BD大数据技术之Redis

(作者：BD大数据研发部)

版本：V1.0

# 第1章 NoSQL简介

NoSQL是Not only SQL的缩写，大意为“不仅仅是SQL”，说明这项技术是传统关系型数据库的补充而非替代。在整个NoSQL技术栈中MemCache、Redis、MongoDB被称为NoSQL三剑客。那么时代为什么需要NoSQL数据库呢？我们来做个对比：

|  | **关系型数据库(MySQL)** | **NoSQL数据库(Redis)** |
| --- | --- | --- |
| **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 | 基于乐观锁的松散事务控制 |

所以NoSQL数据库的最大优势体现为：高性能、高可用性和可伸缩性。

# 第2章 Redis简介

## 2.1 概念

Redis 是一个开源的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持多种类型的数据结构，如字符串（strings），散列（hashes）， 列表（lists），集合（sets），有序集合（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命令参考文档网址：http://redisdoc.com

## 2.2 官方地址

1）官网地址

<https://redis.io/>

2）下载地址

<https://redis.io/download>

3）命令参考地址

<http://redisdoc.com>

## 2.3 应用场景

1）缓存

使用Redis可以建立性能非常出色的缓存服务器，查询请求先在Redis中查找所需要的数据，如果能够查询到（命中）则直接返回，大大减轻关系型数据库的压力。

2）数据临时存储位置

使用token（令牌）作为用户登录系统时的身份标识，这个token就可以在Redis中临时存储。

3）分布式环境下解决Session不一致问题时的Session库

Spring提供了一种技术解决分布式环境下Session不一致问题，叫SpringSession。而Redis就可以为SpringSession提供一个数据存储空间。

4）流式数据去重

在Redis中有一种数据类型是Set，和Java中的Set集合很像，不允许存储重复数据。借助这个特性我们可以在Redis中使用Set类型存储流式数据达到去重的目的。

# 第3章 Redis安装

## 3.1 上传并解压

1）将redis-4.0.2.tar.gz上传至Linux并解压

[atguigu@hadoop102 software]$ tar -zxvf redis-4.0.2.tar.gz -C /opt/module/

## 3.2 安装C语言编译环境

[atguigu@hadoop102 software]$ sudo yum install -y gcc-c++

## 3.3 修改安装位置

进入Redis的解压目录

[atguigu@hadoop102 software]$ cd ../module/redis-4.0.2/

[atguigu@hadoop102 redis-4.0.2]$ vim src/Makefile

修改

PREFIX?=/opt/module/redis

## 3.4 编译安装

1）编译

[atguigu@hadoop102 redis-4.0.2]$ make

... ...

... ...

LINK redis-server

INSTALL redis-sentinel

CC redis-cli.o

LINK redis-cli

CC redis-benchmark.o

LINK redis-benchmark

INSTALL redis-check-rdb

INSTALL redis-check-aof

Hint: It's a good idea to run 'make test' ;)

make[1]: 离开目录“/opt/module/redis-4.0.2/src”

2）安装

[atguigu@hadoop102 redis-4.0.2]$ make install

cd src && make install

make[1]: 进入目录“/opt/module/redis-4.0.2/src”

CC Makefile.dep

make[1]: 离开目录“/opt/module/redis-4.0.2/src”

make[1]: 进入目录“/opt/module/redis-4.0.2/src”

Hint: It's a good idea to run 'make test' ;)

INSTALL install

INSTALL install

INSTALL install

INSTALL install

INSTALL install

make[1]: 离开目录“/opt/module/redis-4.0.2/src”

## 3.5 启动Redis服务器

### 3.5.1 默认配置启动

[atguigu@hadoop102 redis-4.0.2]$ cd ../redis

[atguigu@hadoop102 redis]$ ll

总用量 4

drwxrwxr-x. 2 atguigu atguigu 4096 6月 17 14:14 bin

[atguigu@hadoop102 redis]$ bin/redis-server

### 3.5.2 定制配置启动

1）拷贝配置文件到当前目录

[atguigu@hadoop102 redis]$ cp /opt/module/redis-4.0.2/redis.conf ./

2）修改配置文件

[atguigu@hadoop102 redis]$ vim redis.conf

daemonize yes #后台启动

logfile /opt/module/redis/redis.log

dir /opt/module/redis

3）启动Redis

[atguigu@hadoop102 redis]$ bin/redis-server ./redis.conf

## 3.6 客户端登录

1）启动客户端

[atguigu@hadoop102 redis]$ bin/redis-cli

127.0.0.1:6379>

2）测试

127.0.0.1:6379> ping

PONG

127.0.0.1:6379>

# 第4章 Redis五种常用数据结构

## 4.1 总体结构

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。

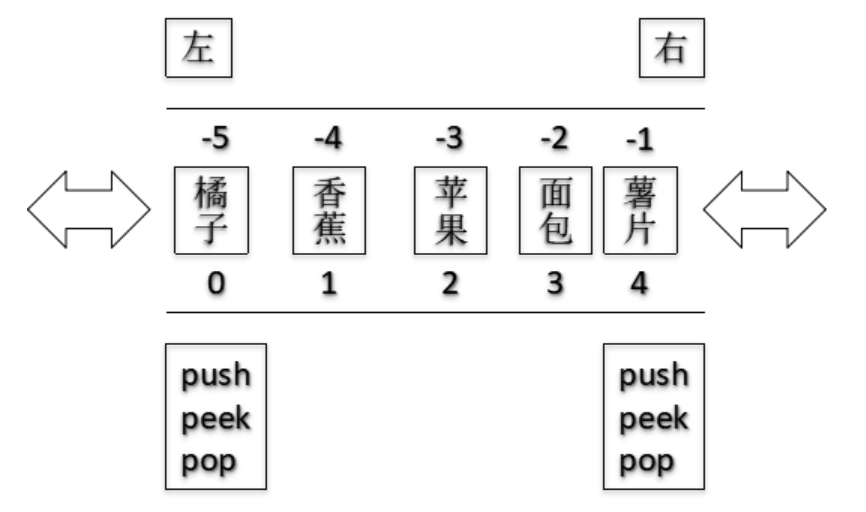
|  |  |
| --- | --- |
| KEY | VALUE |
| string |
| list |
| set |
| hash |
| zset |

## 4.2 String类型

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。

## 4.3 List类型

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。



## 4.4 Set类型

Redis的Set是String类型的无序集合。它是基于哈希表实现的。

## 4.5 Hash类型

本身就是一个键值对集合。可以当做Java中的Map<String,String>对待。

## 4.6 ZSet类型

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。

# 第5章 Redis命令行操作

## 5.1 基本操作

1）切换数据库

Redis默认有16个数据库。

# Set the number of databases. The default database is DB 0, you can select

# a different one on a per-connection basis using SELECT <dbid> where

# dbid is a number between 0 and 'databases'-1

database 16

可以使用select进行切换，数据库索引从0开始

127.0.0.1:6379> select 2

OK

127.0.0.1:6379[2]> select 0

OK

127.0.0.1:6379>

2）查看数据库长度

数据库长度指的是当前数据库里面存放了多少的Key。

127.0.0.1:6379> dbsize

(integer) 3

## 5.2 Key操作

1）查找符合格式的Key

KEYS PATTERN

\*代替多个字符，?代替一个字符

127.0.0.1:6379> KEYS \*

127.0.0.1:6379> KEYS ?

2）返回KEY对应的值的类型

TYPE KEY

3）把一组键值对数据移动到另一个数据库中

MOVE KEY DB

4）根据KEY进行删除，至少要指定一个KEY

DEL KEY [KEY ...]

5）检查指定的KEY是否存在。指定一个KEY时，存在返回1，不存在返回0。可以指定多个，返回存在的KEY的数量。

EXISTS KEY [KEY ...]

6）在现有的KEY中随机返回一个

RANDOMKEY -

7）重命名一个KEY，NEWKEY不管是否是已经存在的都会执行，如果NEWKEY已经存在则会被覆盖。

RENAME KEY NEWKEY

8）只有在NEWKEY不存在时能够执行成功，否则失败

RENAMENX KEY NEWKEY

9）以秒为单位查看KEY还能存在多长时间 -1表示永久有效

TTL KEY

10）给一个KEY设置在SECONDS秒后过期，过期会被Redis移除。

EXPIRE KEY SECONDS

11）移除过期时间，变成永久key

PERSIST KEY

## 5.3 String类型操作

1）给KEY设置一个String类型的值

EX参数用于设置存活的秒数。

PX参数用于设置存活的毫秒数。

NX参数表示当前命令中指定的KEY不存在才执行。(只能新增)

XX参数表示当前命令中指定的KEY存在才执行。(只能修改)

默认：既能新增，也能修改

SET KEY VALUE [EX SECONDS] [PX MILLISECONDS] [NX|XX]

2）根据key得到值

GET KEY

3）直接返回字符串长度

STRLEN KEY

4）把指定的value追加到KEY对应的原来的值后面，返回值是追加后字符串长度

APPEND KEY VALUE

5）自增1

INCR KEY

6）自减1

DECR KEY

7）原值+INCREMENT

INCRBY KEY INCREMENT

8）原值-DECREMENT

DECRBY KEY DECREMENT

9）从字符串中取指定的一段

GETRANGE KEY START END

10）从offset开始使用VALUE进行替换

SETRANGE KEY OFFSET VALUE

11）设置KEY,VALUE时指定存在秒数

SETEX KEY SECONDS VALUE

12）新建字符串类型的键值对（不存在时才能设置成功）

SETNX KEY VALUE

13）一次性设置(新建和修改)一组多个键值对

MSET KEY VALUE [KEY VALUE ...]

14）一次性指定多个KEY，返回它们对应的值，没有值的KEY返回值是(nil)

MGET KEY [KEY ...]

15）一次性新建(不能修改)多个值

MSETNX KEY VALUE [KEY VALUE ...]

16）设置新值，同时能够将旧值返回

GETSET KEY VALUE

## 5.4 List类型操作

1）从左边添加数据

LPUSH key value [value ...]

2）从右边添加数据

RPUSH key value [value ...]

3）根据list集合的索引打印元素数据

正着数：0,1,2,3,...

倒着数：-1,-2,-3,...

LRANGE key start stop

4）list集合的长度

LLEN key

5）从左边弹出一个元素。

弹出=返回+删除。

LPOP key

6）从右边弹出一个元素。

RPOP key

7）从source中RPOP一个元素，LPUSH到destination中

RPOPLPUSH source destination

8）根据索引从集合中取值

LINDEX key index

9）在pivot指定的值前面或后面插入value

LINSERT key BEFORE|AFTER pivot value

10）只能针对存在的list执行LPUSH

LPUSHX key value

11）根据count指定的数量从key对应的list中删除value

LREM key count value

12）把指定索引位置的元素替换为另一个值

LSET key index value

13）仅保留指定区间的数据，两边的数据被删除

LTRIM key start stop

## 5.5 Set类型操作

1）向set集合添加数据

SADD key member [member ...]

2）查看set集合所有数据

SMEMBERS key

3）返回集合中元素的数量

SCARD key

4）检查当前指定member是否是集合中的元素

SISMEMBER key member

5）从集合中删除元素

SREM key member [member ...]

6）将指定的集合进行“交集”操作

集合A：a,b,c

集合B：b,c,d

交集：b,c

SINTER key [key ...]

7）取交集后存入destinationsin

SINTERSTORE destination key [key ...]

8）将指定的集合执行“差集”操作

集合A：a,b,c

集合B：b,c,d

A对B执行diff：a

相当于：A-交集部分

SDIFF key [key ...]

SDIFFSTORE destination key [key ...]

9）将指定的集合执行“并集”操作

集合A：a,b,c

集合B：b,c,d

并集：a,b,c,d

SUNION key [key ...]

SUNIONSTORE destination key [key ...]

10）把member从source移动到destination

SMOVE source destination member

11）从集合中随机返回count个数量的元素，count不指定就返回1个

SRANDMEMBER key [count]

12）从集合中随机弹出count个数量的元素，count不指定就弹出1个

SPOP key [count]

注:集合里的元素弹完以后,集合也会消失.

## 5.6 Hash类型操作

1）添加数据

HSET key field value

2）查询一个key的所有数据

HGETALL key

3）查询一个key对应field的数据

HGET key field

4）查询一个key的field个数

HLEN key

5）查询一个key的所有field

HKEYS key

6）查询一个key的所有value

HVALS key

7）判断一个key中是否存在指定的field

HEXISTS key field

8）删除一个key指定的field

HDEL key field [field ...]

9）给一个key的某个field对应的value增加increment值

HINCRBY key field increment

10）批量获取一个key的多个field

HMGET key field [field ...]

11）批量添加一个key的多个field-value

HMSET key field value [field value ...]

12）在字段(field)不存在时添加数据

HSETNX key field value

## 5.7 ZSet类型操作

1）添加数据

ZADD key [NX|XX] [CH] [INCR] score member [score member ...]

 **XX**: 仅仅更新存在的成员，不添加新成员。

 **NX**: 不更新存在的成员。只添加新成员。

 **CH**: 修改返回值为发生变化的成员总数，原始是返回新添加成员的总数 (CH 是 *changed* 的意思)。更改的元素是**新添加的成员**，已经存在的成员**更新分数**。 所以在命令中指定的成员有相同的分数将不被计算在内。注：在通常情况下，ZADD返回值只计算新添加成员的数量。

 **INCR:** 当ZADD指定这个选项时，成员的操作就等同ZINCRBY命令，对成员的分数进行递增操作

2）在指定范围内返回数据([WITHSCORES]带上分数返回)

ZRANGE key start stop [WITHSCORES]

3）查询key对应的数据个数

ZCARD key

4）返回对应数据的分数

ZSCORE key member

5）给某个元素增加分数

ZINCRBY key increment member

6）在分数的指定区间内返回数据（Limit+返回结果的下标位置+返回个数）

ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]

7）先对分数进行升序排序，返回member的排名

ZRANK key member

8）先对分数进行降序排序，返回member的排名

ZREVRANK key member

9）删除元素

ZREM key member [member ...]

# 第6章 Redis持久化机制

Redis工作时数据都存储在内存中，万一服务器断电，则所有数据都会丢失。针对这种情况，Redis采用持久化机制来增强数据安全性。

## 6.1 RDB

### 6.1.1 机制描述

每隔一定的时间把内存中的数据作为一个快照保存到硬盘上的文件中。Redis默认开启RDB机制。

### 6.1.2 触发时机

1）基于默认配置

save 900 1

save 300 10

save 60 10000

2）使用保存命令

save或bgsave

3）使用flushall命令会清空所有库

这个命令也会产生dump.rdb文件，但里面是空的，没有意义

4）服务器关闭

如果执行SHUTDOWN命令让Redis正常退出，那么此前Redis就会执行一次持久化保存。

### 6.1.3 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 |

### 6.1.4 思考

RDB机制能够保证数据的绝对安全吗？

## 6.2 AOF

### 6.2.1 机制描述

根据配置文件中指定的策略，把生成数据的命令保存到硬盘上的文件中。

1）一个AOF文件的内容可以参照下面的例子：

\*2

$6

SELECT

$1

0

\*3

$3

set

$3

num

$2

10

\*2

$4

incr

$3

num

\*2

$4

incr

$3

num

\*2

$4

incr

$3

num

2）生成上面文件内容的Redis命令

set num 10

incr num

incr num

incr num

### 6.2.2 AOF基本配置

| **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。折中。 |
| no | 由操作系统在适当的时候执行写入操作，最快。 |

### 6.2.3 AOF重写

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。

## 6.3 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

bin/redis-check-aof --fix ./appendonly.aof

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。

## 6.4 扩展阅读

### 6.4.1 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

### 6.4.2 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。

### 6.4.3 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。

### 6.4.4 使用建议

如果Redis仅仅作为缓存可以不使用任何持久化方式。

其他应用方式综合考虑性能和完整性、一致性要求。

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Enable 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同时挂掉，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。

# 第7章 Redis事务控制

## 7.1 Redis事务控制相关命令

| **命令名** | **作用** |
| --- | --- |
| MULTI | 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 | 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 |
| UNWATCH | 放弃监控一个KEY |

## 7.2 命令队列执行失败的两种情况

### 7.2.1 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.

### 7.2.2 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"

### 7.2.3 Redis为什么不支持回滚

官方解释如下：

如果你有使用关系型数据库的经验，那么“Redis在事务失败时不进行回滚，而是继续执行余下的命令”这种做法可能会让你觉得有点奇怪。以下是这种做法的优点：

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。举个例子，如果你本来想通过 INCR 命令将键的值加上1，却不小心加上了2，又或者对错误类型的键执行了INCR，回滚是没有办法处理这些情况的。

## 7.3 Watch

案例演示

127.0.0.1:6379> set salary 3300

OK

127.0.0.1:6379> GET salary

"3300"

127.0.0.1:6379> WATCH salary

OK

127.0.0.1:6379> MULTI

OK

127.0.0.1:6379> INCRBY salary 50

QUEUED

127.0.0.1:6379> INCRBY salary 50

QUEUED

127.0.0.1:6379> INCRBY salary 50

QUEUED

127.0.0.1:6379> INCRBY salary 50

QUEUED

在另一个客户端执行

127.0.0.1:6379> GET salary

"3300"

127.0.0.1:6379> SET salary 5000

OK

127.0.0.1:6379> get salary

"5000"

回到之前的客户端

127.0.0.1:6379> EXEC

(nil)

127.0.0.1:6379> get salary

"5000"

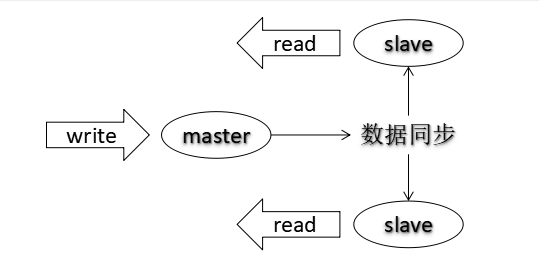
## 7.4 悲观锁和乐观锁

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。

1）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。

2）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。

# 第8章 Redis主从复制机制



## 8.1 读写分离的好处

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。

## 8.2 搭建步骤

### 8.2.1 思路

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。

daemonize yes

dir /opt/module/redis/cluster

不同的参数有：

| **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 | /opt/module/redis/cluster/redis6000.log /opt/module/redis/cluster/redis7000.log /opt/module/redis/cluster/redis8000.log |
| pidfile | pid文件位置 | /var/run/redis\_6000.pid /var/run/redis\_7000.pid /var/run/redis\_8000.pid |

### 8.2.2 步骤

第一步：创建/opt/module/redis/cluster目录

第二步：把原始未经修改的redis.conf复制到/opt/module/redis/cluster目录

第三步：把/opt/module/redis/cluster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 /opt/module/redis/cluster

port 6000

dbfilename dump6000.rdb

logfile /opt/module/redis/cluster/redis6000.log

pidfile /var/run/redis\_6000.pid

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 7000

dbfilename dump7000.rdb

logfile /opt/module/redis/cluster/redis7000.log

pidfile /var/run/redis\_7000.pid

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 8000

dbfilename dump8000.rdb

logfile /opt/module/redis/cluster/redis8000.log

pidfile /var/run/redis\_8000.pid

### 8.2.3 启动

#服务

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

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

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

#客户端

bin/redis-cli -p 6000

bin/redis-cli -p 7000

bin/redis-cli -p 8000

## 8.3 主从关系

### 8.3.1 查看主从关系

127.0.0.1:6000> info replication

# Replication

role:master

connected\_slaves:0

### 8.3.2 设定主从关系(小弟找大哥)

SLAVEOF 127.0.0.1 6000

### 8.3.3 取消主从关系

SLAVEOF NO ONE

## 8.4 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

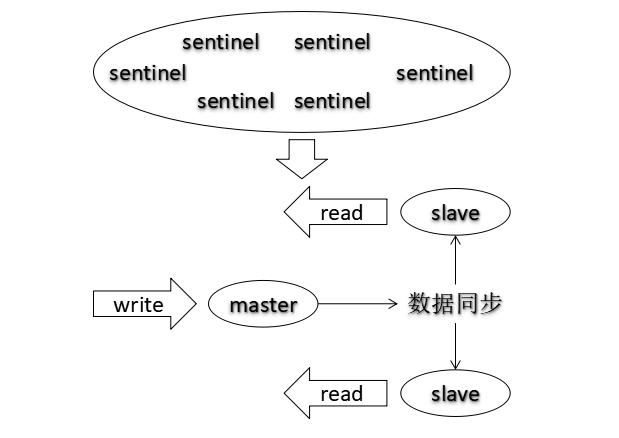
测试4：主机恢复启动，看从机状态

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。

## 8.5 哨兵模式

### 8.5.1 作用

通过哨兵服务器监控master/slave实现主从复制集群的自动管理。



### 8.5.2 相关概念

1）主观下线：1台哨兵检测到某节点服务器下线。

2）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。

### 8.5.3 配置方式

简单起见我们只配置一台哨兵。我们所需要做的就是创建一个哨兵服务器运行所需要的配置文件。

vim /opt/module/redis/cluster/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 |

### 8.5.4 启动哨兵

bin/redis-server cluster/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】

# 第9章 发布订阅（了解）

## 9.1 订阅一个频道

127.0.0.1:6379> SUBSCRIBE cctv

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

1) "subscribe"

2) "cctv"

3) (integer) 1

## 9.2 在一个频道上发布消息

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"

# 第10章 Jedis

## 10.1 一个对比

|  | **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 |

## 10.2 Redis准备

### 10.2.1 理解Redis配置文件中bind配置项的含义

bind后面跟的ip地址是客户端访问Redis时使用的IP地址。看下面例子：

| **bind值** | **访问方式** |
| --- | --- |
| 127.0.0.1 | ./redis-cli -h 127.0.0.1 |
| hadoop102 | ./redis-cli -h hadoop102 |

### 10.2.2 查看Linux系统本机IP

远程客户端访问Linux服务器时不能使用127.0.0.1，要使用网络上的实际IP。可以用ifconfig命令查看。

### 10.2.3 将Redis配置文件中bind配置项设置为本机IP

bind [你的实际IP]

bind hadoop102

## 10.3 Jedis

<dependencies>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.9.0</version>

</dependency>

</dependencies>

//指定Redis服务器的IP地址和端口号

Jedis jedis = new Jedis("hadoop102", 6379);

​

//执行ping命令

String ping = jedis.ping();

​

System.out.println(ping);

​

//关闭连接

jedis.close();

## 10.4 JedisPool

//声明Linux服务器IP地址

String host = " hadoop102";

​

//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();