Redis基础

课程目标

- 能够掌握Redis不同数据类型操作
- 能够使用Java API操作Redis
- 能够理解Redis的两种持久化方式

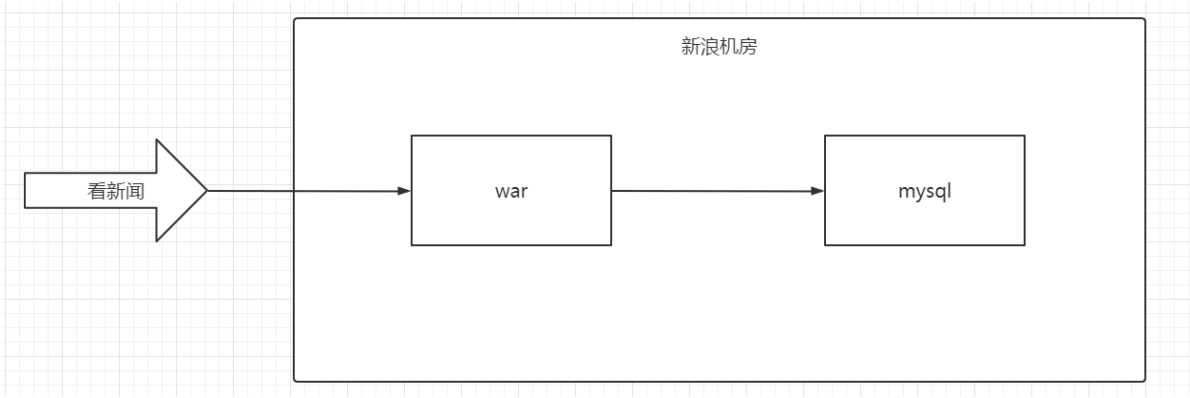
#第一章 NoSQL数据库发展历史简介

#1、 Web的历史发展历程

> web1.0时代简介

- web 1.0是以编辑为特征，网站提供给用户的内容是网站编辑进行编辑处理后提供的，用户阅读网站提供的内容
- 这个过程是网站到用户的**单向行为**
- web1.0时代的代表站点为新浪，搜狐，网易三大门户

性能要求：基本上就是一些简单的静态页面渲染，不会涉及太多复杂业务逻辑，功能简单单一，基本上服务器性能不会有太大压力

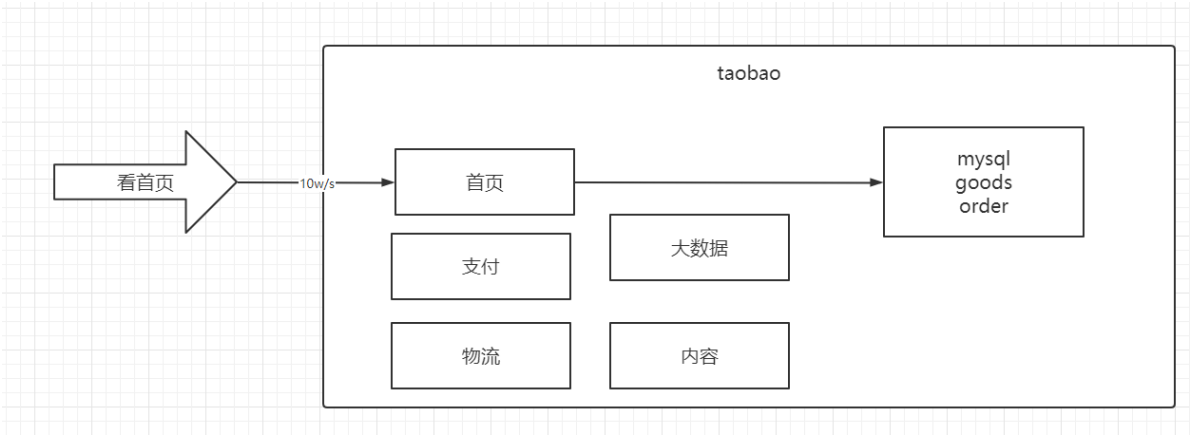


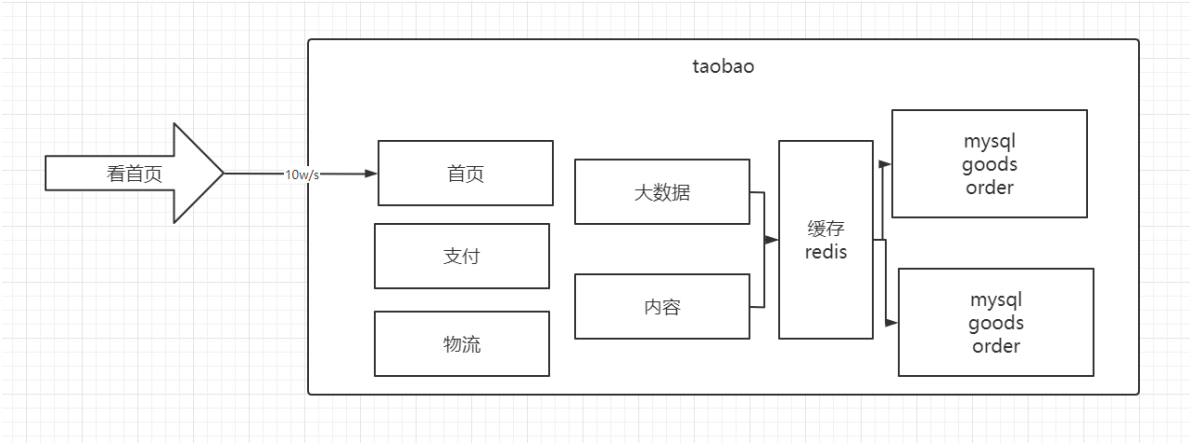
> web2.0时代简介

- 更注重用户的**交互**作用，用户既是网站内容的消费者（浏览者），也是网站内容的制造者
- 加强了网站与用户之间的**互动**，网站内容基于用户提供（微博、天涯社区、自媒体）大数据推荐
- 网站的诸多功能也由用户参与建设，实现了网站与用户双向的交流与参与
- 用户在web2.0网站系统内拥有自己的数据，并完全基于WEB，所有功能都能通过浏览器完成。

性能要求：

- 随着Web2.0时代到来，用户访问量大幅提升，同时产生大量用户数据。加上后来移动设备普及，所有的互联网平台都面临了巨大的性能挑战。**数据库服务器压力越来越大**
- 如何应对数据库的压力成为的关键。不管任何应用，数据始终是核心。传统关系型数据库根本无法承载较高的并发，此时人们就开始**用Redis当成缓存，来缓解数据库的压力**





#2、什么是NoSQL

- NoSQL最常见的解释是"non-relational"， 很多人也说它是"Not Only SQL"
- NoSQL仅仅是一个概念，泛指非关系型的数据库
- 区别于关系数据库，它们不保证关系数据的ACID特性
- NoSQL是一项全新的数据库革命性运动，提倡运用非关系型的数据存储，相对于铺天盖地的关系型数据库运用，这一概念无疑是一种全新的思维的注入

#3、 NoSQL的特点 关系型数据库的补充

应用场景：

- 高并发的读写 10w/s
- 海量数据读写
- 高可扩展性 不限制语言、lua脚本增强
- 速度快

不适用场景：

- 需要事务支持
- 基于sql的结构化查询存储，处理复杂的关系，需要即席查询（用户自定义查询条件的查询）

  政府银行金融项目，还是使用关系型数据库。oracle

#4、 NoSQL数据库

> memcache

- 很早出现的NoSql数据库
- 数据都在内存中，一般不持久化
- 支持简单的key-value模式
- 一般是作为缓存数据库辅助持久化的数据库

> redis介绍

- 几乎覆盖了Memcached的绝大部分功能
- 数据都在内存中，支持持久化，主要用作备份恢复
- 除了支持简单的key-value模式，还支持多种数据结构的存储，比如 list、set、hash、zset等。
- 一般是作为缓存数据库辅助持久化的数据库
- 现在市面上用得非常多的一款内存数据库

> mongoDB介绍

- 高性能、开源、模式自由(schema free)的文档型数据库
- 数据都在内存中， 如果内存不足，把不常用的数据保存到硬盘
- 虽然是key-value模式，但是对value（尤其是json）提供了丰富的查询功能
- 支持二进制数据及大型对象
- 可以根据数据的特点替代RDBMS，成为独立的数据库。或者配合RDBMS，存储特定的数据。

> 列式存储HBase介绍

HBase是**Hadoop**项目中的数据库。它用于需要对大量的数据进行随机、实时读写操作的场景中。HBase的目标就是处理数据量非常庞大的表，可以用普通的计算机处理超过10亿行数据，还可处理有数百万列元素的数据表。

#第二章 Redis介绍

redis官网地址：

https://redis.io/open in new window

中文网站

http://www.redis.cn/open in new window

#1、 Redis的基本介绍

- Redis是当前比较热门的NoSQL系统之一
- 它是一个开源的、使用ANSI C语言编写的**key-value**存储系统（区别于MySQL的二维表格形式存储）
- 和Memcache类似，但很大程度补偿了Memcache的不足，Redis数据都是缓存在计算机**内存**中，不同的是，Memcache只能将数据缓存到内存中，无法自动定期写入硬盘，这就表示，一断电或重启，内存清空，数据丢失

#2、 Redis的应用场景

#2.1 取最新N个数据的操作

比如典型的取网站最新文章，可以将最新的5000条评论ID放在Redis的List集合中，并将超出集合部分从数据库获取

#2.2.排行榜应用，取TOP N操作

这个需求与上面需求的不同之处在于，前面操作以时间为权重，这个是以某个条件为权重，比如按顶的次数排序，可以使用Redis的sorted set，将要排序的值设置成sorted set的score，将具体的数据设置成相应的value，每次只需要执行一条ZADD命令即可。

#2.3需要精准设定过期时间的应用

比如可以把上面说到的sorted set的score值设置成过期时间的时间戳，那么就可以简单地通过过期时间排序，定时清除过期数据了，不仅是清除Redis中的过期数据，你完全可以把Redis里这个过期时间当成是对数据库中数据的索引，用Redis来找出哪些数据需要过期删除，然后再精准地从数据库中删除相应的记录。

#2.4计数器应用

Redis的命令都是原子性的，你可以轻松地利用INCR，DECR命令来构建计数器系统。

#2.5Uniq操作，获取某段时间所有数据排重值

这个使用Redis的set数据结构最合适了，只需要不断地将数据往set中扔就行了，set意为集合，所以会自动排重。

#2.6实时系统，反垃圾系统

通过上面说到的set功能，你可以知道一个终端用户是否进行了某个操作，可以找到其操作的集合并进行分析统计对比等。没有做不到，只有想不到。

#2.7缓存

将数据直接存放到内存中，性能优于Memcached，数据结构更多样化。

#3、Redis的特点

- 高效性 （内存）
  - Redis读取的速度是30w次/s，写的速度是10w次/s
- 原子性 (主逻辑线程是单线程)
  - Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。 pipline
- 支持多种数据结构
  - string（字符串） a->b 配置 color--> red
  - list（列表） a->list 消息队列 msg--->["hello","ydlclass","itlils"]
  - hash（哈希） a->map 购物车 1----->["1"=>"剃须刀"，“2”=>“电脑”]
  - set（集合） a->set 去重 quchong-->["北京"，“山西”，“河北“]
  - zset(有序集合) a->sorted set 排行榜 top10->[”xx拿了金牌,10“,"跑路了,9.5"]
- 稳定性：持久化，主从复制（集群）
- 其他特性：支持过期时间，支持事务，消息订阅。

#第三章 Redis单机环境安装

#1、Windows版Redis安装（了解）

Windows版的安装比较简单，解压Redis压缩包完成即安装完毕，安装的注意事项：

- 解压的目录不要有中文
- 目录结构层次不要太深
- 硬盘空间剩余空间最少要大于你的内存空间，建议20G以上

Redis 目录结构：

目录或文件	作用
redis-benchmark.exe	redis的性能测试工具
redis-check-aof.exe	aof文件的检查和修复工具
redis-check-dump.exe	rdb文件的检查和修复工具
redis-cli.exe	client 客户端访问命令
redis-server.exe	服务器启动程序
redis.window.conf	配置文件，这是个文本文件

Redis 服务启动与关闭：

```
1    1、启动服务器：cmd  redis-server.exe redis.windows.conf
2    2、默认端口号：6379
3    3、关闭服务器：直接关闭窗口
```

```
Microsoft Windows [版本 10.0.19042.1466]
(c) Microsoft Corporation。保留所有权利。

F:\software\redis>redis-server.exe redis.windows.conf
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 3.2.100 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in standalone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 25152
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           http://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'

[25152] 21 Feb 15:39:41.368 # Server started, Redis version 3.2.100
[25152] 21 Feb 15:39:41.370 * The server is now ready to accept connections on port 6379
```

运行【**redis-cli.exe**】客户端：

```
选择管理员: C:\Windows\System32\cmd.exe - redis-cli.exe  -h localhost -p 6379
Microsoft Windows [版本 10.0.19042.1466]
(c) Microsoft Corporation。保留所有权利。

F:\software\redis>redis-cli.exe -h localhost -p 6379
localhost:6379> ping
PONG
localhost:6379>
```

redis端口号 6379来历：

光明版本：发明redis的人，可爱女儿。MERZ,九宫格---6379。 程序员的浪漫。

暗黑版本：MERZ舞女--愚蠢。 发明者再愚蠢的人，也应该会使用redis。

#2、 Linux版Redis安装(运维)

#2.1 下载redis安装包

服务器执行以下命令下载redis安装包

```
cd  /export/software
wget  http://download.redis.io/releases/redis-6.2.6.tar.gz
```

#2.2 解压redis压缩包到指定目录

执行以下命令进行解压redis

```
cd /export/software
tar -zxvf redis-6.2.6.tar.gz -C
```

#2.3 安装C程序运行环境

执行以下命令安装C程序运行环境

```
yum  -y install gcc-c++
```

#2.4 安装较新版本的tcl

下载安装较新版本的tcl

> 使用压缩包进行安装

执行以下命令下载tcl安装包

```
cd  /export/software
wget  http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz
解压tcl
tar  -zxvf tcl8.6.1-src.tar.gz -C ../server/
进入指定目录
cd  ../server/tcl8.6.1/unix/
./configure
make && make install
```

> 在线安装tcl（推荐）

执行以下命令在线安装tcl

```
yum -y   install tcl
```

#2.5 编译redis

执行以下命令进行编译：

```
cd /export/server/redis-6.2.6/
#或者使用命令 make 进行编译
make MALLOC=libc
make test && make install PREFIX=/export/server/redis-6.2.6
```

> 修改redis配置文件

执行以下命令修改redis配置文件

```
 cd  /export/server/redis-6.2.6/
 mkdir  -p /export/server/redis-6.2.6/log
 mkdir  -p /export/server/redis-6.2.6/data

 vim  redis.conf
 #  修改第61行
 bind  localhost
 #  修改第128行  后台
 daemonize  yes
 #  修改第163行
 logfile  "/export/server/redis-6.2.6/log/redis.log"
 #  修改第247行
 dir  /export/server/redis-6.2.6/data
```

#2.6 启动redis

执行以下命令启动redis

```
cd /export/server/redis-6.2.6/
bin/redis-server redis.conf
```

#2.7 关闭redis

```
bin/redis-cli  -h localhost shutdown
```

#2.8 连接redis客户端

执行以下命令连接redis客户端

```
cd  /export/server/redis-6.2.6/
bin/redis-cli -h localhost
```

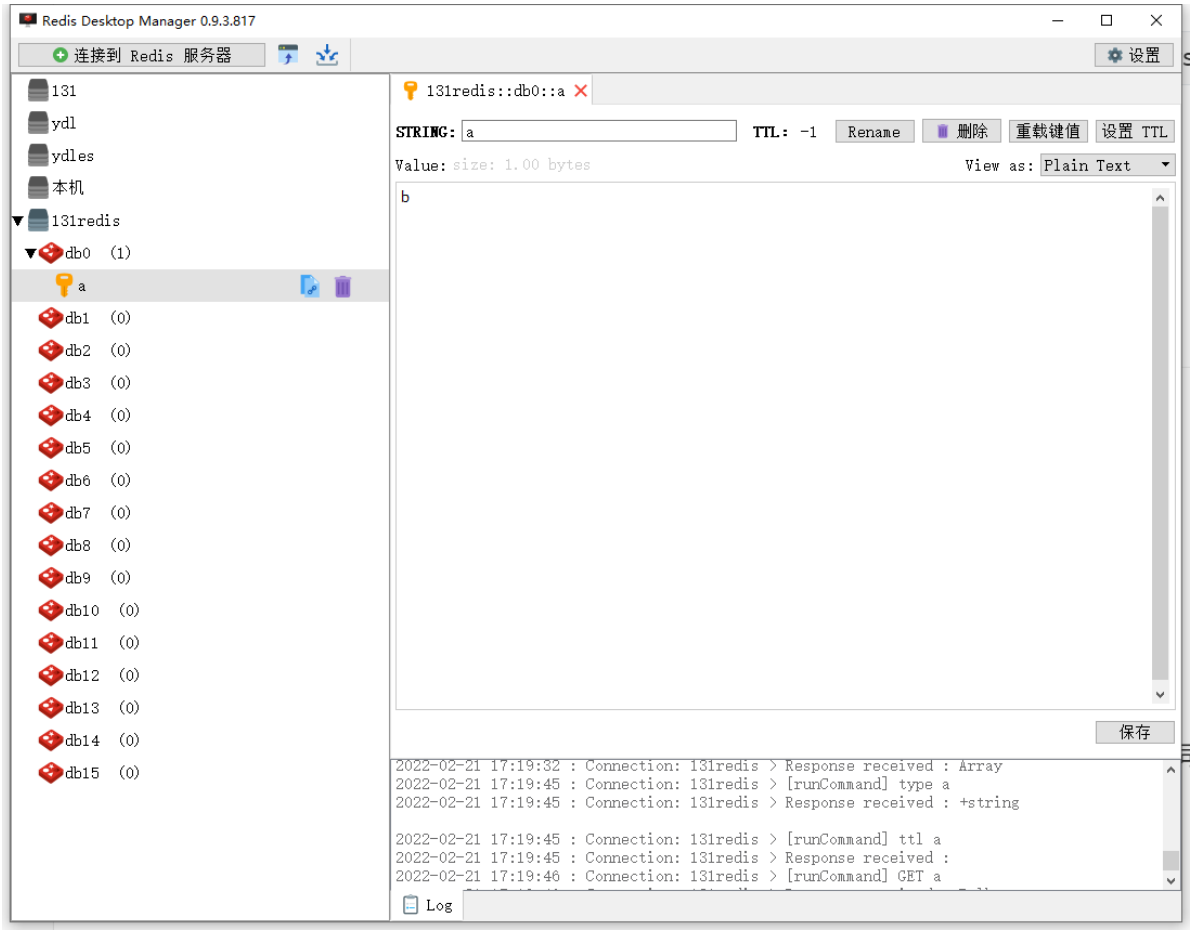
#3、 Redis Desktop Manager

一款基于Qt5的跨平台Redis桌面管理软件，支持：Windows 7+、Mac OS X 10.10+、 Ubuntu 14+，特点：C++ 编写，响应迅速，性能好。

```
下载地址：http://docs.redisdesktop.com/en/latest/install/#windows
```

1

安装客户端，连接Redis服务：



备注说明：Redis Desktoo Manager老版本免费，新版本收费

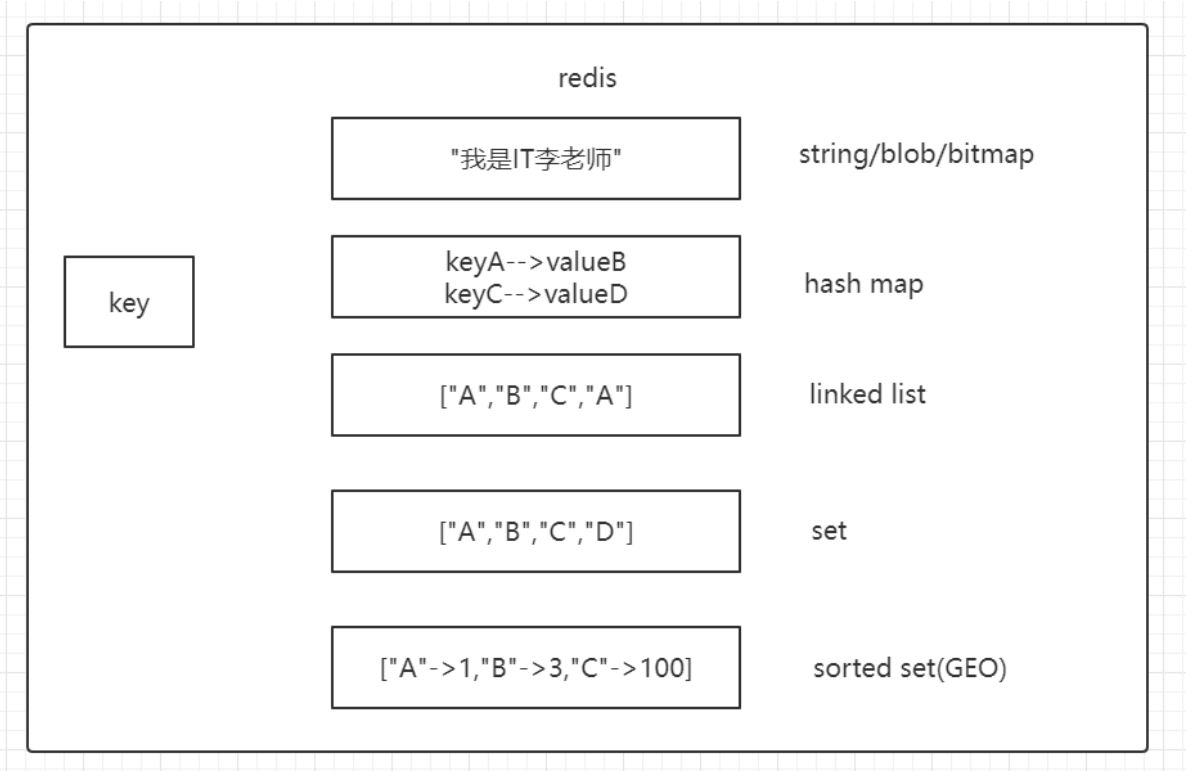
#第四章 Redis的数据类型

redis当中一共支持五种数据类型，分别是：

- string字符串
- list列表
- set集合
- hash表
- zset有序集合

通过这五种不同的数据类型，可以实现各种不同的功能，也可以应用在各种不同的场景。

Redis当中各种数据类型结构如下图：



所有操作看官网：

redis 命令 客户端 文档 社区 下载 支持 许可 更新日志 文章大全 论坛

Redis命令十分丰富，包括的命令组有Cluster、Connection、Geo、Hashes、HyperLogLog、Keys、Lists、Pub/Sub、Scripting、Server、Sets、Sorted Sets、Strings、Transactions一共14个redis命令组两百多个redis命令，Redis中文命令大全。您可以通过下面的检索功能快速查找命令，已下是全部已知的redis命令列表。如果您有兴趣的话也可以查看我们的网站结构图,它以节点图的形式展示了所有redis命令。

过滤命令组: Strings 或者 直接搜索: 例如 PING

APPEND key value
追加一个值到key上

BITCOUNT key [start end]
统计字符串指定起始位置的字节数

BITFIELD key [GET type offset] [SET…
Perform arbitrary bitfield integer operations on strings

BITOP operation destkey key [key ...]
Perform bitwise operations between strings

BITPOS key bit [start] [end]
Find first bit set or clear in a string

DECR key
整数原子减1

DECRBY key decrement
原子减指定的整数

GET key
返回key的value

GETBIT key offset
返回位的值存储在关键的字符串值的偏移量。

GETRANGE key start end
获取存储在key上的值的一个子字符串

GETSET key value
设置一个key的value，并获取设置前的值

INCR key
执行原子加1操作

或者符合过人阅读习惯： https://www.runoob.com/redis/redis-keys.html

RUNOOB.COM 消息队列

首页 HTML CSS JAVASCRIPT VUE BOOTSTRAP NODEJS JQUERY PYTHON JAVA C C++ C# GO SQL LINUX 本地书签

Redis 教程
Redis 教程
Redis 简介
Redis 安装
Redis 配置
Redis 数据类型
Redis 命令
Redis 命令
Redis 键(key)
Redis 字符串(String)
Redis 哈希(Hash)
Redis 列表(List)
Redis 集合(Set)
Redis 有序集合(sorted set)
Redis HyperLogLog
Redis 发布订阅
Redis 事务
Redis 脚本
Redis 连接
Redis 服务器
Redis GEO
Redis Stream

← Redis 命令 Redis 字符串(String) →

Redis 键(key)

Redis 键命令用于管理 redis 的键。

语法

Redis 键命令的基本语法如下：

```
redis 127.0.0.1:6379> COMMAND KEY_NAME
```

实例

```
redis 127.0.0.1:6379> SET runoobkey redis
OK
redis 127.0.0.1:6379> DEL runoobkey
(integer) 1
```

在以上实例中 DEL 是一个命令， **runoobkey** 是一个键。如果键被删除成功，命令执行后输出 **(integer) 1**，否则将输出 **(integer) 0**

Redis keys 命令

下表给出了与 Redis 键相关的基本命令：

序号	命令及描述
1	DEL key 该命令用于在 key 存在时删除 key。
2	DUMP key 序列化给定 key ，并返回被序列化的值。
3	EXISTS key 检查给定 key 是否存在。

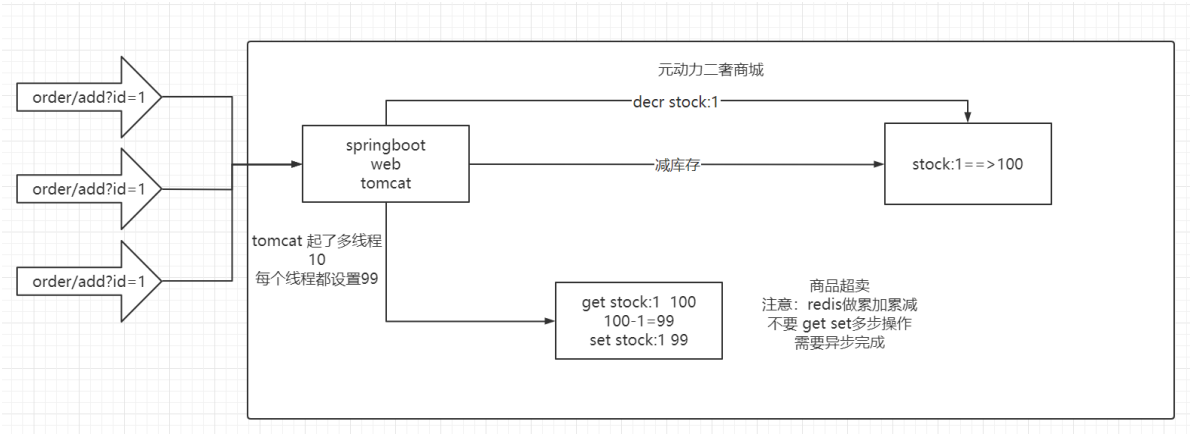
分类导航
HTML / CSS
JavaScript
服务端
数据库
数据分析
移动端
XML 教程
ASP.NET
Web Service
开发工具
网站建设
Advertisement

#1、对字符串string的操作

下表列出了常用的 redis 字符串命令

序号	命令及描述	示例
1	SET key value 设置指定 key 的值	示例：SET hello world
2	GET key 获取指定 key 的值。	示例：GET hello
4	GETSET key value 将给定 key 的值设为 value ，并返回 key 的旧值(old value)。	示例：GETSET hello world2
5	MGET key1 [key2..] 获取所有(一个或多个)给定 key 的值。	示例：MGET hello world
6	SETEX key seconds value 将值 value 关联到 key ，并将 key 的过期时间设为 seconds (以秒为单位)。	示例：SETEX hello 10 world3
7	SETNX key value 只有在 key 不存在时设置 key 的值。	示例：SETNX ydlclass redisvalue
9	STRLEN key 返回 key 所储存的字符串值的长度。	示例：STRLEN ydlclass
10	MSET key value [key value] 同时设置一个或多个 key-value 对。	示例：MSET ydlclass2 ydlclassvalue2 ydlclass3 ydlclassvalue3
12	MSETNX key value key value 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。	示例：MSETNX ydlclass4 ydlclassvalue4 ydlclass5 ydlclassvalue5
13	PSETEX key milliseconds value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。	示例：PSETEX ydlclass6 6000 ydlclass6value
14	INCR key 将 key 中储存的数字值增一。	示例： set ydlclass7 1 INCR ydlclass7 GET ydlclass7
15	INCRBY key increment 将 key 所储存的值加上给定的增量值（increment）	示例：INCRBY ydlclass7 2 get ydlclass7
16	INCRBYFLOAT key increment 将 key 所储存的值加上给定的浮点增量值（increment）	示例：INCRBYFLOAT ydlclass7 0.8
17	DECR key 将 key 中储存的数字值减一。	示例： set ydlclass8 1 DECR ydlclass8 GET ydlclass8
18	DECRBY key decrement key 所储存的值减去给定的减量值（decrement）	示例：DECRBY ydlclass8 3
19	APPEND key value 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。	示例：APPEND ydlclass8 hello

```
1 设置值 获取值
set ydlclass value
get ydlclass
2 mset mget 一次性操作多组数据
mset ydlclass value ydlclass1 value1 ydlclass2 value2
mget ydlclass ydlclass1 ydlclass2
3 没有这个键我们才设置
setnx dlclass value
4 将key的值 加一，减一
incr stock
decr stock
5设置 a值存活时间5秒，值是b     验证码
setex a 5 b
```



#2、对hash列表的操作

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。

Redis 中每个 hash 可以存储 2的32 - 1 键值对（40多亿）

下表列出了 redis hash 基本的相关命令：

序号	命令及描述	示例
1	HSET key field value 将哈希表 key 中的字段 field 的值设为 value 。	示例：HSET key1 field1 value1
2	HSETNX key field value 只有在字段 field 不存在时，设置哈希表字段的值。	示例：HSETNX key1 field2 value2
3	HMSET key field1 value1 [field2 value2] 同时将多个 field-value (域-值)对设置到哈希表 key 中。	示例：HMSET key1 field3 value3 field4 value4
4	HEXISTS key field 查看哈希表 key 中，指定的字段是否存在。	示例： HEXISTS key1 field4 HEXISTS key1 field6
5	HGET key field 获取存储在哈希表中指定字段的值。	示例：HGET key1 field4
6	HGETALL key 获取在哈希表中指定 key 的所有字段和值	示例：HGETALL key1
7	HKEYS key 获取所有哈希表中的字段	示例：HKEYS key1
8	HLEN key 获取哈希表中字段的数量	示例：HLEN key1
9	HMGET key field1 [field2] 获取所有给定字段的值	示例：HMGET key1 field3 field4
10	HINCRBY key field increment 为哈希表 key 中的指定字段的整数值加上增量 increment 。	示例： HSET key2 field1 1 HINCRBY key2 field1 1 HGET key2 field1
11	HINCRBYFLOAT key field increment 为哈希表 key 中的指定字段的浮点数值加上增量 increment 。	示例：HINCRBYFLOAT key2 field1 0.8
12	HVALS key 获取哈希表中所有值	示例：HVALS key1
13	HDEL key field1 [field2] 删除一个或多个哈希表字段	示例： HDEL key1 field3 HVALS key1

```
1设置值 获取值
hset user username itlils
hset user age 18
hget user username
2批量
hmset user1 username itnanls age 19
3获取所有的键值对
hgetall user
4获取所有小key
hkeys  user
5获取所有值
HVALS user
6删除
hdel user age
```

#3、对list列表的操作

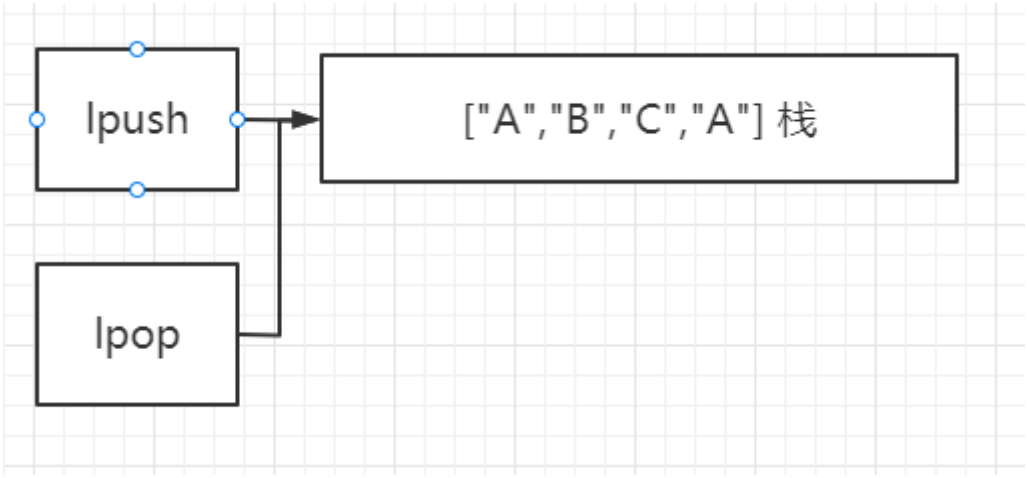
Redis列表是简单的字符串列表，按照插入**顺序**排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）

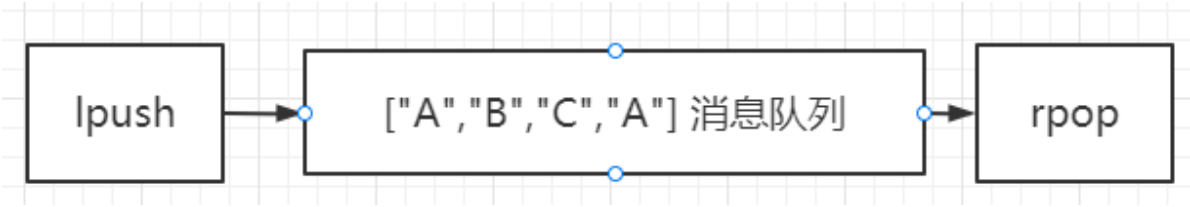
一个列表最多可以包含 2的32 - 1 个元素 (4294967295, 每个列表超过40亿个元素)。

下表列出了列表相关的基本命令：

序号	命令及描述	示例
1	**LPUSH key value1 [value2]** 将一个或多个值插入到列表头部	示例：LPUSH list1 value1 value2
2	**LRANGE key start stop** 查看list当中所有的数据	示例：LRANGE list1 0 -1
3	LPUSHX key value 将一个值插入到已存在的列表头部	示例：LPUSHX list1 value3 LINDEX list1 0
4	RPUSH key value1 [value2] 在列表中添加一个或多个值到尾部	示例： RPUSH list1 value4 value5 LRANGE list1 0 -1
5	RPUSHX key value 为已存在的列表添加单个值到尾部	示例：RPUSHX list1 value6
6	LINSERT key BEFORE|AFTER pivot value 在列表的元素前或者后插入元素	示例：LINSERT list1 BEFORE value3 beforevalue3
7	LINDEX key index 通过索引获取列表中的元素	示例：LINDEX list1 0
8	LSET key index value 通过索引设置列表元素的值	示例：LSET list1 0 hello
9	LLEN key 获取列表长度	示例：LLEN list1
10	LPOP key 移出并获取列表的第一个元素	示例：LPOP list1
11	RPOP key 移除列表的最后一个元素，返回值为移除的元素。	示例：RPOP list1
12	BLPOP key1 [key2] timeout 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。	示例：BLPOP list1 2000
13	BRPOP key1 [key2] timeout 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。	示例：BRPOP list1 2000
14	RPOPLPUSH source destination 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	示例：RPOPLPUSH list1 list2
15	BRPOPLPUSH source destination timeout 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。	示例：BRPOPLPUSH list1 list2 2000
16	LTRIM key start stop 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。	示例：LTRIM list1 0 2
17	DEL key1 key2 删除指定key的列表	示例：DEL list2

```
1 设置值
lpush list1 1 2 3 4 1
rpush list1 6
2查看数据
lrange list1 0 -1
3 移除数据
lpop list1
rpop list1
```





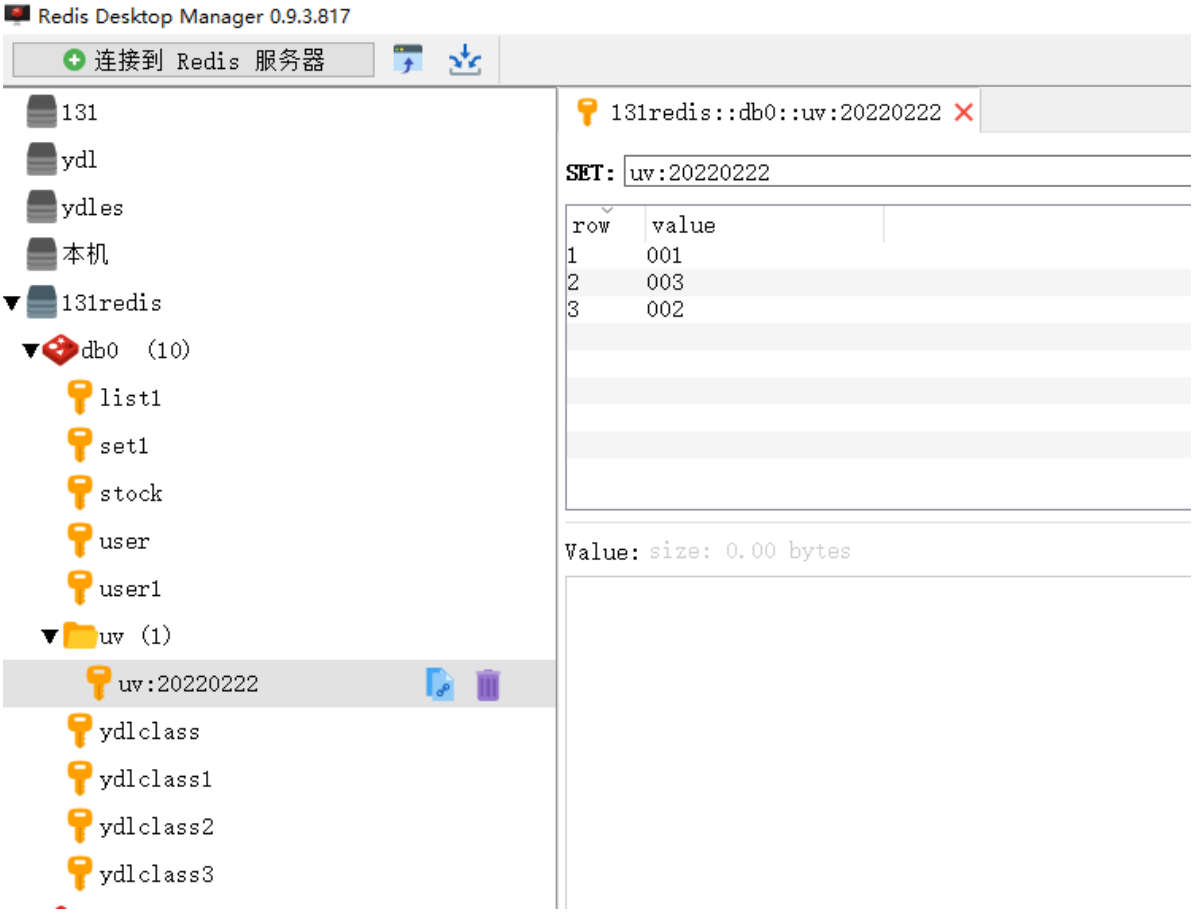
#4、对set集合的操作

- Redis 的 Set 是 String 类型的**无序**集合。集合成员是**唯一**的，这就意味着集合中不能出现重复的数据
- Redis 中集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。
- 集合中最大的成员数为 2的32 - 1 (4294967295, 每个集合可存储40多亿个成员)。

下表列出了 Redis 集合基本命令：

序号	命令及描述	示例
1	SADD key member1 [member2] 向集合添加一个或多个成员	示例：SADD set1 setvalue1 setvalue2
2	SMEMBERS key 返回集合中的所有成员	示例：SMEMBERS set1
3	SCARD key 获取集合的成员数	示例：SCARD set1
4	SDIFF key1 [key2] 返回给定所有集合的差集	示例： SADD set2 setvalue2 setvalue3 SDIFF set1 set2
5	SDIFFSTORE destination key1 [key2] 返回给定所有集合的差集并存储在 destination 中	示例：SDIFFSTORE set3 set1 set2
6	SINTER key1 [key2] 返回给定所有集合的交集	示例：SINTER set1 set2
7	SINTERSTORE destination key1 [key2] 返回给定所有集合的交集并存储在 destination 中	示例：SINTERSTORE set4 set1 set2
8	SISMEMBER key member 判断 member 元素是否是集合 key 的成员	示例：SISMEMBER set1 setvalue1
9	SMOVE source destination member 将 member 元素从 source 集合移动到 destination 集合	示例：SMOVE set1 set2 setvalue1
10	SPOP key 移除并返回集合中的一个随机元素	示例：SPOP set2
11	SRANDMEMBER key [count] 返回集合中一个或多个随机数	示例：SRANDMEMBER set2 2
12	SREM key member1 [member2] 移除集合中一个或多个成员	示例：SREM set2 setvalue1
13	SUNION key1 [key2]] 返回所有给定集合的并集	示例：SUNION set1 set2
14	SUNIONSTORE destination key1 [key2] 所有给定集合的并集存储在 destination 集合中	示例：SUNIONSTORE set5 set1 set2

```
1添加数据
sadd set1 1 2 3 4 5
2获取数据
smembers set1
3获取成员数量
scard set1
4业务 uv 当天登陆用户数
sadd uv:20220222 001 002 003 002
scard uv:20220222
```



#5、对key的操作

下表给出了与 Redis 键相关的基本命令：

序号	命令及描述	示例
1	DEL key 该命令用于在 key 存在时删除 key。	示例：del ydlclass5
2	DUMP key 序列化给定 key ，并返回被序列化的值。	示例：DUMP key1
3	EXISTS key 检查给定 key 是否存在。	示例：exists ydlclass
4	EXPIRE key seconds 为给定 key 设置过期时间，以秒计。	示例：expire ydlclass 5
6	PEXPIRE key milliseconds 设置 key 的过期时间以毫秒计。	示例：PEXPIRE set3 3000
8	KEYS pattern 查找所有符合给定模式(pattern)的 key 。	示例：keys *
10	PERSIST key 移除 key 的过期时间，key 将持久保持。	示例：persist set2
11	PTTL key 以毫秒为单位返回 key 的剩余的过期时间。	示例：pttl set2
12	TTL key 以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)。	示例：ttl set2
13	RANDOMKEY 从当前数据库中随机返回一个 key 。	示例： randomkey
14	RENAME key newkey 修改 key 的名称	示例：rename set5 set8
15	RENAMENX key newkey 仅当 newkey 不存在时，将 key 改名为 newkey 。	示例：renamenx set8 set10
16	TYPE key 返回 key 所储存的值的类型。	示例：type set10

```
1删除
del user1
2查看所有的key
keys *      生产环境下，别用
3存在key
exists user1
4存活时间
expire ydlclass 5
5剩余存活时间    登陆续期
pttl user1
6随机获取 key
randomkey
```

#6、对ZSet的操作-重要（热搜）

- Redis有序集合和集合一样也是string类型元素的集合,且不允许重复的成员
- 它用来保存需要排序的数据，例如排行榜，一个班的语文成绩，一个公司的员工工资，一个论坛的帖子等。
- 有序集合中，每个元素都带有score（权重），以此来对元素进行排序
- 它有三个元素：key、member和score。以语文成绩为例，key是考试名称（期中考试、期末考试等），member是学生名字，score是成绩。
    - 互联网，微博热搜，最热新闻，统计网站pv

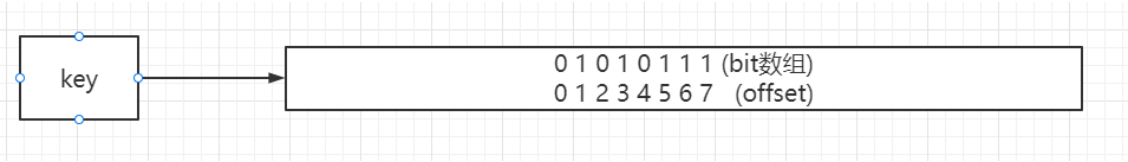
#	命令及描述	示例
1	ZADD key score1 member1 [score2 member2] 向有序集合添加一个或多个成员，或者更新已存在成员的分数	向ZSet中添加页面的PV值 ZADD pv_zset 120 page1.html 100 page2.html 140 page3.html
2	ZCARD key 获取有序集合的成员数	获取所有的统计PV页面数量 ZCARD pv_zset
3	ZCOUNT key min max 计算在有序集合中指定区间分数的成员数	获取PV在120-140在之间的页面数量 ZCOUNT pv_zset 120 140
4	ZINCRBY key increment member 有序集合中对指定成员的分数加上增量 increment	给page1.html的PV值+1 ZINCRBY pv_zset 1 page1.html
5	ZINTERSTORE destination numkeys key [key ...] 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	创建两个保存PV的ZSET： ZADD pv_zset1 10 page1.html 20 page2.html ZADD pv_zset2 5 page1.html 10 page2.html ZINTERSTORE pv_zset_result 2 pv_zset1 pv_zset2
7	ZRANGE key start stop [WITHSCORES] 通过索引区间返回有序集合指定区间内的成员	获取所有的元素，并可以返回每个key对一个的score ZRANGE pv_zset_result 0 -1 WITHSCORES
9	ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT] 通过分数返回有序集合指定区间内的成员	获取ZSET中120-140之间的所有元素 ZRANGEBYSCORE pv_zset 120 140
10	ZRANK key member 返回有序集合中指定成员的索引	获取page1.html的pv排名（升序） ZRANK pv_zset page3.html
11	ZREM key member [member ...] 移除有序集合中的一个或多个成员	移除page1.html ZREM pv_zset page1.html
15	ZREVRANGE key start stop [WITHSCORES] 返回有序集中指定区间内的成员，通过索引，分数从高到低	按照PV降序获取页面 ZREVRANGE pv_zset 0 -1
17	ZREVRANK key member 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	获取page2.html的pv排名（降序） ZREVRANK pv_zset page2.html
18	ZSCORE key member 返回有序集中，成员的分数值	获取page3.html的分数值 ZSCORE pv_zset page3.html

```
1添加
zadd pv 100 page1.html 200 page2.html 300 page3.html
2查看
zcard pv
3查询指定权重范围的成员数
ZCOUNT pv 150 500
4增加权重
ZINCRBY pv 1 page1.html
5交集
ZADD pv_zset1 10 page1.html 20  page2.html
```

```
ZADD pv_zset2 5 page1.html 10  page2.html
ZINTERSTORE pv_zset_result 2 pv_zset1  pv_zset2
6成员的分数值
ZSCORE pv_zset page3.html
7 获取下标范围内的成员。 排序，默认权重由低到高
ZRANGE pv 0 -1
8获取由高到低的几个成员（reverse）使用最多的
效率很高，因为本身zset就是排好序的。
ZREVRANGE key start stop
```

#7、对位图BitMaps的操作

- 计算机最小的存储单位是位bit，Bitmaps是针对位的操作的，相较于String、Hash、Set等存储方式更加节省空间
- Bitmaps不是一种数据结构，操作是基于String结构的，一个String最大可以存储512M，那么一个Bitmaps则可以设置2^32个位
- Bitmaps单独提供了一套命令，所以在Redis中使用Bitmaps和使用字符串的方法不太相同。可以**把Bitmaps想象成一个以位为单位的数组**，数组的每个单元**只能存储0和1**，数组的下标在Bitmaps中叫做偏移量offset



- BitMaps 命令说明：**将每个独立用户是否访问过网站存放在Bitmaps中， 将访问的用户记做1， 没有访问的用户记做0， 用偏移量作为用户的id 。**

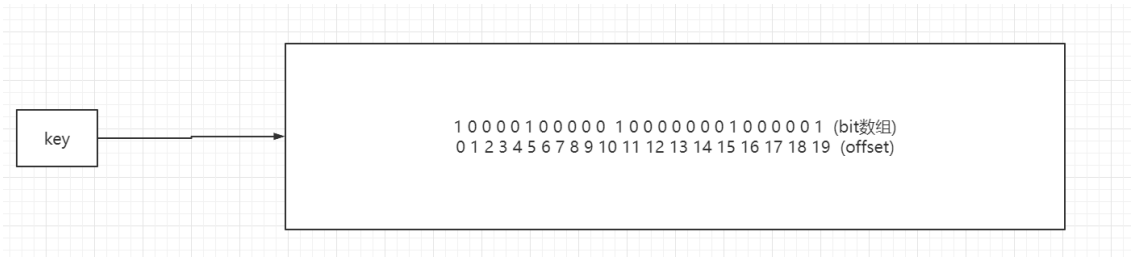
  unique:users:2022-04-05 0 1 0 0

#7.1 设置值

```
SETBIT key offset value
```

setbit命令设置的vlaue只能是**0**或**1**两个值

- 设置键的第offset个位的值（从0算起），假设现在有20个用户，**uid=0，5，11，15，19**的用户对网站进行了访问， 那么当前Bitmaps初始化结果如图所示



- 具体操作过程如下， **unique:users:2022-04-05**代表2022-04-05这天的独立访问用户的Bitmaps

```
setbit unique:users:2022-04-05 0  1
setbit unique:users:2022-04-05 5 1
setbit unique:users:2022-04-05 11 1
setbit unique:users:2022-04-05 15 1
setbit unique:users:2022-04-05 19 1
```

- 很多应用的用户id以一个指定数字（例如10000） 开头， 直接将用户id和Bitmaps的偏移量对应势必会造成一定的浪费， 通常的做法是每次做setbit操作时将用户id减去这个指定数字。

  10000000 10000005 10000011

- 在第一次初始化Bitmaps时， 假如偏移量非常大， 那么整个初始化过程执行会比较慢， 可能会造成Redis的阻塞。

#7.2 获取值

```
GETBIT key offset
```

获取键的第offset位的值（从0开始算），例：下面操作获取id=8的用户是否在2022-04-05这天访问过， 返回0说明没有访问过。

```
getbit unique:users:2022-04-05 8
```

```
192.168.200.131:6379> getbit unique:users:2022-04-05 8
(integer) 0
192.168.200.131:6379> getbit unique:users:2022-04-05 5
(integer) 1
192.168.200.131:6379>
```

#7.3 获取Bitmaps指定范围值为1的个数

```
BITCOUNT key [start end]
```

例：下面操作计算2022-04-05这天的独立访问用户数量：

```
bitcount unique:users:2022-04-05
```

```
192.168.200.131:6379> bitcount unique:users:2022-04-05
(integer) 5
192.168.200.131:6379>
```

#7.4 Bitmaps间的运算

```
BITOP operation destkey key [key, …]
```

bitop是一个复合操作， 它可以做多个Bitmaps的and（交集） 、 or（并集） 、 not（非） 、 xor（异或） 操作并将结果保存在destkey中。

需求：假设2022-04-04访问网站的userid=1， 2， 5， 9， 如图3-13所示：

```
setbit unique:users:2022-04-04 1 1
setbit  unique:users:2022-04-04 2 1
setbit  unique:users:2022-04-04 5 1
setbit  unique:users:2022-04-04 9 1
```

例1：下面操作计算出2022-04-04和2022-04-05两天都访问过网站的用户数量， 如下所示。

```
bitop and unique:users:and:2022-04-04_05  unique:users:2022-04-04
unique:users:2022-04-05
bitcount unique:users:and:2022-04-04_05
```

```
192.168.200.131:6379> bitcount unique:users:and:2022-04-04_05
(integer) 1
192.168.200.131:6379>
```

例2：如果想算出2022-04-04和2022-04-05任意一天都访问过网站的用户数量（例如月活跃就是类似这种），可以使用or求并集，具体命令如下：

```
1    bitop or unique:users:or:2022-04-04_05  unique:users:2022-04-04
   unique:users:2022-04-05
2    bitcount unique:users:or:2022-04-04_05
```

```
192.168.200.131:6379> bitcount unique:users:or:2022-04-04_05
(integer) 8
192.168.200.131:6379>
```

#8、对HyperLogLog结构的操作

#8.1 应用场景

HyperLogLog常用于大数据量的统计，比如页面访问量统计或者用户访问量统计。

```
1    要统计一个页面的访问量（PV），可以直接用redis计数器或者直接存数据库都可以实现，如果要统计一个页面的用户访问量（UV），一个用户一天内如果访问多次的话，也只能算一次，这样，我们可以使用SET集合来做，因为SET集合是有**去重**功能的，key存储页面对应的关键字，value存储对应的userid，这种方法是可行的。但如果访问量较多，假如有几千万的访问量，这就麻烦了。为了统计访问量，要频繁创建SET集合对象。
```

1

Redis实现HyperLogLog算法，HyperLogLog 这个数据结构的发明人 是Philippe Flajolet（菲利普·弗拉若莱）教授。Redis 在 2.8.9 版本添加了 HyperLogLog 结构。

#8.2 UV计算示例

```
1   node1.ydlclass.cn:6379> help @hyperloglog
2
3   PFADD  key element [element ...]
4   summary:  Adds the specified elements to the specified HyperLogLog.
5   since:  2.8.9
6
7   PFCOUNT  key [key ...]
8   summary:  Return the approximated cardinality（基数） of the set(s) observed by the
    HyperLogLog at  key(s).
9   since:  2.8.9
10
11  PFMERGE  destkey sourcekey [sourcekey ...]
12  summary:  Merge N different HyperLogLogs into a single one.
13  since:  2.8.9
```

Redis集成的HyperLogLog使用语法主要有pfadd和pfcount，顾名思义，一个是来添加数据，一个是来统计的。为什么用**pf**？是因为HyperLogLog 这个数据结构的发明人 是Philippe Flajolet教授 ，所以用发明人的英文缩写，这样容易记住这个语法了。

下面我们通过一个示例，来演示如何计算uv。

```
node1.ydlclass.cn:6379> pfadd uv user1
(integer) 1
node1.ydlclass.cn:6379> keys *
1) "uv"
node1.ydlclass.cn:6379> pfcount uv
(integer) 1
node1.ydlclass.cn:6379> pfadd uv user2
(integer) 1
node1.ydlclass.cn:6379> pfcount uv
(integer) 2
node1.ydlclass.cn:6379> pfadd uv user3
(integer) 1
node1.ydlclass.cn:6379> pfcount uv
(integer) 3
node1.ydlclass.cn:6379> pfadd uv user4
(integer) 1
node1.ydlclass.cn:6379> pfcount uv
(integer) 4
node1.ydlclass.cn:6379> pfadd uv user5 user6  user7 user8 user9 user10
(integer) 1  node1.ydlclass.cn:6379> pfcount uv
(integer) 10
```

HyperLogLog算法一开始就是为了大数据量的统计而发明的，所以很适合那种数据量很大，然后又没要求不能有一点误差的计算，HyperLogLog 提供不精确的去重计数方案，虽然不精确但是也不是非常不精确，标准误差是 0.81%，不过这对于页面用户访问量是没影响的，因为这种统计可能是访问量非常巨大，但是又没必要做到绝对准确，访问量对准确率要求没那么高，但是性能存储方面要求就比较高了，而HyperLogLog正好符合这种要求，不会占用太多存储空间，同时性能不错

pfadd和**pfcount**常用于统计，需求：假如两个页面很相近，现在想统计这两个页面的用户访问量呢？这里就可以用**pfmerge**合并统计了，语法如例子：

```
node1.ydlclass.cn:6379> pfadd page1 user1 user2  user3 user4 user5
(integer) 1
node1.ydlclass.cn:6379> pfadd page2 user1 user2  user3 user6 user7
(integer) 1
node1.ydlclass.cn:6379> pfmerge page1+page2  page1 page2
OK
node1.ydlclass.cn:6379> pfcount page1+page2
(integer) 7
```

#8.3 HyperLogLog为什么适合做大量数据的统计

- Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。
- 在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。
- 但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素。

什么是基数?

比如：数据集{1, 3, 5, 7, 5, 7, 8}，那么这个数据集的基数集{1, 3, 5, 7, 8}，基数（不重复元素）为5。基数估计就是在误差可接受的范围内，快速计算基数。

#第五章 Redis Java API操作

Redis不仅可以通过命令行进行操作，也可以通过JavaAPI操作，通过使用Java API来对Redis数据库中的各种数据类型操作。

#1、创建maven工程并导入依赖

#1.1 创建Maven工程

groupId	com.ydlclass
artifactId	redis_op

#1.2 导入POM依赖

```
<dependencies>
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>2.9.0</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>6.14.3</version>
        <scope>test</scope>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.0</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>UTF-8</encoding>
                <!--    <verbal>true</verbal>-->
            </configuration>
        </plugin>
    </plugins>
</build>
```

#2、创建包结构和类

1. 在test目录创建 com.ydlclass.redis.api_test 包结构
2. 创建RedisTest类

#3、连接以及关闭redis客户端

因为后续测试都需要用到Redis连接，所以，我们先创建一个JedisPool用于获取Redis连接。此处，我们基于TestNG来测试各类的API。使用@BeforeTest在执行测试用例前，创建Redis连接池。使用@AfterTest在执行测试用例后，关闭连接池。

实现步骤：

1. 创建JedisPoolConfig配置对象，指定最大空闲连接为10个、最大等待时间为3000毫秒、最大连接数为50、最小空闲连接5个
2. 创建JedisPool
3. 使用@Test注解，编写测试用例，查看Redis中所有的key

   a) 从Redis连接池获取Redis连接

   b) 调用keys方法获取所有的key

   c) 遍历打印所有key

```
package com.ydlclass.redis.api_test;

import org.junit.After;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;

import java.util.List;
import java.util.Set;

/**
 * 1.   创建JedisPoolConfig配置对象，指定最大空闲连接为10个、最大等待时间为3000毫秒、最大连接数为50、最小空闲连接5个
 * 2.   创建JedisPool
 * 3.   使用@Test注解，编写测试用例，查看Redis中所有的key
 * a)   从Redis连接池获取Redis连接
 * b)   调用keys方法获取所有的key
 * c)   遍历打印所有key
 */
public class RedisTest {

    private JedisPool jedisPool;

    @BeforeTest
    public void beforeTest() {
        // JedisPoolConfig配置对象
        JedisPoolConfig config = new JedisPoolConfig();
        // 指定最大空闲连接为10个
        config.setMaxIdle(10);
        // 最小空闲连接5个
        config.setMinIdle(5);
```

```
        // 最大等待时间为3000毫秒
        config.setMaxWaitMillis(3000);
        // 最大连接数为50
        config.setMaxTotal(50);

        jedisPool = new JedisPool(config, "192.168.200.131", 6379);
    }

    @Test
    public void keysTest() {
        // 从Redis连接池获取Redis连接
        Jedis jedis = jedisPool.getResource();
        // 调用keys方法获取所有的key
        Set<String> keySet = jedis.keys("*");

        for (String key : keySet) {
            System.out.println(key);
        }
    }


    @AfterTest
    public void afterTest() {
        // 关闭连接池
        jedisPool.close();
    }
}
```

#4、操作string类型数据

1. 添加一个string类型数据，key为pv，用于保存pv的值，初始值为0
2. 查询该key对应的数据
3. 修改pv为1000
4. 实现整形数据原子自增操作 +1
5. 实现整形该数据原子自增操作 +1000

```
@Test
    public void stringTest() {
        // 获取Jedis连接
        Jedis jedis = jedisPool.getResource();

        // 1.添加一个string类型数据，key为pv，用于保存pv的值，初始值为0
        jedis.set("pv", "0");

        // 2.查询该key对应的数据
        System.out.println("pv:" + jedis.get("pv"));

        // 3.修改pv为1000
        jedis.set("pv", "1000");

        // 4.实现整形数据原子自增操作 +1
        jedis.incr("pv");

        // 5.实现整形该数据原子自增操作 +1000
        jedis.incrBy("pv", 1000);

        System.out.println(jedis.get("pv"));
```

```
22
23          // 将jedis对象放回到连接池
24          jedis.close();
25      }
```

#5、操作hash列表类型数据

1. 往Hash结构中添加以下商品库存

   a) iphone11 => 10000

   b) macbookpro => 9000

2. 获取Hash中所有的商品
3. 新增3000个macbookpro库存
4. 删除整个Hash的数据

```
1   @Test
2       public void testHash(){
3           //从池子里哪一个连接
4           Jedis jedis = jedisPool.getResource();
5
6           //1. 往Hash结构中添加以下商品库存 goods
7           //   a)   iphone13 => 10000
8           //   b)   macbookpro => 9000
9           jedis.hset("goods", "iphone13", "10000");
10          jedis.hset("goods", "macbookpro", "9000");
11          //2. 获取Hash中所有的商品
12          Set<String> goods = jedis.hkeys("goods");
13          for (String good : goods) {
14              System.out.println(good);
15          }
16          //3. 新增3000个macbookpro库存
17          //String hget = jedis.hget("goods", "macbookpro");
18          //int stock=Integer.parseInt(hget)+3000;
19          //jedis.hset("goods", "macbookpro", stock+"");
20          jedis.hincrBy("goods", "macbookpro", 3000);
21
22          String hget = jedis.hget("goods", "macbookpro");
23          System.out.println(hget);
24
25          //4. 删除整个Hash的数据
26          jedis.del("goods");
27      }
```

#6、操作list类型数据

1. 向list的左边插入以下三个手机号码：18511310002、18912301233、18123123314
2. 从右边移除一个手机号码
3. 获取list所有的值

```
1   @Test
2       public void listTest() {
3           // 获取Jedis连接
4           Jedis jedis = jedisPool.getResource();
5
6           // 1.   向list的左边插入以下三个手机号码：18511310001、18912301231、18123123312
```

```
        jedis.lpush("tel_list", "18511310001", "18912301231", "18123123312");

        // 2.   从右边移除一个手机号码
        jedis.rpop("tel_list");

        // 3.   获取list所有的值
        List<String> telList = jedis.lrange("tel_list", 0, -1);
        for (String tel : telList) {
            System.out.println(tel);
        }

        jedis.close();
    }
```

#7、操作set类型的数据

使用set来保存uv值，为了方便计算，将用户名保存到uv中。

1. 往一个set中添加页面 page1 的uv，用户user1访问一次该页面
2. user2访问一次该页面
3. user1再次访问一次该页面
4. 最后获取 page1的uv值

```
@Test
    public void setTest(){
        // 获取Jedis连接
        Jedis jedis = jedisPool.getResource();

        // 求UV就是求独立有多少个（不重复）
        // 1.   往一个set中添加页面 page1 的uv，用户user1访问一次该页面
        jedis.sadd("uv", "user1");
        // jedis.sadd("uv", "user3");
        // jedis.sadd("uv", "user1");
        // 2.   user2访问一次该页面
        jedis.sadd("uv", "user2");

        // 3.   user1再次访问一次该页面
        jedis.sadd("uv", "user1");

        // 4.   最后获取 page1的uv值
        System.out.println("uv:" + jedis.scard("uv"));

        jedis.close();
    }
```

作业：

jedis操作bitmap，HyperLogLog。自己做一下。

#Redis进阶

课程目标(面试、运维)

- 能够理解Redis的持久化
- 能够理解Redis的主从复制架构
- 能够理解Redis的Sentinel架构
- 能够理解Redis cluster集群架构

- redis缓存常见面试题
- 常见问题

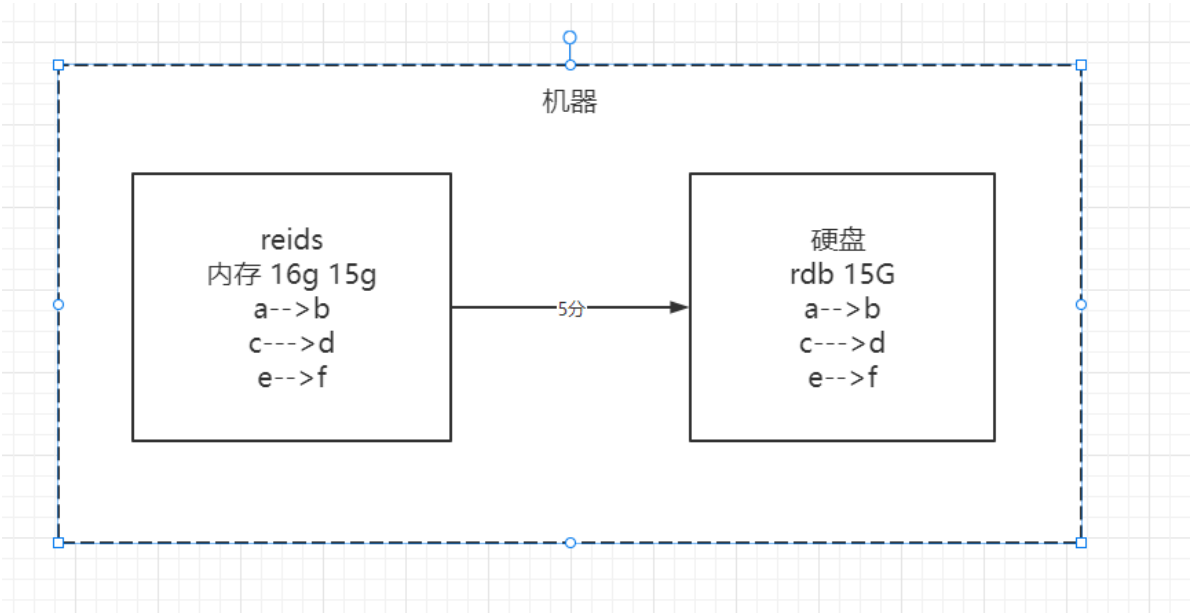
#第一章 Redis的持久化

由于redis是一个内存数据库，所有的数据都是保存在内存当中的，内存当中的数据极易丢失，所以redis的数据持久化就显得尤为重要，在redis当中，提供了两种数据持久化的方式，分别为**RDB**以及**AOF**，且Redis默认开启的数据持久化方式为RDB方式。

#1、RDB持久化方案

#1.1 介绍

Redis会定期保存数据快照至一个rbd文件中，并在启动时自动加载rdb文件，恢复之前保存的数据。



可以在配置文件中配置Redis进行快照保存的时机：

```
save [seconds] [changes]
```

意为在seconds秒内如果发生了changes次数据修改，则进行一次RDB快照保存，例如

```
save  60 100
save  600 500
```

会让Redis每60秒检查一次数据变更情况，如果发生了100次或以上的数据变更，则进行RDB快照保存。可以配置多条save指令，让Redis执行多级的快照保存策略。Redis默认开启RDB快照。也可以通过SAVE或者BGSAVE命令手动触发RDB快照保存。SAVE 和 BGSAVE 两个命令都会调用 rdbSave 函数，但它们调用的方式各有不同：

- SAVE 直接调用 rdbSave ，阻塞 Redis 主进程，直到保存完成为止。在主进程阻塞期间，服务器不能处理客户端的任何请求。
- BGSAVE 则 fork 出一个子进程，子进程负责调用 rdbSave ，并在保存完成之后向主进程发送信号，通知保存已完成。 Redis 服务器在BGSAVE 执行期间仍然可以继续处理客户端的请求。

#1.2 RDB方案优点

1. 对性能影响最小。如前文所述，Redis在保存RDB快照时会fork出子进程进行，几乎不影响Redis处理客户端请求的效率。
2. 每次快照会生成一个完整的数据快照文件，所以可以辅以其他手段保存多个时间点的快照（例如把每天0点的快照备份至其他存储媒介中），作为非常可靠的灾难恢复手段。
3. 使用RDB文件进行数据恢复比使用AOF要快很多。

#1.3 RDB方案缺点

1. 快照是定期生成的，所以在Redis crash时或多或少会丢失一部分数据
2. 如果数据集非常大且CPU不够强（比如单核CPU），Redis在fork子进程时可能会消耗相对较长的时间，影响Redis对外提供服务的能力

#1.4 RDB配置

1. 修改redis的配置文件

```
cd
/export/server/redis-6.2.6/
vim  redis.conf
# 第 行
save  900 1
save  300 10
save  60 10000
save 5 1
```

这三个选项是redis的配置文件默认自带的存储机制。表示每隔多少秒，有多少个key发生变化就生成一份dump.rdb文件，作为redis的快照文件

例如：save 60 10000 表示在60秒内，有10000个key发生变化，就会生成一份redis的快照

1. 重新启动redis服务

每次生成新的dump.rdb都会覆盖掉之前的老的快照

```
 ps -ef | grep redis
 bin/redis-cli -h 192.168.200.131 shutdown
 bin/redis-server redis.conf
```

#2、AOF持久化方案

#2.1 介绍

采用AOF持久方式时，Redis会把每一个写请求都记录在一个日志文件里。在Redis重启时，会把AOF文件中记录的所有写操作顺序执行一遍，确保数据恢复到最新。

#2.2 开启AOF

AOF默认是关闭的，如要开启，进行如下配置：

```
  # 第594行
  appendonly  yes
```

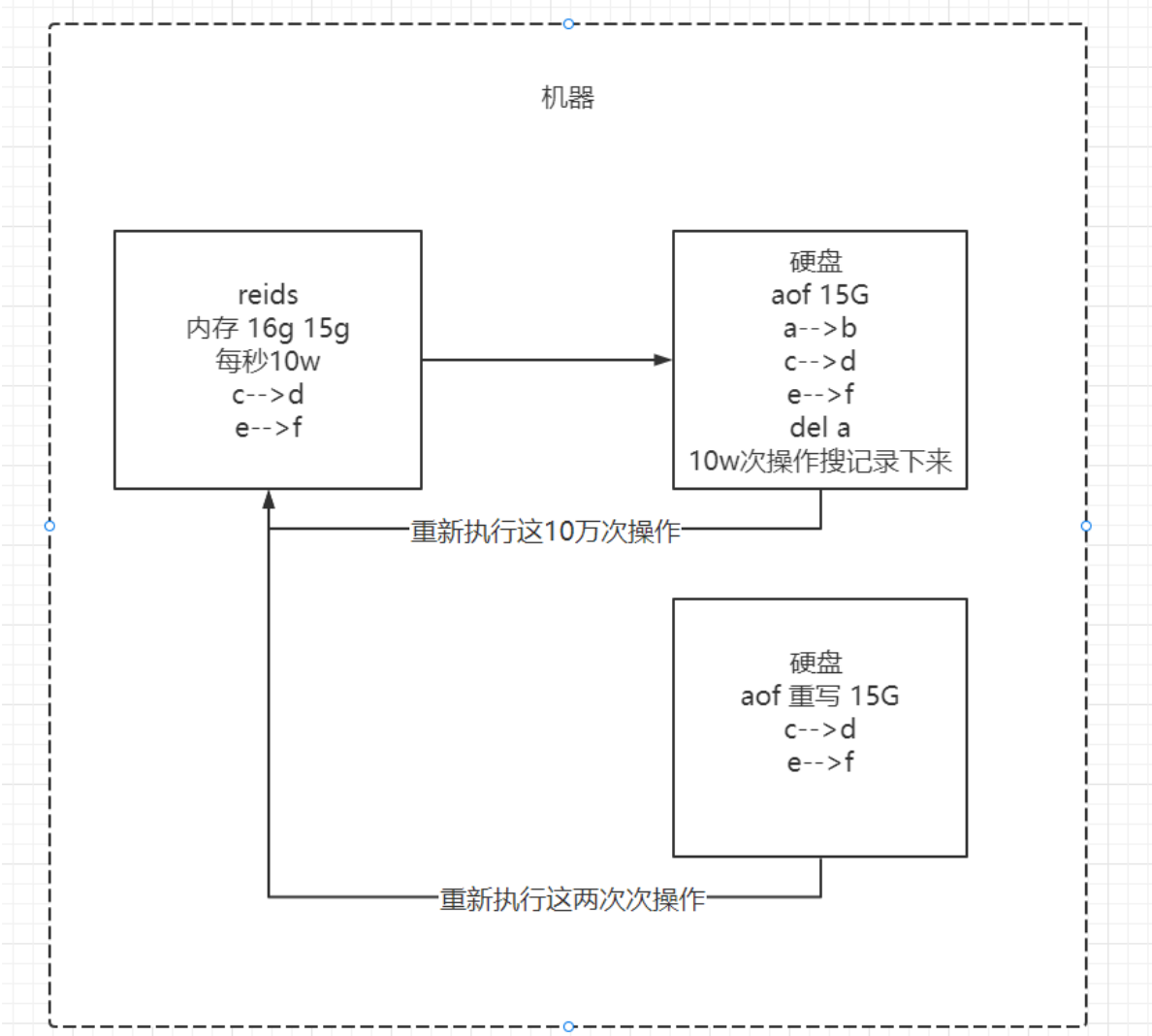
#2.3 配置AOF

AOF提供了三种fsync配置：always/everysec/no，通过配置项[appendfsync]指定：

1. **appendfsync no**：不进行fsync，将flush文件的时机交给OS决定，速度最快
2. **appendfsync always**：每写入一条日志就进行一次fsync操作，数据安全性最高，但速度最慢
3. **appendfsync everysec**：折中的做法，交由后台线程每秒fsync一次

#2.4 AOF rewrite

随着AOF不断地记录写操作日志，因为所有的写操作都会记录，所以必定会出现一些无用的日志。大量无用的日志会让AOF文件过大，也会让数据恢复的时间过长。不过Redis提供了AOF rewrite功能，可以重写AOF文件，只保留能够把数据恢复到最新状态的最小写操作集。



AOF rewrite可以通过BGREWRITEAOF命令触发，也可以配置Redis定期自动进行：

```
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

- Redis在每次AOF rewrite时，会记录完成rewrite后的AOF日志大小，当AOF日志大小在该基础上增长了100%后，自动进行AOF rewrite 32m--->10万-->64M--->40M--100万--80--》50M-->>100m
- auto-aof-rewrite-min-size最开始的AOF文件必须要触发这个文件才触发，后面的每次重写就不会根据这个变量了。该变量仅初始化启动Redis有效。

#2.5 AOF优点

1. 最安全，在启用appendfsync为always时，任何已写入的数据都不会丢失，使用在启用appendfsync everysec也至多只会丢失1秒的数据
2. AOF文件在发生断电等问题时也不会损坏，即使出现了某条日志只写入了一半的情况，也可以使用redis-check-aof工具轻松修复
3. AOF文件易读，可修改，在进行某些错误的数据清除操作后，只要AOF文件没有rewrite，就可以把AOF文件备份出来，把错误的命令删除，然后恢复数据。

#2.6 AOF的缺点

1. AOF文件通常比RDB文件更大
2. 性能消耗比RDB高
3. 数据恢复速度比RDB慢

Redis的数据持久化工作本身就会带来延迟，需要根据数据的安全级别和性能要求制定合理的持久化策略：

- AOF + fsync always的设置虽然能够绝对确保数据安全，但每个操作都会触发一次fsync，会对Redis的性能有比较明显的影响
- AOF + fsync every second是比较好的折中方案，每秒fsync一次
- AOF + fsync never会提供AOF持久化方案下的最优性能

使用RDB持久化通常会提供比使用AOF更高的性能，但需要注意RDB的策略配置

#3、 RDB or AOF

每一次RDB快照和AOF Rewrite都需要Redis主进程进行fork操作。fork操作本身可能会产生较高的耗时，与CPU和Redis占用的内存大小有关。根据具体的情况合理配置RDB快照和AOF Rewrite时机，避免过于频繁的fork带来的延迟。

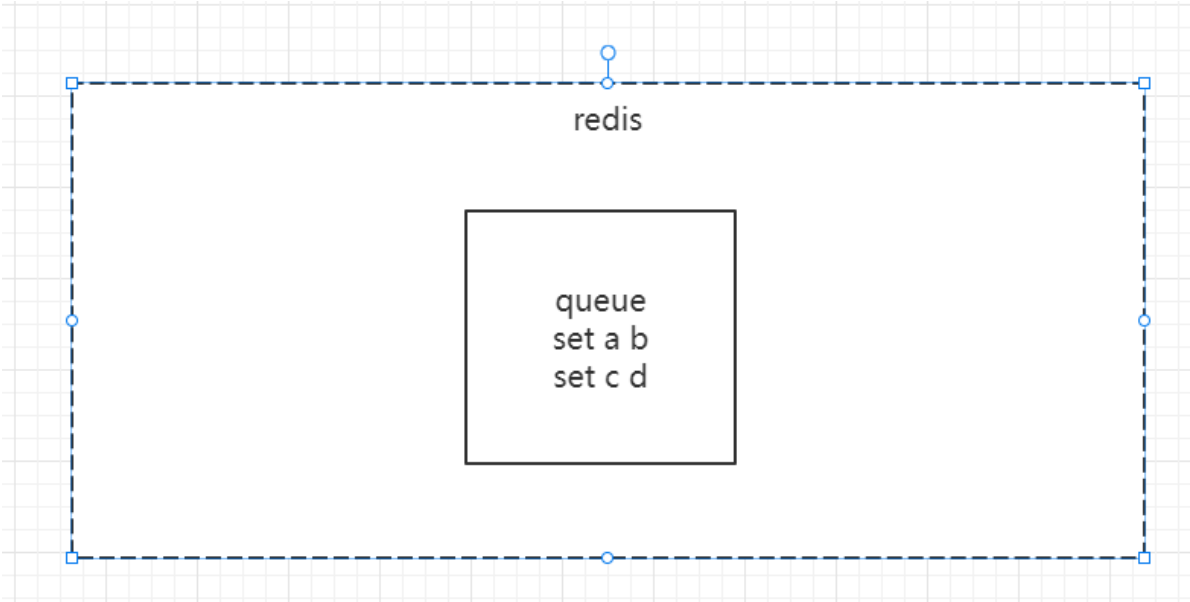
Redis在fork子进程时需要将内存分页表拷贝至子进程，以占用了24GB内存的Redis实例为例，共需要拷贝48MB的数据。在使用单Xeon 2.27Ghz的物理机上，这一fork操作耗时216ms。

本人以前的公司，最后的从机上，rdb aof都开启。

#第二章 Redis 事务

#1、 Redis事务简介

Redis 事务的本质是一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。



总结说：Redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令

- **Redis事务没有隔离级别的概念**

批量操作在发送 EXEC 命令前被放入队列缓存，并不会被实际执行，也就不存在事务内的查询要看到事务里的更新，事务外查询不能看到。

- **Redis不保证原子性**

Redis中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。

一个事务从开始到执行会经历以下三个阶段：

- 第一阶段：开始事务
- 第二阶段：命令入队
- 第三阶段：执行事务

Redis事务相关命令：

- MULTI

开启事务，redis会将后续的命令逐个放入队列中，然后使用EXEC命令来原子化执行这个命令队列

- EXEC

执行事务中的所有操作命令

- DISCARD

取消事务，放弃执行事务块中的所有命令

- WATCH

监视一个或多个key，如果事务在执行前，这个key（或多个key）被其他命令修改，则事务被中断，不会执行事务中的任何命令

- UNWATCH

取消WATCH对所有key的监视

#2、 Redis事务演示

1. MULTI开始一个事务：给k1、k2分别赋值，在事务中修改k1、k2，执行事务后，查看k1、k2值都被修改。

```
192.168.200.131:6379> set key1 v1
OK
192.168.200.131:6379> set key2 v2
OK
192.168.200.131:6379> multi
OK
192.168.200.131:6379(TX)> set key1 11
QUEUED
192.168.200.131:6379(TX)> set key2 22
QUEUED
192.168.200.131:6379(TX)> exec
1) OK
2) OK
192.168.200.131:6379> get key1
"11"
192.168.200.131:6379> get key2
"22"
192.168.200.131:6379>
```

2. 事务失败处理：语法错误（编译器错误），在开启事务后，修改k1值为11，k2值为22，但k2语法错误，最终导致事务提交失败，k1、k2保留原值。

```
192.168.200.131:6379> set key1 v1
OK
192.168.200.131:6379> set key2 v2
OK
192.168.200.131:6379> multi
OK
192.168.200.131:6379(TX)> set key1 11
QUEUED
192.168.200.131:6379(TX)> sets key2 22
(error) ERR unknown command `sets`, with args beginning with: `key2`, `22`,
192.168.200.131:6379(TX)> exec
(error) EXECABORT Transaction discarded because of previous errors.
192.168.200.131:6379> get key1
"v1"
192.168.200.131:6379> get key2
"v2"
192.168.200.131:6379>
```

3.Redis类型错误（运行时错误），在开启事务后，修改k1值为11，k2值为22，但将k2的类型作为List，在运行时检测类型错误，最终导致事务提交失败，此时事务并没有回滚，而是跳过错误命令继续执行， 结果k1值改变、k2保留原值。

```
192.168.200.131:6379> set key1 v1
OK
192.168.200.131:6379> set key2 v2
OK
192.168.200.131:6379> multi
OK
192.168.200.131:6379(TX)> set key1 11
QUEUED
192.168.200.131:6379(TX)> lpush key2 22
```

```
QUEUED
192.168.200.131:6379(TX)> exec
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
192.168.200.131:6379> get key1
"11"
192.168.200.131:6379> get key2
"v2"
192.168.200.131:6379>
```

4 **DISCARD**取消事务

```
192.168.200.131:6379> set key1 v1
OK
192.168.200.131:6379> set key2 v2
OK
192.168.200.131:6379> multi
OK
192.168.200.131:6379> set key1 v1
QUEUED
192.168.200.131:6379> set key2 v2
QUEUED
192.168.200.131:6379> discard
OK
192.168.200.131:6379> get key1
"v1"
192.168.200.131:6379> get key2
"v2"
```

#3、为什么Redis不支持事务回滚?

多数事务失败是由语法错误或者数据结构类型错误导致的，语法错误说明在命令入队前就进行检测的，而类型错误是在执行时检测的，Redis为提升性能而采用这种简单的事务，这是不同于关系型数据库的，特别要注意区分。Redis之所以保持这样简易的事务，完全是为了保证高并发下的核心问题——**性能**。

#第三章 数据删除与淘汰策略

#1、过期数据

1.1 Redis中的数据特征

Redis是一种内存级数据库，所有数据均存放在内存中，内存中的数据可以通过TTL指令获取其状态

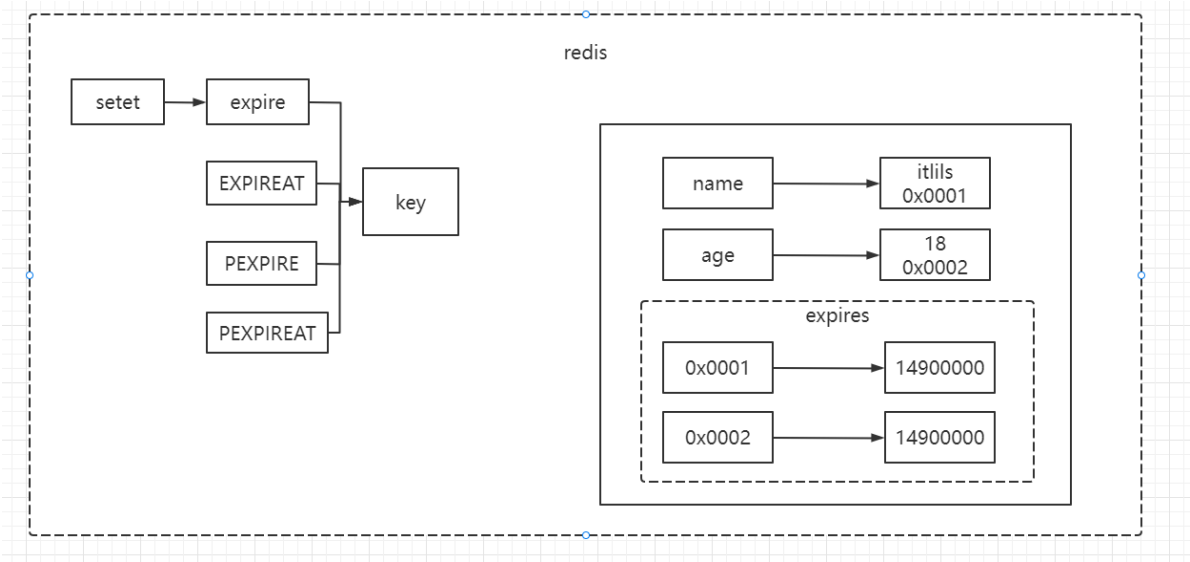
TTL返回的值有三种情况：正数，-1，-2

- **正数**：代表该数据在内存中还能存活的时间
- **-1**：永久有效的数据
- **-2**： 已经过期的数据 或被删除的数据 或 未定义的数据

删除策略就是针对已过期数据的处理策略，已过期的数据是真的就立即删除了吗？其实也不是，我们会有多种删除策略，是分情况的，在不同的场景下使用不同的删除方式会有不同效果，这也正是我们要将的数据的删除策略的问题。

#1.2 时效性数据的存储结构

在Redis中，如何给数据设置它的失效周期呢？数据的时效在redis中如何存储呢？看下图：



过期数据是一块独立的存储空间，Hash结构，field是内存地址，value是过期时间，保存了所有key的过期描述，在最终进行过期处理的时候，对该空间的数据进行检测， 当时间到期之后通过field找到内存该地址处的数据，然后进行相关操作。

#2、数据删除策略

#2.1 数据删除策略的目标

在内存占用与CPU占用之间寻找一种平衡，顾此失彼都会造成整体redis性能的下降，甚至引发服务器宕机或 内存泄露

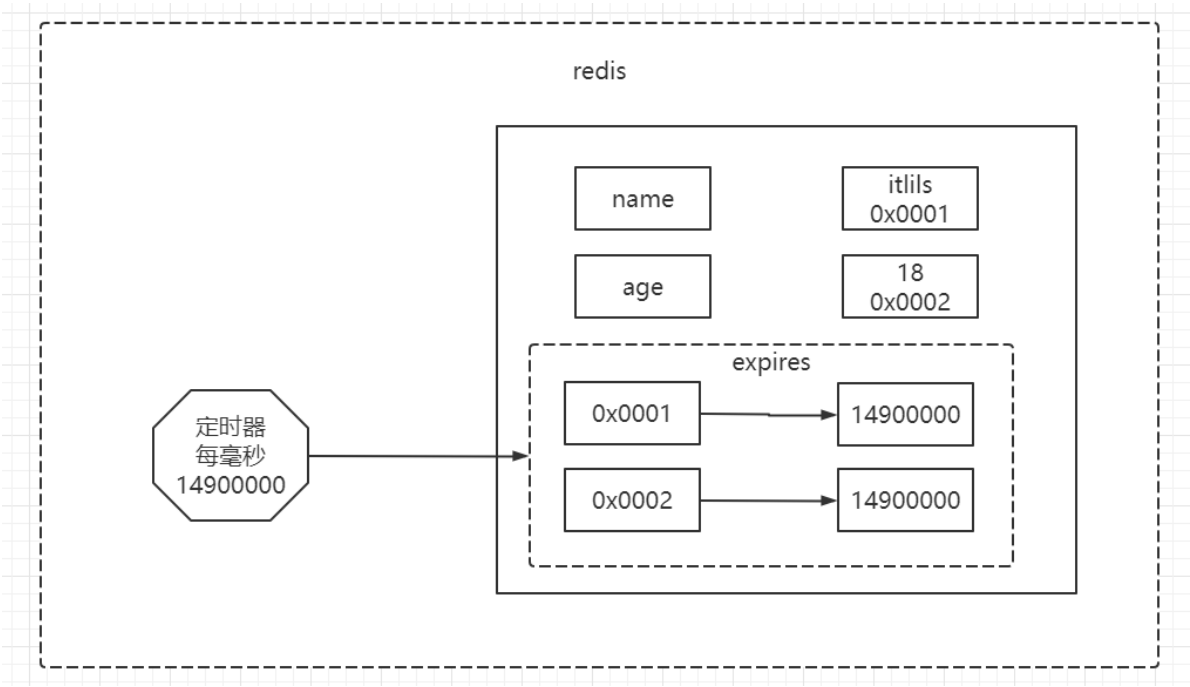
针对过期数据要进行删除的时候都有哪些删除策略呢？

- 1.定时删除
- 2.惰性删除
- 3.定期删除

#2.2 定时删除

创建一个定时器，当key设置有过期时间，且过期时间到达时，由定时器任务立即执行对键的删除操作

- **优点**：节约内存，到时就删除，快速释放掉不必要的内存占用
- **缺点**：CPU压力很大，无论CPU此时负载量多高，均占用CPU，会影响redis服务器响应时间和指令吞吐量
- **总结**：用处理器性能换取存储空间（拿时间换空间）

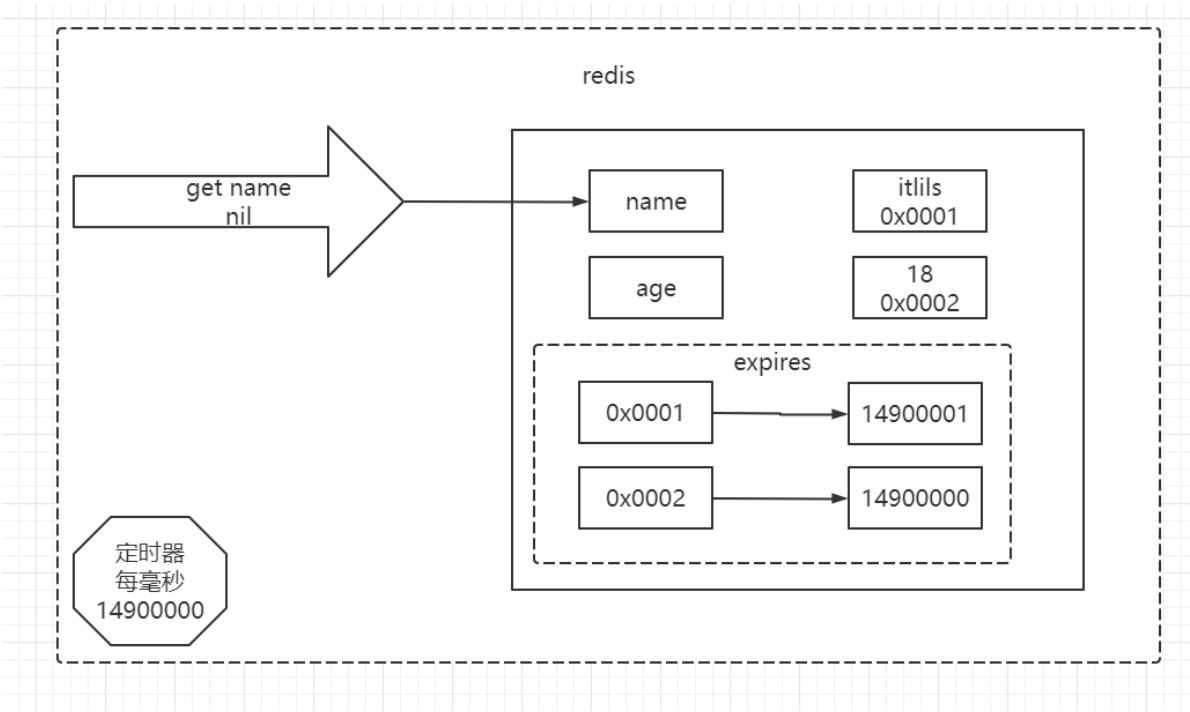


#2.3 惰性删除

数据到达过期时间，不做处理。等下次访问该数据时，我们需要判断

1. 如果未过期，返回数据
2. 发现已过期，删除，返回不存在

- **优点**：节约CPU性能，发现必须删除的时候才删除
- **缺点**：内存压力很大，出现长期占用内存的数据
- **总结**：用存储空间换取处理器性能（拿时间换空间）



#2.4 定期删除

定时删除和惰性删除这两种方案都是走的极端，那有没有折中方案?

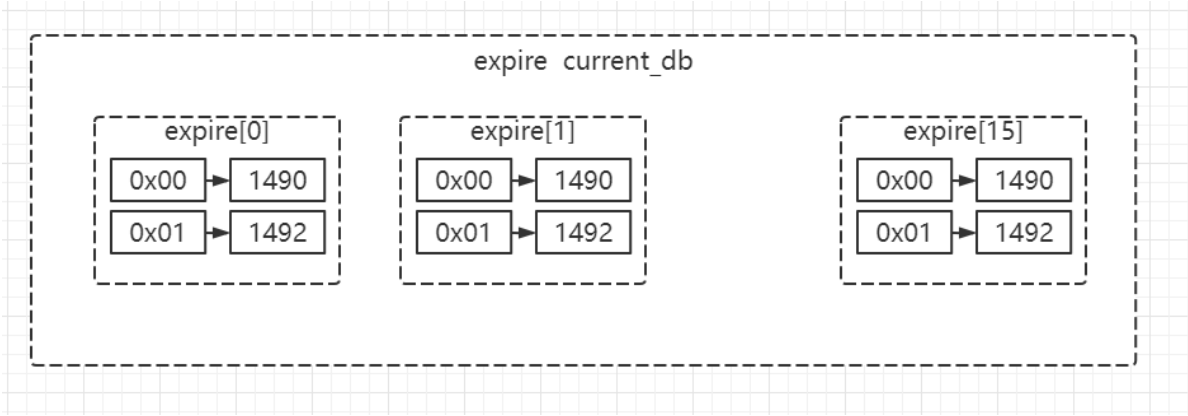
我们来讲redis的定期删除方案：

- Redis启动服务器初始化时，读取配置server.hz的值，默认为10
- 每秒钟执行server.hz次**serverCron()**--------**>databasesCron()**---------**>activeExpireCycle()**

- **activeExpireCycle()**对每个expires[*]逐一进行检测，每次执行耗时：250ms/server.hz
- 对某个expires[*]检测时，随机挑选W个key检测

```
    如果key超时，删除key

    如果一轮中删除的key的数量>W*25%，循环该过程

    如果一轮中删除的key的数量≤W*25%，检查下一个expires[*]，0-15循环

    W取值=ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP属性值
```

- 参数current_db用于记录**activeExpireCycle()** 进入哪个expires[*] 执行
- 如果activeExpireCycle()执行时间到期，下次从current_db继续向下执行



总的来说：定期删除就是周期性轮询redis库中的时效性数据，采用随机抽取的策略，利用过期数据占比的方式控制删除频度

- **特点1**：CPU性能占用设置有峰值，检测频度可自定义设置
- **特点2**：内存压力不是很大，长期占用内存的冷数据会被持续清理
- **总结**：周期性抽查存储空间（随机抽查，重点抽查）

#2.5 删除策略对比

1：定时删除：

```
    节约内存，无占用，
    不分时段占用CPU资源，频度高，
    拿时间换空间
```

2：惰性删除：

```
    内存占用严重
    延时执行，CPU利用率高
    拿空间换时间
```

3：定期删除：

```
    内存定期随机清理
    每秒花费固定的CPU资源维护内存
    随机抽查，重点抽查
```

#3、数据淘汰策略（逐出算法）-面试

#3.1 淘汰策略概述

什么叫数据淘汰策略？什么样的应用场景需要用到数据淘汰策略？

当新数据进入redis时，如果内存不足怎么办？在执行每一个命令前，会调用**freeMemoryIfNeeded()**检测内存是否充足。如果内存不满足新 加入数据的最低存储要求，redis要临时删除一些数据为当前指令清理存储空间。清理数据的策略称为逐出算法。

注意：逐出数据的过程不是100%能够清理出足够的可使用的内存空间，如果不成功则反复执行。当对所有数据尝试完毕， 如不能达到内存清理的要求，将出现错误信息如下

```
(error) OOM command not allowed when used memory >'maxmemory'
```

#3.2 策略配置

影响数据淘汰的相关配置如下：

1：最大可使用内存，即占用物理内存的比例，默认值为0，表示不限制。生产环境中根据需求设定，通常设置在50%以上

```
maxmemory ?mb
```

2：每次选取待删除数据的个数，采用随机获取数据的方式作为待检测删除数据

```
maxmemory-samples count
```

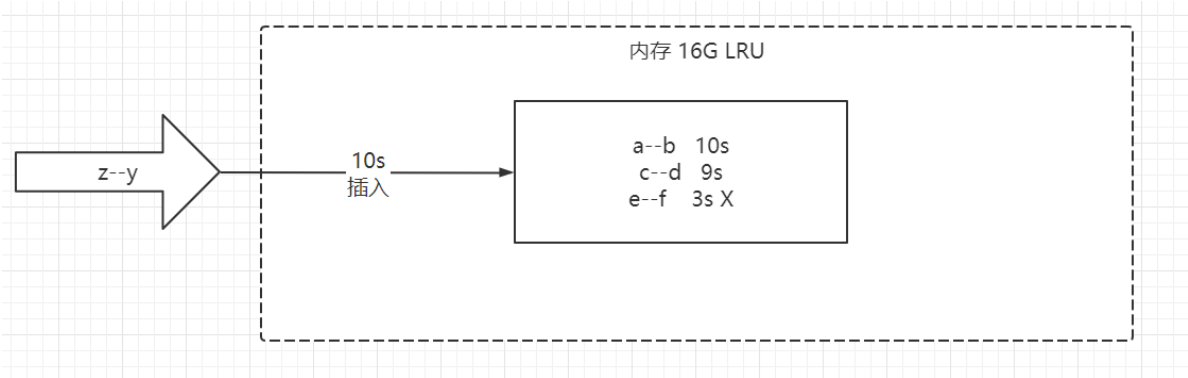
3：对数据进行删除的选择策略

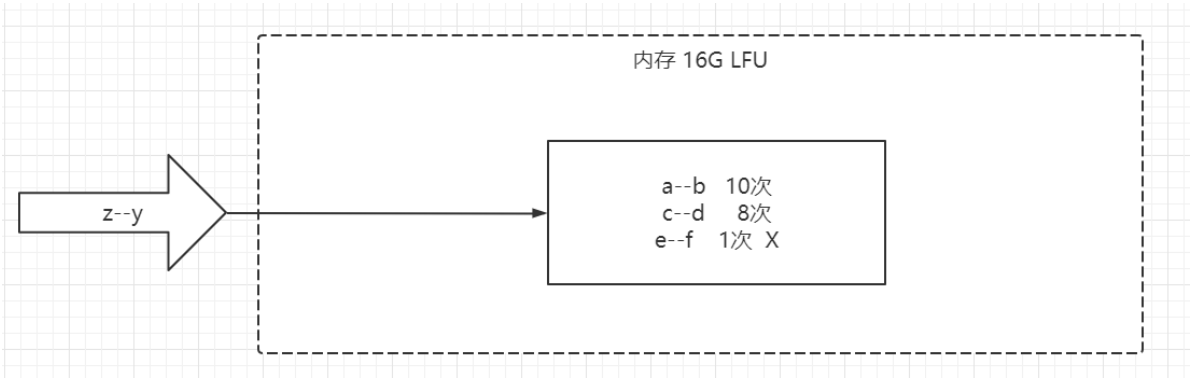
```
maxmemory-policy policy
```

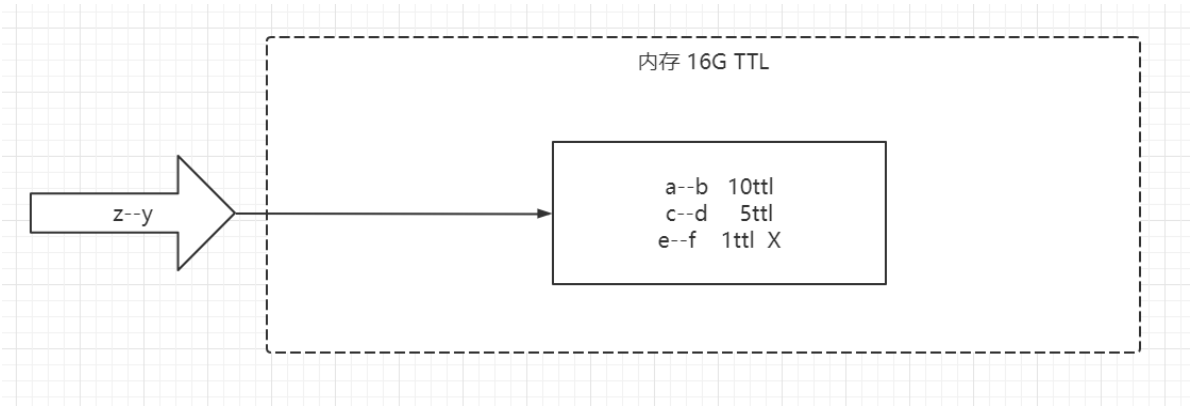
那数据删除的策略policy到底有几种呢？一共是**3类8种**

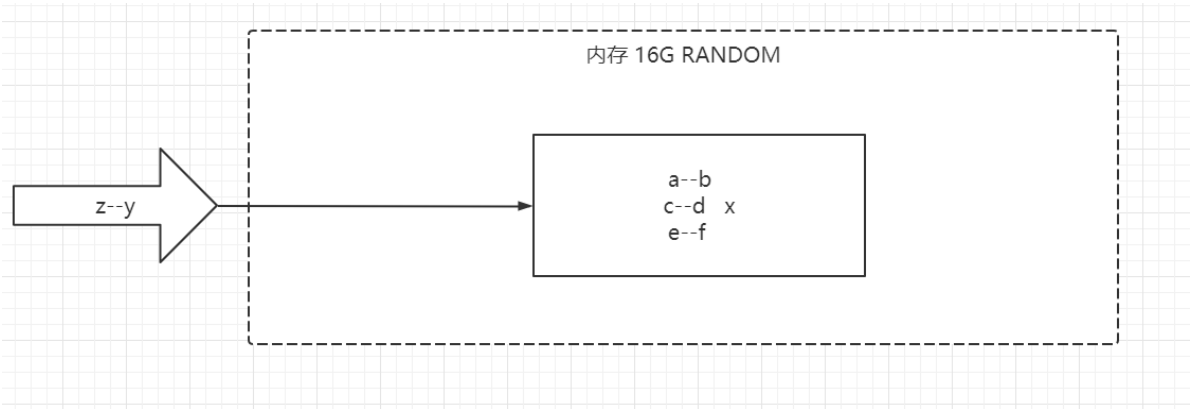
第一类：检测易失数据（可能会过期的数据集server.db[i].expires ） 同一个库

```
volatile-lru：挑选最近最少使用的数据淘汰        least recently used
volatile-lfu：挑选最近使用次数最少的数据淘汰    least frequently used
volatile-ttl：挑选将要过期的数据淘汰
volatile-random：任意选择数据淘汰
```









第二类：检测全库数据（所有数据集server.db[i].dict ）

```
allkeys-lru：挑选最近最少使用的数据淘汰
allkeLyRs-lfu：：挑选最近使用次数最少的数据淘汰
allkeys-random：任意选择数据淘汰，相当于随机
```

第三类：放弃数据驱逐

```
no-enviction（驱逐）：禁止驱逐数据(redis4.0中默认策略)，会引发OOM(Out Of Memory)
```

注意：这些策略是配置到哪个属性上？怎么配置？如下所示

```
maxmemory-policy volatile-lru
```

数据淘汰策略配置依据

使用INFO命令输出监控信息，查询缓存 hit 和 miss 的次数，根据业务需求调优Redis配置

#第四章 Redis的主从复制架构

#1、主从复制简介

#1.1 高可用

首先我们要理解互联网应用因为其独有的特性我们演化出的**三高**架构

- 高并发

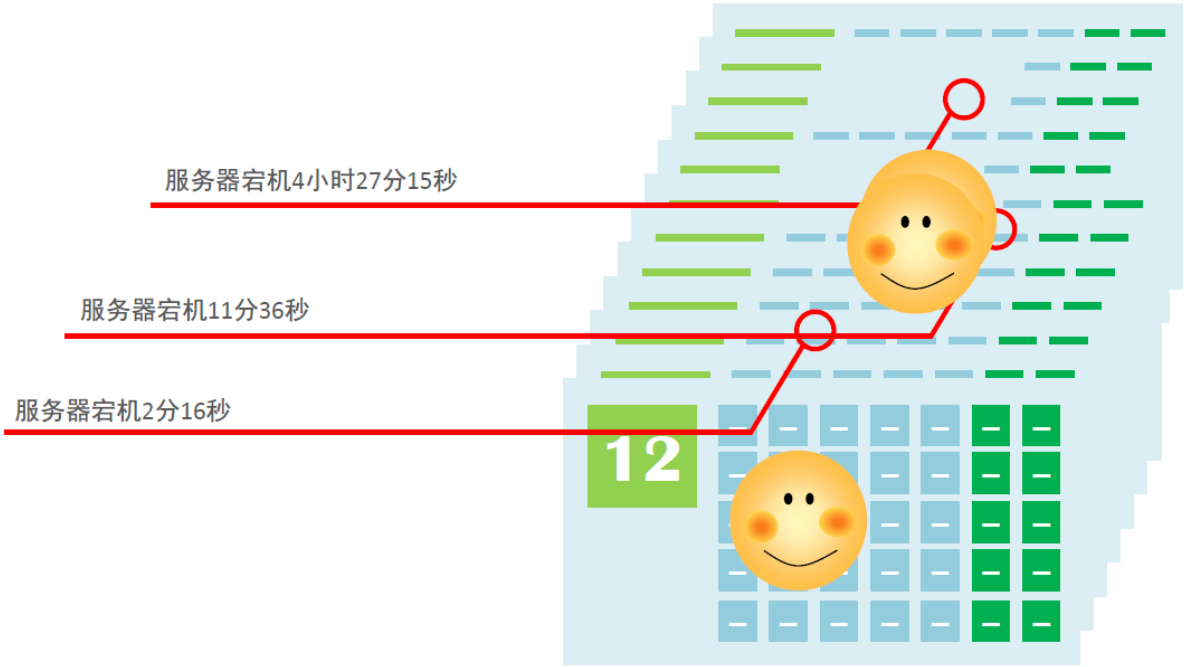
> 应用要提供某一业务要能支持很多客户端同时访问的能力，我们称为并发，高并发意思就很明确了

- 高性能

> 性能带给我们最直观的感受就是：速度快，时间短

- 高可用

可用性：一年中应用服务正常运行的时间占全年时间的百分比，如下图：表示了应用服务在全年宕机的时间



我们把这些时间加在一起就是全年应用服务不可用的时间，然后我们可以得到应用服务全年可用的时间

> 4小时27分15秒+11分36秒+2分16秒=4小时41分7秒=16867秒
>
> 1年=365 *24* 60*60=31536000秒
>
> 可用性=（31536000-16867）/31536000*100%=99.9465151%

业界可用性目标**5个9，即**99.999%，即服务器年宕机时长低于315秒，约5.25分钟

支付宝

Amazon 半天宕机 全球60% 云服务器上

#1.2 主从复制概念

知道了三高的概念之后，我们想：你的“Redis”是否高可用？那我们要来分析单机redis的风险与问题

问题1.机器故障

- 现象：硬盘故障、系统崩溃
- 本质：数据丢失，很可能对业务造成灾难性打击
- 结论：基本上会放弃使用redis.

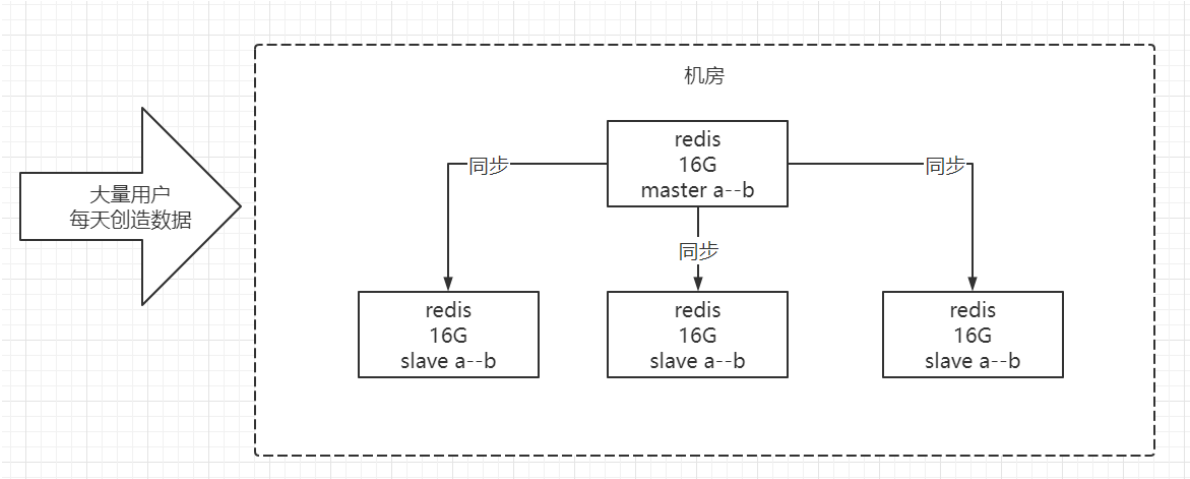
问题2.容量瓶颈

- 现象：内存不足，从16G升级到64G，从64G升级到128G，无限升级内存
- 本质：穷，硬件条件跟不上
- 结论：放弃使用redis

结论：

为了避免单点Redis服务器故障，准备多台服务器，互相连通。将数据复制多个副本保存在不同的服务器上，连接在一起，并保证数据是同步的。即使有其中一台服务器宕机，其他服务器依然可以继续提供服务，实现Redis的高可用，同时实现数据冗余备份。

多台服务器连接方案：



- 提供数据方： **master**

主服务器，主节点，主库主客户端

- 接收数据方： **slave**

从服务器，从节点，从库

从客户端

- 需要解决的问题：

数据同步（master的数据复制到slave中）

这里我们可以来解释主从复制的概念：

概念：主从复制即将master中的数据即时、有效的复制到slave中

特征：一个master可以拥有多个slave，一个slave只对应一个master

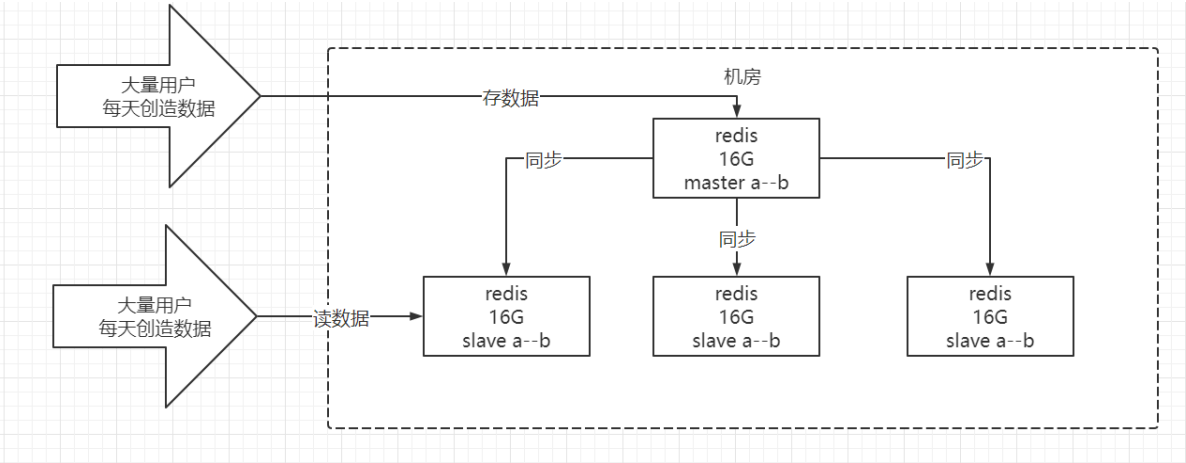
职责：master和slave各自的职责不一样

master:

```
1    写数据
2    执行写操作时，将出现变化的数据自动同步到slave
3    读数据（可忽略）
```

slave:

```
1    读数据
2    写数据（禁止）
```



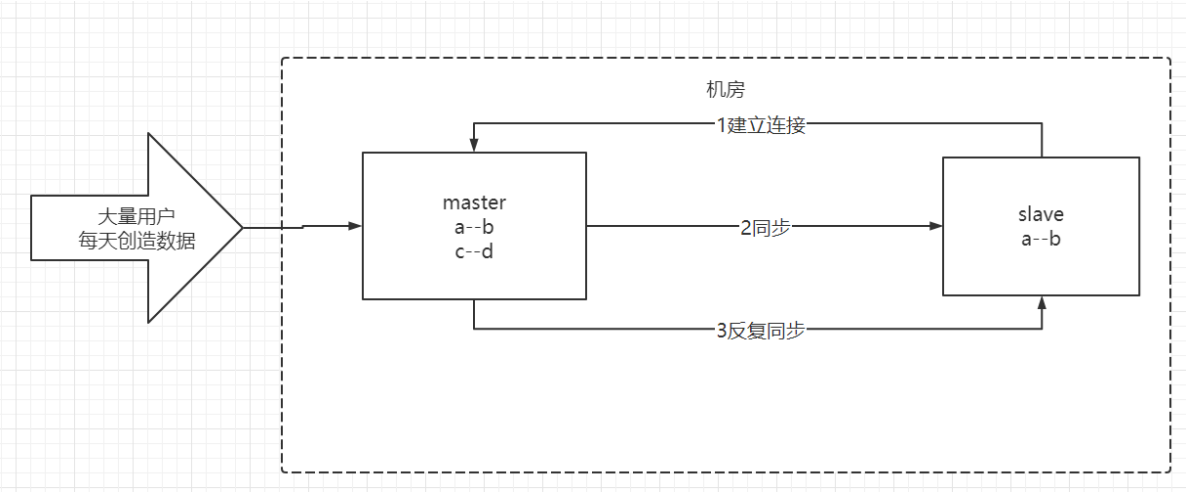
#1.3 主从复制的作用

- 读写分离：master写、slave读，提高服务器的读写负载能力
- 负载均衡：基于主从结构，配合读写分离，由slave分担master负载，并根据需求的变化，改变slave的数 量，通过多个从节点分担数据读取负载，大大提高Redis服务器并发量与数据吞吐量
- 故障恢复：当master出现问题时，由slave提供服务，实现快速的故障恢复
- 数据冗余：实现数据热备份，是持久化之外的一种数据冗余方式
- 高可用基石：基于主从复制，构建哨兵模式与集群，实现Redis的高可用方案

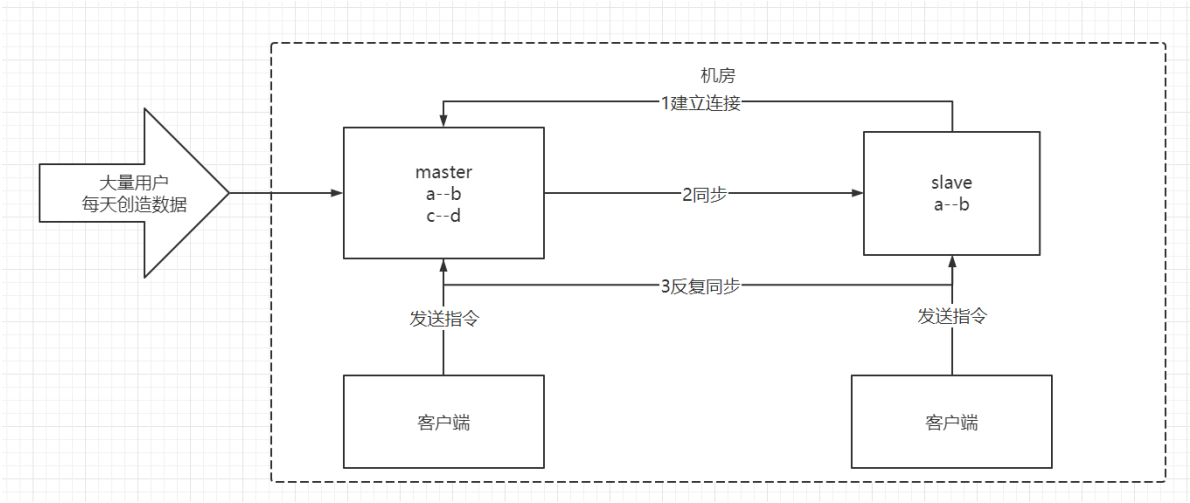
#2、主从复制工作流程

主从复制过程大体可以分为3个阶段

- 建立连接阶段（即准备阶段）
- 数据同步阶段
- 命令传播阶段（反复同步）



而命令的传播其实有4种，分别如下：



#2.1 主从复制的工作流程（三个阶段）

> 阶段一：建立连接

建立slave到master的连接，使master能够识别slave，并保存slave端口号

流程如下：

1. 步骤1：设置master的地址和端口，保存master信息
2. 步骤2：建立socket连接
3. 步骤3：发送ping命令（定时器任务）
4. 步骤4：身份验证
5. 步骤5：发送slave端口信息

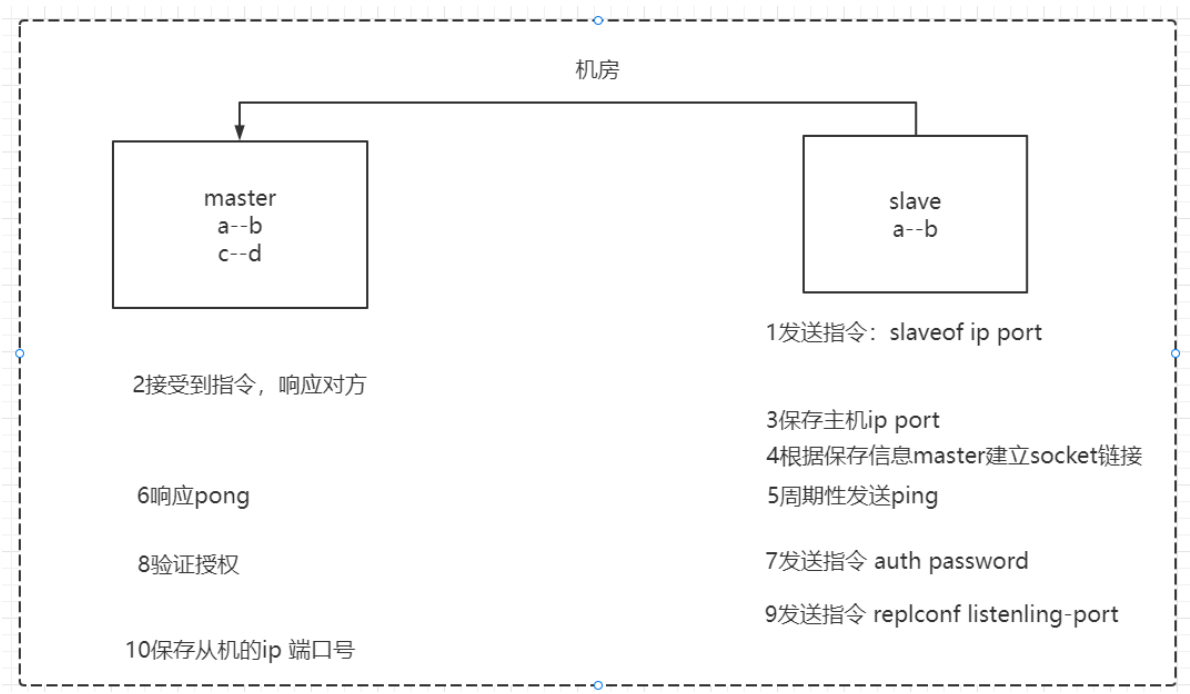
至此，主从连接成功！

当前状态：

slave：保存master的地址与端口

master：保存slave的端口

总体：之间创建了连接的socket



master和slave互联

接下来就要通过某种方式将master和slave连接到一起

方式一：客户端发送命令

```
slaveof masterip masterport
```

方式二：启动服务器参数

```
redis-server --slaveof masterip masterport
```

方式三：服务器配置（**主流方式**）

```
slaveof masterip masterport
```

slave系统信息

```
master_link_down_since_seconds
masterhost & masterport
```

master系统信息

```
uslave_listening_port(多个)
```

主从断开连接

断开slave与master的连接，slave断开连接后，不会删除已有数据，只是不再接受master发送的数据

```
slaveof no one
```

授权访问

master客户端发送命令设置密码

```
requirepass password
```

master配置文件设置密码

```
config set requirepass password
config get requirepass
```

slave客户端发送命令设置密码

```
auth password
```

slave配置文件设置密码

```
masterauth password
```

slave启动服务器设置密码

```
redis-server -a password
```

> 阶段二：数据同步

- 在slave初次连接master后，复制master中的所有数据到slave
- 将slave的数据库状态更新成master当前的数据库状态

同步过程如下：

1. 步骤1：请求同步数据

2. 步骤2：创建RDB同步数据
3. 步骤3：恢复RDB同步数据
4. 步骤4：请求部分同步数据
5. 步骤5：恢复部分同步数据

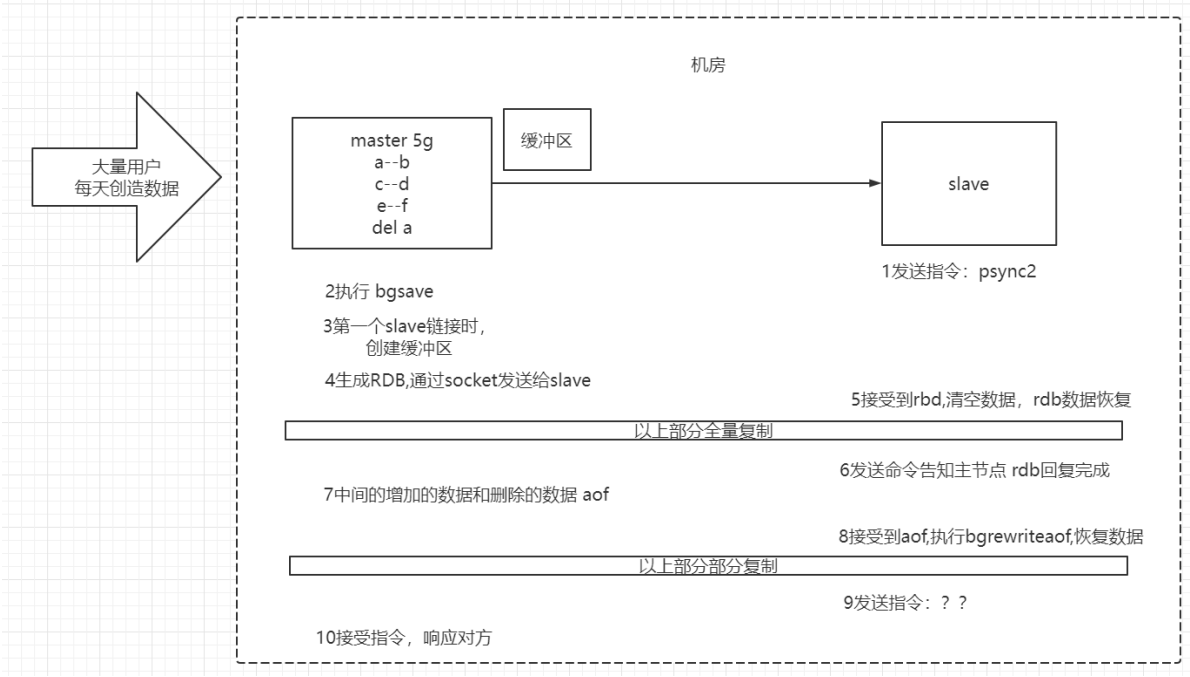
至此，数据同步工作完成！

当前状态：

slave：具有master端全部数据，包含RDB过程接收的数据

master：保存slave当前数据同步的位置

总体：之间完成了数据克隆



数据同步阶段master说明

1：如果master数据量巨大，数据同步阶段应避开流量高峰期，避免造成master阻塞，影响业务正常执行

2：复制缓冲区大小设定不合理，会导致数据溢出。如进行全量复制周期太长，进行部分复制时发现数据已经存在丢失的情况，必须进行第二次全量复制，致使slave陷入死循环状态。

```
repl-backlog-size ?mb
```

1. master单机内存占用主机内存的比例不应过大，建议使用50%-70%的内存，留下30%-50%的内存用于执 行bgsave命令和创建复制缓冲区

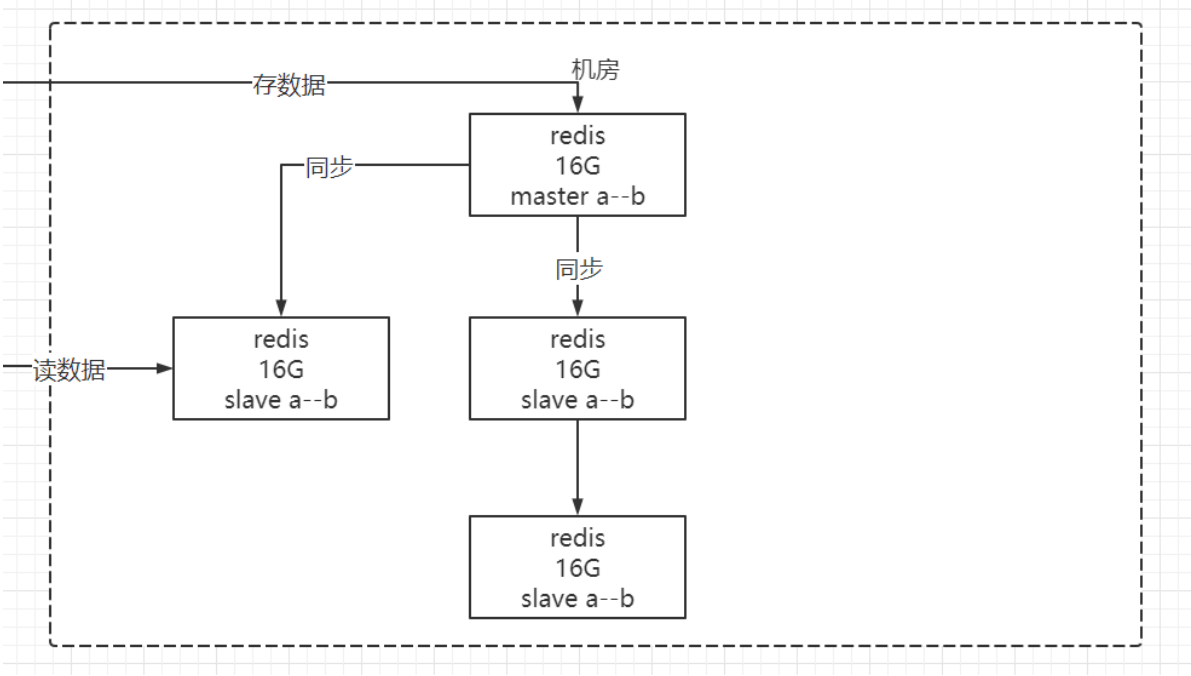
数据同步阶段slave说明

1. 为避免slave进行全量复制、部分复制时服务器响应阻塞或数据不同步，建议关闭此期间的对外服务

```
slave-serve-stale-data yes|no
```

1. 数据同步阶段，master发送给slave信息可以理解master是slave的一个客户端，主动向slave发送命令
2. 多个slave同时对master请求数据同步，master发送的RDB文件增多，会对带宽造成巨大冲击，如果master带宽不足，因此数据同步需要根据业务需求，适量错峰
3. slave过多时，建议调整拓扑结构，由一主多从结构变为树状结构，中间的节点既是master，也是slave。注意使用树状结构时，由于层级深度，导致深度越高的slave与最顶层master间数据同步延

迟较大，数据一致性变差，应谨慎选择



生产环境：一开始，想好redis架构，一开始就把n主m从的redis服务都启动起来。

> 阶段三：命令传播

- 当master数据库状态被修改后，导致主从服务器数据库状态不一致，此时需要让主从数据同步到一致的状态，同步的动作称为**命令传播**
- master将接收到的数据变更命令发送给slave，slave接收命令后执行命令

命令传播阶段的部分复制

命令传播阶段出现了断网现象：

网络闪断闪连：忽略

短时间网络中断：部分复制

长时间网络中断：全量复制

这里我们主要来看部分复制，部分复制的三个核心要素

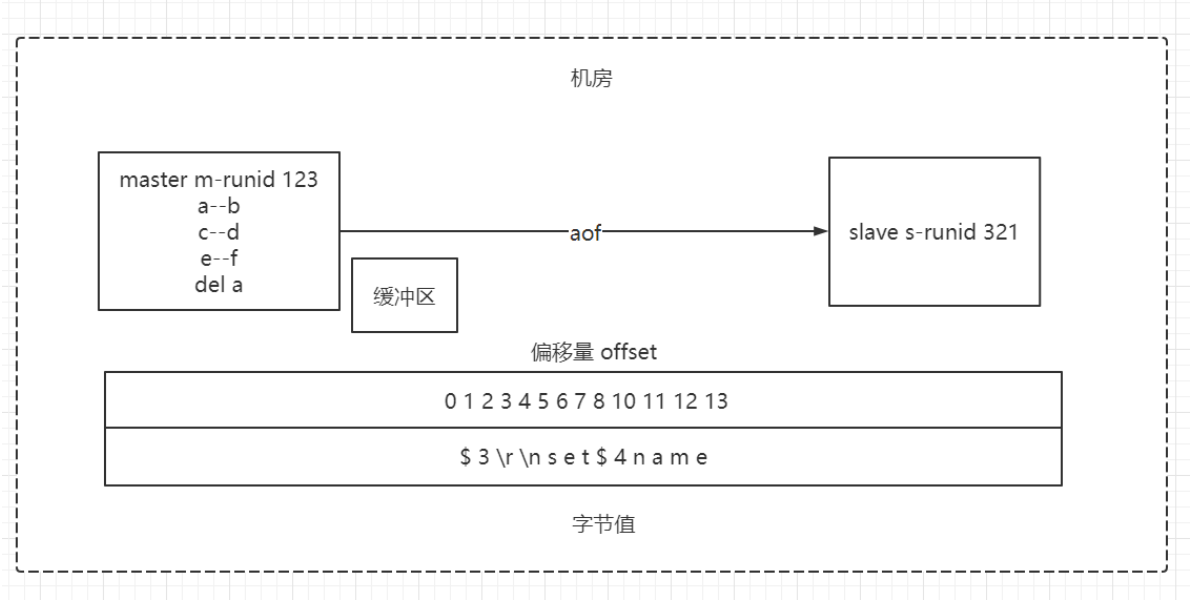
1. 服务器的运行 id（run id）
2. 主服务器的复制积压缓冲区
3. 主从服务器的复制偏移量

- 服务器运行ID（runid）

```
概念：服务器运行ID是每一台服务器每次运行的身份识别码，一台服务器多次运行可以生成多个运行id

组成：运行id由40位字符组成，是一个随机的十六进制字符
例如：fdc9ff13b9bbaab28db42b3d50f852bb5e3fcdce

作用：运行id被用于在服务器间进行传输，识别身份
如果想两次操作均对同一台服务器进行，必须每次操作携带对应的运行id，用于对方识别

实现方式：运行id在每台服务器启动时自动生成的，master在首次连接slave时，会将自己的运行ID发送给slave，
slave保存此ID，通过info Server命令，可以查看节点的runid
```

- 复制缓冲区

```
概念：复制缓冲区，又名复制积压缓冲区，是一个先进先出（FIFO）的队列，用于存储服务器执行过的命令，每次传播命令，master都会将传播的命令记录下来，并存储在复制缓冲区
    复制缓冲区默认数据存储空间大小是1M
    当入队元素的数量大于队列长度时，最先入队的元素会被弹出，而新元素会被放入队列
作用：用于保存master收到的所有指令（仅影响数据变更的指令，例如set，select）

数据来源：当master接收到主客户端的指令时，除了将指令执行，会将该指令存储到缓冲区中
```



复制缓冲区内部工作原理：

组成

- 偏移量

> 概念：一个数字，描述复制缓冲区中的指令字节位置
>
> 分类：
>
> - master复制偏移量：记录发送给所有slave的指令字节对应的位置（多个）
> - slave复制偏移量：记录slave接收master发送过来的指令字节对应的位置（一个）
>
> 作用：同步信息，比对master与slave的差异，当slave断线后，恢复数据使用
>
> 数据来源：
>
> - master端：发送一次记录一次
> - slave端：接收一次记录一次

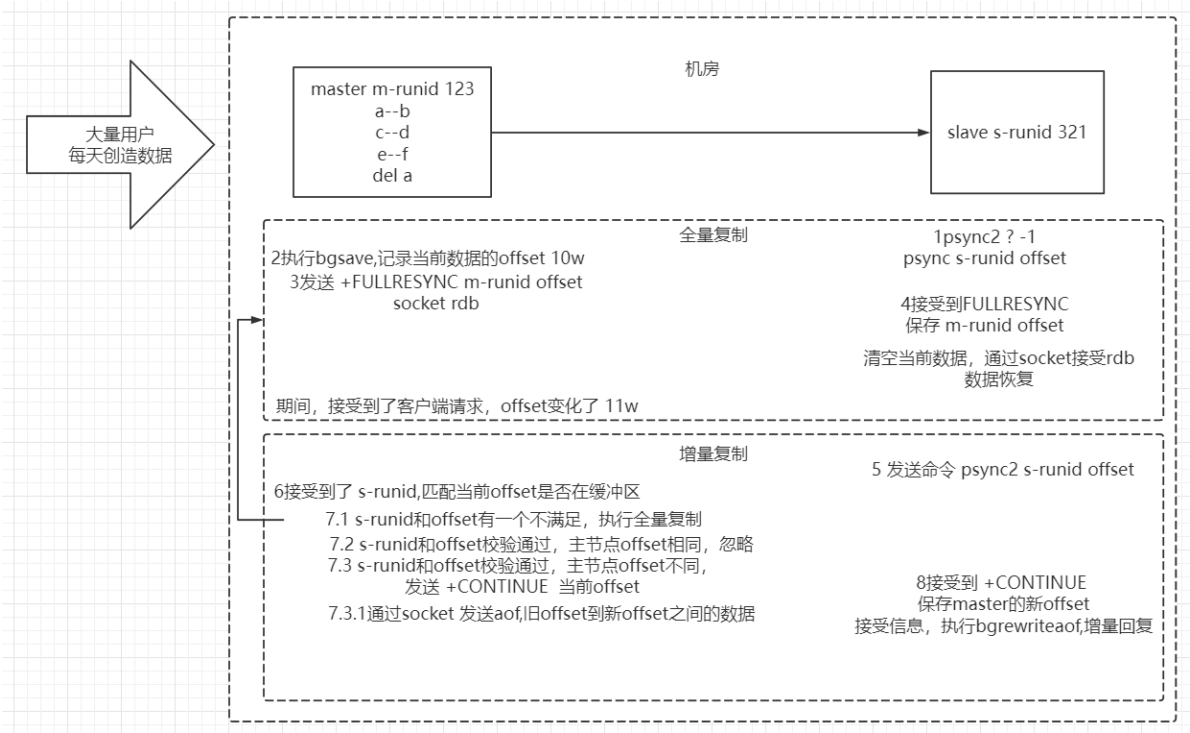
- 字节值

工作原理

- 通过offset区分不同的slave当前数据传播的差异
- master记录已发送的信息对应的offset
- slave记录已接收的信息对应的offset

#2.2 流程更新(全量复制/部分复制)

我们再次的总结一下主从复制的三个阶段的工作流程：



#2.3 心跳机制

什么是心跳机制?

进入命令传播阶段候，master与slave间需要进行信息交换，使用心跳机制进行维护，实现双方连接保持在线

master心跳：

- 内部指令：PING
- 周期：由repl-ping-slave-period决定，默认10秒
- 作用：判断slave是否在线
- 查询：INFO replication 获取slave最后一次连接时间间隔，lag项维持在0或1视为正常

slave心跳任务

- 内部指令：REPLCONF ACK {offset}
- 周期：1秒
- 作用1：汇报slave自己的复制偏移量，获取最新的数据变更指令
- 作用2：判断master是否在线

心跳阶段注意事项：

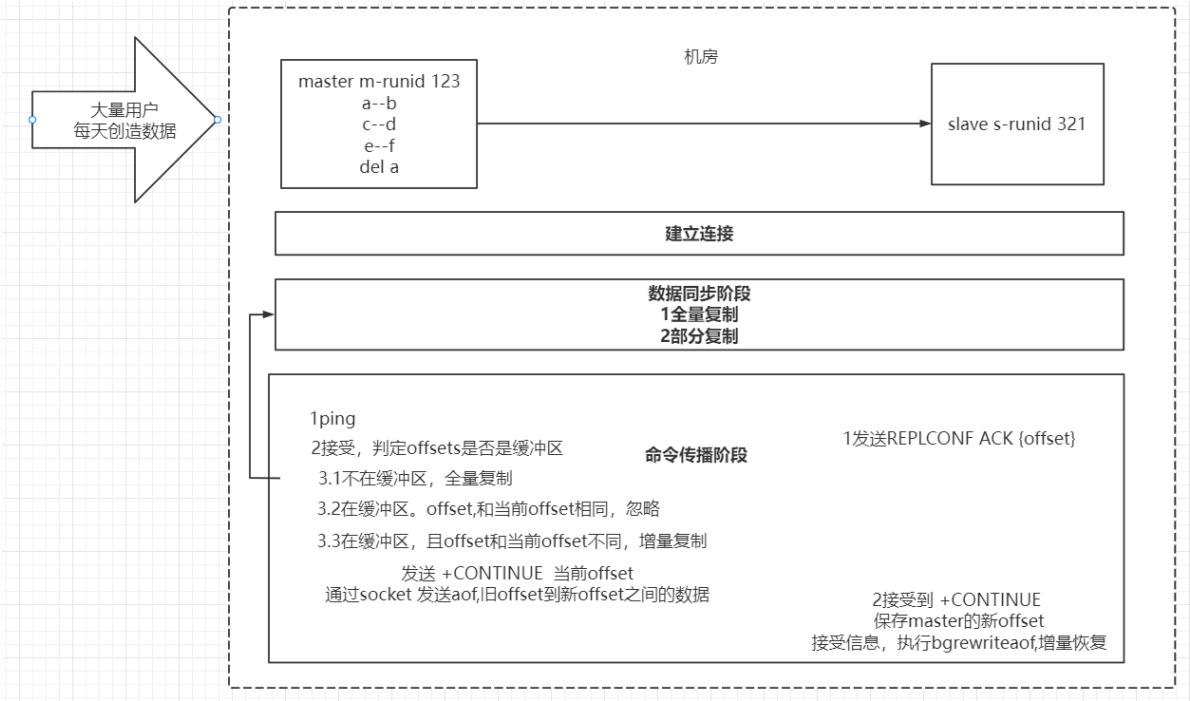
- 当slave多数掉线，或延迟过高时，master为保障数据稳定性，将拒绝所有信息同步

```
1   min-slaves-to-write 2
2   min-slaves-max-lag 8
```

slave数量少于2个，或者所有slave的延迟都大于等于8秒时，强制关闭master写功能，停止数据同步

- slave数量由slave发送REPLCONF ACK命令做确认
- slave延迟由slave发送REPLCONF ACK命令做确认

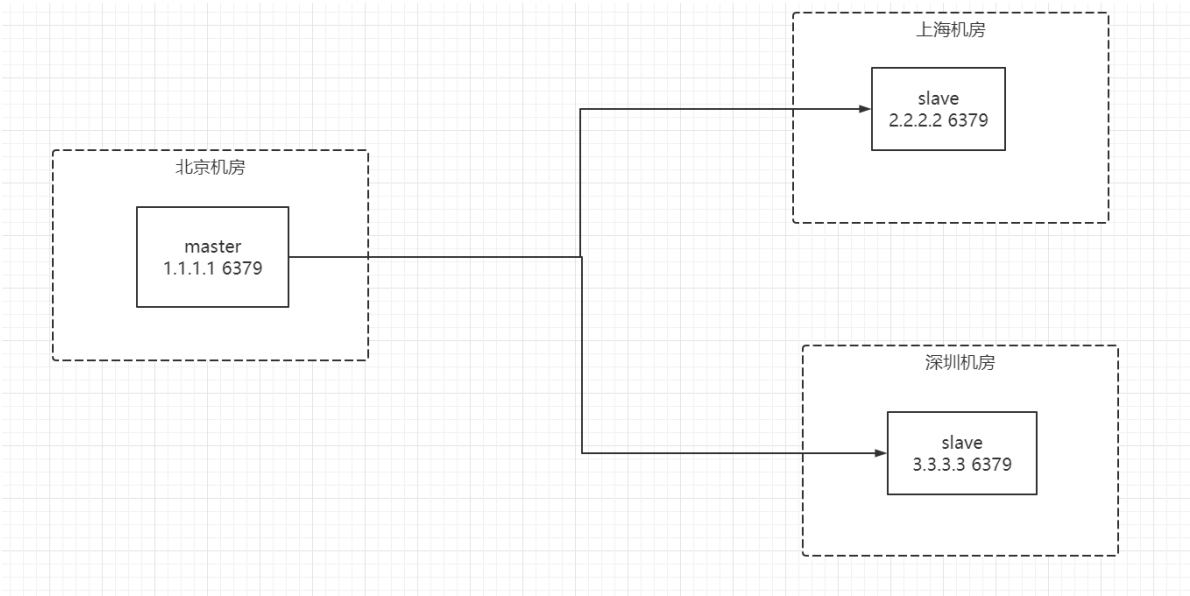
至此：我们可以总结出完整的主从复制流程：



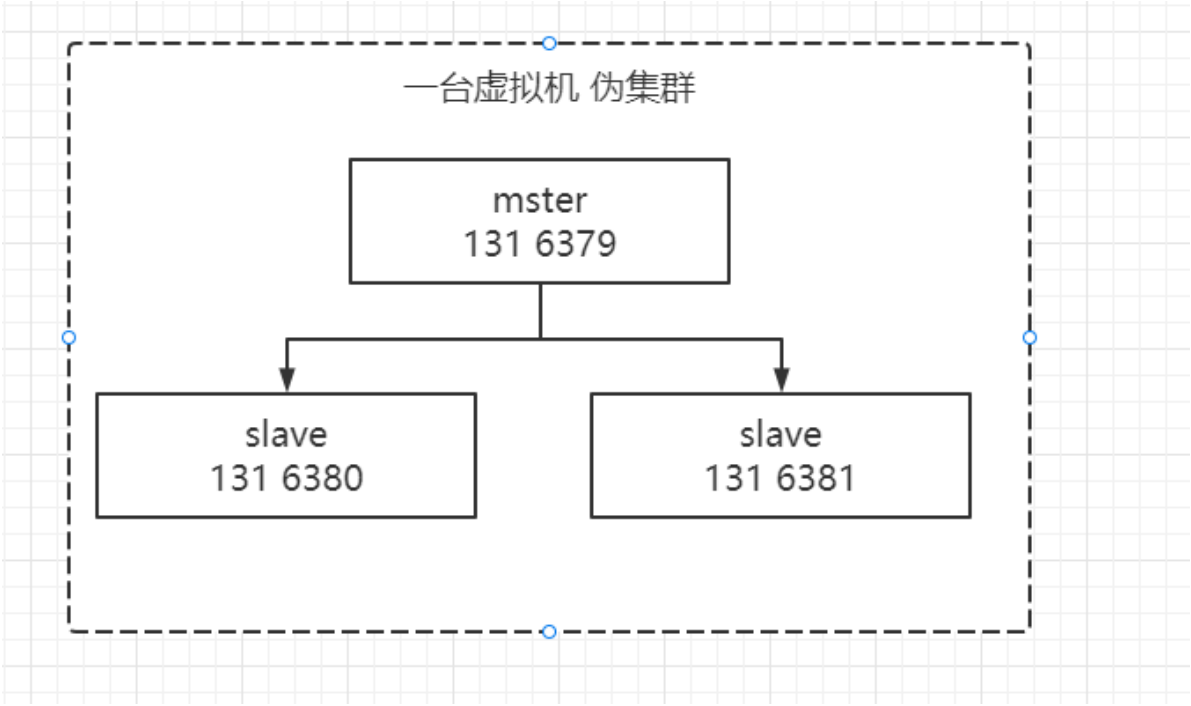
#3、搭建主从架构

#3.1背景

真实的生产环境，主从部署在不同的物理地方



教学



#3.2 操作

1复制两套redis redis6380 redis 6381

```
cp redis-6.2.6/ redis6380
cp redis-6.2.6/ redis6381
```

2分别修改配置文件：

```
port 6380
pidfile /var/run/redis_6380.pid
logfile "/export/server/redis6380/log/redis.log"
dir /export/server/redis6380/data
```

```
port 6381
pidfile /var/run/redis_6381.pid
logfile "/export/server/redis6381/log/redis.log"
dir /export/server/redis6381/data
```

3**启动**：

```
./bin/redis-server redis.conf slaveof 192.168.200.131 6379
```

4效果

登陆6380

```
./bin/redis-cli -h 192.168.200.131 -p 6380
```

```
info
```

```
# Replication
role:slave
master_host:192.168.200.131
master_port:6379
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_read_repl_offset:14
slave_repl_offset:14
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:e3faf93488cc543c52b7589f42e2967b32bd551d
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:14
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
```

5测试

1 主节点写数据，从节点获取数据

```
192.168.200.131:6379> set a b
OK
```

```
192.168.200.131:6380> get a
"b"
```

2从节点不能写数据

```
192.168.200.131:6381> set aa bb
(error) READONLY You can't write against a read only replica.
```

#4、主从复制常见问题-架构师

#4.1 频繁的全量复制

- 伴随着系统的运行，master的数据量会越来越大，一旦master重启，runid将发生变化，会导致全部slave的全量复制操作

内部优化调整方案：

1：master内部创建master_replid变量，使用runid相同的策略生成，长度41位，并发送给所有slave

2：在master关闭时执行命令shutdown save，进行RDB持久化,将runid与offset保存到RDB文件中

```
repl-id  repl-offset

通过redis-check-rdb命令可以查看该信息
```

3：master重启后加载RDB文件，恢复数据，重启后，将RDB文件中保存的repl-id与repl-offset加载到内存中

```
master_repl_id=repl  master_repl_offset =repl-offset

通过info命令可以查看该信息
```

作用：本机保存上次runid，重启后恢复该值，使所有slave认为还是之前的master

- 第二种出现频繁全量复制的问题现象：网络环境不佳，出现网络中断，slave不提供服务

问题原因：复制缓冲区过小，断网后slave的offset越界，触发全量复制

最终结果：slave反复进行全量复制

解决方案：修改复制缓冲区大小

```
repl-backlog-size 20mb
```

建议设置如下：

1.测算从master到slave的重连平均时长second 10s

2.获取master平均每秒产生写命令数据总量write_size_per_second 10w

3.最优复制缓冲区空间 = 2 * second * write_size_per_second 10w*10b 20mb

#4.2 频繁的网络中断

- 问题现象：master的CPU占用过高 或 slave频繁断开连接

问题原因

```
slave每1秒发送REPLCONFACK命令到master

当slave接到了慢查询时（keys * ，hgetall等），会大量占用CPU性能

master每1秒调用复制定时函数replicationCron()，比对slave发现长时间没有进行响应
```

最终结果：master各种资源（输出缓冲区、带宽、连接等）被严重占用

解决方案：通过设置合理的超时时间，确认是否释放slave

```
repl-timeout seconds
```

该参数定义了超时时间的阈值（默认60秒），超过该值，释放slave

- 问题现象：slave与master连接断开

问题原因

```
master发送ping指令频度较低

master设定超时时间较短

ping指令在网络中存在丢包
```

解决方案：提高ping指令发送的频度

```
repl-ping-slave-period seconds
```

超时时间repl-time的时间至少是ping指令频度的5到10倍，否则slave很容易判定超时

#4.3 数据不一致

问题现象：多个slave获取相同数据不同步

问题原因：网络信息不同步，数据发送有延迟

解决方案

```
优化主从间的网络环境，通常放置在同一个机房部署，如使用阿里云等云服务器时要注意此现象
拉专线。vpn。4m 10万 2万/年
监控主从节点延迟（通过offset）判断，如果slave延迟过大，暂时屏蔽程序对该slave的数据访问
```

```
slave-serve-stale-data  yes|no
```

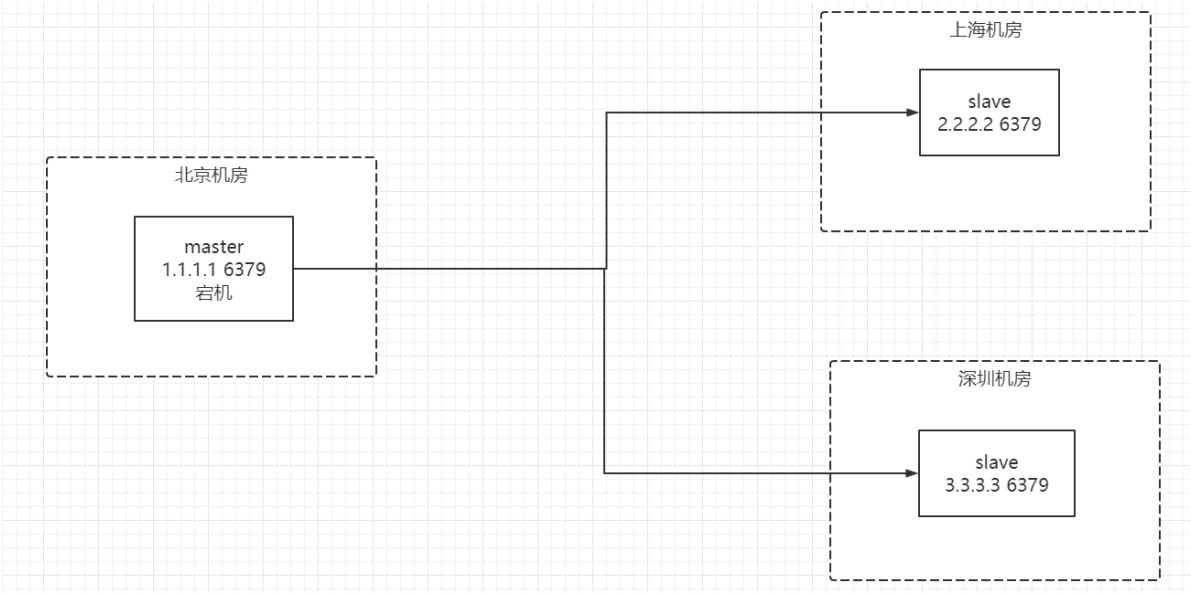
开启后仅响应info、slaveof等少数命令（慎用，除非对数据一致性要求很高）

#第五章 Redis中的Sentinel架构

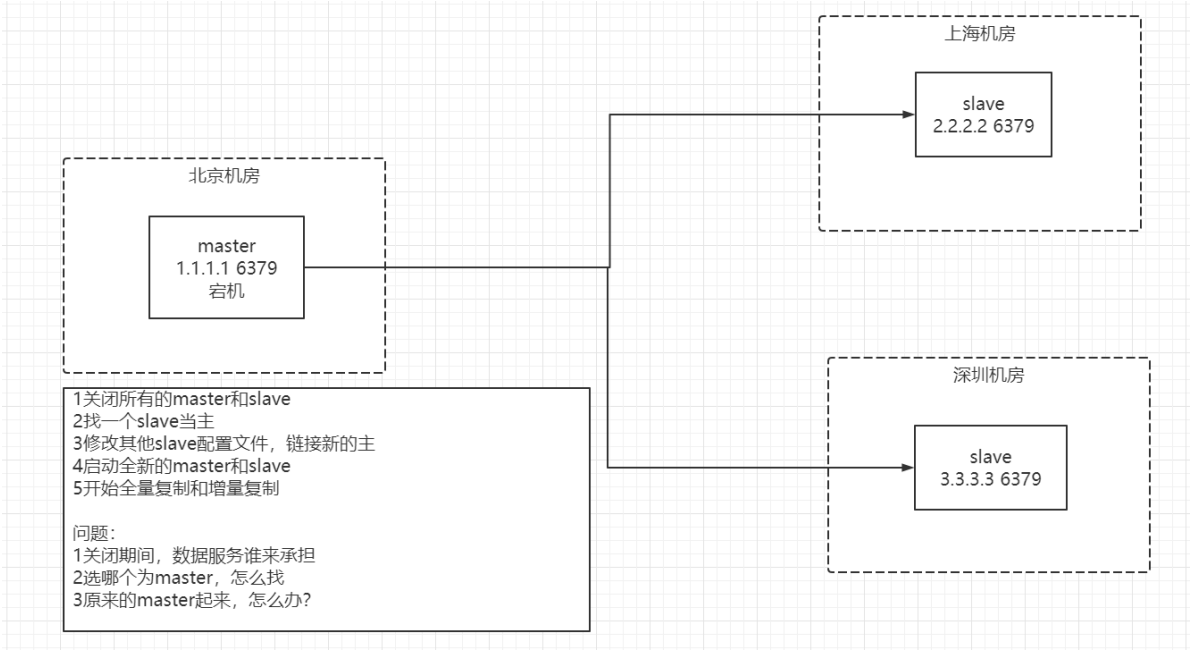
#1、哨兵简介

#1.1 哨兵概念

首先我们来看一个业务场景：如果redis的master宕机了，此时应该怎么办？



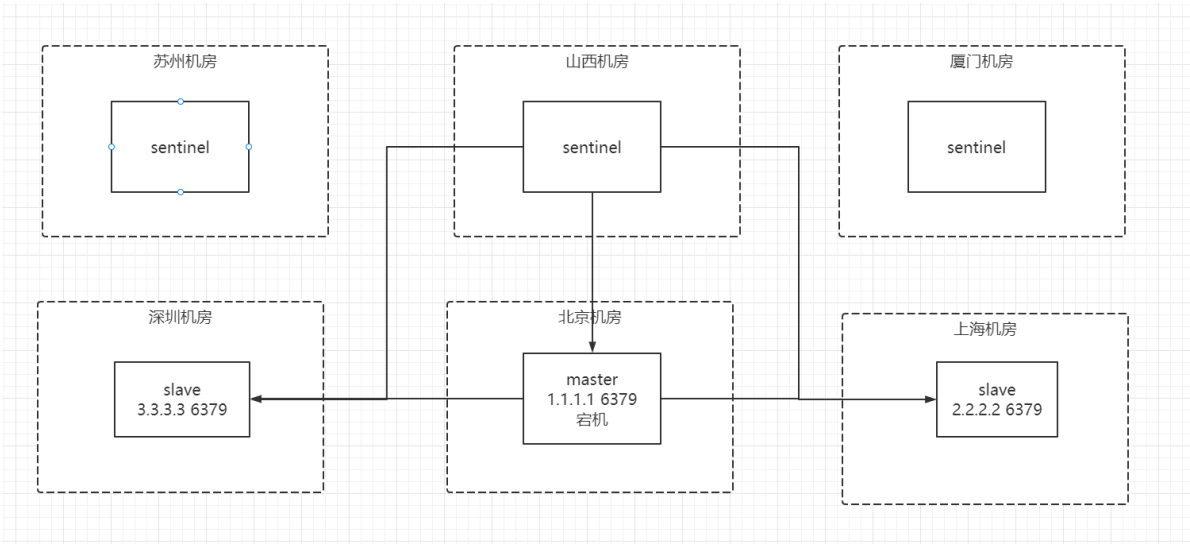
那此时我们可能需要从一堆的slave中重新选举出一个新的master，那这个操作过程是什么样的呢？这里面会有什么问题出现呢？



要实现这些功能，我们就需要redis的哨兵，那哨兵是什么呢?

哨兵

哨兵(sentinel) 是一个分布式系统，用于对主从结构中的每台服务器进行**监控**，当出现故障时通过**投票**机制**选择**新的master并将所有slave连接到新的master。



#1.2 哨兵作用

哨兵的作用：

- 监控：监控master和slave

  不断的检查master和slave是否正常运行

  master存活检测、master与slave运行情况检测
- 通知（提醒）：当被监控的服务器出现问题时，向其他（哨兵间，客户端）发送通知
- **自动故障转移**：断开master与slave连接，选取一个slave作为master，将其他slave连接新的master，并告知客户端新的服务器地址

注意：哨兵也是一台redis服务器，只是不提供数据相关服务，通常哨兵的数量配置为单数

#2、启用哨兵

配置哨兵

- 配置一拖二的主从结构（利用之前的方式启动即可）
- 配置三个哨兵（配置相同，端口不同），参看sentinel.conf

端口号

1：设置哨兵监听的主服务器信息， sentinel_number表示参与投票的哨兵数量

```
sentinel monitor master_name  master_host   master_port  sentinel_number
```

2：设置判定服务器宕机时长，该设置控制是否进行主从切换

```
sentinel down-after-milliseconds master_name    million_seconds
```

3：设置故障切换的最大超时时长

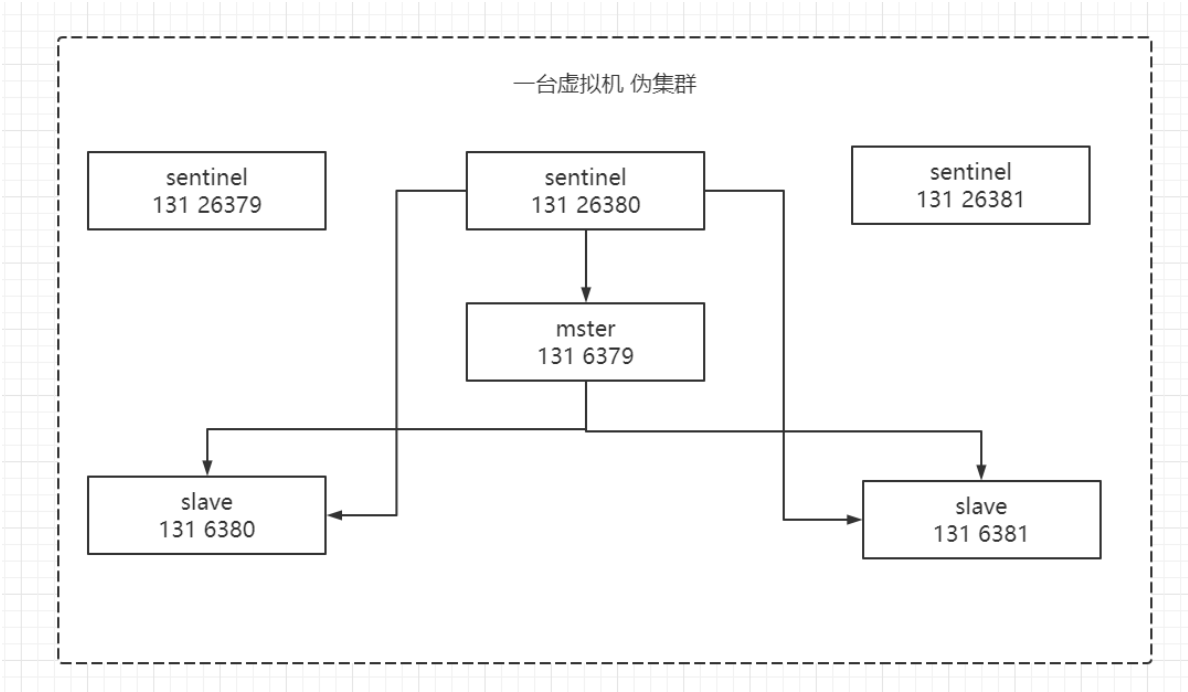
```
sentinel failover-timeout master_name   million_seconds
```

4：设置主从切换后，同时进行数据同步的slave数量，数值越大，要求网络资源越高，数值越小，同步时间越长

```
sentinel parallel-syncs master_name sync_slave_number
```

- 启动哨兵

```
redis-sentinel filename
```



#3、搭建哨兵

1copy出三份 redis运行包

```
cp -R redis6380 sentinel26379
cp -R redis6380 sentinel26380
cp -R redis6380 sentinel26381
```

2分别修改sentinel.conf

```
port 26379
daemonize yes
pidfile "/var/run/redis-sentinel26379.pid"
logfile "/export/server/sentinel26379/log/log.log"
dir "/export/server/sentinel26379/data"
sentinel monitor mymaster 192.168.200.131 6379 2

sentinel resolve-hostnames no
sentinel announce-hostnames no
```

3分别启动sentinel

```
./bin/redis-sentinel sentinel.conf
```

4链接sentinel

```
./bin/redis-cli -h 192.168.200.131 -p 26379
```

5查看状态

```
info
```

```
# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
sentinel_simulate_failure_flags:0
master0:name=mymaster,status=ok,address=192.168.200.131:6379,slaves=2,sentinels=3
192.168.200.131:26379>
```

6尝试故障迁移 杀掉主

```
[root@localhost ~]# ps -ef|grep redis
root        3730       1  0 07:39 ?        00:00:39 ./bin/redis-server 192.168.200.131:6379
root       13759       1  0 18:05 ?        00:00:11 ./bin/redis-server 192.168.200.131:6381
root       63442       1  0 22:02 ?        00:00:00 ./bin/redis-sentinel *:26379 [sentinel]
root       65528       1  0 22:03 ?        00:00:00 ./bin/redis-sentinel *:26380 [sentinel]
root       67011       1  0 22:03 ?        00:00:00 ./bin/redis-sentinel *:26381 [sentinel]
root       70670   15079  0 22:04 pts/2    00:00:00 ./bin/redis-cli -h 192.168.200.131 -p 26379
root       96840   10730  0 22:08 pts/8    00:00:00 grep --color=auto redis
root      102704   15078  0 20:48 pts/1    00:00:00 ./bin/redis-cli -h 192.168.200.131
root      129823       1  0 18:01 ?        00:00:11 ./bin/redis-server 192.168.200.131:6380
[root@localhost ~]# kill -9 3730
[root@localhost ~]#
```

登陆6380 info

```
# Replication
role:master
connected_slaves:1
slave0:ip=192.168.200.131,port=6381,state=online,offset=98186,lag=0
master_failover_state:no-failover
master_replid:126066e0dcd352770bb9e796896427ebf226f171
master_replid2:e3faf93488cc543c52b7589f42e2967b32bd551d
master_repl_offset:98331
second_repl_offset:95105
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:98331
```

7尝试6379恢复

```
[root@localhost redis-6.2.6]# ./bin/redis-server redis.conf
```

登陆6380 info

```
# Replication
role:master
connected_slaves:2
slave0:ip=192.168.200.131,port=6381,state=online,offset=123607,lag=0
slave1:ip=192.168.200.131,port=6379,state=online,offset=123317,lag=1
master_failover_state:no-failover
master_replid:126066e0dcd352770bb9e796896427ebf226f171
master_replid2:e3faf93488cc543c52b7589f42e2967b32bd551d
master_repl_offset:123607
second_repl_offset:95105
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:123607
```

#4、java代码链接sentinel

#4.1 原生

1. 在 创建一个新的类 ReidsSentinelTest
2. 构建JedisPoolConfig配置对象
3. 创建一个HashSet，用来保存哨兵节点配置信息（记得一定要写端口号）
4. 构建JedisSentinelPool连接池
5. 使用sentinelPool连接池获取连接

```
1  package com.ydlclass.redis;
2
3  import org.testng.annotations.AfterTest;
4  import org.testng.annotations.BeforeTest;
5  import org.testng.annotations.Test;
```

```java
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPoolConfig;
import redis.clients.jedis.JedisSentinelPool;

import java.util.HashSet;
import java.util.Set;

/**
 * @Created by IT李老师
 * 公主号 “IT李哥交朋友”
 * 个人微 itlils
 */
public class ReidsSentinelTest {
    JedisSentinelPool jedisSentinelPool;

    //1. 在 创建一个新的类 ReidsSentinelTest
    //2. 构建JedisPoolConfig配置对象
    //3. 创建一个HashSet，用来保存哨兵节点配置信息（记得一定要写端口号）
    //4. 构建JedisSentinelPool连接池
    //5. 使用sentinelPool连接池获取连接
    @BeforeTest
    public void beforeTest(){
        //创建jedis连接池
        JedisPoolConfig config=new JedisPoolConfig();
        //最大空闲连接
        config.setMaxIdle(10);
        //最小空闲连接
        config.setMinIdle(5);
        //最大空闲时间
        config.setMaxWaitMillis(3000);
        //最大连接数
        config.setMaxTotal(50);


        Set<String> sentinels=new HashSet<>();
        sentinels.add("192.168.200.131:26379");
        sentinels.add("192.168.200.131:26380");
        sentinels.add("192.168.200.131:26381");

        jedisSentinelPool= new JedisSentinelPool("mymaster",sentinels,config);
    }

    @Test
    public void keysTest(){
        Jedis jedis = jedisSentinelPool.getResource();
        Set<String> keys = jedis.keys("*");
        for (String key : keys) {
            System.out.println(key);
        }
    }


    @AfterTest
    public void afterTest(){
        jedisSentinelPool.close();
    }

}
```

#4.2 springboot（后边学了springboot就知道了）

```
spring:
  redis:
    sentinel:
      nodes: 192.168.200.131:26379,192.168.200.131:26380,192.168.200.131:26381 //哨兵的ip和端口
      master: mymaster //这个就是哨兵配置文件中 sentinel monitor mymaster 192.168.200.131 6379 2 配置的mymaster
```

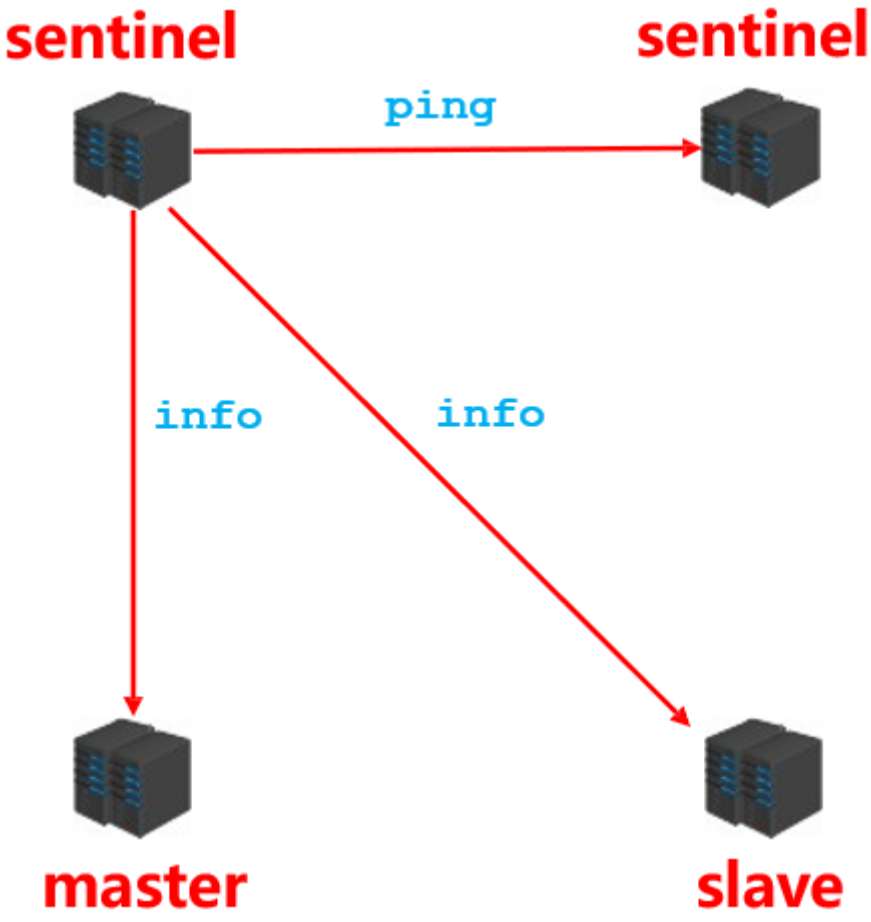
#5、哨兵工作原理-面试

哨兵在进行主从切换过程中经历三个阶段

- 监控
- 通知
- 故障转移

#5.1 监控

用于同步各个节点的状态信息



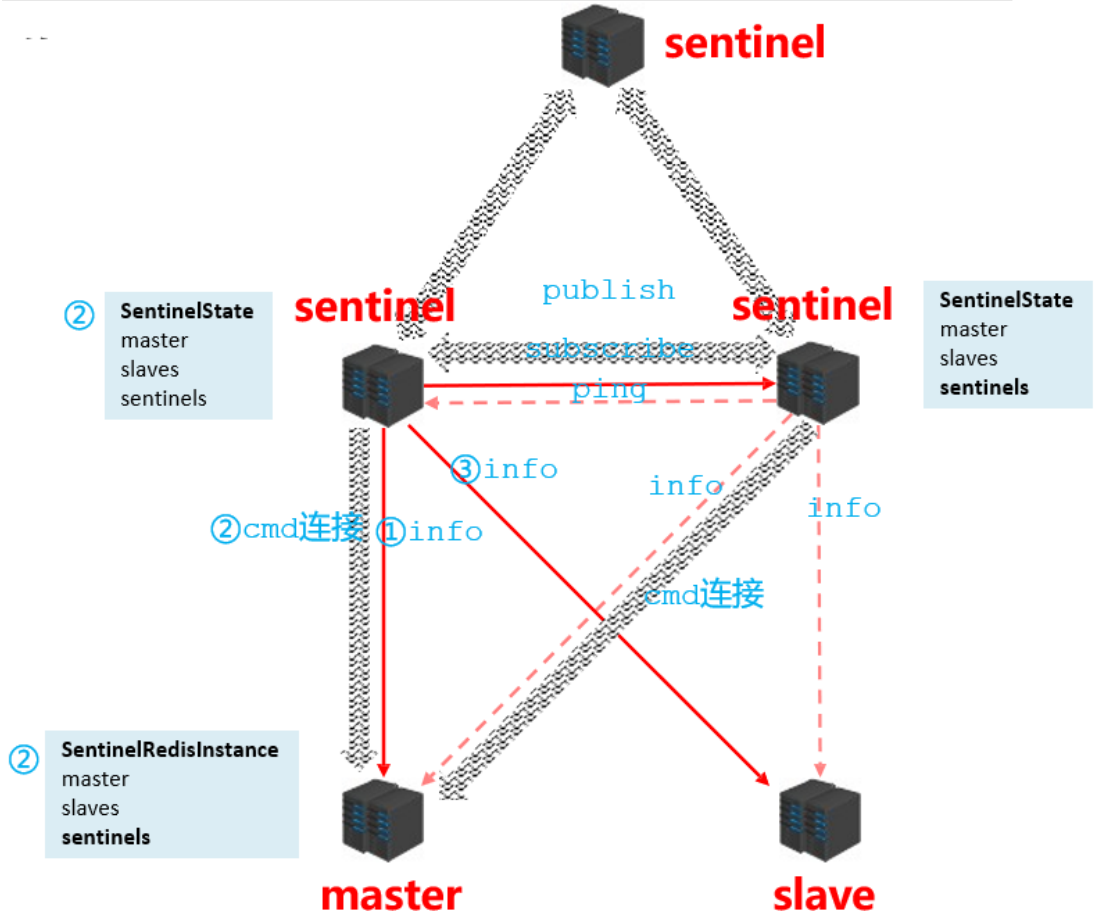
- 获取各个sentinel的状态（是否在线）
- 获取master的状态

```
master属性
    prunid
    prole: master
各个slave的详细信息
```

- 获取所有slave的状态（根据master中的slave信息）

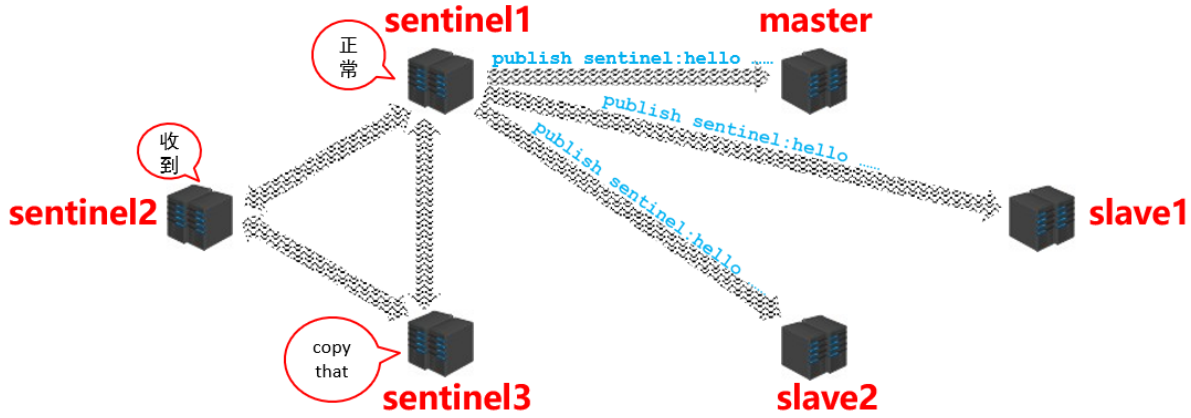
```
slave属性
    prunid
    prole：slave
    pmaster_host、master_port
    poffset
```

其内部的工作原理具体如下：



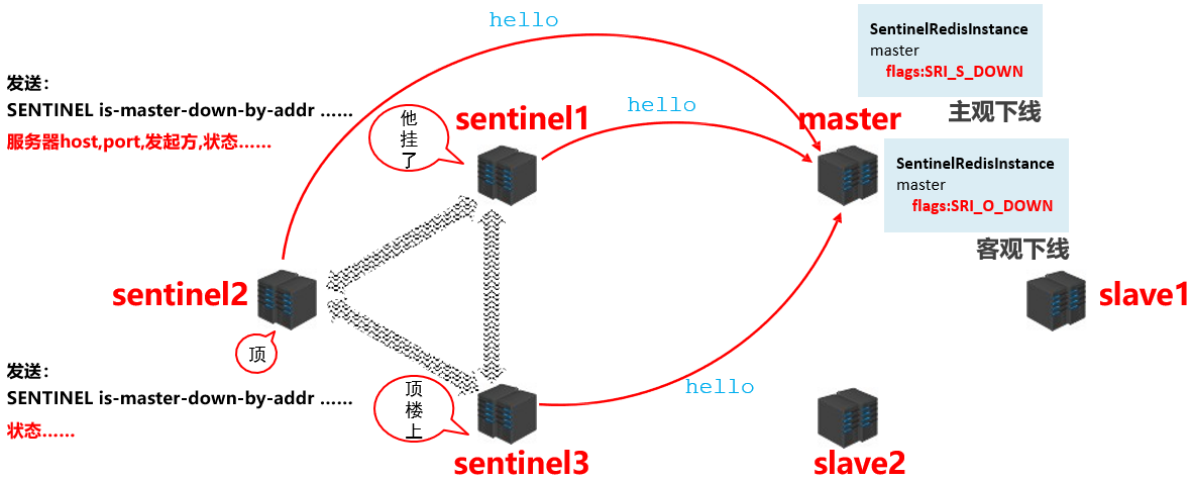
#5.2 通知

sentinel在通知阶段要不断的去获取master/slave的信息，然后在各个sentinel之间进行共享，具体的流程如下：

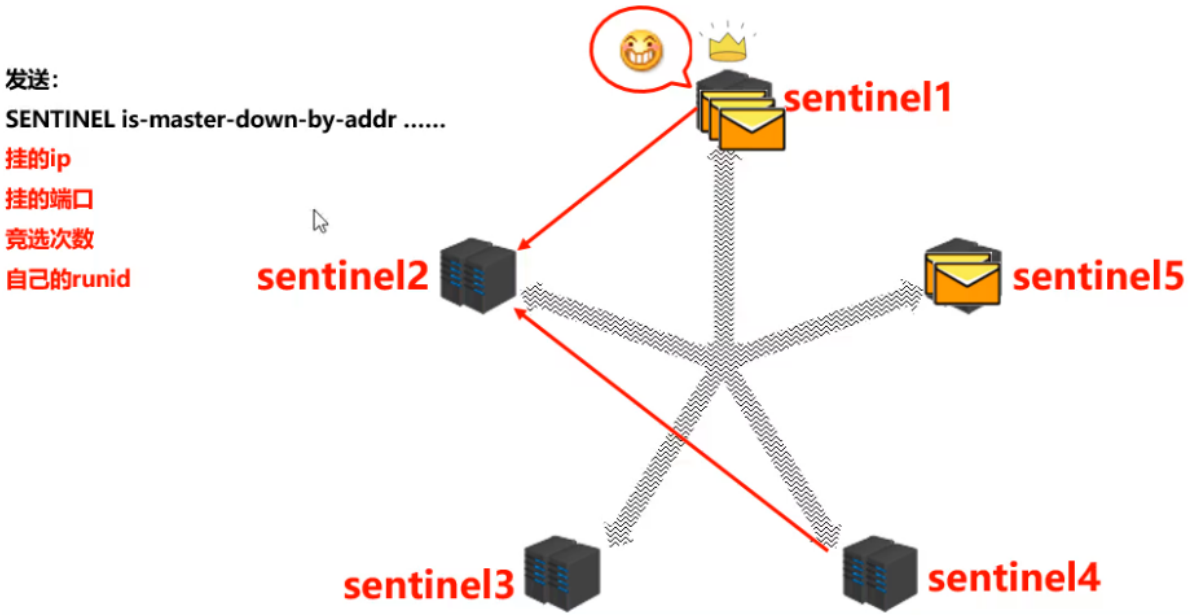


#5.3 故障转移

当master宕机后sentinel是如何知晓并判断出master是真的宕机了呢？我们来看具体的操作流程



当sentinel认定master下线之后，此时需要决定更换master，那这件事由哪个sentinel来做呢？这时候sentinel之间要进行选举，如下图所示：



在选举的时候每一个人手里都有一票，而每一个人的又都想当这个处理事故的人，那怎么办？大家就开始抢，于是每个人都会发出一个指令，在内网里边告诉大家我要当选举人，比如说现在的sentinel1和sentinel4发出这个选举指令了，那么sentinel2既能接到sentinel1的也能接到sentinel4的，接到了他们的申请以后呢，sentinel2他就会把他的一票投给其中一方，投给谁呢？谁先过来我投给谁，假设sentinel1先过来，所以这个票就给到了sentinel1。那么给过去以后呢，现在sentinel1就拿到了一票，按照这样的一种形式，最终会有一个选举结果。对应的选举最终得票多的，那自然就成为了处理事故的人。需要注意在这个过程中有可能会存在失败的现象，就是一轮选举完没有选取，那就会接着进行第二轮第三轮直到完成选举。

接下来就是由选举胜出的sentinel去从slave中选一个新的master出来的工作，这个流程是什么样的呢？

首先它有一个在服务器列表中挑选备选master的原则

- 不在线的OUT
- 响应慢的OUT
- 与原master断开时间久的OUT
- 优先原则

  优先级 offset runid

选出新的master之后，发送指令（sentinel）给其他的slave：

- 向新的master发送slaveof no one
- 向其他slave发送slaveof 新masterIP端口

总结：故障转移阶段

1. 发现问题，主观下线与客观下线
2. 竞选负责人
3. 优选新master
4. 新master上任，其他slave切换master，原master作为slave故障恢复后连接

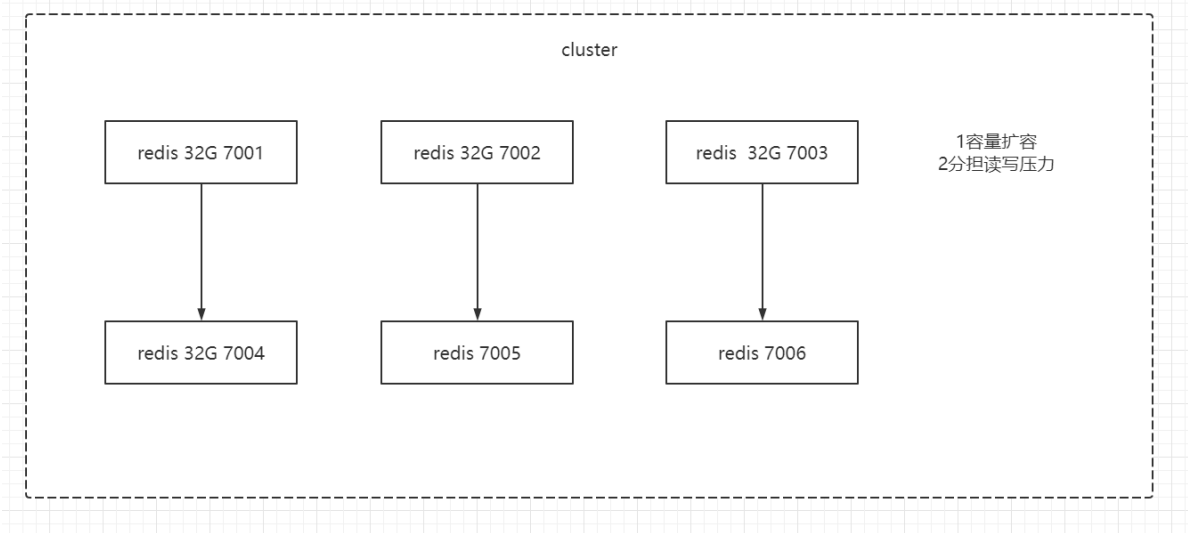
#第六章 Redis cluster集群

现状问题：业务发展过程中遇到的峰值瓶颈

- redis提供的服务OPS可以达到10万/秒，当前业务OPS已经达到10万/秒
- 内存单机容量达到256G，当前业务需求内存容量1T
- 使用集群的方式可以快速解决上述问题

#1、集群简介

集群就是使用网络将若干台计算机联通起来，并提供统一的管理方式，使其对外呈现单机的服务效果



集群作用：

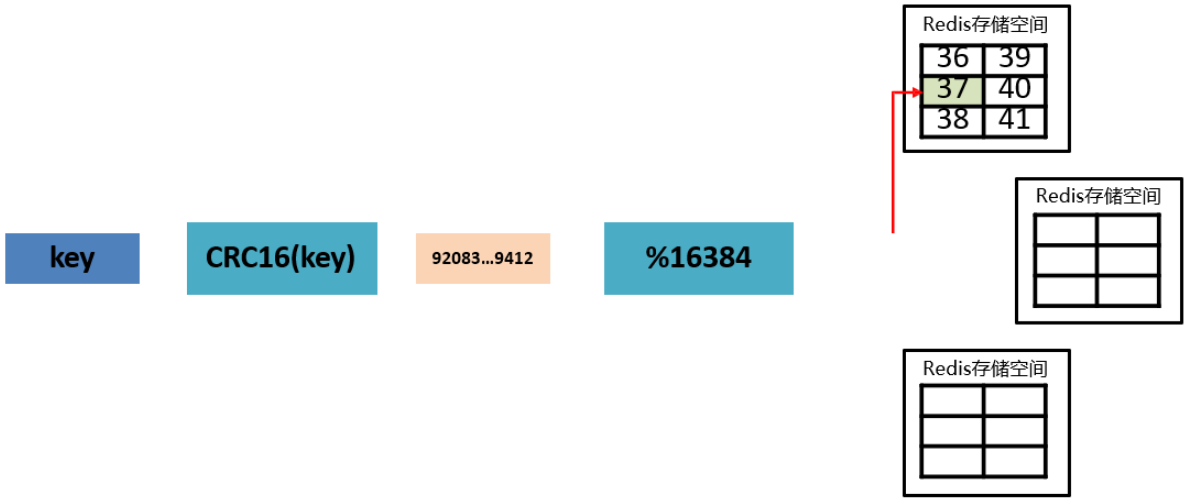
- 分散单台服务器的访问压力，实现负载均衡
- 分散单台服务器的存储压力，实现可扩展性
- 降低单台服务器宕机带来的业务灾难

#2、Cluster集群结构设计

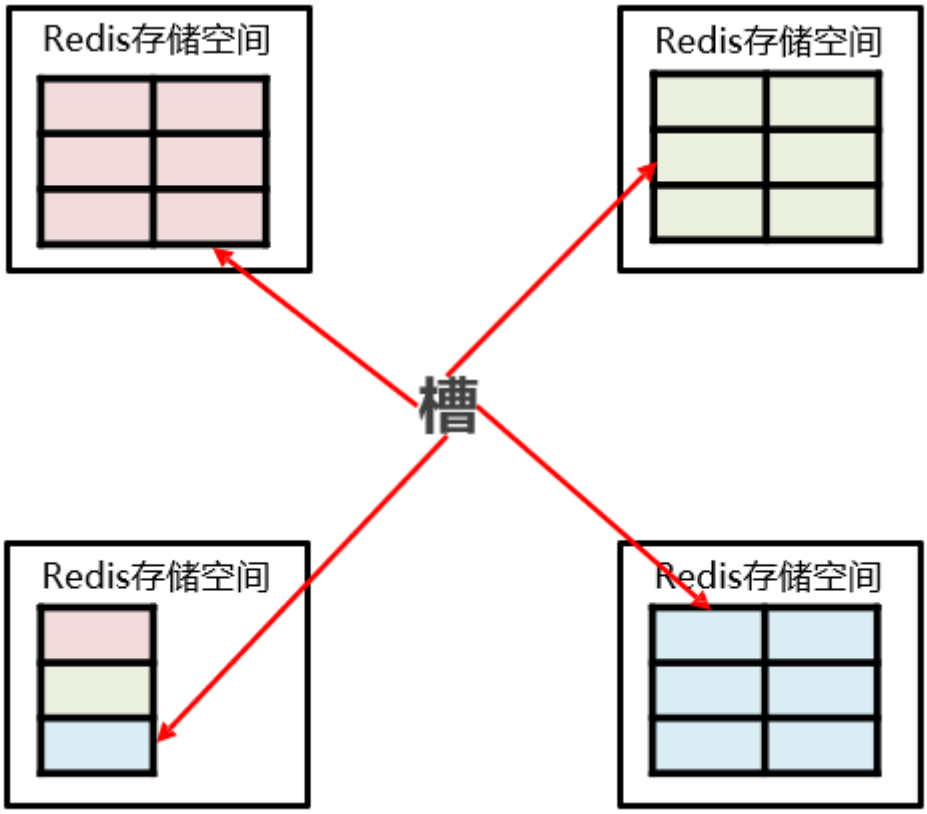
数据存储设计：

1. 通过算法设计，计算出key应该保存的位置
2. 将所有的存储空间计划切割成16384份，每台主机保存一部分

   注意：每份代表的是一个存储空间，不是一个key的保存空间
3. 将key按照计算出的结果放到对应的存储空间

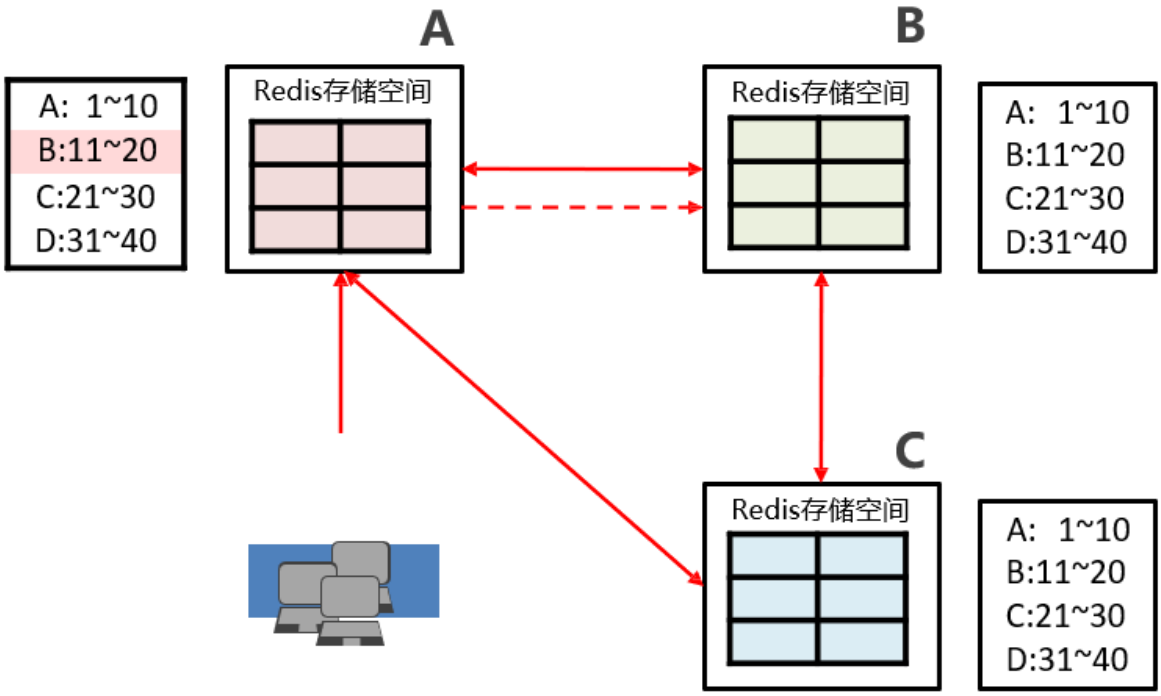


那redis的集群是如何增强可扩展性的呢？譬如我们要增加一个集群节点



当我们查找数据时，集群是如何操作的呢？

- 各个数据库相互通信，保存各个库中槽的编号数据
- 一次命中，直接返回
- 一次未命中，告知具体位置



#3、Cluster集群结构搭建

首先要明确的几个要点:

- 配置服务器（3主3从）
- 建立通信（Meet）
- 分槽（Slot）
- 搭建主从（master-slave）

1 创建6个redis单体服务 7001-7006。修改redis.conf

```
port 7001
bind 192.168.200.131
protected-mode no
daemonize yes
pidfile /var/run/redis_7001.pid
logfile "/export/server/redis7001/log/redis.log"
dir /export/server/redis7001/data/
appendonly yes
cluster-enabled yes
cluster-config-file nodes-7001.conf
cluster-node-timeout 15000
```

2让六台机器组成集群

```
redis-cli --cluster create 192.168.200.131:7001 192.168.200.131:7002
192.168.200.131:7003 192.168.200.131:7004 192.168.200.131:7005
192.168.200.131:7006 --cluster-replicas 1
```

```
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 192.168.200.131:7005 to 192.168.200.131:7001
Adding replica 192.168.200.131:7006 to 192.168.200.131:7002
Adding replica 192.168.200.131:7004 to 192.168.200.131:7003
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: c4bf6866cdd913dad5fcc535afe4889b0643d1d8 192.168.200.131:7001
   slots:[0-5460] (5461 slots) master
M: 0a31b6154a98ca3db59a9f5d2a66592b77b95511 192.168.200.131:7002
   slots:[5461-10922] (5462 slots) master
M: 6ee693561d0d3146fb7f333e7b39427f4549e1e6 192.168.200.131:7003
   slots:[10923-16383] (5461 slots) master
S: bb27da86c2e5e8c4c08327c3f4a074ff4407883d 192.168.200.131:7004
   replicates c4bf6866cdd913dad5fcc535afe4889b0643d1d8
S: 95f0f1961177238c130f3e4d0f47c29ee378efe8 192.168.200.131:7005
   replicates 0a31b6154a98ca3db59a9f5d2a66592b77b95511
S: c9b116c4dbc8fd357b6cb7a03b7648c2a772aaba 192.168.200.131:7006
   replicates 6ee693561d0d3146fb7f333e7b39427f4549e1e6
Can I set the above configuration? (type 'yes' to accept): yes
```

输入yes

```
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
.
>>> Performing Cluster Check (using node 192.168.200.131:7001)
M: c4bf6866cdd913dad5fcc535afe4889b0643d1d8 192.168.200.131:7001
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: 6ee693561d0d3146fb7f333e7b39427f4549e1e6 192.168.200.131:7003
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: 95f0f1961177238c130f3e4d0f47c29ee378efe8 192.168.200.131:7005
   slots: (0 slots) slave
   replicates 0a31b6154a98ca3db59a9f5d2a66592b77b95511
S: c9b116c4dbc8fd357b6cb7a03b7648c2a772aaba 192.168.200.131:7006
   slots: (0 slots) slave
   replicates 6ee693561d0d3146fb7f333e7b39427f4549e1e6
M: 0a31b6154a98ca3db59a9f5d2a66592b77b95511 192.168.200.131:7002
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
S: bb27da86c2e5e8c4c08327c3f4a074ff4407883d 192.168.200.131:7004
   slots: (0 slots) slave
   replicates c4bf6866cdd913dad5fcc535afe4889b0643d1d8
[OK] All nodes agree about slots configuration.
```

客户端连接cluster

```
./redis-cli -c -h 192.168.200.131 -p 7001
```

```
[root@localhost bin]# ./redis-cli -c  -h 192.168.200.131 -p 7001
192.168.200.131:7001> set a b
-> Redirected to slot [15495] located at 192.168.200.131:7003
OK
192.168.200.131:7003>
```

设置值，跳转到正确的服务器上。

Cluster配置

- 是否启用cluster，加入cluster节点

```
cluster-enabled yes|no
```

- cluster配置文件名，该文件属于自动生成，仅用于快速查找文件并查询文件内容

```
cluster-config-file filename
```

- 节点服务响应超时时间，用于判定该节点是否下线或切换为从节点

```
cluster-node-timeout milliseconds
```

- master连接的slave最小数量

```
cluster-migration-barrier min_slave_number
```

Cluster节点操作命令

- 查看集群节点信息

```
cluster nodes
```

- 更改slave指向新的master

```
cluster replicate master-id
```

- 发现一个新节点，新增master

```
cluster meet ip:port
```

- 忽略一个没有solt的节点

```
cluster forget server_id
```

- 手动故障转移

```
cluster failover
```

集群操作命令：

- 创建集群

```
redis-cli –-cluster create masterhost1:masterport1 masterhost2:masterport2
masterhost3:masterport3 [masterhostn:masterportn …] slavehost1:slaveport1
slavehost2:slaveport2 slavehost3:slaveport3 -–cluster-replicas n
```

注意：master与slave的数量要匹配，一个master对应n个slave，由最后的参数n决定

master与slave的匹配顺序为第一个master与前n个slave分为一组，形成主从结构

- 添加master到当前集群中，连接时可以指定任意现有节点地址与端口

```
redis-cli --cluster add-node new-master-host:new-master-port now-host:now-port
```

- 添加slave

```
redis-cli --cluster add-node new-slave-host:new-slave-port master-host:master-port
--cluster-slave --cluster-master-id masterid
```

- 删除节点，如果删除的节点是master，必须保障其中没有槽slot

```
redis-cli --cluster del-node del-slave-host:del-slave-port del-slave-id
```

- 重新分槽，分槽是从具有槽的master中划分一部分给其他master，过程中不创建新的槽

```
redis-cli --cluster reshard new-master-host:new-master:port --cluster-from src-
master-id1, src-master-id2, src-master-idn --cluster-to target-master-id --
cluster-slots slots
```

注意：将需要参与分槽的所有masterid不分先后顺序添加到参数中，使用，分隔

指定目标得到的槽的数量，所有的槽将平均从每个来源的master处获取

- 重新分配槽，从具有槽的master中分配指定数量的槽到另一个master中，常用于清空指定master中的槽

```
redis-cli --cluster reshard src-master-host:src-master-port --cluster-from src-master-id --cluster-to target-master-id --cluster-slots slots --cluster-yes
```

#4、java操作

1原生

JedisCluster

```
package com.ydlclass.redis;

import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;
import redis.clients.jedis.HostAndPort;
import redis.clients.jedis.JedisCluster;
import redis.clients.jedis.JedisPoolConfig;

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

/**
 * @Created by IT李老师
 * 公主号“IT李哥交朋友”
 * 个人微 itlils
 */
public class RedisClusterTest {
    JedisCluster jedisCluster;

    @BeforeTest
    public void beforeTest(){
        //创建jedis连接池
        JedisPoolConfig config=new JedisPoolConfig();
        //最大空闲连接
        config.setMaxIdle(10);
        //最小空闲连接
        config.setMinIdle(5);
        //最大空闲时间
        config.setMaxWaitMillis(3000);
        //最大连接数
        config.setMaxTotal(50);

        Set<HostAndPort> nodes=new HashSet<>();
        nodes.add(new HostAndPort("192.168.200.131", 7001));
        nodes.add(new HostAndPort("192.168.200.131", 7002));
        nodes.add(new HostAndPort("192.168.200.131", 7003));
        nodes.add(new HostAndPort("192.168.200.131", 7004));
        nodes.add(new HostAndPort("192.168.200.131", 7005));
        nodes.add(new HostAndPort("192.168.200.131", 7006));

        jedisCluster= new JedisCluster(nodes,config);
    }
```

```
45
46      @Test
47      public void addTest(){
48          jedisCluster.set("c", "d");
49          String str = jedisCluster.get("c");
50          System.out.println(str);
51      }
52
53      @AfterTest
54      public void afterTest(){
55          try {
56              jedisCluster.close();
57          } catch (IOException e) {
58              e.printStackTrace();
59          }
60      }
61
62  }
```

2springboot

配置文件

```
1   spring:
2     redis:
3       pool:
4         max-idle: 100
5         min-idle: 1
6         max-active: 1000
7         max-wait: -1
8       cluster:
9         nodes:
10          - 192.168.200.131:7001
11          - 192.168.200.131:7002
12          - 192.168.200.131:7003
13          - 192.168.200.131:7004
14          - 192.168.200.131:7005
15          - 192.168.200.131:7006
16      database: 0
17      timeout: 15000
18      connect-timeout: 5000
```

#第七章 Redis高频面试题

#1、缓存预热

场景：“宕机”

服务器启动后迅速宕机

问题排查：

1.请求数量较高，大量的请求过来之后都需要去从缓存中获取数据，但是缓存中又没有，此时从数据库中查找数据然后将数据再存入缓存，造成了短期内对redis的高强度操作从而导致问题

2.主从之间数据吞吐量较大，数据同步操作频度较高

解决方案：

- 前置准备工作：

1.日常例行统计数据访问记录，统计访问频度较高的热点数据

2.利用LRU数据删除策略，构建数据留存队列例如：storm与kafka配合

- 准备工作：

1.将统计结果中的数据分类，根据级别，redis优先加载级别较高的热点数据

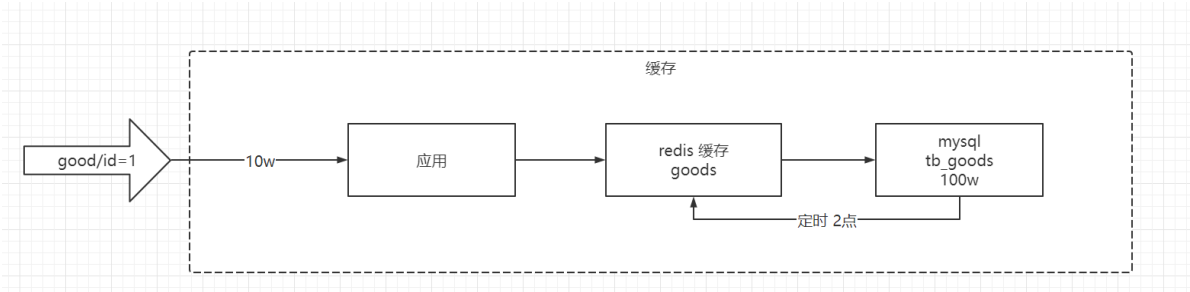
2.利用分布式多服务器同时进行数据读取，提速数据加载过程

3.热点数据主从同时预热

- 实施：

4.使用脚本程序固定触发数据预热过程

5.如果条件允许，使用了CDN（内容分发网络），效果会更好



总的来说：缓存预热就是系统启动前，提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

#2、缓存雪崩

场景：数据库服务器崩溃，一连串的场景会随之儿来

1.系统平稳运行过程中，忽然数据库连接量激增

2.应用服务器无法及时处理请求

3.大量408，500错误页面出现

4.客户反复刷新页面获取数据

5.数据库崩溃

6.应用服务器崩溃

7.重启应用服务器无效

8.Redis服务器崩溃

9.Redis集群崩溃

10.重启数据库后再次被瞬间流量放倒

问题排查：

1.在一个较短的时间内，缓存中较多的key集中过期

2.此周期内请求访问过期的数据，redis未命中，redis向数据库获取数据

3.数据库同时接收到大量的请求无法及时处理

4.Redis大量请求被积压，开始出现超时现象

5.数据库流量激增，数据库崩溃

6.重启后仍然面对缓存中无数据可用

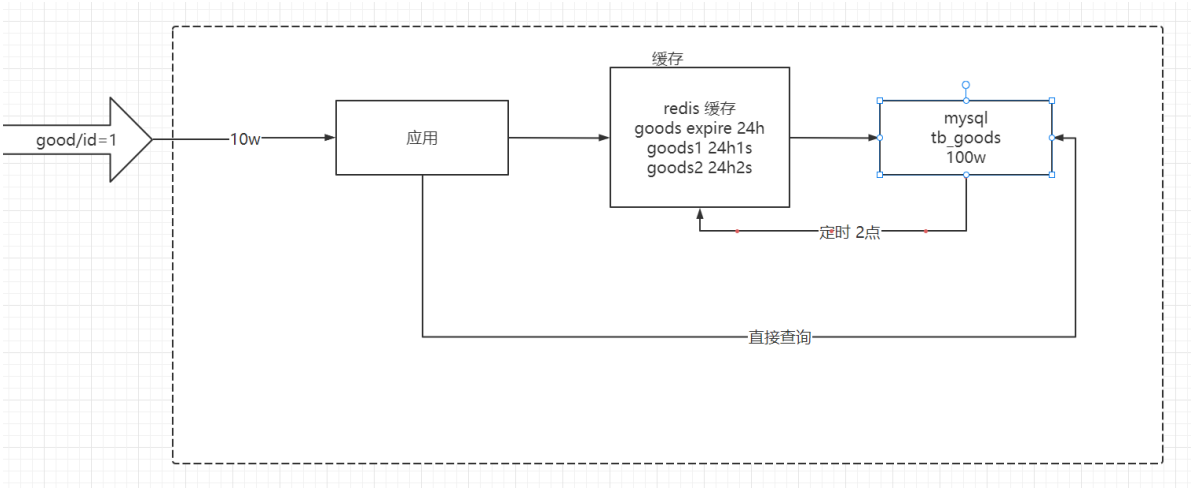
7.Redis服务器资源被严重占用，Redis服务器崩溃

8.Redis集群呈现崩塌，集群瓦解

9.应用服务器无法及时得到数据响应请求，来自客户端的请求数量越来越多，应用服务器崩溃

10.应用服务器，redis，数据库全部重启，效果不理想

总而言之就两点：短时间范围内，大量key集中过期



解决方案

- 思路：

1.更多的页面静态化处理

2.构建多级缓存架构

Nginx缓存+redis缓存+ehcache缓存

3.检测Mysql严重耗时业务进行优化

对数据库的瓶颈排查：例如超时查询、耗时较高事务等

4.灾难预警机制

监控redis服务器性能指标

CPU占用、CPU使用率

内存容量

查询平均响应时间

线程数

5.限流、降级

短时间范围内牺牲一些客户体验，限制一部分请求访问，降低应用服务器压力，待业务低速运转后再逐步放开访问

- 落地实践：

1.LRU与LFU切换

2.数据有效期策略调整

根据业务数据有效期进行分类错峰，A类90分钟，B类80分钟，C类70分钟

过期时间使用固定时间+随机值的形式，稀释集中到期的key的数量

3.超热数据使用永久key

4.定期维护（自动+人工）

对即将过期数据做访问量分析，确认是否延时，配合访问量统计，做热点数据的延时

5.加锁：慎用！

总的来说：缓存雪崩就是瞬间过期数据量太大，导致对数据库服务器造成压力。如能够有效避免过期时间集中，可以有效解决雪崩现象的 出现（约40%），配合其他策略一起使用，并监控服务器的运行数据，根据运行记录做快速调整。

#3、缓存击穿

场景：还是数据库服务器崩溃，但是跟之前的场景有点不太一样

1.系统平稳运行过程中

2.数据库连接量瞬间激增

3.Redis服务器无大量key过期

4.Redis内存平稳，无波动

5.Redis服务器CPU正常

6.数据库崩溃

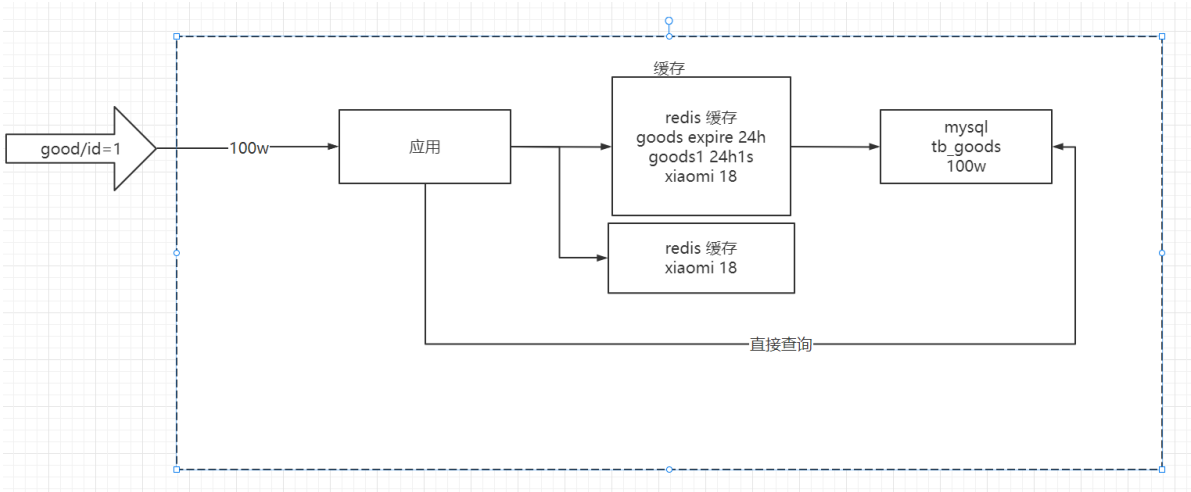
问题排查：

1.Redis中某个key过期，该key访问量巨大

2.多个数据请求从服务器直接压到Redis后，均未命中

3.Redis在短时间内发起了大量对数据库中同一数据的访问

总而言之就两点：单个key高热数据，key过期



解决方案：

1.预先设定

以电商为例，每个商家根据店铺等级，指定若干款主打商品，在购物节期间，加大此类信息key的过期时长 注意：购物节不仅仅指当天，以及后续若干天，访问峰值呈现逐渐降低的趋势

2.现场调整

监控访问量，对自然流量激增的数据延长过期时间或设置为永久性key

3.后台刷新数据

启动定时任务，高峰期来临之前，刷新数据有效期，确保不丢失

4.二级缓存

设置不同的失效时间，保障不会被同时淘汰就行

5.加锁

分布式锁，防止被击穿，但是要注意也是性能瓶颈，慎重！

总的来说：缓存击穿就是单个高热数据过期的瞬间，数据访问量较大，未命中redis后，发起了大量对同一数据的数据库访问，导致对数 据库服务器造成压力。应对策略应该在业务数据分析与预防方面进行，配合运行监控测试与即时调整策略，毕竟单个key的过 期监控难度较高，配合雪崩处理策略即可。

#4、缓存穿透

场景：数据库服务器又崩溃了，跟之前的一样吗？

1.系统平稳运行过程中

2.应用服务器流量随时间增量较大

3.Redis服务器命中率随时间逐步降低

4.Redis内存平稳，内存无压力

5.Redis服务器CPU占用激增

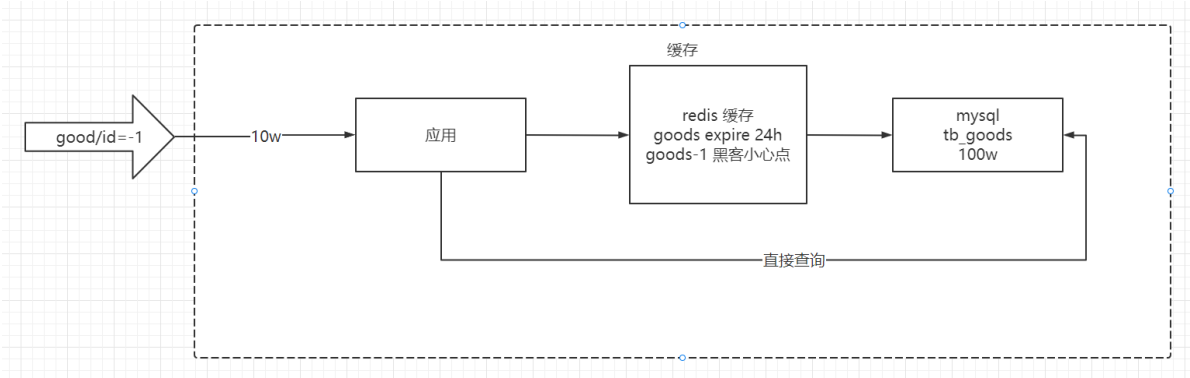
6.数据库服务器压力激增

7.数据库崩溃

问题排查：

1.Redis中大面积出现未命中

2.出现非正常URL访问



问题分析：

- 获取的数据在数据库中也不存在，数据库查询未得到对应数据
- Redis获取到null数据未进行持久化，直接返回
- 下次此类数据到达重复上述过程

- 出现黑客攻击服务器

解决方案：

1.缓存null

对查询结果为null的数据进行缓存（长期使用，定期清理），设定短时限，例如30-60秒，最高5分钟

2.白名单策略

提前预热各种分类数据id对应的bitmaps，id作为bitmaps的offset，相当于设置了数据白名单。当加载正常数据时放行，加载异常数据时直接拦截（效率偏低）

使用布隆过滤器（有关布隆过滤器的命中问题对当前状况可以忽略）

2.实施监控

实时监控redis命中率（业务正常范围时，通常会有一个波动值）与null数据的占比

非活动时段波动：通常检测3-5倍，超过5倍纳入重点排查对象

活动时段波动：通常检测10-50倍，超过50倍纳入重点排查对象

根据倍数不同，启动不同的排查流程。然后使用黑名单进行防控（运营）

4.key加密

问题出现后，临时启动防灾业务key，对key进行业务层传输加密服务，设定校验程序，过来的key校验

例如每天随机分配60个加密串，挑选2到3个，混淆到页面数据id中，发现访问key不满足规则，驳回数据访问。

总的来说：缓存击穿是指访问了不存在的数据，跳过了合法数据的redis数据缓存阶段，每次访问数据库，导致对数据库服务器造成压力。通常此类数据的出现量是一个较低的值，当出现此类情况以毒攻毒，并及时报警。应对策略应该在临时预案防范方面多做文章。

无论是黑名单还是白名单，都是对整体系统的压力，警报解除后尽快移除。

#5、Redis的命名规范

- 使用统一的命名规范
  - 一般使用业务名(或数据库名)为前缀，用冒号分隔，例如，业务名:表名:id。
  - 例如：shop:usr:msg_code（电商:用户:验证码）
- 控制key名称的长度，不要使用过长的key
  - 在保证语义清晰的情况下，尽量减少Key的长度。有些常用单词可使用缩写，例如，user缩写为u，messages缩写为msg。
- 名称中不要包含特殊字符
  - 包含空格、单双引号以及其他转义字符

#6、性能指标监控

redis中的监控指标如下：

- 性能指标：Performance

> 响应请求的平均时间:
>
> ```
> 1   latency
> ```
>
> 平均每秒处理请求总数

```
instantaneous_ops_per_sec
```

缓存查询命中率（通过查询总次数与查询得到非nil数据总次数计算而来）

```
hit_rate(calculated)
```

- 内存指标：Memory

当前内存使用量

```
used_memory
```

内存碎片率（关系到是否进行碎片整理）

```
mem_fragmentation_ratio
```

为避免内存溢出删除的key的总数量

```
evicted_keys
```

基于阻塞操作（BLPOP等）影响的客户端数量

```
blocked_clients
```

- 基本活动指标：Basic_activity

当前客户端连接总数

```
connected_clients
```

当前连接slave总数

```
connected_slaves
```

最后一次主从信息交换距现在的秒

```
master_last_io_seconds_ago
```

key的总数

```
keyspace
```

- 持久性指标：Persistence

当前服务器最后一次RDB持久化的时间

```
rdb_last_save_time
```

当前服务器最后一次RDB持久化后数据变化总量

```
rdb_changes_since_last_save
```

- 错误指标：Error

被拒绝连接的客户端总数（基于达到最大连接值的因素）

```
rejected_connections
```

key未命中的总次数

```
keyspace_misses
```

主从断开的秒数

```
master_link_down_since_seconds
```

要对redis的相关指标进行监控，我们可以采用一些用具：

- CloudInsight Redis
- Prometheus
- Redis-stat
- Redis-faina
- RedisLive
- zabbix

也有一些命令工具：

- benchmark

测试当前服务器的并发性能

```
redis-benchmark [-h ] [-p ] [-c ] [-n <requests]> [-k ]
```

范例1：50个连接，10000次请求对应的性能

```
redis-benchmark
```

范例2：100个连接，5000次请求对应的性能

```
redis-benchmark -c 100 -n 5000
```

序号	选项	描述	默认值
1	-h	指定服务器主机名	127.0.0.1
2	-p	指定服务器端口	6379
3	-s	指定服务器 socket	
4	-c	指定并发连接数	50
5	-n	指定请求数	10000
6	-d	以字节的形式指定 SET/GET 值的数据大小	2
7	-k	1=keep alive 0=reconnect	1
8	-r	SET/GET/INCR 使用随机 key, SADD 使用随机值	
9	-P	通过管道传输 <numreq> 请求	1
10	-q	强制退出 redis。仅显示 query/sec 值	
11	--csv	以 CSV 格式输出	
12	-l	生成循环，永久执行测试	
13	-t	仅运行以逗号分隔的测试命令列表。	
14	-I	Idle 模式。仅打开 N 个 idle 连接并等待。	

- redis-cli

  monitor：启动服务器调试信息

```
monitor
```

```
slowlog：慢日志
```

获取慢查询日志

```
slowlog [operator]
```

get： 获取慢查询日志信息

len： 获取慢查询日志条目数

reset： 重置慢查询日志

相关配置

```
slowlog-log-slower-than 1000 #设置慢查询的时间下线，单位：微妙
slowlog-max-len 100 #设置慢查询命令对应的日志显示长度，单位：命令数
```

#第八章 常见问题

make: cc：命令未找到，make: *** [adlist.o] 错误 127

解决方法：安装gcc，命令如下：

```
yum install gcc
```

> Redis编译错误Killing still running Redis server 4966

```
Killing still  running Redis server 4966   Killing still running Redis server 4971   Killing still running Redis server 4976   Killing still running Redis server 4978   Killing still running Redis server 4980   Killing still running Redis server 4983   Killing still running Redis server 4990   Killing still running Redis server 4991   Killing still running Redis server 4998   Killing still running Redis server 5001   Killing still running Redis server 5014   Killing still running Redis server 5134   Killing still running Redis server 5187   Killing still running Redis server 5208   Killing still running Redis server 5224   Killing still running Redis server 5253   Killing still running Redis server 5265   make[1]: *** [test] Error 1   make[1]: Leaving directory `/opt/redis/redis-6.2.6/src'   make: *** [test] Error 2
```

解决方案：

```
vim  tests/integration/replication-2.tc
```

将after 1000修改为after 10000

after是tcl脚本中的命令，表示延迟程序执行或者在后台执行命令。

> MISCONF Redis is configured to save RDB snapshots

异常信息：

```
redis.clients.jedis.exceptions.JedisDataException:  MISCONF Redis is configured to save RDB snapshots, but is currently not able  to persist on disk. Commands that may modify the data set are disabled.  Please check Redis logs for details about the error.
```

解决方案：

因为强制关闭Redis快照导致不能持久化。可以使用kill -9再次强制关闭掉redis，然后在重新启动。

> Connection error: Connection refused

修改配置文件/etc/redis.conf，并注释掉bind 127.0.0.1这一行

> Connection: Connection error: The remote host closed the connection

修改配置文件/etc/redis.conf，将protected-mode为no。

> Node 192.168.200.131:7002 is not empty. Either the node already knows other nodes (check with CLUSTER NODES) or contains some key in database 0.

1. 先将redis 进程干掉 ps -ef | grep redis kill pid
2. 将每个节点下aof、rdb、nodes.conf本地备份文件删除,redis.conf appendfilename；
3. 之后再执行脚本，成功执行；