog)。

## Jedis

Jedis是用于Java和Redis交互的Java类，类似于jdbc中的Connection。

Jedis是Redis的Java实现的客户端，其API提供了比较全面的Redis命令的支持；Redisson实现了分布式和可扩展的Java数据结构，和Jedis相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等Redis特性。Redisson的宗旨是促进使用者对Redis的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

|  | **MySQL** | **Redis** |
| --- | --- | --- |
| 连接 | Connection | Jedis |
| 连接池 | C3P0等等 | JedisPool |
| 操作完成 | 关闭连接 | 关闭连接 |

在使用Jedis前需要将redis的配置文件中bind配置修改为当前Linux服务器的IP地址，否则会无法访问redis服务器。在连接前还要关闭Linux服务器的防火墙，否则也会造成time out错误，导致无法连接。

|  |
| --- |
| public void jedisTest(){  //指定Redis服务器的IP地址和端口号  Jedis jedis = new Jedis("192.168.235.23", 6379);  //执行ping命令  String ping = jedis.ping();  System.*err*.println(ping);  //关闭连接  jedis.close(); } |

如果出现无法连接，则可能是IP地址错误或redis服务器没有开启。

JedisPool

|  |
| --- |
| public void jedisPoolTest(){  //声明Linux服务器IP地址  String host = "192.168.235.23";  //声明Redis端口号  int port = Protocol.*DEFAULT\_PORT*;  //创建连接池对象  JedisPool jedisPool = new JedisPool(host, port);  //获取Jedis对象连接Redis  Jedis jedis = jedisPool.getResource();  //执行具体操作  String ping = jedis.ping();  System.*out*.println(ping);  //关闭连接  jedisPool.close(); } |

## Lettuce

# 待补充

## Redis零散知识点

20、Redis集群会有写操作丢失吗？为什么？

Redis并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

21、Redis集群之间是如何复制的？

异步复制

22、Redis集群最大节点个数是多少？

16384个。

23、Redis集群如何选择数据库？

Redis集群目前无法做数据库选择，默认在0数据库。

24、怎么测试Redis的连通性？

ping

32、Redis如何做大量数据插入？

Redis2.6开始redis-cli支持一种新的被称之为pipe mode的新模式用于执行大量数据插入工作。

36、Redis持久化数据和缓存怎么做扩容？

如果Redis被当做缓存使用，使用一致性哈希实现动态扩容缩容。

如果Redis被当做一个持久化存储使用，必须使用固定的keys-to-nodes映射关系，节点的数量一旦确定不能变化。否则的话(即Redis节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有Redis集群可以做到这样。

37、分布式Redis是前期做还是后期规模上来了再做好？为什么？

既然Redis是如此的轻量（单实例只使用1M内存）,为防止以后的扩容，最好的办法就是一开始就启动较多实例。即便你只有一台服务器，你也可以一开始就让Redis以分布式的方式运行，使用分区，在同一台服务器上启动多个实例。

一开始就多设置几个Redis实例，例如32或者64个实例，对大多数用户来说这操作起来可能比较麻烦，但是从长久来看做这点牺牲是值得的。

这样的话，当你的数据不断增长，需要更多的Redis服务器时，你需要做的就是仅仅将Redis实例从一台服务迁移到另外一台服务器而已（而不用考虑重新分区的问题）。一旦你添加了另一台服务器，你需要将你一半的Redis实例从第一台机器迁移到第二台机器。

39、支持一致性哈希的客户端有哪些？

Redis-rb、Predis等。

40、Redis与其他key-value存储有什么不同？

Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。

Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，应为数据量不能大于硬件内存。在内存数据库方面的另一个优点是， 相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。 同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

41、Redis的内存占用情况怎么样？

给你举个例子： 100万个键值对（键是0到999999值是字符串“hello world”）在我的32位的Mac笔记本上 用了100MB。同样的数据放到一个key里只需要16MB， 这是因为键值有一个很大的开销。 在Memcached上执行也是类似的结果，但是相对Redis的开销要小一点点，因为Redis会记录类型信息引用计数等等。

当然，大键值对时两者的比例要好很多。

64位的系统比32位的需要更多的内存开销，尤其是键值对都较小时，这是因为64位的系统里指针占用了8个字节。 但是，当然，64位系统支持更大的内存，所以为了运行大型的Redis服务器或多或少的需要使用64位的系统。

42、都有哪些办法可以降低Redis的内存使用情况呢？

如果你使用的是32位的Redis实例，可以好好利用Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的Key-Value可以用更紧凑的方式存放到一起。

43、查看Redis使用情况及状态信息用什么命令？

info

44、Redis的内存用完了会发生什么？

如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以将Redis当缓存来使用配置淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

45、Redis是单线程的，如何提高多核CPU的利用率？

可以在同一个服务器部署多个Redis的实例，并把他们当作不同的服务器来使用，在某些时候，无论如何一个服务器是不够的， 所以，如果你想使用多个CPU，你可以考虑一下分片（shard）。

46、一个Redis实例最多能存放多少的keys？List、Set、Sorted Set他们最多能存放多少元素？

理论上Redis可以处理多达232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。我们正在测试一些较大的值。

任何list、set、和sorted set都可以放232个元素。

换句话说，Redis的存储极限是系统中的可用内存值。

47、Redis常见性能问题和解决方案？

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

50、修改配置不重启Redis会实时生效吗？

针对运行实例，有许多配置选项可以通过 CONFIG SET 命令进行修改，而无需执行任何形式的重启。 从 Redis 2.2 开始，可以从 AOF 切换到 RDB 的快照持久性或其他方式而不需要重启 Redis。检索 ‘CONFIG GET \*’ 命令获取更多信息。

但偶尔重新启动是必须的，如为升级 Redis 程序到新的版本，或者当你需要修改某些目前 CONFIG 命令还不支持的配置参数的时候。

1）String

常用命令：set/get/decr/incr/mget等；

应用场景：String是最常用的一种数据类型，普通的key/value存储都可以归为此类；

实现方式：String在redis内部存储默认就是一个字符串，被redisObject所引用，当遇到incr、decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。

2）Hash

常用命令：hget/hset/hgetall等

应用场景：我们要存储一个用户信息对象数据，其中包括用户ID、用户姓名、年龄和生日，通过用户ID我们希望获取该用户的姓名或者年龄或者生日；

实现方式：Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口。如图所示，Key是用户ID, value是一个Map。这个Map的key是成员的属性名，value是属性值。这样对数据的修改和存取都可以直接通过其内部Map的Key(Redis里称内部Map的key为field), 也就是通过 key(用户ID) + field(属性标签) 就可以操作对应属性数据。当前HashMap的实现有两种方式：当HashMap的成员比较少时Redis为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的HashMap结构，这时对应的value的redisObject的encoding为zipmap，当成员数量增大时会自动转成真正的HashMap,此时encoding为ht。

hash

3）List

常用命令：lpush/rpush/lpop/rpop/lrange等；

应用场景：Redis list的应用场景非常多，也是Redis最重要的数据结构之一，比如twitter的关注列表，粉丝列表等都可以用Redis的list结构来实现；

实现方式：Redis list的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。

4）Set

常用命令：sadd/spop/smembers/sunion等；

应用场景：Redis set对外提供的功能与list类似是一个列表的功能，特殊之处在于set是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的；

实现方式：set 的内部实现是一个 value永远为null的HashMap，实际就是通过计算hash的方式来快速排重的，这也是set能提供判断一个成员是否在集合内的原因。

5）Sorted Set

常用命令：zadd/zrange/zrem/zcard等；

应用场景：Redis sorted set的使用场景与set类似，区别是set不是自动有序的，而sorted set可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择sorted set数据结构，比如twitter 的public timeline可以以发表时间作为score来存储，这样获取时就是自动按时间排好序的。

实现方式：Redis sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

7、redis的缓存失效策略和主键失效机制

　　作为缓存系统都要定期清理无效数据，就需要一个主键失效和淘汰策略.

　　在Redis当中，有生存期的key被称为volatile。在创建缓存时，要为给定的key设置生存期，当key过期的时候（生存期为0），它可能会被删除。

　　1、影响生存时间的一些操作

　　生存时间可以通过使用 DEL 命令来删除整个 key 来移除，或者被 SET 和 GETSET 命令覆盖原来的数据，也就是说，修改key对应的value和使用另外相同的key和value来覆盖以后，当前数据的生存时间不同。

　　比如说，对一个 key 执行INCR命令，对一个列表进行LPUSH命令，或者对一个哈希表执行HSET命令，这类操作都不会修改 key 本身的生存时间。另一方面，如果使用RENAME对一个 key 进行改名，那么改名后的 key的生存时间和改名前一样。

　　RENAME命令的另一种可能是，尝试将一个带生存时间的 key 改名成另一个带生存时间的 another\_key ，这时旧的 another\_key (以及它的生存时间)会被删除，然后旧的 key 会改名为 another\_key ，因此，新的 another\_key 的生存时间也和原本的 key 一样。使用PERSIST命令可以在不删除 key 的情况下，移除 key 的生存时间，让 key 重新成为一个persistent key 。

　　2、如何更新生存时间

　　可以对一个已经带有生存时间的 key 执行EXPIRE命令，新指定的生存时间会取代旧的生存时间。过期时间的精度已经被控制在1ms之内，主键失效的时间复杂度是O（1），

　　EXPIRE和TTL命令搭配使用，TTL可以查看key的当前生存时间。设置成功返回 1；当 key 不存在或者不能为 key 设置生存时间时，返回 0 。

　　最大缓存配置

　　在 redis 中，允许用户设置最大使用内存大小

　　server.maxmemory

　　默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。redis 内存数据集大小上升到一定大小的时候，就会实行数据淘汰策略。

　　redis 提供 6种数据淘汰策略：

　　． volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

　　． volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

　　． volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

　　． allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

　　． allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

　　． no-enviction（驱逐）：禁止驱逐数据

　　注意这里的6种机制，volatile和allkeys规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的lru、ttl以及random是三种不同的淘汰策略，再加上一种no-enviction永不回收的策略。

　　使用策略规则：

　　1、如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用allkeys-lru

　　2、如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用allkeys-random

　　三种数据淘汰策略：

　　ttl和random比较容易理解，实现也会比较简单。主要是Lru最近最少使用淘汰策略，设计上会对key 按失效时间排序，然后取最先失效的key进行淘汰

8.为什么redis需要把所有数据放到内存中?

　　　Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。在内存越来越便宜的今天，redis将会越来越受欢迎。

　　　如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

9.Redis是单进程单线程的

　　　redis利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销

10.redis的并发竞争问题如何解决?

　　　Redis为单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争，但是在Jedis客户端对Redis进行并发访问时会发生连接超时、数据转换错误、阻塞、客户端关闭连接等问题，这些问题均是

　　　　　由于客户端连接混乱造成。对此有2种解决方法：

　　　1.客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。

　　　2.服务器角度，利用setnx实现锁。

　　　注：对于第一种，需要应用程序自己处理资源的同步，可以使用的方法比较通俗，可以使用synchronized也可以使用lock；第二种需要用到Redis的setnx命令，但是需要注意一些问题。

11、redis常见性能问题和解决方案：

　　　1).Master写内存快照，save命令调度rdbSave函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以Master最好不要写内存快照。

　　　2).Master AOF持久化，如果不重写AOF文件，这个持久化方式对性能的影响是最小的，但是AOF文件会不断增大，AOF文件过大会影响Master重启的恢复速度。Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久

　　　　化,如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。

　　　3).Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。

　　　4). Redis主从复制的性能问题，为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内。

12.redis事物的了解CAS(check-and-set 操作实现乐观锁 )?

　　　　和众多其它数据库一样，Redis作为NoSQL数据库也同样提供了事务机制。在Redis中，MULTI/EXEC/DISCARD/WATCH这四个命令是我们实现事务的基石。相信对有关系型数据库开发经验的开发者而言这一概念并不陌生，即便如此，我们还是会简要的列出

　　　　Redis中

　　事务的实现特征：

　　　　1). 在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。

　　　　2). 和关系型数据库中的事务相比，在Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行。

　　　　3). 我们可以通过MULTI命令开启一个事务，有关系型数据库开发经验的人可以将其理解为"BEGIN TRANSACTION"语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行EXEC/DISCARD命令来提交/回滚该事务内的所有操作。这两个Redis命令可被视为等同于关系型数据库中的COMMIT/ROLLBACK语句。

　　　　4). 在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行EXEC命令之后，那么该事务中的所有命令都会被服务器执行。

　　　　5). 当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了。

13.WATCH命令和基于CAS的乐观锁?

　　　在Redis的事务中，WATCH命令可用于提供CAS(check-and-set)功能。假设我们通过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Null multi-bulk应答以通知调用者事务

　执行失败。例如，我们再次假设Redis中并未提供incr命令来完成键值的原子性递增，如果要实现该功能，我们只能自行编写相应的代码。其伪码如下：

　　val = GET mykey

　　val = val + 1

　　SET mykey $val

　　以上代码只有在单连接的情况下才可以保证执行结果是正确的，因为如果在同一时刻有多个客户端在同时执行该段代码，那么就会出现多线程程序中经常出现的一种错误场景--竞态争用(race condition)。比如，客户端A和B都在同一时刻读取了mykey的原有值，假设该值为10，此后两个客户端又均将该值加一后set回Redis服务器，这样就会导致mykey的结果为11，而不是我们认为的12。为了解决类似的问题，我们需要借助WATCH命令的帮助，见如下代码：

　　WATCH mykey

　　val = GET mykey

　　val = val + 1

　　MULTI

　　SET mykey $val

　　EXEC

　　和此前代码不同的是，新代码在获取mykey的值之前先通过WATCH命令监控了该键，此后又将set命令包围在事务中，这样就可以有效的保证每个连接在执行EXEC之前，如果当前连接获取的mykey的值被其它连接的客户端修改，那么当前连接的EXEC命令将执行失败。这样调用者在判断返回值后就可以获悉val是否被重新设置成功。

14.使用过Redis分布式锁么，它是什么回事？

　　先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。

　　这时候对方会告诉你说你回答得不错，然后接着问如果在setnx之后执行expire之前进程意外crash或者要重启维护了，那会怎么样？

　　这时候你要给予惊讶的反馈：唉，是喔，这个锁就永远得不到释放了。紧接着你需要抓一抓自己得脑袋，故作思考片刻，好像接下来的结果是你主动思考出来的，然后回答：我记得set指令有非常复杂的参数，这个应该是可以同时把setnx和expire合成一条指令来用的！对方这时会显露笑容，心里开始默念：摁，这小子还不错。

15.假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来？

　　使用keys指令可以扫出指定模式的key列表。

　　对方接着追问：如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题？

　　这个时候你要回答redis关键的一个特性：redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

16.使用过Redis做异步队列么，你是怎么用的？

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

　　如果对方追问可不可以不用sleep呢？list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。

　　如果对方追问能不能生产一次消费多次呢？使用pub/sub主题订阅者模式，可以实现1:N的消息队列。

　　如果对方追问pub/sub有什么缺点？在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。

　　如果对方追问redis如何实现延时队列？我估计现在你很想把面试官一棒打死如果你手上有一根棒球棍的话，怎么问的这么详细。但是你很克制，然后神态自若的回答道：使用sortedset，拿时间戳作为score，消息内容作为key调用zadd来生产消息，消费者用zrangebyscore指令获取N秒之前的数据轮询进行处理。

　　到这里，面试官暗地里已经对你竖起了大拇指。但是他不知道的是此刻你却竖起了中指，在椅子背后。

17.如果有大量的key需要设置同一时间过期，一般需要注意什么？

　　如果大量的key过期时间设置的过于集中，到过期的那个时间点，redis可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值，使得过期时间分散一些。

18.Redis如何做持久化的？

　　bgsave做镜像全量持久化，aof做增量持久化。因为bgsave会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要aof来配合使用。在redis实例重启时，会使用bgsave持久化文件重新构建内存，再使用aof重放近期的操作指令来实现完整恢复重启之前的状态。

　　对方追问那如果突然机器掉电会怎样？取决于aof日志sync属性的配置，如果不要求性能，在每条写指令时都sync一下磁盘，就不会丢失数据。但是在高性能的要求下每次都sync是不现实的，一般都使用定时sync，比如1s1次，这个时候最多就会丢失1s的数据。

　　对方追问bgsave的原理是什么？你给出两个词汇就可以了，fork和cow。fork是指redis通过创建子进程来进行bgsave操作，cow指的是copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

19.Pipeline有什么好处，为什么要用pipeline？

　　可以将多次IO往返的时间缩减为一次，前提是pipeline执行的指令之间没有因果相关性。使用redis-benchmark进行压测的时候可以发现影响redis的QPS峰值的一个重要因素是pipeline批次指令的数目。

20.Redis的同步机制了解么？

　　Redis可以使用主从同步，从从同步。第一次同步时，主节点做一次bgsave，并同时将后续修改操作记录到内存buffer，待完成后将rdb文件全量同步到复制节点，复制节点接受完成后将rdb镜像加载到内存。加载完成后，再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

21.是否使用过Redis集群，集群的原理是什么？

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

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

1，什么是Redis？

Redis是一款开源的、高性能的键-值（key-value）数据库。Redis是Remote Dictionary Server的简称。

2，Redis有哪些数据类型？

字符串、集合、有序集合、列表、哈希

如果对redis有更深入了解，加上 HyperLogLog , Geo , Pub/Sub

另外， 可以加上 Redis 的 BloomFilter（布隆过滤器) ,Bit操作，Lua脚本等等。

提示： 除了5种基本类型，剩下的类型当然是说的越多越好，但是注意，前提是你真的用过，因为面试官可能眼前一亮，然后往深了问。这时候如果答不上来就GG了。

3，Redis有什么特点？

Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过 10万次读写操作，是已知性能最快的Key-Value DB。

Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存多种数据结构，此外单个value的最大限制是1GB，不像 memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性 能消息队列服务，用他的Set可以做高性能的tag系统等等。

另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一 个功能加强版的memcached来用。

4，Redis有什么缺点？

Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

5，什么叫缓存雪崩？

第一种情况：如果缓存数据设置的过期时间是相同的，并且Redis恰好将这部分数据全部删掉，这就会导致这段时间之内，这些缓存数据失效，那么所有请求全部走数据库。

第二种情况：Redis全部挂掉，所有的请求也是走数据库的。

6，怎么解决缓存雪崩？

由于缓存雪崩会导致所有请求直接走数据库，这样数据库压力会很大，甚至拖垮数据库，怎么办呢？

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。

常用解决方法：

加锁排队. 限流-- 限流算法

1.计数 2.滑动窗口 3. 令牌桶Token Bucket 4.漏桶 leaky bucket [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。

业界比较常用的做法，是使用mutex。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。

SETNX，是「SET if Not eXists」的缩写，也就是只有不存在的时候才设置，可以利用它来实现锁的效果。

数据预热

可以通过缓存reload机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

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。

缓存永远不过期

这里的“永远不过期”包含两层意思：

(1) 从缓存上看，确实没有设置过期时间，这就保证了，不会出现热点key过期问题，也就是“物理”不过期。

(2) 从功能上看，如果不过期，那不就成静态的了吗？所以我们把过期时间存在key对应的value里，如果发现要过期了，通过一个后台的异步线程进行缓存的构建，也就是“逻辑”过期。从实战看，这种方法对于性能非常友好，唯一不足的就是构建缓存时候，其余线程(非构建缓存的线程)可能访问的是老数据，但是对于一般的互联网功能来说这个还是可以忍受。

对于“挂机”的这种雪崩：

事发前：实现Redis的高可用(主从架构+Sentinel 或者Redis Cluster)，尽量避免Redis挂掉这种情况发生。

事发中：万一Redis真的挂了，我们可以设置本地缓存(ehcache)+限流(hystrix)，尽量避免我们的数据库被干掉(起码能保证我们的服务还是能正常工作的)

事发后：redis持久化，重启后自动从磁盘上加载数据，快速恢复缓存数据。

7，Redis数据淘汰策略有几种？

noeviction:返回错误当内存限制达到，并且客户端尝试执行会让更多内存被使用的命令。

allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

allkeys-random: 回收随机的键使得新添加的数据有空间存放。

volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

8，Redis有哪些使用场景？

会话缓存

最常用的一种使用Redis的情景是会话缓存（session cache），用Redis缓存会话比其他存储（如Memcached）的优势在于：Redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用Redis来缓存会话的文档。甚至广为人知的商业平台Magento也提供Redis的插件。

全页缓存

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。

再次以Magento为例，Magento提供一个插件来使用Redis作为全页缓存后端。

此外，对WordPress的用户来说，Pantheon有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如Python）对 list 的 push/pop 操作。

如果你快速的在Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用Redis创建非常好的后端工具，以满足各种队列需求。例如，Celery有一个后台就是使用Redis作为broker，你可以从这里去查看。

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。

发布/订阅

最后（但肯定不是最不重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！

9，Redis相比memcached的优势有哪些？

(1) memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型

(2) redis的速度比memcached快很多

(3) redis可以持久化其数据

10，Redis为什么快？

Redis是纯内存数据库，一般都是简单的存取操作，线程占用的时间很多，时间的花费主要集中在IO上，所以读取速度快。

Redis使用的是非阻塞IO，IO多路复用，使用了单线程来轮询描述符，将数据库的开、关、读、写都转换成了事件，减少了线程切换时上下文的切换和竞争。

Redis采用了单线程的模型，保证了每个操作的原子性，也减少了线程的上下文切换和竞争。

数据结构也帮了不少忙，Redis全程使用hash结构，读取速度快，还有一些特殊的数据结构，对数据存储进行了优化，如压缩表，对短数据进行压缩存储，再如，跳表，使用有序的数据结构加快读取的速度。

Redis采用自己实现的事件分离器，效率比较高，内部采用非阻塞的执行方式，吞吐能力比较大。

11，Redis的java实现客户端有哪些？

Redisson,Jedis，lettuce等等，官方推荐使用Redisson。

12，Redis的持久化有几种方式？各有什么不同？

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 提供了两种持久化方式:

RDB（默认）

AOF

RDB：

rdb是Redis DataBase缩写

功能核心函数rdbSave(生成RDB文件)和rdbLoad（从文件加载内存）两个函数

AOF:

Aof是Append-only file缩写

每当执行服务器(定时)任务或者函数时flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作

aof写入保存：

WRITE：根据条件，将 aof\_buf 中的缓存写入到 AOF 文件

SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

存储结构:

内容是redis通讯协议(RESP )格式的命令文本存储。

比较：

1、aof文件比rdb更新频率高，优先使用aof还原数据。

2、aof比rdb更安全也更大

3、rdb性能比aof好

4、如果两个都配了优先加载AOF

13,什么是缓存穿透？如何避免？

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）。一些恶意的请求会故意查询不存在的key,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。

## Redis数据淘汰策略

为了保证redis中的数据都是热点数据，redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

noeviction:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但DEL和几个例外）

allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

allkeys-random: 回收随机的键使得新添加的数据有空间存放。

volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

### Redis回收进程

一个客户端运行了新的命令，添加了新的数据。

Redi检查内存使用情况，如果大于maxmemory的限制, 则根据设定好的策略进行回收。

一个新的命令被执行，等等。

所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地回收回到边界以下。

如果一个命令的结果导致大量内存被使用（例如很大的集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超越。

### Redis回收算法

LRU算法

## Redis适用场景

（1）、会话缓存（Session Cache）

最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）的优势在于Redis提供持久化。

（2）、全页缓存（FPC）

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。

再次以Magento为例，Magento提供一个插件来使用Redis作为全页缓存后端。

此外，对WordPress的用户来说，Pantheon有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

（3）、队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。

（4），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，Set和Sorted Set也使得执行这些操作变的非常简单，Redis只是正好提供了这两种数据结构。要从排序集合中获取到排名最靠前的10个用户，只需要像下面一样执行即可：

当然，这是假定根据用户的分数做递增的排序。如果想返回用户及用户的分数，需要这样执行：

ZRANGE user\_scores 0 10 WITHSCORES

（5）、发布/订阅

发布/订阅的使用场景非常多，可以在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！

## Redis管道

一次请求/响应服务器能实现处理新的请求即使旧的请求还未被响应。这样就可以将多个命令发送到服务器，而不用等待回复，最后在一个步骤中读取该答复。

这就是管道（pipelining），是一种几十年来广泛使用的技术。例如许多POP3协议已经实现支持这个功能，大大加快了从服务器下载新邮件的过程。

## Redis分区

分区可以让Redis管理更大的内存，Redis将可以使用所有机器的内存。如果没有分区，最多只能使用一台机器的内存。分区使Redis的计算能力通过简单地增加计算机得到成倍提升,Redis的网络带宽也会随着计算机和网卡的增加而成倍增长。

### Redis分区实现方案

**方案一：客户端分区**

客户端分区就是在客户端就已经决定数据会被存储到哪个redis节点或者从哪个redis节点读取。大多数客户端已经实现了客户端分区。

**方案二：代理分区**

代理分区意味着客户端将请求发送给代理，然后代理决定去哪个节点写数据或者读数据。代理根据分区规则决定请求哪些Redis实例，然后根据Redis的响应结果返回给客户端。redis和memcached的一种代理实现就是Twemproxy。

Twemproxy支持自动分区，如果其代理的其中一个Redis节点不可用时，会自动将该节点排除（这将改变原来的keys-instances的映射关系，所以应该仅在把Redis当缓存时使用Twemproxy)。 Twemproxy本身不存在单点问题，可以启动多个Twemproxy实例，然后让客户端去连接任意一个Twemproxy实例。 Twemproxy是Redis客户端和服务器端的一个中间层，由它来处理分区功能不算复杂且比较可靠。

**方案三：查询路由**

查询路由(Query routing) 的意思是客户端随机地请求任意一个redis实例，然后由Redis将请求转发给正确的Redis节点。Redis Cluster实现了一种混合形式的查询路由，但并不是直接将请求从一个redis节点转发到另一个redis节点，而是在客户端的帮助下直接redirected到正确的redis节点。

### Redis分区缺点

涉及多个key的操作通常不会被支持。例如你不能对两个集合求交集，因为他们可能被存储到不同的Redis实例（实际上这种情况也有办法，但是不能直接使用交集指令）。

同时操作多个key,则不能使用Redis事务.

分区使用的粒度是key，不能使用一个非常长的排序key存储一个数据集（The partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set）.

当使用分区的时候，数据处理会非常复杂，例如为了备份你必须从不同的Redis实例和主机同时收集RDB / AOF文件。

分区时动态扩容或缩容可能非常复杂。Redis集群在运行时增加或者删除Redis节点，能做到最大程度对用户透明地数据再平衡，但其他一些客户端分区或者代理分区方法则不支持这种特性。然而，有一种预分片的技术也可以较好的解决这个问题。