

狂神聊Redis

学习方式：不是为了面试和工作学习！仅仅是为了兴趣！兴趣才是最好的老师！

- 基本的理论先学习，然后将知识融汇贯通！

狂神的Redis课程安排：

- nosql 讲解
- 阿里巴巴架构演进
- nosql 数据模型
- Nosql 四大分类
- CAP
- BASE
- Redis 入门
- Redis安装 (Window & Linux服务器)
- 五大基本数据类型
 - String
 - List
 - Set
 - Hash
 - Zset
- 三种特殊数据类型
 - geo
 - hyperloglog
 - bitmap
- Redis 配置详解
- Redis 持久化
 - RDB
 - AOF
- Redis 事务操作
- Redis 实现订阅发布
- Redis 主从复制
- Redis 哨兵模式（现在公司中所有的集群都用哨兵模式）
- 缓存穿透及解决方案
- 缓存击穿及解决方案
- 缓存雪崩及解决方案
- 基础API 之 Jedis 详解
- SpringBoot 集成 Redis 操作
- Redis 的实践分析

Nosql概述

为什么要用Nosql

1、单机MySQL的年代！



90年代，一个基本的网站访问量一般不会太大，单个数据库完全足够！

那个时候，更多的去使用静态网页 Html ~ 服务器根本没有太大的压力！

思考一下，这种情况下：整个网站的瓶颈是什么？

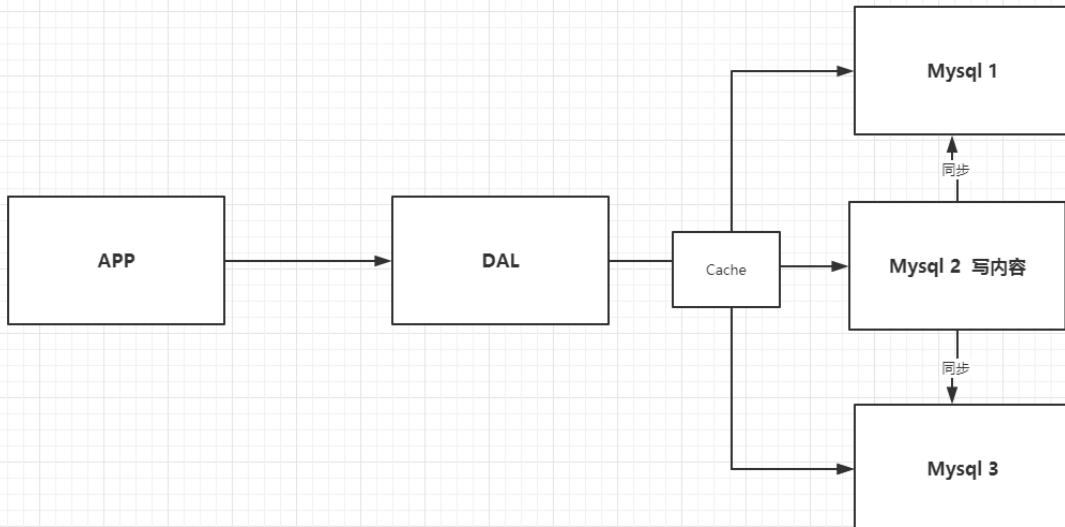
- 1、数据量如果太大、一个机器放不下了！
- 2、数据的索引（B+ Tree），一个机器内存也放不下
- 3、访问量（读写混合），一个服务器承受不了~

只要你开始出现以上的三种情况之一，那么你就必须要晋级！

2、Memcached（缓存）+ MySQL + 垂直拆分（读写分离）

网站80%的情况都是在读，每次都要去查询数据库的话就十分的麻烦！所以说我们希望减轻数据的压力，我们可以使用缓存来保证效率！

发展过程：优化数据结构和索引--> 文件缓存（IO）--> Memcached（当时最热门的技术！）



3、分库分表 + 水平拆分 + MySQL集群

技术和业务在发展的同时，对人的要求也越来越高！

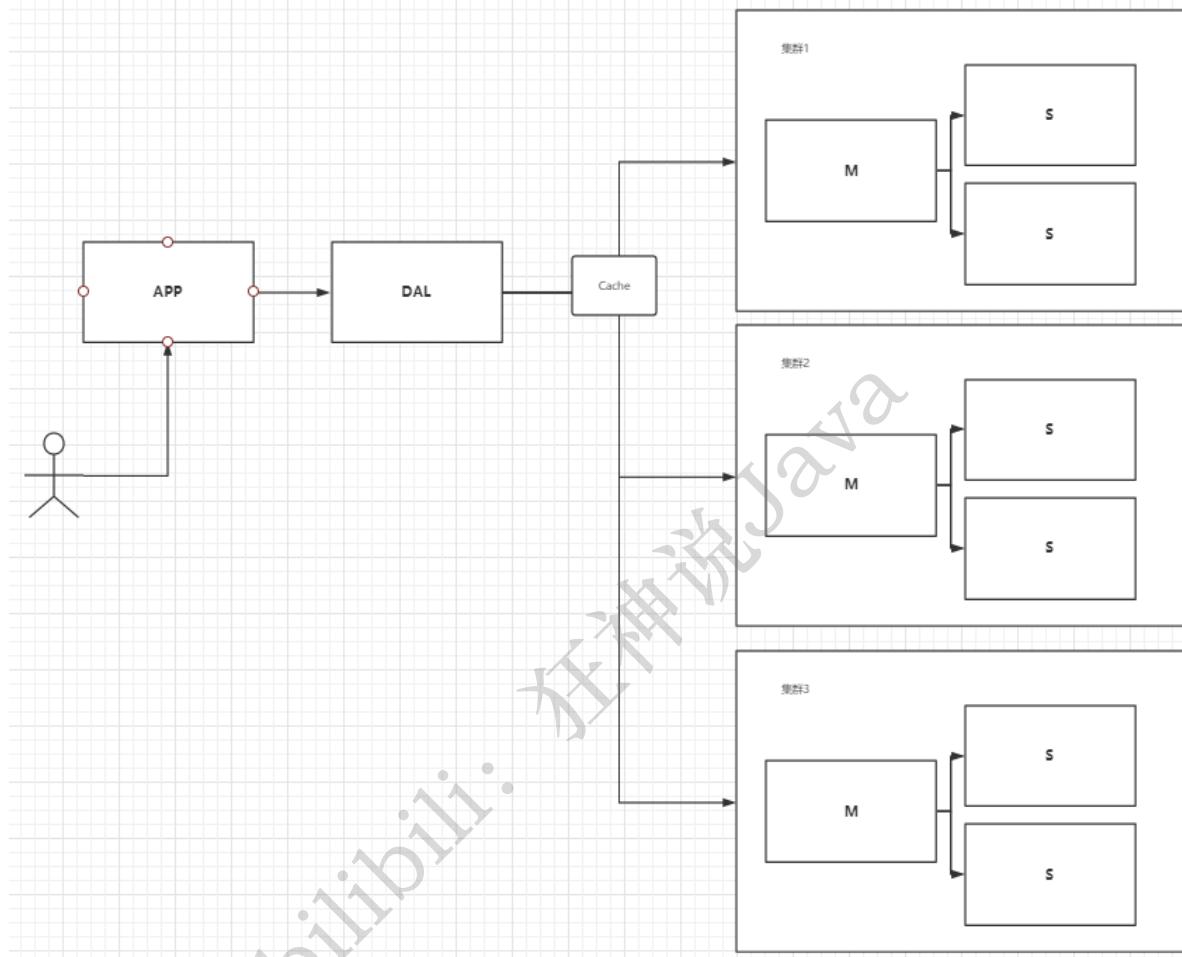
本质：数据库（读，写）

早些年MyISAM：表锁，十分影响效率！高并发下就会出现严重的锁问题

转战Innodb：行锁

慢慢的就开始使用分库分表来解决写的压力！MySQL 在哪个年代推出了表分区！这个并没有多少公司使用！

MySQL 的集群，很好满足哪个年代的所有需求！



4、如今最近的年代

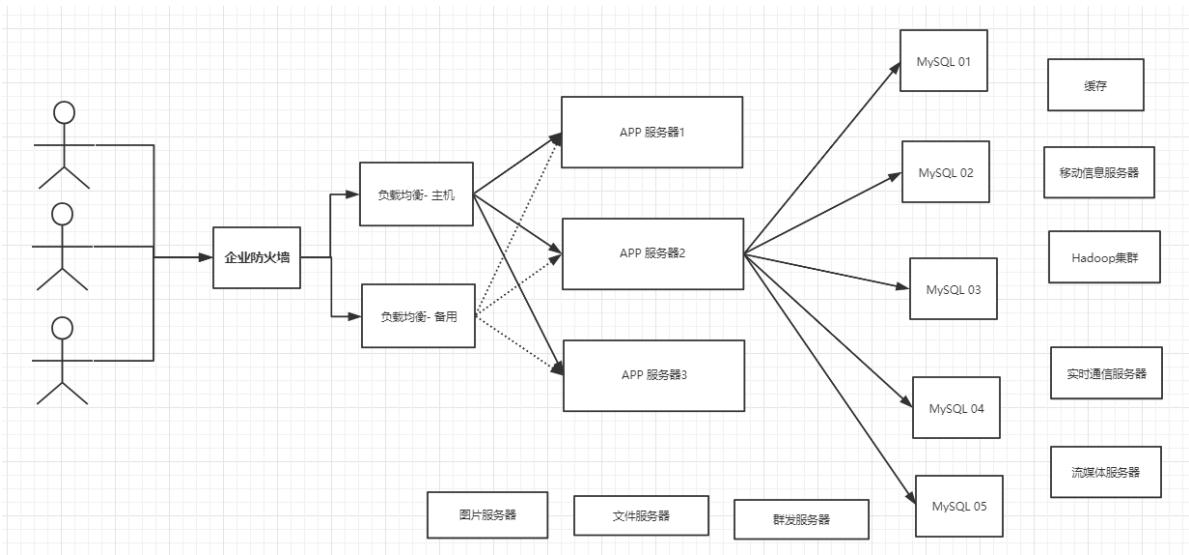
2010--2020 十年之间，世界已经发生了翻天覆地的变化；（定位，也是一种数据，音乐，热榜！）

MySQL 等关系型数据库就不够用了！数据量很多，变化很快~！

MySQL 有的使用它来存储一些比较大的文件，博客，图片！数据库表很大，效率就低了！如果有一种数据库来专门处理这种数据，

MySQL压力就变得十分小（研究如何处理这些问题！）大数据的IO压力下，表几乎没法更大！

目前一个基本的互联网项目！



为什么要用NoSQL！

用户的个人信息，社交网络，地理位置。用户自己产生的数据，用户日志等等爆发式增长！

这时候我们就需要使用NoSQL数据库的，Nosql 可以很好的处理以上的情况！

什么是NoSQL

NoSQL

NoSQL = Not Only SQL (不仅仅是SQL)

关系型数据库：表格，行，列

泛指非关系型数据库的，随着web2.0互联网的诞生！传统的关系型数据库很难对付web2.0时代！尤其是超大规模的高并发的社区！暴露出来很多难以克服的问题，NoSQL在当今大数据环境下发展的十分迅速，Redis是发展最快的，而且是我们当下必须要掌握的一个技术！

很多的数据类型用户的个人信息，社交网络，地理位置。这些数据类型的存储不需要一个固定的格式！不需要多月的操作就可以横向扩展的！ Map<String, Object> 使用键值对来控制！

NoSQL 特点

解耦！

- 1、方便扩展（数据之间没有关系，很好扩展！）
- 2、大数据量高性能（Redis一秒写8万次，读取11万，NoSQL的缓存记录级，是一种细粒度的缓存，性能会比较高！）
- 3、数据类型是多样型的！（不需要事先设计数据库！随取随用！如果是数据量十分大的表，很多人就无法设计了！）
- 4、传统 RDBMS 和 NoSQL

传统的 RDBMS

- 结构化组织
- SQL
- 数据和关系都存在单独的表中 row col
- 操作操作，数据定义语言
- 严格的一致性
- 基础的事务
-

Nosql

- 不仅仅是数据
- 没有固定的查询语言
- 键值对存储，列存储，文档存储，图形数据库（社交关系）
- 最终一致性，
- CAP定理和BASE （异地多活） 初级架构师！（狂神理念：只要学不死，就往死里学！）
- 高性能，高可用，高可扩
-

了解：3V+3高

大数据时代的3V：主要是描述问题的

1. 海量Volume
2. 多样Variety
3. 实时Velocity

大数据时代的3高：主要是对程序的要求

1. 高并发
2. 高可扩
3. 高性能

真正在公司中的实践：NoSQL + RDBMS 一起使用才是最强的，阿里巴巴的架构演进！

技术没有高低之分，就看你如何去使用！（提升内功，思维的提高！）

阿里巴巴演进分析

思考问题：这么多东西难道都是在一个数据库中的吗？



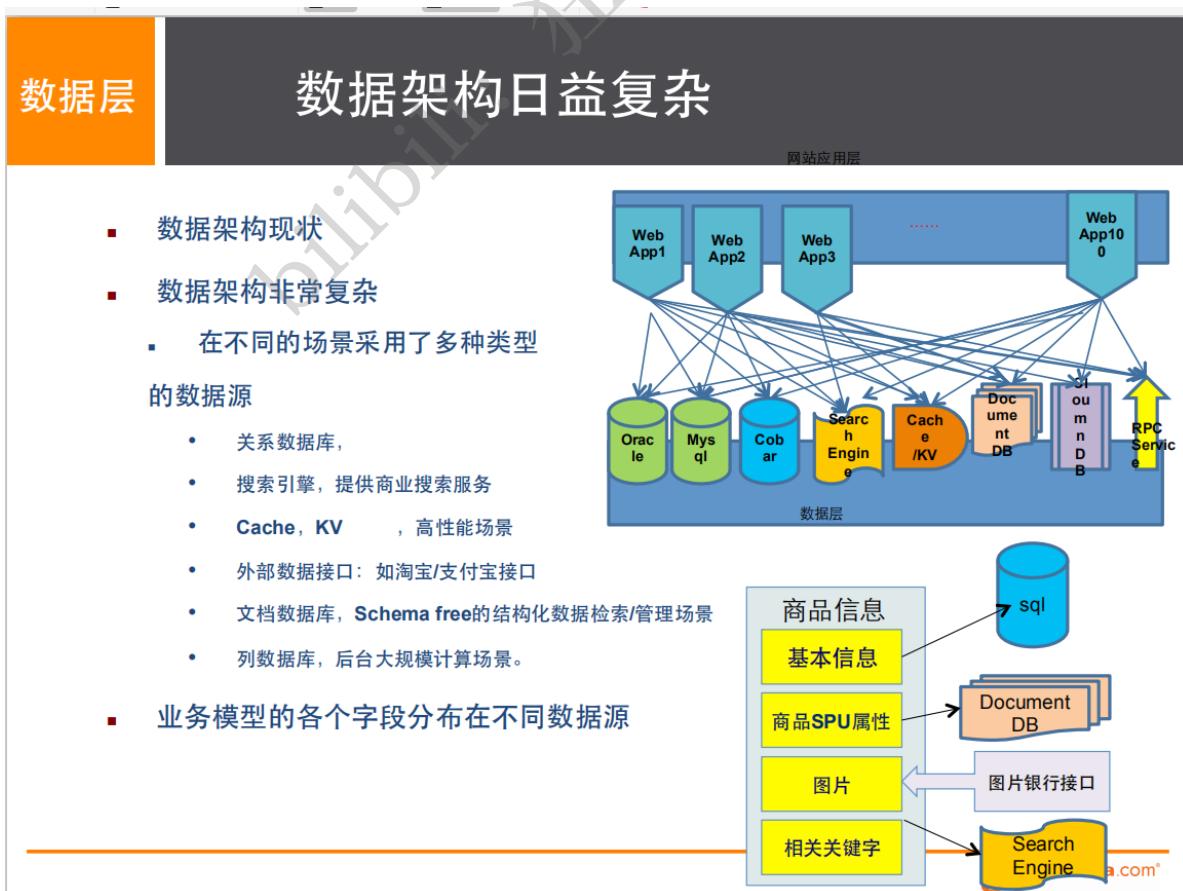
技术急不得，越是慢慢学，才能越扎实！

开源才是技术的王道！

任何一家互联网的公司，都不可能只是简简单单让用户能用就好了！

大量公司做的都是相同的业务；（竞品协议）

随着这样的竞争，业务是越来越完善，然后对于开发者的要求也是越来越高！



如果你未来相当一个架构师：没有什么是加一层解决不了的！

1、商品的基本信息

名称、价格、商家信息；

关系型数据库就可以解决了！ MySQL / Oracle （淘宝早年就去IOE了！ - 王坚：推荐文章：阿里云的这群疯子：40分钟重要！）

淘宝内部的 MySQL 不是大家用的 MySQL

2、商品的描述、评论（文字比较多）

文档型数据库中，MongoDB

3、图片

分布式文件系统 FastDFS

- 淘宝自己的 TFS
- Google的 GFS
- Hadoop HDFS
- 阿里云的 OSS

4、商品的关键字（搜索）

- 搜索引擎 solr elasticsearch

- ISerach: 多隆（多去了解一下这些技术大佬！）

所有牛逼的人都有一段苦逼的岁月！但是你只要像SB一样的去坚持，终将牛逼！

5、商品热门的波段信息

- 内存数据库
- Redis Tair、Memache...

6、商品的交易，外部的支付接口

- 三方应用

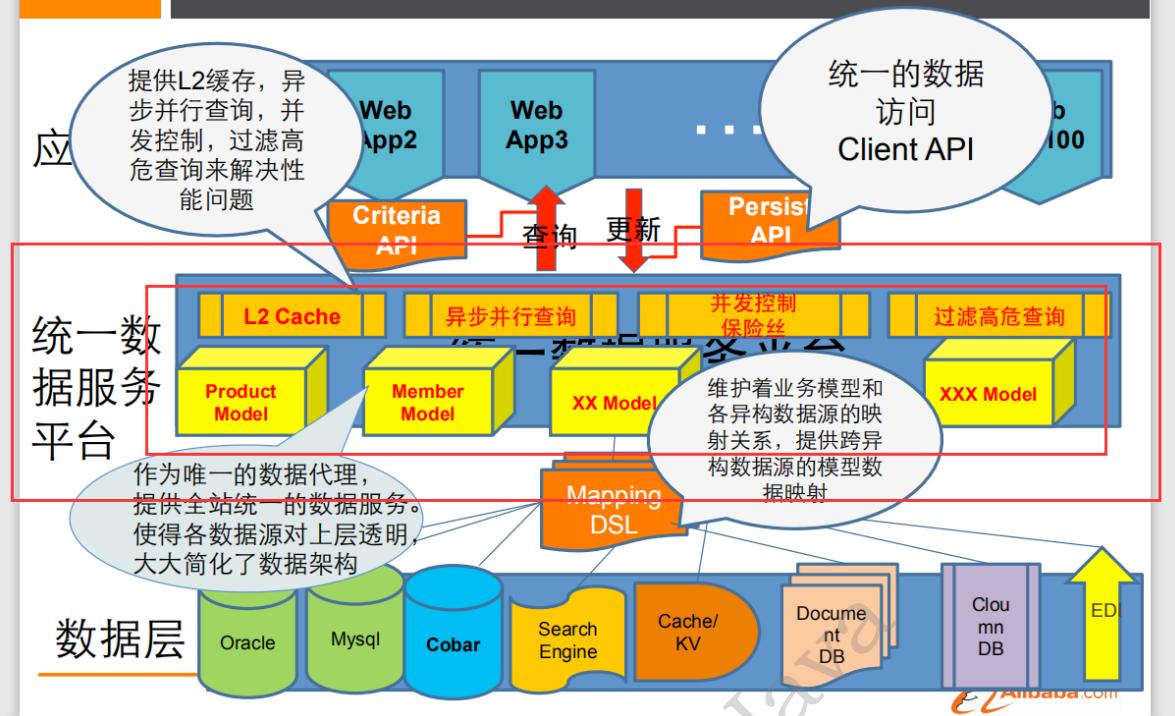
要知道，一个简单地网页背后的技术一定不是大家所想的那么简单！

大型互联网应用问题：

- 数据类型太多了！
- 数据源繁多，经常重构！
- 数据要改造，大面积改造？

解决问题：

数据层 解决方案 : UDSL (统一数据服务平台)



数据层 UDSL: 热点缓存设计



以上都是 NoSQL 入门概述，不仅能够提高大家的知识，还可以帮助大家了解大厂的工作内容！

NoSQL 的四大分类

KV 键值对：

- 新浪 : Redis
- 美团 : Redis + Tair

- 阿里、百度：Redis + memecache

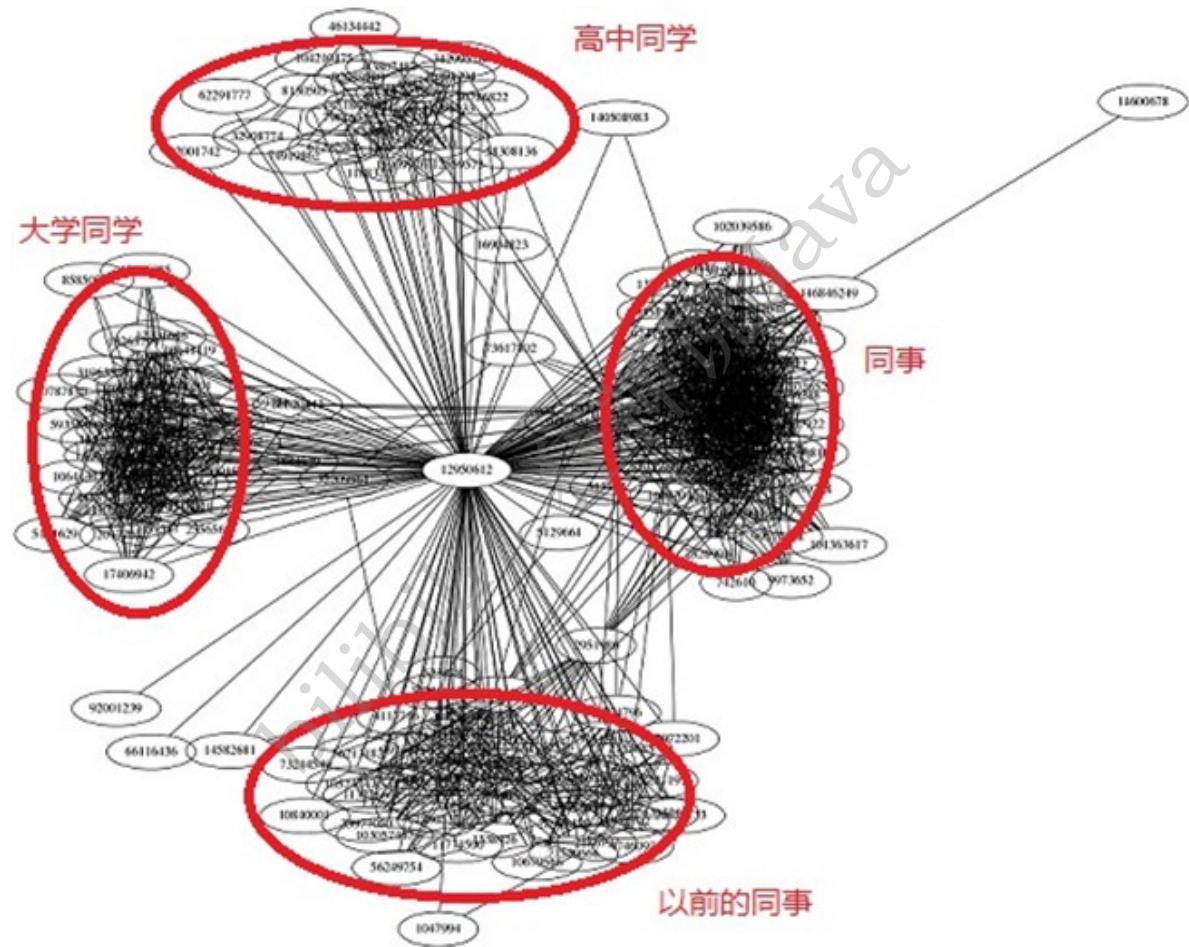
文档型数据库 (bson格式 和json一样) :

- **MongoDB** (一般必须要掌握)
 - MongoDB 是一个基于分布式文件存储的数据库，C++ 编写，主要用来处理大量的文档！
 - MongoDB 是一个介于关系型数据库和非关系型数据中中间的产品！MongoDB 是非关系型数据库中功能最丰富，最像关系型数据库的！
- ComtDB

列存储数据库

- **HBase**
- 分布式文件系统

图关系数据库



- 他不是存图形，放的是关系，比如：朋友圈社交网络，广告推荐！
- **Neo4j** , InfoGrid ;

四者对比！

分类	Examples举例	典型应用场景	数据模型	优点	缺点
键值 (key-value) [3]	Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB	内容缓存，主要用于处理大量数据的高访问负载，也用于一些日志系统等等。[3]	Key 指向 Value 的键值对，通常用 hash table 来实现 [3]	查找速度快	数据无结构化，通常只被当作字符串或者二进制数据 [3]
列存储数据库 [3]	Cassandra, HBase, Riak	分布式的文件系统 +	以列簇式存储，将同一列数据存在一起	查找速度快，可扩展性强，更容易进行分布式扩展	功能相对局限
文档型数据库 [3]	CouchDB, MongoDB	Web 应用（与 Key-Value 类似，Value 是结构化的，不同的是数据库能够了解 Value 的内容）	Key-Value 对应的键值对，Value 为结构化数据	数据结构要求不严格，表结构可变，不需要像关系型数据库一样需要预先定义表结构	查询性能不高，而且缺乏统一的查询语法。
图形(Graph)数据库 [3]	Neo4J, InfoGrid, Infinite Graph	社交网络，推荐系统等。专注于构建关系图谱	图结构	利用图结构相关算法。比如最短路径寻址，N 度关系查找等	很多时候需要对整个图做计算才能得出需要的信息，而且这种结构不太好做分布式集群方案。[3]

敬畏之心可以使人进步！宇宙！科幻！

活着的意义？追求幸福（帮助他人，感恩之心），探索未知（努力的学习，不要这个社会抛弃）

Redis入门

概述

Redis 是什么？

Redis (Remote Dictionary Server)，即远程字典服务！

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

许多语言都包含 Redis 支持，包括：[1]

• ActionScript	• Common Lisp	• Haxe	• Objective-C	• R
• C	• Dart	• Io	• Perl	• Ruby
• C++	• Erlang	• Java	• PHP	• Scala
• C#	• Go	• Node.js	• Pure Data	• Smalltalk
• Clojure	• Haskell	• Lua	• Python	• Tcl

redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave(主从) 同步。

免费和开源！是当下最热门的 NoSQL 技术之一！也被人们称之为结构化数据库！

Redis 能干嘛？

- 1、内存存储、持久化，内存中是断电即失、所以说持久化很重要 (rdb、 aof)
- 2、效率高，可以用于高速缓存
- 3、发布订阅系统
- 4、地图信息分析
- 5、计时器、计数器 (浏览量！)
- 6、

特性

- 1、多样的数据类型
- 2、持久化
- 3、集群
- 4、事务

.....

学习中需要用到的东西

- 1、狂神的公众号 : 狂神说
- 2、官网 : <https://redis.io/>

Redis 介绍

Redis 是一个开源 (BSD 许可) 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)。

你可以对这些类型执行 原子操作，例如：字符串 (strings) 的append命令；散列 (hashes) 的incrby命令；列表 (lists) 的push命令；集合 (sets) 计算交集sinter命令，计算并集union命令 和 计算差集sdiff命令；或者 在有序集合 (sorted sets) 里面获取成员的最高排名zrangebyscore命令。

为了实现其卓越的性能，Redis 采用运行在 内存中的数据集工作方式。根据您的使用情况，您可以每隔一定时间将 数据集导出到磁盘，或者 追加到命令日志中。您也可以关闭持久化功能，将Redis作为一个高效的网络的缓存数据功能使用。

Redis 同样支持 主从复制 (能自动重连和网络断开时自动重新同步)，并且第一次同步是快速的非阻塞式的同步。

其他功能包括：

- 事务 (Transactions)

3、中文网：<http://www.redis.cn/>

4、下载地址：通过官网下载即可！



Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster. [Learn more →](#)

Try it

Ready for a test drive? Check this [interactive tutorial](#) that will walk you through the most important features of Redis.

Download it

[Redis 5.0.8 is the latest stable version.](#)
Interested in release candidates or unstable versions? Check the [downloads page](#).

Quick links

Follow day-to-day Redis on [Twitter](#) and [GitHub](#). Get help or help others by subscribing to [our mailing list](#), we are 5,000 and counting!

注意：Wdinow在Github上下载（停更很久了！）

Redis推荐都是在Linux服务器上搭建的，我们是基于Linux学习！

Windows安装

1、下载安装包：<https://github.com/dmajkic/redis/releases>

2、下载完毕得到压缩包：

名称	修改日期	类型	大小
jedis-3.2.0.jar	2020/1/19 星期...	Executable Jar File	641 KB
redis-5.0.7.tar.gz	2019/12/11 星期...	WinRAR 压缩文件	1,938 KB
Redis-x64-3.2.100.zip	2019/12/11 星期...	WinRAR ZIP 压缩...	5,102 KB
阿里巴巴中文站架构设计实践(何凌).pdf	2014/10/29 星期...	WPS PDF 文档	2,529 KB

3、解压到自己电脑上的环境目录下的就可以的！Redis 十分的小，只有5M

名称	修改日期	类型	大小
EventLog.dll	2016/7/1 星期五 ...	应用程序扩展	1 KB
Redis on Windows Release Notes.doc...	2016/7/1 星期五 ...	DOCX 文档	13 KB
Redis on Windows.docx	2016/7/1 星期五 ...	DOCX 文档	17 KB
redis.windows.conf	2016/7/1 星期五 ...	CONF 文件	48 KB
redis.windows-service.conf	2016/7/1 星期五 ...	CONF 文件	48 KB
redis-benchmark.exe	2016/7/1 星期五 ...	应用程序	400 KB
redis-benchmark.pdb	2016/7/1 星期五 ...	PDB 文件	4,268 KB
redis-check-aof.exe	2016/7/1 星期五 ...	应用程序	251 KB
redis-check-aof.pdb	2016/7/1 星期五 ...	PDB 文件	3,436 KB
redis-cli.exe	2016/7/1 星期五 ...	应用程序	488 KB
redis-cli.pdb	2016/7/1 星期五 ...	PDB 文件	4,420 KB
redis-server.exe	2016/7/1 星期五 ...	应用程序	1,628 KB
redis-server.pdb	2016/7/1 星期五 ...	PDB 文件	6,916 KB
Windows Service Documentation.docx	2016/7/1 星期五 ...	DOCX 文档	14 KB

4、开启Redis，双击运行服务即可！

```
D:\Environment\Redis-x64-3.2.100\redis-server.exe
[11748] 26 Mar 22:45:20.885 # Warning: no config file specified, using the default config. In order to specify a config file use D:\Environment\Redis-x64-3.2.100\redis-server.exe /path/to/redis.conf

Redis 3.2.100 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 11748

http://redis.io

[11748] 26 Mar 22:45:20.891 # Server started, Redis version 3.2.100
[11748] 26 Mar 22:45:20.891 * The server is now ready to accept connections on port 6379
```

搜狗拼音输入法 全 :

5、使用redis客户端来连接redis

```
选择D:\Environment\Redis-x64-3.2.100\redis-cli.exe
127.0.0.1:6379> ping ← 测试连接
PONG
127.0.0.1:6379> set name kuangshen ← set基本值 key value
OK
127.0.0.1:6379> get name ← get key 获取值
"kuangshen"
127.0.0.1:6379> -
```

记住一句话，Window下使用确实简单，但是Redis 推荐我们使用Linux去开发使用！

You can use Redis from most programming languages out there.

Redis is written in **ANSI C** and works in most POSIX systems like Linux, *BSD, OS X without external dependencies.

Linux and OS X are the two operating systems where Redis is developed and tested the most, and we **recommend**

using Linux for deploying. Redis may work in Solaris-derived systems like SmartOS, but the support is *best effort*.

There is no official support for Windows builds.

Linux安装

1、下载安装包！ `redis-5.0.8.tar.gz`

2、解压Redis的安装包！ 程序/opt

```
[root@kuangshen opt]# ls
containerd  redis-5.0.8  redis-5.0.8.tar.gz
[root@kuangshen opt]#
```

3、进入解压后的文件，可以看到我们redis的配置文件

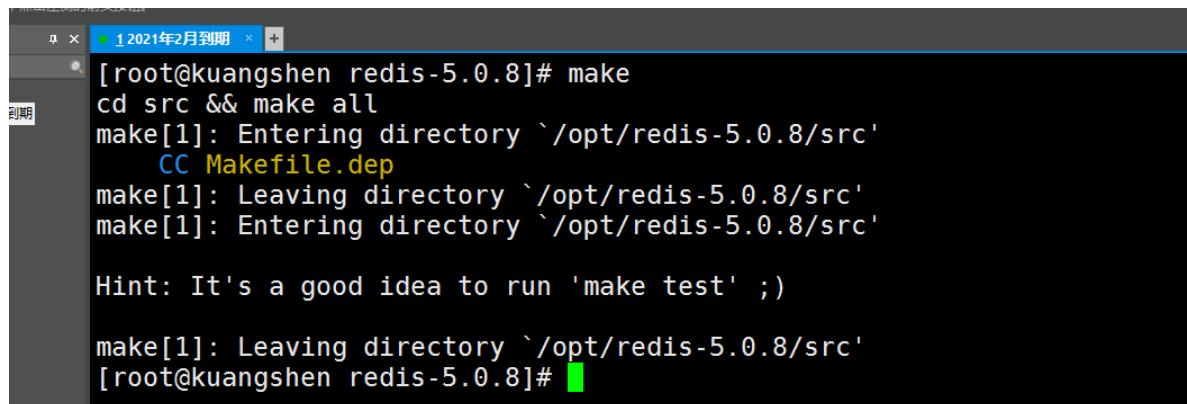
```
[root@kuangshen redis-5.0.8]# ls
00-RELEASENOTES  deps  README.md  runtest-moduleapi  tests
BUGS            INSTALL  redis.conf  runtest-sentinel  utils
CONTRIBUTING    Makefile  runtest
COPYING          MANIFESTO runtest-cluster  sentinel.conf
[root@kuangshen redis-5.0.8]#
```

4、基本的环境安装

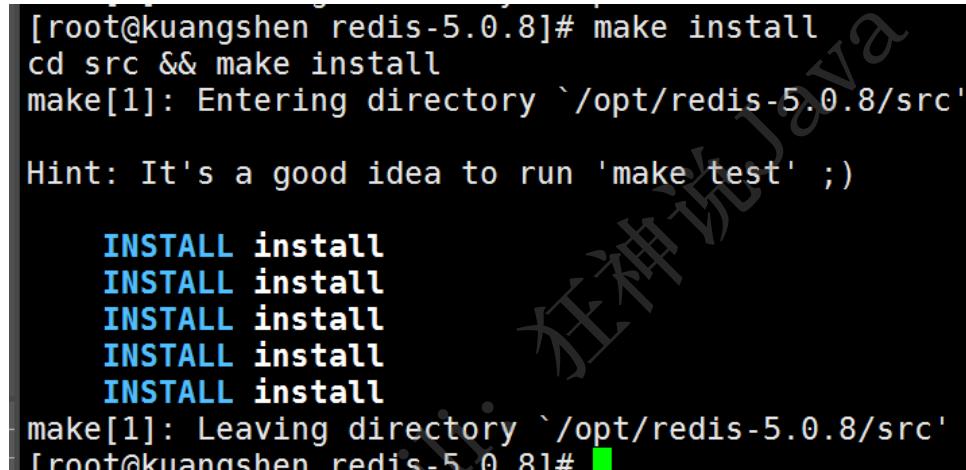
```
yum install gcc-c++
```

```
make
```

```
make install
```



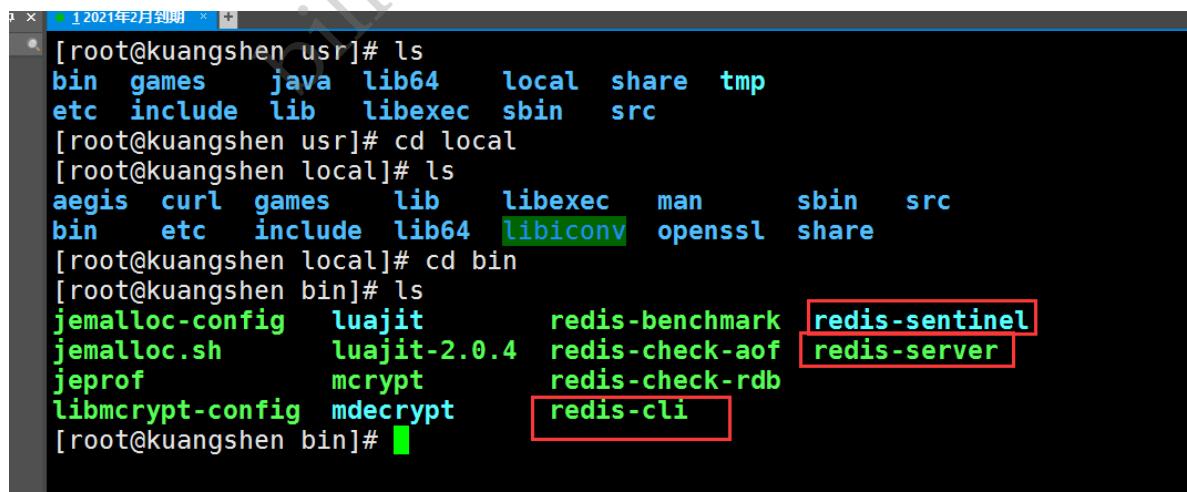
```
[root@kuangshen redis-5.0.8]# make
cd src && make all
make[1]: Entering directory `/opt/redis-5.0.8/src'
  CC Makefile.dep
make[1]: Leaving directory `/opt/redis-5.0.8/src'
make[1]: Entering directory `/opt/redis-5.0.8/src'
Hint: It's a good idea to run 'make test' ;)
make[1]: Leaving directory `/opt/redis-5.0.8/src'
[root@kuangshen redis-5.0.8]#
```



```
[root@kuangshen redis-5.0.8]# make install
cd src && make install
make[1]: Entering directory `/opt/redis-5.0.8/src'
Hint: It's a good idea to run 'make test' ;)

  INSTALL install
  INSTALL install
  INSTALL install
  INSTALL install
  INSTALL install
make[1]: Leaving directory `/opt/redis-5.0.8/src'
[root@kuangshen redis-5.0.8]#
```

5、redis的默认安装路径 /usr/local/bin



```
[root@kuangshen usr]# ls
bin games java lib64 local share tmp
etc include lib libexec sbin src
[root@kuangshen usr]# cd local
[root@kuangshen local]# ls
aegis curl games lib libexec man sbin src
bin etc include lib64 libiconv openssl share
[root@kuangshen local]# cd bin
[root@kuangshen bin]# ls
jemalloc-config luajit redis-benchmark redis-sentinel
jemalloc.sh luajit-2.0.4 redis-check-aof redis-server
jeprof mcrypt redis-check-rdb
libmcrypt-config mdecrypt redis-cli
[root@kuangshen bin]#
```

6、将redis配置文件。复制到我们当前目录下

```
[root@kuangshen local]# cd bin
[root@kuangshen bin]# ls
jemalloc-config  luajit      redis-benchmark  redis-sentinel
jemalloc.sh       luajit-2.0.4  redis-check-aof  redis-server
jeprof          mcrypt      redis-check-rdb
libmcrypt-config mdecrypt    redis-cli
[root@kuangshen bin]# mkdir kconfig
[root@kuangshen bin]# cp /opt/redis-5.0.8/redis.conf kconfig
[root@kuangshen bin]# cd kconfig/
[root@kuangshen kconfig]# ls
redis.conf  我们之后就使用这个配置文件进行启动
[root@kuangshen kconfig]#
```

7、redis默认不是后台启动的，修改配置文件！

```
# 
# A reasonable value for this option is 300 seconds, which is the new
# Redis default starting with Redis 3.2.1.
tcp-keepalive 300

#####
# GENERAL #####
# 

# By default Redis does not run as a daemon. Use 'yes' if you need it.
# Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
daemonize yes 改为yes

# If you run Redis from upstart or systemd, Redis can interact with your
# supervision tree. Options:
# supervised no      - no supervision interaction
# supervised yes     - supervised by upstart or systemd
# supervised systemd - supervised by upstart or systemd via the
#                      Redis init script (https://github.com/antirez/redis/wiki/Upstart-and-Systemd)
```

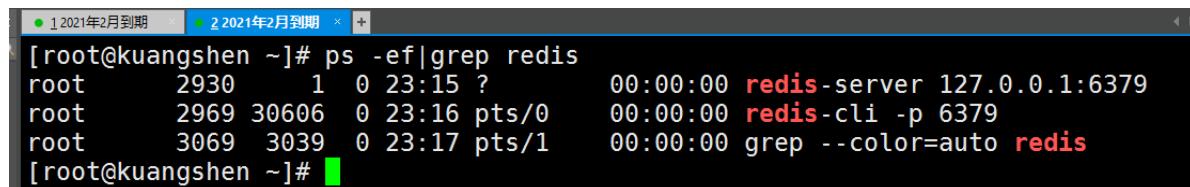
8、启动Redis服务！

```
月到期
所有会话
文件夹
2
• [root@kuangshen bin]# pwd      通过指定的配置文件启动服务
/usr/local/bin
[root@kuangshen bin]# redis-server kconfig/redis.conf
2929:C 26 Mar 2020 23:15:36.825 # o000o000o000o Redis is starting o000o000o000
o
2929:C 26 Mar 2020 23:15:36.825 # Redis version=5.0.8, bits=64, commit=0000000
0, modified=0, pid=2929, just started
2929:C 26 Mar 2020 23:15:36.825 # Configuration loaded
[root@kuangshen bin]#
```

9、使用redis-cli 进行连接测试！

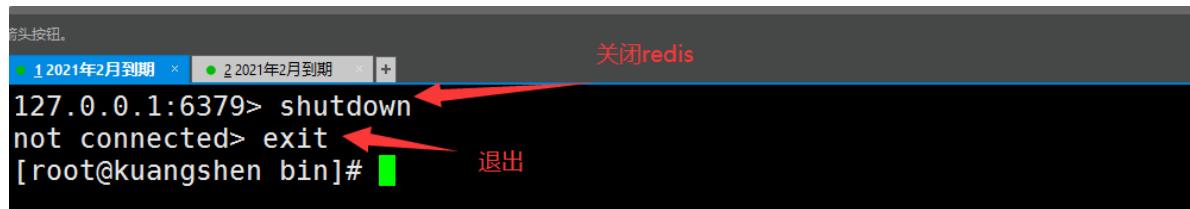
```
[root@kuangshen bin]# ls
jemalloc-config  libmcrypt-config  mdecrypt      redis-cli
jemalloc.sh       luajit        redis-benchmark  redis-sentinel
jeprof          luajit-2.0.4   redis-check-aof  redis-server
kconfig          mcrypt        redis-check-rdb
[root@kuangshen bin]# redis-cli -p 6379 使用Redis客户端进行连接
127.0.0.1:6379> ping ←
PONG
127.0.0.1:6379> set name kuangshen ←
OK
127.0.0.1:6379> get nage
(nil)
127.0.0.1:6379> get name
"kuangshen"
127.0.0.1:6379> keys * ←      查看所有的key
1) "name"
127.0.0.1:6379>
```

10、查看redis的进程是否开启！



```
[root@kuangshen ~]# ps -ef|grep redis
root      2930      1  0 23:15 ?        00:00:00 redis-server 127.0.0.1:6379
root      2969  30606  0 23:16 pts/0    00:00:00 redis-cli -p 6379
root      3069  3039  0 23:17 pts/1    00:00:00 grep --color=auto redis
[root@kuangshen ~]#
```

11、如何关闭Redis服务呢？ shutdown

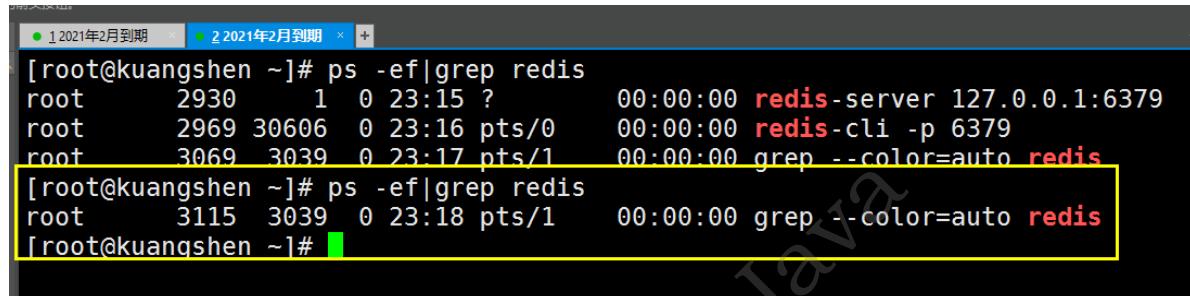


命令按钮。

127.0.0.1:6379> shutdown
not connected> exit

[root@kuangshen bin]# 退出

12、再次查看进程是否存在



```
[root@kuangshen ~]# ps -ef|grep redis
root      2930      1  0 23:15 ?        00:00:00 redis-server 127.0.0.1:6379
root      2969  30606  0 23:16 pts/0    00:00:00 redis-cli -p 6379
root      3069  3039  0 23:17 pts/1    00:00:00 grep --color=auto redis
[root@kuangshen ~]# ps -ef|grep redis
root      3115  3039  0 23:18 pts/1    00:00:00 grep --color=auto redis
[root@kuangshen ~]#
```

13、后面我们会使用单机多Redis启动集群测试！

测试性能

redis-benchmark 是一个压力测试工具！

官方自带的性能测试工具！

redis-benchmark 命令参数！

图片来自菜鸟教程：

redis 性能测试工具可选参数如下所示：

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。	

我们来简单测试下：

```
# 测试：100个并发连接 100000请求  
redis-benchmark -h localhost -p 6379 -c 100 -n 100000
```

```
[root@kuangshen bin]# redis-benchmark -h localhost -p 6379 -c 100 -n 100000  
===== PING_INLINE =====  
100000 requests completed in 1.69 seconds  
100 parallel clients  
3 bytes payload  
keep alive: 1  
  
28.93% <= 1 milliseconds  
99.85% <= 2 milliseconds  
99.98% <= 3 milliseconds  
100.00% <= 3 milliseconds  
59276.82 requests per second  
  
===== PING_BULK =====  
100000 requests completed in 1.66 seconds  
100 parallel clients  
3 bytes payload  
keep alive: 1  
  
32.62% <= 1 milliseconds  
99.89% <= 2 milliseconds  
100.00% <= 2 milliseconds  
60168.47 requests per second
```

如何查看这些分析呢？

```

===== SET =====
100000 requests completed in 1.68 seconds 对我们的10万个请求进行写入测试
100 parallel clients 100个并发客户端
3 bytes payload 每次写入3个字节
keep alive: 1 只有一台服务器来处理这些请求，单机性能

29.03% <= 1 milliseconds
99.90% <= 2 milliseconds
100.00% <= 3 milliseconds ← 所有请求在3毫秒处理完成
59382.42 requests per second ← 每秒处理 59382.42次请求

```

基础的知识

redis默认有16个数据库

```

# syslog-facility local0
# Set the number of databases. The default database is DB 0, you can set
# a different one on a per-connection basis using SELECT <dbid> where
# dbid is a number between 0 and 'databases'-1
databases 16

# By default Redis shows an ASCII art logo only when started to log to
# standard output and if the standard output is a TTY. Basically this means
# that normally a logo is displayed only in interactive sessions

```

默认使用的是第0个

可以使用 select 进行切换数据库！

```

127.0.0.1:6379> select 3 # 切换数据库
OK
127.0.0.1:6379[3]> DBSIZE # 查看DB大小!
(integer) 0

```

```

127.0.0.1:6379[3]> DBSIZE
(integer) 0
127.0.0.1:6379[3]> set name qinjiang
OK
127.0.0.1:6379[3]> DBSIZE
(integer) 1
127.0.0.1:6379[3]> select 7
OK
127.0.0.1:6379[7]> DBSIZE
(integer) 0
127.0.0.1:6379[7]> get name
(nil)
127.0.0.1:6379[7]> select 3
OK
127.0.0.1:6379[3]> get name
"qinjiang"
127.0.0.1:6379[3]>

```

```

127.0.0.1:6379[3]> keys * # 查看数据库所有的key
1) "name"

```

清除当前数据库 **flushdb**

清除全部数据库的内容 **FLUSHALL**

```
127.0.0.1:6379[3]> flushdb
OK
127.0.0.1:6379[3]> keys *
(empty list or set)
```

思考：为什么redis是6379！粉丝效应！（了解一下即可！）

Redis 是单线程的！

明白Redis是很快的，官方表示，Redis是基于内存操作，CPU不是Redis性能瓶颈，Redis的瓶颈是根据机器的内存和网络带宽，既然可以使用单线程来实现，就使用单线程了！所有就使用了单线程了！

Redis 是C 语言写的，官方提供的数据为 100000+ 的QPS，完全不比同样是使用 key-vale 的 Memecache 差！

Redis 为什么单线程还这么快？

- 1、误区1：高性能的服务器一定是多线程的？
- 2、误区2：多线程（CPU上下文会切换！）一定比单线程效率高！

先去CPU>内存>硬盘的速度要有所了解！

核心：redis 是将所有的数据全部放在内存中的，所以说使用单线程去操作效率就是最高的，多线程（CPU上下文会切换：耗时的操作！！！），对于内存系统来说，如果没有上下文切换效率就是最高的！多次读写都是在一个CPU上的，在内存情况下，这个就是最佳的方案！

五大数据类型

官网文档



Commands Clients Documentation Community Download Modules Support

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster. [Learn more →](#)

全段翻译：

Redis 是一个开源（BSD 许可）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MQ。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）。

我们现在讲解的所有命令大家一定要全部记住，后面我们使用SpringBoot。Jedis，所有方法就是这些命令！

单点登录

Redis-Key

```
127.0.0.1:6379> keys *      # 查看所有的key
(empty list or set)
127.0.0.1:6379> set name kuangshen  # set key
OK
127.0.0.1:6379> keys *
1) "name"
127.0.0.1:6379> set age 1
OK
127.0.0.1:6379> keys *
1) "age"
2) "name"
127.0.0.1:6379> EXISTS name  # 判断当前的key是否存在
(integer) 1
127.0.0.1:6379> EXISTS name1
(integer) 0
127.0.0.1:6379> move name 1  # 移除当前的key
(integer) 1
127.0.0.1:6379> keys *
1) "age"
127.0.0.1:6379> set name qinjiang
OK
127.0.0.1:6379> keys *
1) "age"
2) "name"
127.0.0.1:6379> clear
127.0.0.1:6379> keys *
1) "age"
2) "name"
127.0.0.1:6379> get name
"qinjiang"
127.0.0.1:6379> EXPIRE name 10  # 设置key的过期时间，单位是秒
(integer) 1
127.0.0.1:6379> ttl name  # 查看当前key的剩余时间
(integer) 4
127.0.0.1:6379> ttl name
(integer) 3
127.0.0.1:6379> ttl name
(integer) 2
127.0.0.1:6379> ttl name
(integer) 1
127.0.0.1:6379> ttl name
(integer) -2
127.0.0.1:6379> get name
(nil)
127.0.0.1:6379> type name  # 查看当前key的一个类型！
string
127.0.0.1:6379> type age
string
```

后面如果遇到不会的命令，可以在官网查看帮助文档！



The screenshot shows the Redis official website's header with the Redis logo and navigation links: 命令 (Commands), 客户端 (Client), 文档 (Documentation), 社区 (Community), 下载 (Download), 支持 (Support), 许可 (License), 更新日志 (Change Log), 文章大全 (Article Catalog), and 论坛 (Forum). A red arrow points to the '命令' link.

String (字符串)

90% 的 java程序员使用 redis 只会使用一个String类型！

```
#####
127.0.0.1:6379> set key1 v1      # 设置值
OK
127.0.0.1:6379> get key1        # 获得值
"v1"
127.0.0.1:6379> keys *          # 获得所有的key
1) "key1"
127.0.0.1:6379> EXISTS key1    # 判断某一个key是否存在
(integer) 1
127.0.0.1:6379> APPEND key1 "hello"  # 追加字符串, 如果当前key不存在, 就相当于setkey
(integer) 7
127.0.0.1:6379> get key1
"v1hello"
127.0.0.1:6379> STRLEN key1   # 获取字符串的长度!
(integer) 7
127.0.0.1:6379> APPEND key1 ",kaungshen"
(integer) 17
127.0.0.1:6379> STRLEN key1
(integer) 17
127.0.0.1:6379> get key1
"v1hello,kaungshen"
#####
# i++
# 步长 i+=
127.0.0.1:6379> set views 0  # 初始浏览量为0
OK
127.0.0.1:6379> get views
"0"
127.0.0.1:6379> incr views  # 自增1 浏览量变为1
(integer) 1
127.0.0.1:6379> incr views
(integer) 2
127.0.0.1:6379> get views
"2"
127.0.0.1:6379> decr views # 自减1 浏览量-1
(integer) 1
127.0.0.1:6379> decr views
(integer) 0
127.0.0.1:6379> decr views
(integer) -1
127.0.0.1:6379> get views
"-1"
127.0.0.1:6379> INCRBY views 10 # 可以设置步长, 指定增量!
(integer) 9
127.0.0.1:6379> INCRBY views 10
(integer) 19
127.0.0.1:6379> DECRBY views 5
```

```
(integer) 14

#####
# 字符串范围 range
127.0.0.1:6379> set key1 "hello,kuangshen" # 设置 key1 的值
OK
127.0.0.1:6379> get key1
"hello,kuangshen"
127.0.0.1:6379> GETRANGE key1 0 3      # 截取字符串 [0,3]
"hell"
127.0.0.1:6379> GETRANGE key1 0 -1    # 获取全部的字符串 和 get key是一样的
"hello,kuangshen"

# 替换!
127.0.0.1:6379> set key2 abcdefg
OK
127.0.0.1:6379> get key2
"abcdefg"
127.0.0.1:6379> SETRANGE key2 1 xx   # 替换指定位置开始的字符串!
(integer) 7
127.0.0.1:6379> get key2
"axxdefg"
#####
# setex (set with expire)    # 设置过期时间
# setnx (set if not exist)  # 不存在在设置 (在分布式锁中会常常使用!)
127.0.0.1:6379> setex key3 30 "hello" # 设置key3 的值为 hello,30秒后过期
OK
127.0.0.1:6379> ttl key3
(integer) 26
127.0.0.1:6379> get key3
"hello"
127.0.0.1:6379> setnx mykey "redis"   # 如果mykey 不存在, 创建mykey
(integer) 1
127.0.0.1:6379> keys *
1) "key2"
2) "mykey"
3) "key1"
127.0.0.1:6379> ttl key3
(integer) -2
127.0.0.1:6379> setnx mykey "MongoDB"   # 如果mykey存在, 创建失败!
(integer) 0
127.0.0.1:6379> get mykey
"redis"

#####
mset
mget

127.0.0.1:6379> mset k1 v1 k2 v2 k3 v3 # 同时设置多个值
OK
127.0.0.1:6379> keys *
1) "k1"
2) "k2"
3) "k3"
127.0.0.1:6379> mget k1 k2 k3   # 同时获取多个值
1) "v1"
2) "v2"
3) "v3"
```

```

127.0.0.1:6379> msetnx k1 v1 k4 v4 # msetnx 是一个原子性的操作，要么一起成功，要么一起失败！
(integer) 0
127.0.0.1:6379> get k4
(nil)

# 对象
set user:1 {name:zhangsan,age:3} # 设置一个user:1 对象 值为 json字符来保存一个对象！

# 这里的key是一个巧妙的设计： user:{id}:{field} ，如此设计在Redis中是完全OK了！

127.0.0.1:6379> mset user:1:name zhangsan user:1:age 2
OK
127.0.0.1:6379> mget user:1:name user:1:age
1) "zhangsan"
2) "2"

#####
getset # 先get然后在set

127.0.0.1:6379> getset db redis # 如果不存在值，则返回 nil
(nil)
127.0.0.1:6379> get db
"redis"
127.0.0.1:6379> getset db mongodb # 如果存在值，获取原来的值，并设置新的值
"redis"
127.0.0.1:6379> get db
"mongodb"

```

数据结构是相同的！

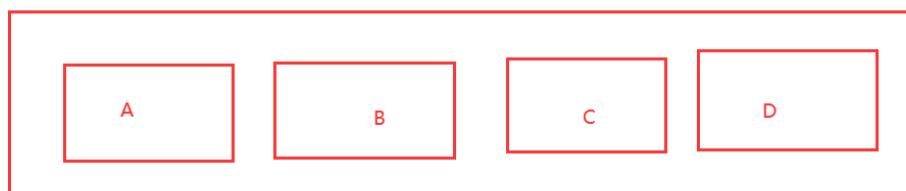
String类似的使用场景：value除了是我们的字符串还可以是我们的数字！

- 计数器
- 统计多单位的数量
- 粉丝数
- 对象缓存存储！

List (列表)

基本的数据类型，列表

List



在redis里面，我们可以把list玩成，栈、队列、阻塞队列！

所有的list命令都是用l开头的，Redis不区分大小命令

```
#####
127.0.0.1:6379> LPUSH list one # 将一个值或者多个值，插入到列表头部（左）
(integer) 1
127.0.0.1:6379> LPUSH list two
(integer) 2
127.0.0.1:6379> LPUSH list three
(integer) 3
127.0.0.1:6379> LRANGE list 0 -1 # 获取list中值！
1) "three"
2) "two"
3) "one"
127.0.0.1:6379> LRANGE list 0 1 # 通过区间获取具体的值！
1) "three"
2) "two"
127.0.0.1:6379> Rpush list righr # 将一个值或者多个值，插入到列表位部（右）
(integer) 4
127.0.0.1:6379> LRANGE list 0 -1
1) "three"
2) "two"
3) "one"
4) "righr"

#####
LPOP
RPOP
127.0.0.1:6379> LRANGE list 0 -1
1) "three"
2) "two"
3) "one"
4) "righr"
127.0.0.1:6379> Lpop list # 移除list的第一个元素
"three"
127.0.0.1:6379> Rpop list # 移除list的最后一个元素
"righr"
127.0.0.1:6379> LRANGE list 0 -1
1) "two"
2) "one"
#####
Lindex

127.0.0.1:6379> LRANGE list 0 -1
1) "two"
2) "one"
127.0.0.1:6379> lindex list 1 # 通过下标获得 list 中的某一个值！
"one"
127.0.0.1:6379> lindex list 0
"two"

#####
Llen

127.0.0.1:6379> Lpush list one
(integer) 1
127.0.0.1:6379> Lpush list two
```

```
(integer) 2
127.0.0.1:6379> Lpush list three
(integer) 3
127.0.0.1:6379> Llen list # 返回列表的长度
(integer) 3

#####
移除指定的值!
取关 uid

Lrem
127.0.0.1:6379> LRANGE list 0 -1
1) "three"
2) "three"
3) "two"
4) "one"
127.0.0.1:6379> lrem list 1 one # 移除list集合中指定个数的value, 精确匹配
(integer) 1
127.0.0.1:6379> LRANGE list 0 -1
1) "three"
2) "three"
3) "two"
127.0.0.1:6379> lrem list 1 three
(integer) 1
127.0.0.1:6379> LRANGE list 0 -1
1) "three"
2) "two"
127.0.0.1:6379> Lpush list three
(integer) 3
127.0.0.1:6379> lrem list 2 three
(integer) 2
127.0.0.1:6379> LRANGE list 0 -1
1) "two"

#####
trim 修剪。; list 截断!

127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> Rpush mylist "hello"
(integer) 1
127.0.0.1:6379> Rpush mylist "hello1"
(integer) 2
127.0.0.1:6379> Rpush mylist "hello2"
(integer) 3
127.0.0.1:6379> Rpush mylist "hello3"
(integer) 4
127.0.0.1:6379> ltrim mylist 1 2 # 通过下标截取指定的长度, 这个list已经被改变了, 截断了
只剩下截取的元素!
OK
127.0.0.1:6379> LRANGE mylist 0 -1
1) "hello1"
2) "hello2"

#####
rpoplpush # 移除列表的最后一个元素, 将他移动到新的列表中!

127.0.0.1:6379> rpush mylist "hello"
```

```

(integer) 1
127.0.0.1:6379> rpush mylist "hello1"
(integer) 2
127.0.0.1:6379> rpush mylist "hello2"
(integer) 3
127.0.0.1:6379> rpoplpush mylist myotherlist # 移除列表的最后一个元素，将他移动到新的
列表中！
"hello2"
127.0.0.1:6379> lrange mylist 0 -1 # 查看原来的列表
1) "hello"
2) "hello1"
127.0.0.1:6379> lrange myotherlist 0 -1 # 查看目标列表中，确实存在改值！
1) "hello2"

#####
lset 将列表中指定下标的值替换为另外一个值，更新操作
127.0.0.1:6379> EXISTS list # 判断这个列表是否存在
(integer) 0
127.0.0.1:6379> lset list 0 item # 如果不存在列表我们去更新就会报错
(error) ERR no such key
127.0.0.1:6379> lpush list value1
(integer) 1
127.0.0.1:6379> LRANGE list 0 0
1) "value1"
127.0.0.1:6379> lset list 0 item # 如果存在，更新当前下标的值
OK
127.0.0.1:6379> LRANGE list 0 0
1) "item"
127.0.0.1:6379> lset list 1 other # 如果不存在，则会报错！
(error) ERR index out of range
#####
linsert # 将某个具体的value插入到列把你中某个元素的前面或者后面！

127.0.0.1:6379> Rpush mylist "hello"
(integer) 1
127.0.0.1:6379> Rpush mylist "world"
(integer) 2
127.0.0.1:6379> LINSERT mylist before "world" "other"
(integer) 3
127.0.0.1:6379> LRANGE mylist 0 -1
1) "hello"
2) "other"
3) "world"
127.0.0.1:6379> LINSERT mylist after world new
(integer) 4
127.0.0.1:6379> LRANGE mylist 0 -1
1) "hello"
2) "other"
3) "world"
4) "new"

```

小结

- 他实际上是一个链表，before Node after , left , right 都可以插入值
- 如果key不存在，创建新的链表
- 如果key存在，新增内容

- 如果移除了所有值，空链表，也代表不存在！
- 在两边插入或者改动值，效率最高！中间元素，相对来说效率会低一点~

消息排队！消息队列（Lpush Rpop），栈（Lpush Lpop）！

Set (集合)

set中的值是不能重读的！

```
#####
127.0.0.1:6379> sadd myset "hello"      # set集合中添加匀速
(integer) 1
127.0.0.1:6379> sadd myset "kuangshen"
(integer) 1
127.0.0.1:6379> sadd myset "lovekuangshen"
(integer) 1
127.0.0.1:6379> SMEMBERS myset      # 查看指定set的所有值
1) "hello"
2) "lovekuangshen"
3) "kuangshen"
127.0.0.1:6379> SISMEMBER myset hello    # 判断某一个值是不是在set集合中！
(integer) 1
127.0.0.1:6379> SISMEMBER myset world
(integer) 0

#####
127.0.0.1:6379> scard myset  # 获取set集合中的内容元素个数！
(integer) 4

#####
rem

127.0.0.1:6379> srem myset hello  # 移除set集合中的指定元素
(integer) 1
127.0.0.1:6379> scard myset
(integer) 3
127.0.0.1:6379> SMEMBERS myset
1) "lovekuangshen2"
2) "lovekuangshen"
3) "kuangshen"

#####
set 无序不重复集合。抽随机！

127.0.0.1:6379> SMEMBERS myset
1) "lovekuangshen2"
2) "lovekuangshen"
3) "kuangshen"
127.0.0.1:6379> SRANDMEMBER myset  # 随机抽出一个元素
"kuangshen"
127.0.0.1:6379> SRANDMEMBER myset
"kuangshen"
127.0.0.1:6379> SRANDMEMBER myset
"kuangshen"
127.0.0.1:6379> SRANDMEMBER myset
"kuangshen"
127.0.0.1:6379> SRANDMEMBER myset 2  # 随机抽出指定个数的元素
```

```
1) "lovekuangshen"
2) "lovekuangshen2"
127.0.0.1:6379> SRANDMEMBER myset 2
1) "lovekuangshen"
2) "lovekuangshen2"
127.0.0.1:6379> SRANDMEMBER myset      # 随机抽选出一个元素
"lovekuangshen2"

#####
删除定的key, 随机删除key!

127.0.0.1:6379> SMEMBERS myset
1) "lovekuangshen2"
2) "lovekuangshen"
3) "kuangshen"
127.0.0.1:6379> spop myset  # 随机删除一些set集合中的元素!
"lovekuangshen2"
127.0.0.1:6379> spop myset
"lovekuangshen"
127.0.0.1:6379> SMEMBERS myset
1) "kuangshen"

#####
将一个指定的值, 移动到另外一个set集合!
127.0.0.1:6379> sadd myset "hello"
(integer) 1
127.0.0.1:6379> sadd myset "world"
(integer) 1
127.0.0.1:6379> sadd myset "kuangshen"
(integer) 1
127.0.0.1:6379> sadd myset2 "set2"
(integer) 1
127.0.0.1:6379> smove myset myset2 "kuangshen" # 将一个指定的值, 移动到另外一个set集合!
(integer) 1
127.0.0.1:6379> SMEMBERS myset
1) "world"
2) "hello"
127.0.0.1:6379> SMEMBERS myset2
1) "kuangshen"
2) "set2"

#####
微博, B站, 共同关注! (并集)
数字集合类:
- 差集 SDIFF
- 交集
- 并集
127.0.0.1:6379> SDIFF key1 key2      # 差集
1) "b"
2) "a"
127.0.0.1:6379> SINTER key1 key2    # 交集    共同好友就可以这样实现
1) "c"
127.0.0.1:6379> SUNION key1 key2    # 并集
1) "b"
2) "c"
3) "e"
4) "a"
```

5) "d"

微博，A用户将所有关注的人放在一个set集合中！将它的粉丝也放在一个集合中！

共同关注，共同爱好，二度好友，推荐好友！（六度分割理论）

Hash (哈希)

Map集合，key-map! 时候这个值是一个map集合！本质和String类型没有太大区别，还是一个简单的key-vlaue！

set myhash field kuangshen

```
#####
127.0.0.1:6379> hset myhash field1 kuangshen # set一个具体 key-vlaue
(integer) 1
127.0.0.1:6379> hget myhash field1 # 获取一个字段值
"kuangshen"
127.0.0.1:6379> hmset myhash field1 hello field2 world # set多个 key-vlaue
OK
127.0.0.1:6379> hmget myhash field1 field2 # 获取多个字段值
1) "hello"
2) "world"
127.0.0.1:6379> hgetall myhash # 获取全部的数据
1) "field1"
2) "hello"
3) "field2"
4) "world"
127.0.0.1:6379> hdel myhash field1 # 删除hash指定key字段！对应的value值也就消失了！
(integer) 1
127.0.0.1:6379> hgetall myhash
1) "field2"
2) "world"
#####
hlen

127.0.0.1:6379> hmset myhash field1 hello field2 world
OK
127.0.0.1:6379> HGETALL myhash
1) "field2"
2) "world"
3) "field1"
4) "hello"
127.0.0.1:6379> hlen myhash # 获取hash表的字段数量！
(integer) 2

#####
127.0.0.1:6379> HEXISTS myhash field1 # 判断hash中指定字段是否存在！
(integer) 1
127.0.0.1:6379> HEXISTS myhash field3
(integer) 0

#####
# 只获得所有field
# 只获得所有value
127.0.0.1:6379> hkeys myhash # 只获得所有field
1) "field2"
2) "field1"
```

```
127.0.0.1:6379> hvals myhash # 只获得所有value
1) "world"
2) "hello"
#####
incr  decr

127.0.0.1:6379> hset myhash field3 5      #指定增量!
(integer) 1
127.0.0.1:6379> HINCRBY myhash field3 1
(integer) 6
127.0.0.1:6379> HINCRBY myhash field3 -1
(integer) 5
127.0.0.1:6379> hsetnx myhash field4 hello # 如果不存在则可以设置
(integer) 1
127.0.0.1:6379> hsetnx myhash field4 world # 如果存在则不能设置
(integer) 0
```

hash变更的数据 user name age,尤其是是用户信息之类的，经常变动的信息！hash 更适合于对象的存储，String更加适合字符串存储！

Zset (有序集合)

在set的基础上，增加了一个值，set k1 v1 zset k1 score1 v1

```
127.0.0.1:6379> zadd myset 1 one      # 添加一个值
(integer) 1
127.0.0.1:6379> zadd myset 2 two 3 three # 添加多个值
(integer) 2
127.0.0.1:6379> ZRANGE myset 0 -1
1) "one"
2) "two"
3) "three"
#####
排序如何实现
```

```
127.0.0.1:6379> zadd salary 2500 xiaohong # 添加三个用户
(integer) 1
127.0.0.1:6379> zadd salary 5000 zhangsan
(integer) 1
127.0.0.1:6379> zadd salary 500 kaungshen
(integer) 1
# ZRANGEBYSCORE key min max
127.0.0.1:6379> ZRANGEBYSCORE salary -inf +inf # 显示全部的用户 从小到大!
1) "kaungshen"
2) "xiaohong"
3) "zhangsan"
127.0.0.1:6379> ZREVRANGE salary 0 -1 # 从大到进行排序!
1) "zhangsan"
2) "kaungshen"
127.0.0.1:6379> ZRANGEBYSCORE salary -inf +inf withscores # 显示全部的用户并且附带成绩
1) "kaungshen"
2) "500"
3) "xiaohong"
```

```

4) "2500"
5) "zhangsan"
6) "5000"
127.0.0.1:6379> ZRANGEBYSCORE salary -inf 2500 withscores # 显示工资小于2500员工的升
序排序!
1) "kaungshen"
2) "500"
3) "xiaohong"
4) "2500"

#####
# 移除rem中的元素

127.0.0.1:6379> zrange salary 0 -1
1) "kaungshen"
2) "xiaohong"
3) "zhangsan"
127.0.0.1:6379> zrem salary xiaohong # 移除有序集合中的指定元素
(integer) 1
127.0.0.1:6379> zrange salary 0 -1
1) "kaungshen"
2) "zhangsan"
127.0.0.1:6379> zcard salary # 获取有序集合中的个数
(integer) 2

#####
127.0.0.1:6379> zadd myset 1 hello
(integer) 1
127.0.0.1:6379> zadd myset 2 world 3 kuangshen
(integer) 2
127.0.0.1:6379> zcount myset 1 3 # 获取指定区间的成员数量!
(integer) 3
127.0.0.1:6379> zcount myset 1 2
(integer) 2

```

其与的一些API，通过我们的学习吗，你们剩下的如果工作中有需要，这个时候你可以去查看官方文档！

案例思路：set 排序 存储班级成绩表，工资表排序！

普通消息，1，重要消息2，带权重进行判断！

排行榜应用实现，取Top N 测试！

三种特殊数据类型

Geospatial 地理位置

朋友的定位，附近的人，打车距离计算？

Redis 的 Geo 在Redis3.2 版本就推出了！这个功能可以推算地理位置的信息，两地之间的距离，方圆几里的人！

可以查询一些测试数据：<http://www.jsons.cn/lngcodeinfo/0706D99C19A781A3/>

只有六个命令：

元素数量。

key中。这些数据将
IUS或者
。这些坐标的限制是
II。具体的限制，由

相关命令

- [GEOADD](#)
- [GEODIST](#)
- [GEOHASH](#)
- [GEOPOS](#)
- [GEORADIUS](#)
- [GEORADIUSBYMEMBER](#)

官方文档：<https://www.redis.net.cn/order/3685.html>

getadd

```
# getadd 添加地理位置
# 规则：两级无法直接添加，我们一般会下载城市数据，直接通过java程序一次性导入！
# 有效的经度从-180度到180度。
# 有效的纬度从-85.05112878度到85.05112878度。
# 当坐标位置超出上述指定范围时，该命令将会返回一个错误。
# 127.0.0.1:6379> geoadd china:city 39.90 116.40 beijing
(error) ERR invalid longitude,latitude pair 39.900000,116.400000

# 参数 key 值()
127.0.0.1:6379> geoadd china:city 116.40 39.90 beijing
(integer) 1
127.0.0.1:6379> geoadd china:city 121.47 31.23 shanghai
(integer) 1
127.0.0.1:6379> geoadd china:city 106.50 29.53 chongqi 114.05 22.52 shengzhen
(integer) 2
127.0.0.1:6379> geoadd china:city 120.16 30.24 hangzhou 108.96 34.26 xian
(integer) 2
```

getpos

获得当前定位：一定是一个坐标值！

```
127.0.0.1:6379> GEOPOS china:city beijing # 获取指定的城市的经度和纬度!
1) 1) "116.39999896287918091"
   2) "39.90000009167092543"
127.0.0.1:6379> GEOPOS china:city beijing chongqi
1) 1) "116.39999896287918091"
   2) "39.90000009167092543"
2) 1) "106.49999767541885376"
   2) "29.52999957900659211"
```

GEODIST

两人之间的距离！

单位：

- **m** 表示单位为米。
- **km** 表示单位为千米。
- **mi** 表示单位为英里。
- **ft** 表示单位为英尺。

```
127.0.0.1:6379> GEODIST china:city beijing shanghai km # 查看上海到北京的直线距离  
"1067.3788"  
127.0.0.1:6379> GEODIST china:city beijing chongqi km # 查看重庆到北京的直线距离  
"1464.0708"
```

georadius 以给定的经纬度为中心，找出某一半径内的元素

我附近的人？（获得所有附近的人的地址，定位！）通过半径来查询！

获得指定数量的人，200

所有数据应该都录入：china:city，才会让结果更加请求！

```
127.0.0.1:6379> GEORADIUS china:city 110 30 1000 km # 以110, 30 这个经纬度为中心，寻找方圆1000km内的城市  
1) "chongqi"  
2) "xian"  
3) "shengzhen"  
4) "hangzhou"  
127.0.0.1:6379> GEORADIUS china:city 110 30 500 km  
1) "chongqi"  
2) "xian"  
127.0.0.1:6379> GEORADIUS china:city 110 30 500 km withdist # 显示到中间距离的位置  
1) 1) "chongqi"  
   2) "341.9374"  
2) 1) "xian"  
   2) "483.8340"  
127.0.0.1:6379> GEORADIUS china:city 110 30 500 km withcoord # 显示他人的定位信息  
1) 1) "chongqi"  
   2) 1) "106.49999767541885376"  
      2) "29.52999957900659211"  
2) 1) "xian"  
   2) 1) "108.96000176668167114"  
      2) "34.25999964418929977"  
127.0.0.1:6379> GEORADIUS china:city 110 30 500 km withdist withcoord count 1 # 筛选出指定的结果！  
1) 1) "chongqi"  
   2) "341.9374"  
   3) 1) "106.49999767541885376"  
      2) "29.52999957900659211"  
127.0.0.1:6379> GEORADIUS china:city 110 30 500 km withdist withcoord count 2  
1) 1) "chongqi"  
   2) "341.9374"  
   3) 1) "106.49999767541885376"  
      2) "29.52999957900659211"  
2) 1) "xian"  
   2) "483.8340"  
   3) 1) "108.96000176668167114"  
      2) "34.25999964418929977"
```

GEORADIUSBYMEMBER

```
# 找出位于指定元素周围的其他元素！  
127.0.0.1:6379> GEORADIUSBYMEMBER china:city beijing 1000 km  
1) "beijing"  
2) "xian"  
127.0.0.1:6379> GEORADIUSBYMEMBER china:city shanghai 400 km  
1) "hangzhou"  
2) "shanghai"
```

GEOHASH 命令 - 返回一个或多个位置元素的 Geohash 表示

该命令将返回11个字符的Geohash字符串!

```
# 将二维的经纬度转换为一维的字符串，如果两个字符串越接近，那么则距离越近！  
127.0.0.1:6379> geohash china:city beijing chongqi  
1) "wx4fbxxfke0"  
2) "wm5xzrybty0"
```

GEO 底层的实现原理其实就是 Zset ! 我们可以使用Zset命令来操作geo !

```
127.0.0.1:6379> ZRANGE china:city 0 -1 # 查看地图中全部的元素  
1) "chongqi"  
2) "xian"  
3) "shengzhen"  
4) "hangzhou"  
5) "shanghai"  
6) "beijing"  
127.0.0.1:6379> zrem china:city beijing # 移除指定元素！  
(integer) 1  
127.0.0.1:6379> ZRANGE china:city 0 -1  
1) "chongqi"  
2) "xian"  
3) "shengzhen"  
4) "hangzhou"  
5) "shanghai"
```

Hyperloglog

什么是基数？

A {1,3,5,7,8,7}

B{1 , 3,5,7,8}

基数（不重复的元素） = 5，可以接受误差！

简介

Redis 2.8.9 版本就更新了 Hyperloglog 数据结构！

Redis Hyperloglog 基数统计的算法！

优点：占用的内存是固定， 2^{64} 不同的元素的技术，只需要废 12KB内存！如果要从内存角度来比较的话 Hyperloglog 首选！

网页的 UV（一个人访问一个网站多次，但是还是算作一个人！）

传统的方式，set 保存用户的id，然后就可以统计 set 中的元素数量作为标准判断！

这种方式如果保存大量的用户id，就会比较麻烦！我们的目的是为了计数，而不是保存用户id；

0.81% 错误率！统计UV任务，可以忽略不计的！

测试使用

```
127.0.0.1:6379> PFadd mykey a b c d e f g h i j    # 创建第一组元素 mykey
(integer) 1
127.0.0.1:6379> PFCOUNT mykey  # 统计 mykey 元素的基本数量
(integer) 10
127.0.0.1:6379> PFadd mykey2 i j z x c v b n m    # 创建第二组元素 mykey2
(integer) 1
127.0.0.1:6379> PFCOUNT mykey2
(integer) 9
127.0.0.1:6379> PFMERGE mykey3 mykey mykey2  # 合并两组 mykey mykey2 => mykey3 并集
OK
127.0.0.1:6379> PFCOUNT mykey3  # 看并集的数量!
(integer) 15
```

如果允许容错，那么一定可以使用 Hyperloglog！

如果不允许容错，就使用 set 或者自己的数据类型即可！

Bitmap

为什么其他教程都不喜欢讲这些？这些在生活中或者开发中，都有十分多的应用场景，学习了，就是就是多一个思路！

技多不压身！

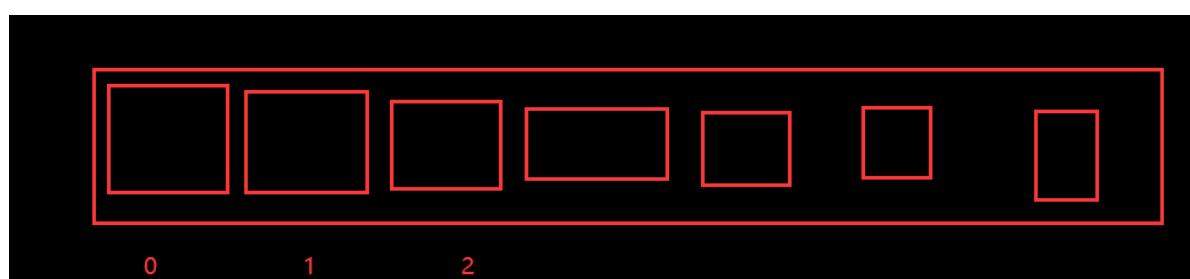
位存储

统计用户信息，活跃，不活跃！登录、未登录！打卡，365打卡！两个状态的，都可以使用 Bitmaps！

Bitmap 位图，数据结构！都是操作二进制位来进行记录，就只有0 和 1 两个状态！

365 天 = 365 bit 1字节 = 8bit 46 个字节左右！

测试



使用bitmap 来记录 周一到周日的打卡 !

周一 : 1 周二 : 0 周三 : 0 周四 : 1

```
127.0.0.1:6379> setbit sign 0 1
(integer) 0
127.0.0.1:6379> setbit sign 1 0
(integer) 0
127.0.0.1:6379> setbit sign 2 0
(integer) 0
127.0.0.1:6379> setbit sign 3 1
(integer) 0
127.0.0.1:6379> setbit sign 4 1
(integer) 0
127.0.0.1:6379> setbit sign 5 0
(integer) 0
127.0.0.1:6379> setbit sign 6 0
(integer) 0
```

查看某一天是否有打卡 !

```
127.0.0.1:6379> getbit sign 3
(integer) 1
127.0.0.1:6379> getbit sign 6
(integer) 0
```

统计操作 , 统计 打卡的天数 !

```
127.0.0.1:6379> bitcount sign # 统计这周的打卡记录 , 就可以看到是否有全勤 !
(integer) 3
```

事务

Redis 事务本质 : 一组命令的集合 ! 一个事务中的所有命令都会被序列化 , 在事务执行过程的中 , 会按照顺序执行 !

一次性、顺序性、排他性 ! 执行一些列的命令 !

----- 队列 set set set 执行 -----

Redis 事务没有隔离级别的概念 !

所有的命令在事务中 , 并没有直接被执行 ! 只有发起执行命令的时候才会执行 ! Exec

Redis 单条命令式保存原子性的 , 但是事务不保证原子性 !

redis 的事务 :

- 开启事务 (multi)
- 命令入队 (.....)
- 执行事务 (exec)

正常执行事务 !

```
127.0.0.1:6379> multi      # 开启事务
OK
# 命令入队
127.0.0.1:6379> set k1 v1
```

```
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> get k2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> exec # 执行事务
1) OK
2) OK
3) "v2"
4) OK
```

放弃事务！

```
127.0.0.1:6379> multi # 开启事务
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k4 v4
QUEUED
127.0.0.1:6379> DISCARD # 取消事务
OK
127.0.0.1:6379> get k4 # 事务队列中命令都不会被执行!
(nil)
```

编译型异常（代码有问题！命令有错！），事务中所有的命令都不会被执行！

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> getset k3 # 错误的命令
(error) ERR wrong number of arguments for 'getset' command
127.0.0.1:6379> set k4 v4
QUEUED
127.0.0.1:6379> set k5 v5
QUEUED
127.0.0.1:6379> exec # 执行事务报错!
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get k5 # 所有的命令都不会被执行!
(nil)
```

运行时异常（1/0），如果事务队列中存在语法性，那么执行命令的时候，其他命令是可以正常执行的，错误命令抛出异常！

```
127.0.0.1:6379> set k1 "v1"
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> incr k1 # 会执行的时候失败!
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> get k3
QUEUED
127.0.0.1:6379> exec
1) (error) ERR value is not an integer or out of range # 虽然第一条命令报错了，但是
依旧正常执行成功了！
2) OK
3) OK
4) "v3"
127.0.0.1:6379> get k2
"v2"
127.0.0.1:6379> get k3
"v3"
```

监控！Watch（面试常问！）

悲观锁：

- 很悲观，认为什么时候都会出问题，无论做什么都会加锁！

乐观锁：

- 很乐观，认为什么时候都不会出问题，所以不会上锁！更新数据的时候去判断一下，在此期间是否有人修改过这个数据，
- 获取version
- 更新的时候比较version

Redis监视测试

正常执行成功！

```
127.0.0.1:6379> set money 100
OK
127.0.0.1:6379> set out 0
OK
127.0.0.1:6379> watch money # 监视 money 对象
OK
127.0.0.1:6379> multi      # 事务正常结束，数据期间没有发生变动，这个时候就正常执行成功！
OK
127.0.0.1:6379> DECRBY money 20
QUEUED
127.0.0.1:6379> INCRBY out 20
QUEUED
127.0.0.1:6379> exec
1) (integer) 80
2) (integer) 20
```

测试多线程修改值，使用watch可以当做redis的乐观锁操作！

```
127.0.0.1:6379> watch money # 监视 money
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> DECRBY money 10
QUEUED
127.0.0.1:6379> INCRBY out 10
QUEUED
127.0.0.1:6379> exec # 执行之前，另外一个线程，修改了我们的值，这个时候，就会导致事务执行失败！
(nil)
```

如果修改失败，获取最新的值就好

```
127.0.0.1:6379> UNWATCH → 1、如果发现事务执行失败，就先解锁
OK
127.0.0.1:6379> WATCH money → 2、获取最新的值，再次监视，select version
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> DECRBY money 1
QUEUED
127.0.0.1:6379> incrBY money 1
QUEUED
127.0.0.1:6379> exec → 3、比对监视的值是否发生了变化，如果没有变化，那么可以执行成功，如果变量就执行失败！
1) (integer) 999
2) (integer) 1000
127.0.0.1:6379>
```

Jedis

我们要使用 Java 来操作 Redis，知其然并知其所以然，授人以渔！学习不能急躁，慢慢来会很快！

什么是Jedis 是 Redis 官方推荐的 java连接开发工具！ 使用Java 操作Redis 中间件！如果你要使用 java操作redis，那么一定要对Jedis 十分的熟悉！

测试

1、导入对应的依赖

```
<!--导入jedis的包-->
<dependencies>
    <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>3.2.0</version>
    </dependency>
    <!--fastjson-->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>fastjson</artifactId>
        <version>1.2.62</version>
    </dependency>
</dependencies>
```

2、编码测试：

- 连接数据库
- 操作命令
- 断开连接！

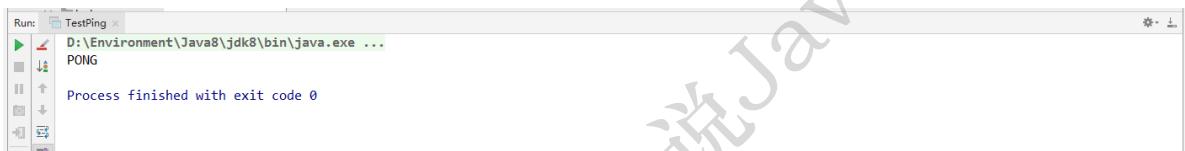
```
package com.kuang;

import redis.clients.jedis.Jedis;

public class TestPing {
    public static void main(String[] args) {
        // 1. new Jedis 对象即可
        Jedis jedis = new Jedis("127.0.0.1", 6379);
        // jedis 所有的命令就是我们之前学习的所有指令！所以之前的指令学习很重要！

        System.out.println(jedis.ping());
    }
}
```

输出：



常用的API

String

List

Set

Hash

Zset

所有的api命令，就是我们对应的上面学习的指令，一个都没有变化！

事务

```
public class TestTX {
    public static void main(String[] args) {
        Jedis jedis = new Jedis("127.0.0.1", 6379);

        jedis.flushDB();

        JSONObject jsonObject = new JSONObject();
        jsonObject.put("hello", "world");
        jsonObject.put("name", "kuangshen");
        // 开启事务
        Transaction multi = jedis.multi();
        String result = jsonObject.toJSONString();
```

```

        // jedis.watch(result)
        try {
            multi.set("user1", result);
            multi.set("user2", result);
            int i = 1/0 ; // 代码抛出异常事务，执行失败!
            multi.exec(); // 执行事务!
        } catch (Exception e) {
            multi.discard(); // 放弃事务
            e.printStackTrace();
        } finally {
            System.out.println(jedis.get("user1"));
            System.out.println(jedis.get("user2"));
            jedis.close(); // 关闭连接
        }
    }
}

```

SpringBoot整合

SpringBoot 操作数据：spring-data jpa jdbc mongodb redis !

SpringData 也是和 SpringBoot 齐名的项目！

说明：在 SpringBoot2.x 之后，原来使用的jedis 被替换为了 lettuce?

jedis：采用的直连，多个线程操作的话，是不安全的，如果想要避免不安全的，使用 jedis pool 连接池！更像 BIO 模式

lettuce：采用netty，实例可以再多个线程中进行共享，不存在线程不安全的情况！可以减少线程数据了，更像 NIO 模式

源码分析：

```

@Bean
@ConditionalOnMissingBean(name = "redisTemplate") // 我们可以自己定义一个
redisTemplate来替换这个默认的!
public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory)
    throws UnknownHostException {
    // 默认的 RedisTemplate 没有过多的设置，redis 对象都是需要序列化!
    // 两个泛型都是 Object, Object 的类型，我们后使用需要强制转换 <String, Object>
    RedisTemplate<Object, Object> template = new RedisTemplate<>();
    template.setConnectionFactory(redisConnectionFactory);
    return template;
}

@Bean
@ConditionalOnMissingBean // 由于 String 是redis中最常使用的类型，所以说单独提出来了一个bean!
public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory
redisConnectionFactory)
    throws UnknownHostException {
    StringRedisTemplate template = new StringRedisTemplate();
    template.setConnectionFactory(redisConnectionFactory);
    return template;
}

```

```
}
```

整合测试一下

1、导入依赖

```
<!-- 操作redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2、配置连接

```
# 配置redis
spring.redis.host=127.0.0.1
spring.redis.port=6379
```

3、测试！

```
@SpringBootTest
class Redis02SpringbootApplicationTests {

    @Autowired
    private RedisTemplate redisTemplate;

    @Test
    void contextLoads() {

        // redisTemplate 操作不同的数据类型，api和我们的指令是一样的
        // opsForValue 操作字符串 类似String
        // opsForList 操作List 类似List
        // opsForSet
        // opsForHash
        // opsForZSet
        // opsForGeo
        // opsForHyperLogLog

        // 除了进本的操作，我们常用的方法都可以直接通过redisTemplate操作，比如事务，和基本的
        // CRUD

        // 获取redis的连接对象
        //     RedisConnection connection =
        redisTemplate.getConnectionFactory().getConnection();
        //     connection.flushDb();
        //     connection.flushAll();

        redisTemplate.opsForValue().set("mykey", "关注狂神说公众号");
        System.out.println(redisTemplate.opsForValue().get("mykey"));

    }

}
```

Navigate Code Analyze Refactor Build Run Tools VCS Window Help

redis-2.2.5.RELEASE-sources.jar org springframework data redis core RedisTemplate Redis02SpringbootApplicationTests.contextLoads Redis02SpringbootApplicationTests.java RedisAutoConfiguration.java RedisTemplate.java LettuceConnectionConfiguration.java application.properties

```

private boolean enableTransactionSupport = false;
private boolean exposeConnection = false;
private boolean initialized = false;
private boolean enableDefaultSerializer = true;
private @Nullable RedisSerializer<?> defaultSerializer;
private @Nullable ClassLoader classLoader;

/rawtypes/ private @Nullable RedisSerializer keySerializer = null;
/rawtypes/ private @Nullable RedisSerializer valueSerializer = null;
/rawtypes/ private @Nullable RedisSerializer hashKeySerializer = null;
/rawtypes/ private @Nullable RedisSerializer hashValueSerializer = null;
private RedisSerializer<String> stringSerializer = RedisSerializer.string();

private @Nullable ScriptExecutor<K> scriptExecutor;

private final ValueOperations<K, V> valueOps = new DefaultValueOperations<>(& template: this);

boolean defaultUsed = false;

if (defaultSerializer == null) 默认的序列化方式是JDK序列化，我们可能会使用Json来序列化！

    defaultSerializer = new JdkSerializationRedisSerializer(
        classLoader != null ? classLoader : this.getClass().getClassLoader
    }

if (enableDefaultSerializer) {

```

关于对象的保存：

```

@java
com
  com
    kuang
      config
      pojo
        User
        Redis02SpringbootApplication
resources
  static
  templates
  application.properties

@test
java
com
  com
    kuang
      Redis02SpringbootApplication

Redis02SpringbootApplicationTests.java
  44
  45
  46
  47
  48
  49
  50
  51
  52
  53
  54

@Test
public void test() throws JsonProcessingException {
    // 真实的开发一般都使用json来传递对象
    User user = new User(name: "狂神说", age: 3);
    String jsonUser = new ObjectMapper().writeValueAsString(user);
    redisTemplate.opsForValue().set("user", user);
    System.out.println(redisTemplate.opsForValue().get("user"));
}

}

Redis02SpringbootApplicationTests.test
  Tests failed: 1 of 1 test - 204 ms
  at com.kuang.Redis02SpringbootApplicationTests.test(Redis02SpringbootApplicationTests.java:49) <31 internal calls>
  at java.util.ArrayList.forEach(ArrayList.java:1257) <9 internal calls>
  at java.util.ArrayList.forEach(ArrayList.java:1257) <21 internal calls>
  Caused by: org.springframework.core.serializer.support.SerializationFailedException: Failed to serialize object using DefaultSerializer
  at org.springframework.core.serializer.support.SerializingConverter.convert(SerializingConverter.java:68)
  at org.springframework.core.serializer.support.SerializingConverter.convert(SerializingConverter.java:35)
  at org.springframework.data.redis.serializer.JdkSerializationRedisSerializer.serialize(JdkSerializationRedisSerializer.java:...
  ... 66 more
  Caused by: java.lang.IllegalArgumentException: DefaultSerializer requires a Serializable payload but received an object of type
  at org.springframework.core.serializer.DefaultSerializer.serialize(DefaultSerializer.java:43)
  at org.springframework.core.serializer.support.SerializingConverter.convert(SerializingConverter.java:63)
  ... 68 more

```

所有的对象，需要序列化

我们来编写一个自己的 RedisTemplate

```

package com.kuang.config;

import com.fasterxml.jackson.annotation.JsonAutoDetect;
import com.fasterxml.jackson.annotation.PropertyAccessor;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
public class RedisConfig {

```

```

// 这是我给大家写好的一个固定模板，大家在企业中，拿去就可以直接使用！
// 自己定义了一个 RedisTemplate
@Bean
@SuppressWarnings("all")
public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
factory) {
    // 我们为了自己开发方便，一般直接使用 <String, Object>
    RedisTemplate<String, Object> template = new RedisTemplate<String,
Object>();
    template.setConnectionFactory(factory);

    // Json序列化配置
    Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new
Jackson2JsonRedisSerializer(Object.class);
    ObjectMapper om = new ObjectMapper();
    om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
    om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
    jackson2JsonRedisSerializer.setObjectMapper(om);
    // String 的序列化
    StringRedisSerializer stringRedisSerializer = new
StringRedisSerializer();

    // key采用String的序列化方式
    template.setKeySerializer(stringRedisSerializer);
    // hash的key也采用String的序列化方式
    template.setHashKeySerializer(stringRedisSerializer);
    // value序列化方式采用jackson
    template.setValueSerializer(jackson2JsonRedisSerializer);
    // hash的value序列化方式采用jackson
    template.setHashValueSerializer(jackson2JsonRedisSerializer);
    template.afterPropertiesSet();

    return template;
}

}

```

所有的redis操作，其实对于java开发人员来说，十分的简单，更重要是要去理解redis的思想和每一种数据结构的用处和应用场景！

Redis.conf详解

启动的时候，就通过配置文件来启动！

工作中，一些小小的配置，可以让你脱颖而出！

行家有没有，出手就知道

单位

```
Redis configuration file example.  
#  
# Note that in order to read the configuration file, Redis must be  
# started with the file path as first argument:  
#  
# ./redis-server /path/to/redis.conf  
  
# Note on units: when memory size is needed, it is possible to specify  
# it in the usual form of 1k 5GB 4M and so forth:  
#  
# 1k => 1000 bytes  
# 1kb => 1024 bytes  
# 1m => 1000000 bytes  
# 1mb => 1024*1024 bytes  
# 1g => 1000000000 bytes  
# 1gb => 1024*1024*1024 bytes  
#  
# units are case insensitive so 1GB 1Gb 1gB are all the same.  
  
##### INCLUDES #####
```

1、配置文件 unit单位 对大小写不敏感！

包含

```
##### INCLUDES #####  
#  
  
# Include one or more other config files here. This is useful if you  
# have a standard template that goes to all Redis servers but also need  
# to customize a few per-server settings. Include files can include  
# other files, so use this wisely.  
#  
# Notice option "include" won't be rewritten by command "CONFIG REWRITE"  
# from admin or Redis Sentinel. Since Redis always uses the last processed  
# line as value of a configuration directive, you'd better put includes  
# at the beginning of this file to avoid overwriting config change at runtime.  
#  
# If instead you are interested in using includes to override configuration  
# options, it is better to use include as the last line.  
#  
# include /path/to/local.conf  
# include /path/to/other.conf
```

就是好比我们学习Spring、Improt，include

网络

```
bind 127.0.0.1      # 绑定的ip  
protected-mode yes # 保护模式  
port 6379    # 端口设置
```

通用 GENERAL

```
daemonize yes      # 以守护进程的方式运行，默认是 no，我们需要自己开启为yes!  
  
pidfile /var/run/redis_6379.pid  # 如果以后台的方式运行，我们就需要指定一个 pid 文件!  
  
# 日志  
# Specify the server verbosity level.  
# This can be one of:
```

```
# debug (a lot of information, useful for development/testing)
# verbose (many rarely useful info, but not a mess like the debug level)
# notice (moderately verbose, what you want in production probably) 生产环境
# warning (only very important / critical messages are logged)
loglevel notice
logfile "" # 日志的文件位置名
databases 16 # 数据库的数量, 默认是 16 个数据库
always-show-logo yes # 是否总是显示LOGO
```

快照

持久化 , 在规定的时间内 , 执行了多少次操作 , 则会持久化到文件 .rdb. aof

redis 是内存数据库 , 如果没有持久化 , 那么数据断电及失 !

```
# 如果900s内, 如果至少有一个1 key进行了修改, 我们及进行持久化操作
save 900 1
# 如果300s内, 如果至少10 key进行了修改, 我们及进行持久化操作
save 300 10
# 如果60s内, 如果至少10000 key进行了修改, 我们及进行持久化操作
save 60 10000
# 我们之后学习持久化, 会自己定义这个测试!

stop-writes-on-bgsave-error yes # 持久化如果出错, 是否还需要继续工作!

rdbcompression yes # 是否压缩 rdb 文件, 需要消耗一些cpu资源!

rdbchecksum yes # 保存rdb文件的时候, 进行错误的检查校验!

dir ./ # rdb 文件保存的目录!
```

REPLICATION 复制 , 我们后面讲解主从复制的 , 时候再进行讲解

SECURITY 安全

可以在这里设置redis的密码 , 默认是没有密码 !

```
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> config get requirepass # 获取redis的密码
1) "requirepass"
2) ""
127.0.0.1:6379> config set requirepass "123456" # 设置redis的密码
OK
127.0.0.1:6379> config get requirepass # 发现所有的命令都没有权限了
(error) NOAUTH Authentication required.
127.0.0.1:6379> ping
(error) NOAUTH Authentication required.
127.0.0.1:6379> auth 123456 # 使用密码进行登录!
OK
127.0.0.1:6379> config get requirepass
1) "requirepass"
2) "123456"
```

限制 CLIENTS

```
maxclients 10000 # 设置能连接上redis的最大客户端的数量  
maxmemory <bytes> # redis 配置最大的内存容量  
maxmemory-policy noevasion # 内存到达上限之后的处理策略  
1、volatile-lru: 只对设置了过期时间的key进行LRU(默认值)  
2、allkeys-lru : 删除lru算法的key  
3、volatile-random: 随机删除即将过期key  
4、allkeys-random: 随机删除  
5、volatile-ttl : 删除即将过期的  
6、noevasion : 永不过期, 返回错误
```

APPEND ONLY 模式 aof配置

```
appendonly no # 默认是不开启aof模式的, 默认是使用rdb方式持久化的, 在大部分所有的情况下,  
rdb完全够用!  
appendfilename "appendonly.aof" # 持久化的文件的名字  
  
# appendfsync always # 每次修改都会 sync。消耗性能  
appendfsync everysec # 每秒执行一次 sync, 可能会丢失这1s的数据!  
# appendfsync no # 不执行 sync, 这个时候操作系统自己同步数据, 速度最快!
```

具体的配置，我们在 Redis持久化 中去给大家详细讲解！

Redis持久化

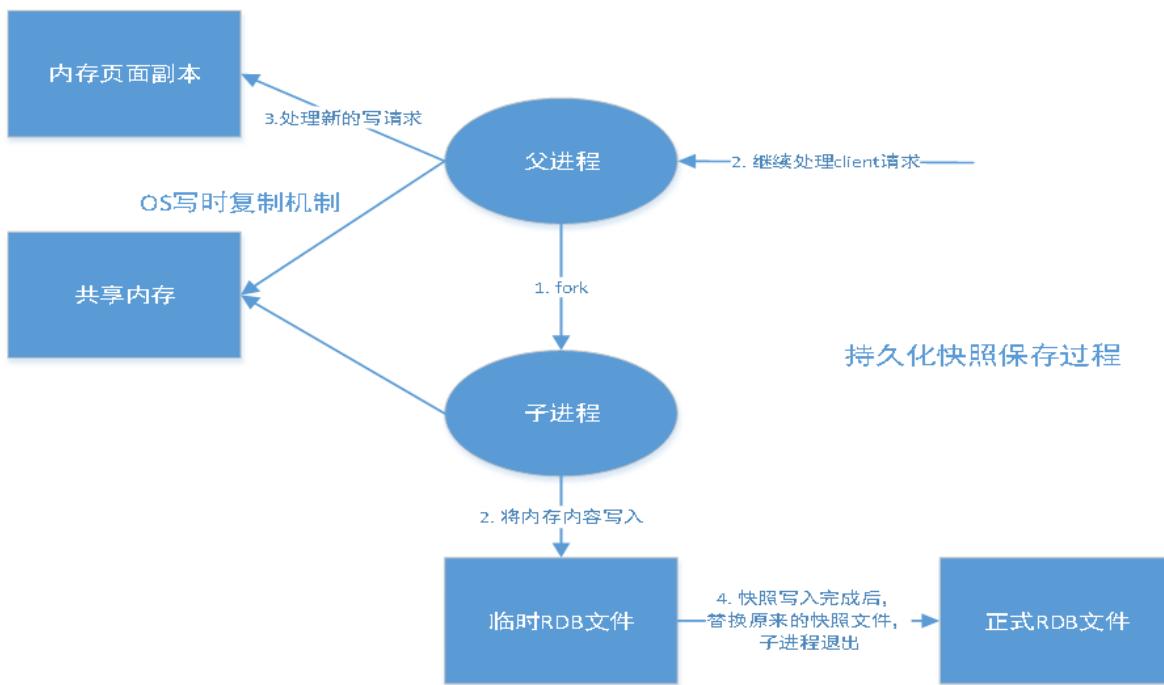
面试和工作，持久化都是重点！

Redis 是内存数据库，如果不将内存中的数据库状态保存到磁盘，那么一旦服务器进程退出，服务器中的数据库状态也会消失。所以 Redis 提供了持久化功能！

RDB (Redis DataBase)

什么是RDB

在主从复制中，rdb就是备用了！从机上面！



在指定的时间间隔内将内存中的数据集快照写入磁盘，也就是行话讲的Snapshot快照，它恢复时是将快照文件直接读到内存里。

Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的。这就确保了极高的性能。如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。RDB的缺点是最后一次持久化后的数据可能丢失。我们默认的就是RDB，一般情况下不需要修改这个配置！

有时候在生产环境我们会将这个文件进行备份！

redis保存的文件是dump.rdb 都是在我们的配置文件中快照中进行配置的！

```
# The filename where to dump the DB
dbfilename dump.rdb

# save 900 1
# save 300 10
# save 60 10000
save 60 5
# By default Redis will stop accepting writes if RDB snapshots ar
```

触发机制

- 1、save的规则满足的情况下，会自动触发rdb规则
- 2、执行 flushall 命令，也会触发我们的rdb规则！
- 3、退出redis，也会产生 rdb 文件！

备份就自动生成一个 dump.rdb

```
[root@kuangshen bin]# ls
dump.rdb          kconfig      mcrypt      redis-check-rdb
jemalloc-config   libmcrypt-config mdecrypt   redis-cli
jemalloc.sh       luajit       redis-benchmark redis-sentinel
jeprof           luajit-2.0.4  redis-check-aof  redis-server
[root@kuangshen bin]#
```

如果恢复rdb文件！

1、只需要将rdb文件放在我们redis启动目录就可以，redis启动的时候会自动检查dump.rdb 恢复其中的数据！

2、查看需要存在的位置

```
127.0.0.1:6379> config get dir
1) "dir"
2) "/usr/local/bin" # 如果在这个目录下存在 dump.rdb 文件，启动就会自动恢复其中的数据
```

几乎就他自己默认的配置就够了，但是我们还是需要去学习！

优点：

- 1、适合大规模的数据恢复！
- 2、对数据的完整性要不高！

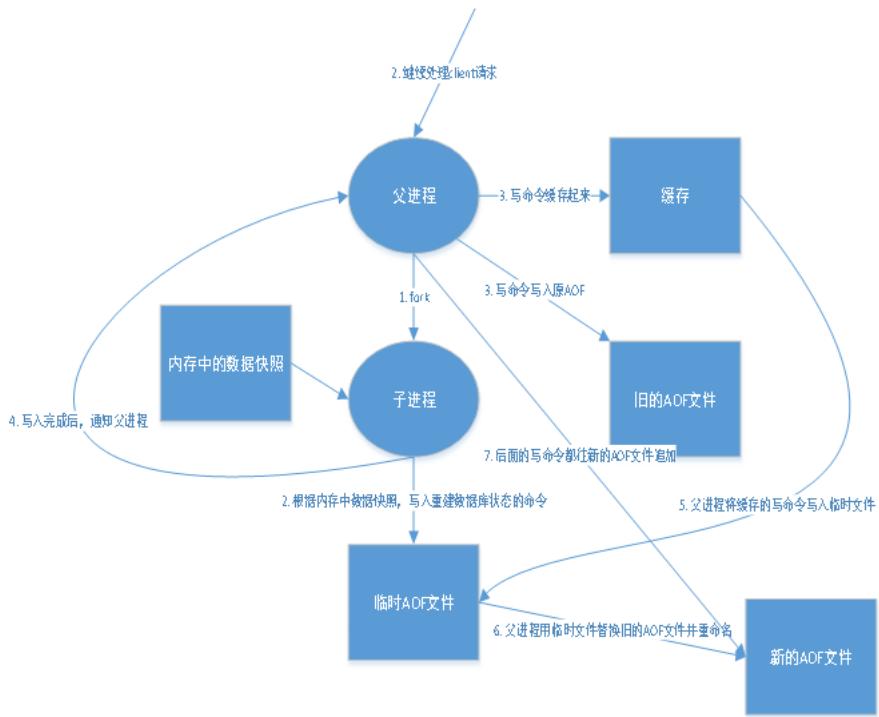
缺点：

- 1、需要一定的时间间隔进程操作！如果redis意外宕机了，这个最后一次修改数据就没有的了！
- 2、fork进程的时候，会占用一定的内容空间！！

AOF (Append Only File)

将我们的所有命令都记录下来，history，恢复的时候把这个文件全部在执行一遍！

是什么



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

Aof保存的是 appendonly.aof 文件

append

```
#####
APPEND ONLY MODE #####
# By default Redis asynchronously dumps the dataset on disk. This mode is
# good enough in many applications, but an issue with the Redis process or
# a power outage may result into a few minutes of writes lost (depending on
# the configured save points).
#
# The Append Only File is an alternative persistence mode that provides
# much better durability. For instance using the default data fsync policy
# (see later in the config file) Redis can lose just one second of writes in a
# dramatic event like a server power outage, or a single write if something
# wrong with the Redis process itself happens, but the operating system is
# still running correctly.
#
# AOF and RDB persistence can be enabled at the same time without problems.
# If the AOF is enabled on startup Redis will load the AOF, that is the file
# with the better durability guarantees.
#
# Please check http://redis.io/topics/persistence for more information.

appendonly no
```

默认是不开启的，我们需要手动进行配置！我们只需要将 appendonly 改为yes就开启了 aof！

重启，redis 就可以生效了！

如果这个 aof 文件有错位，这时候 redis 是启动不起来的吗，我们需要修复这个aof文件

redis 给我们提供了一个工具 `redis-check-aof --fix`

```
[root@kuangshen bin]# redis-check-aof --fix appendonly.aof
0x          a4: Expected \r\n, got: 6461
AOF analyzed: size=185, ok_up_to=139, diff=46
This will shrink the AOF from 185 bytes, with 46 bytes, to 139 bytes
Continue? [y/N]: y
Successfully truncated AOF
[root@kuangshen bin]#
```

如果文件正常，重启就可以直接恢复了！

```
[root@kuangshen bin]# ps -ef|grep redis
root      25945 15221  0 23:02 pts/0    00:00:00 grep --color=auto redis
[root@kuangshen bin]# redis-server kconfig/redis.conf
25948:C 30 Mar 2020 23:02:05.489 # o000o000o000o Redis is starting o000o000o000o
25948:C 30 Mar 2020 23:02:05.489 # Redis version=5.0.8, bits=64, commit=00000000
, modified=0, pid=25948, just started
25948:C 30 Mar 2020 23:02:05.489 # Configuration loaded
[root@kuangshen bin]# redis-cli -p 6379
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> get k1
"v1"
127.0.0.1:6379>
```

重写规则说明

aof 默认就是文件的无限追加，文件会越来越大！

```
no-appendfsync-on-rewrite no
# Automatic rewrite of the append only file.
# Redis is able to automatically rewrite the log file implicitly c
# BGREWRITEAOF when the AOF log size grows by the specified percen
#
# This is how it works: Redis remembers the size of the AOF file a
# latest rewrite (if no rewrite has happened since the restart, th
# the AOF at startup is used).
#
# This base size is compared to the current size. If the current s
# bigger than the specified percentage, the rewrite is triggered.
# you need to specify a minimal size for the AOF file to be rewrit
# is useful to avoid rewriting the AOF file even if the percentage
# is reached but it is still pretty small.
#
# Specify a percentage of zero in order to disable the automatic A
# rewrite feature.

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

如果 aof 文件大于 64m，太大了！ fork一个新的进程来将我们的文件进行重写！

优点和缺点！

```
appendonly no      # 默认是不开启aof模式的，默认是使用rdb方式持久化的，在大部分所有的情况下，  
rdb完全够用！  
appendfilename "appendonly.aof"  # 持久化的文件的名字  
  
# appendfsync always    # 每次修改都会 sync。消耗性能  
appendfsync everysec   # 每秒执行一次 sync，可能会丢失这1s的数据！  
# appendfsync no        # 不执行 sync，这个时候操作系统自己同步数据，速度最快！  
  
# rewrite 重写，
```

优点：

- 1、每一次修改都同步，文件的完整会更加好！
- 2、每秒同步一次，可能会丢失一秒的数据
- 3、从不同步，效率最高的！

缺点：

- 1、相对于数据文件来说，aof远远大于 rdb，修复的速度也比 rdb慢！
- 2、Aof 运行效率也要比 rdb 慢，所以我们redis默认的配置就是rdb持久化！

扩展：

- 1、RDB 持久化方式能够在指定的时间间隔内对你的数据进行快照存储
- 2、AOF 持久化方式记录每次对服务器写的操作，当服务器重启的时候会重新执行这些命令来恢复原始的数据，AOF命令以Redis 协议追加保存每次写的操作到文件末尾，Redis还能对AOF文件进行后台重写，使得AOF文件的体积不至于过大。
- 3、**只做缓存，如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化**
- 4、同时开启两种持久化方式
 - 在这种情况下，当redis重启的时候会优先载入AOF文件来恢复原始的数据，因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。
 - RDB 的数据不实时，同时使用两者时服务器重启也只会找AOF文件，那要不要只使用AOF呢？作者建议不要，因为RDB更适合用于备份数据库（AOF在不断变化不好备份），快速重启，而且不会有AOF可能潜在的Bug，留着作为一个万一的手段。

5、性能建议

- 因为RDB文件只用作后备用途，建议只在Slave上持久化RDB文件，而且只要15分钟备份一次就够了，只保留 save 900 1 这条规则。
- 如果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%大小重写可以改到适当的数值。
- 如果不Enable AOF，仅靠 Master-Slave Replication 实现高可用性也可以，能省掉一大笔IO，也减少了rewrite时带来的系统波动。代价是如果Master/Slave 同时倒掉，会丢失十几分钟的数据，启动脚本也要比较两个 Master/Slave 中的 RDB文件，载入较新的那个，微博就是这种架构。

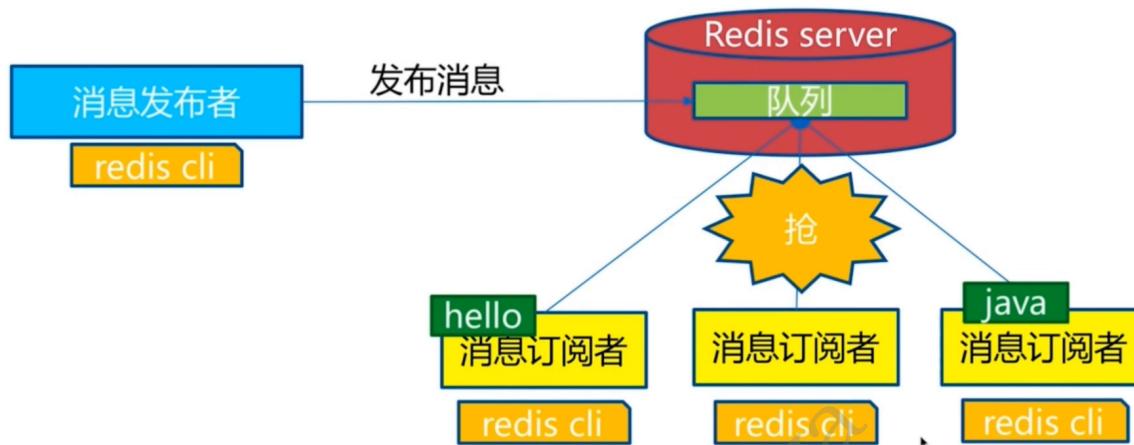
Redis发布订阅

Redis 发布订阅(pub/sub)是一种**消息通信模式**：发送者(pub)发送消息，订阅者(sub)接收消息。微信、微博、关注系统！

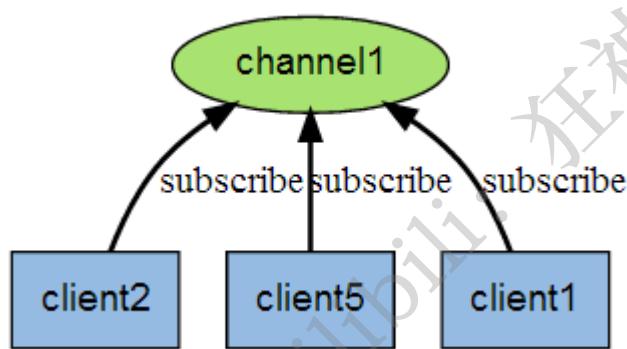
Redis 客户端可以订阅任意数量的频道。

订阅/发布消息图：

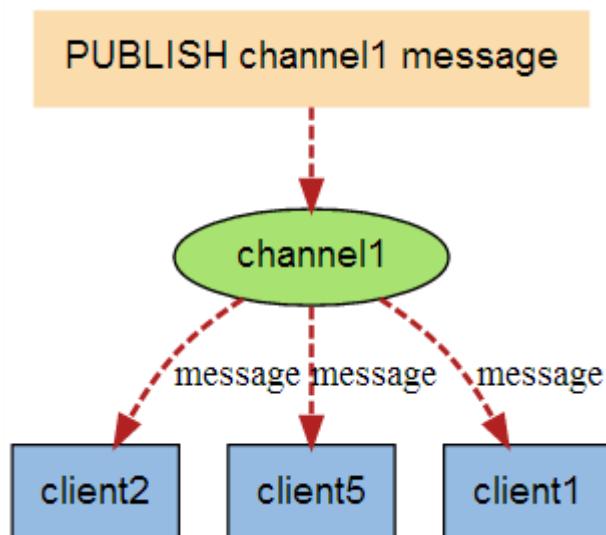
第一个：消息发送者，第二个：频道 第三个：消息订阅者！



下图展示了频道 channel1，以及订阅这个频道的三个客户端——client2、client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时，这个消息就会被发送给订阅它的三个客户端：



命令

这些命令被广泛用于构建即时通信应用，比如网络聊天室(chatroom)和实时广播、实时提醒等。

序号	命令及描述
1	<u>PSUBSCRIBE pattern [pattern ...]</u> 订阅一个或多个符合给定模式的频道。
2	<u>PUBSUB subcommand [argument [argument ...]]</u> 查看订阅与发布系统状态。
3	<u>PUBLISH channel message</u> 将信息发送到指定的频道。
4	<u>PUNSUBSCRIBE [pattern [pattern ...]]</u> 退订所有给定模式的频道。
5	<u>SUBSCRIBE channel [channel ...]</u> 订阅给定的一个或多个频道的信息。
6	<u>UNSUBSCRIBE [channel [channel ...]]</u> 指退订给定的频道。

测试

订阅端：

```
127.0.0.1:6379> SUBSCRIBE kuangshenshuo # 订阅一个频道 kuangshenshuo
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "kuangshenshuo"
3) (integer) 1
# 等待读取推送的信息
1) "message" # 消息
2) "kuangshenshuo" # 那个频道的消息
3) "hello,kuangshen" # 消息的具体内容

1) "message"
2) "kuangshenshuo"
3) "hello,redis"
```

发送端：

```
127.0.0.1:6379> PUBLISH kuangshenshuo "hello,kuangshen" # 发布者发布消息到频道!
(integer) 1
127.0.0.1:6379> PUBLISH kuangshenshuo "hello,redis" # 发布者发布消息到频道!
(integer) 1
127.0.0.1:6379>
```

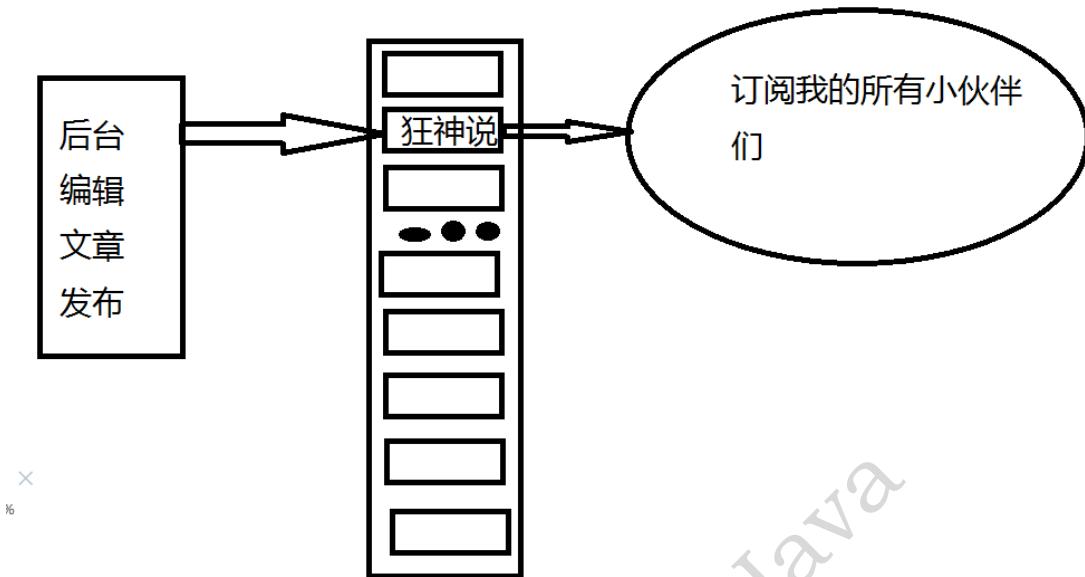
原理

Redis是使用C实现的，通过分析 Redis 源码里的 pubsub.c 文件，了解发布和订阅机制的底层实现，籍此加深对 Redis 的理解。

Redis 通过 PUBLISH 、 SUBSCRIBE 和 PSUBSCRIBE 等命令实现发布和订阅功能。

微信：

通过 SUBSCRIBE 命令订阅某频道后，redis-server 里维护了一个字典，字典的键就是一个个 频道！，而字典的值则是一个链表，链表中保存了所有订阅这个 channel 的客户端。SUBSCRIBE 命令的关键，就是将客户端添加到给定 channel 的订阅链表中。



通过 PUBLISH 命令向订阅者发送消息，redis-server 会使用给定的频道作为键，在它所维护的 channel 字典中查找记录了订阅这个频道的所有客户端的链表，遍历这个链表，将消息发布给所有订阅者。



Pub/Sub 从字面上理解就是发布 (Publish) 与订阅 (Subscribe) ，在Redis中，你可以设定对某一个 key值进行消息发布及消息订阅，当一个key值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

使用场景：

- 1、实时消息系统！
 - 2、事实聊天！（频道当做聊天室，将信息回显给所有人即可！）
 - 3、订阅，关注系统都是可以的！
- 稍微复杂的场景我们就会使用 消息中间件 MQ ()

Redis主从复制

概念

主从复制，是指将一台Redis服务器的数据，复制到其他的Redis服务器。前者称为主节点(master/leader)，后者称为从节点(slave/follower)；**数据的复制是单向的，只能由主节点到从节点。**Master以写为主，Slave以读为主。

默认情况下，每台Redis服务器都是主节点；

且一个主节点可以有多个从节点(或没有从节点)，但一个从节点只能有一个主节点。()

主从复制的作用主要包括：

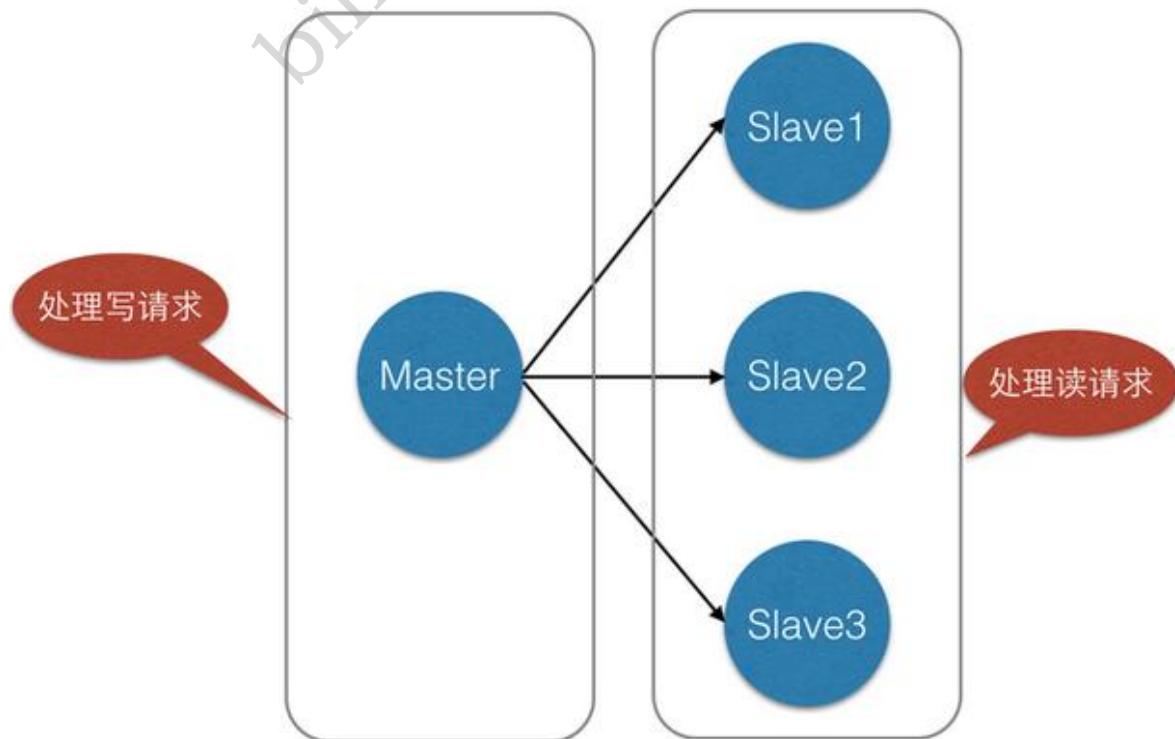
- 1、数据冗余：主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。
- 2、故障恢复：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复；实际上是一种服务的冗余。
- 3、负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写Redis数据时应用连接主节点，读Redis数据时应用连接从节点），分担服务器负载；尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高Redis服务器的并发量。
- 4、高可用（集群）基石：除了上述作用以外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是Redis高可用的基础。

一般来说，要将Redis运用于工程项目中，只使用一台Redis是万万不能的（宕机），原因如下：

- 1、从结构上，单个Redis服务器会发生单点故障，并且一台服务器需要处理所有的请求负载，压力较大；
- 2、从容量上，单个Redis服务器内存容量有限，就算一台Redis服务器内存容量为256G，也不能将所有内存用作Redis存储内存，一般来说，**单台Redis最大使用内存不应该超过20G。**

电商网站上的商品，一般都是一次上传，无数次浏览的，说专业点也就是“多读少写”。

对于这种场景，我们可以使如下这种架构：



主从复制，读写分离！ 80% 的情况下都是在进行读操作！减缓服务器的压力！架构中经常使用！ 一主二从！

只要在公司中，主从复制就是必须要使用的，因为在真实的项目中不可能单机使用Redis！

环境配置

只配置从库，不用配置主库！

```
127.0.0.1:6379> info replication # 查看当前库的信息
# Replication
role:master # 角色 master
connected_slaves:0 # 没有从机
master_replid:b63c90e6c501143759cb0e7f450bd1eb0c70882a
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

复制3个配置文件，然后修改对应的信息

- 1、端口
- 2、pid名字
- 3、log文件名字
- 4、dump.rdb名字

修改完毕之后，启动我们的3个redis服务器，可以通过进程信息查看！



The screenshot shows a terminal window with four tabs labeled 1, 2, 3, and 4, all set to expire in February 2021. The current tab (4) contains the command:

```
[root@kuangshen bin]# ps -ef|grep redis
root    24596    1  0 20:33 ?        00:00:00 redis-server 127.0.0.1:6379
root    24623    1  0 20:34 ?        00:00:00 redis-server 127.0.0.1:6380
root    24640    1  0 20:34 ?        00:00:00 redis-server 127.0.0.1:6381
root    24654 24281  0 20:34 pts/3    00:00:00 grep --color=auto redis
[root@kuangshen bin]#
```

The last line of the output, "grep --color=auto redis", is highlighted with a red rectangle.

一主二从

默认情况下，每台Redis服务器都是主节点；我们一般情况下只用配置从机就好了！

认老大！ 一主（79）二从（80, 81）

```
127.0.0.1:6380> SLAVEOF 127.0.0.1 6379 # SLAVEOF host 6379 找谁当自己的老大!
OK
127.0.0.1:6380> info replication
# Replication
role:slave # 当前角色是从机
master_host:127.0.0.1 # 可以的看到主机的信息
master_port:6379
```

如果两个都配置完了，就是有两个从机的

真实的从主配置应该在配置文件中配置，这样的话是永久的，我们这里使用的是命令，暂时的！

细节

主机可以写，从机不能写只能读！主机中的所有信息和数据，都会自动被从机保存！

主机写：

```

1 2021年2月到期 x 2 2021年2月到期 x 3 2021年2月到期 x 4 2021年2月到期 + 
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> set k1 v1
OK
127.0.0.1:6379> 

```

从机只能读取内容！

```

127.0.0.1:6381> keys *
(empty list or set)
127.0.0.1:6381> keys *
1) "k1"
127.0.0.1:6381> get k1
"v1"
127.0.0.1:6381> set k2 v2
(error) READONLY You can't write against a read only replica.
127.0.0.1:6381> 

```

测试：主机断开连接，从机依旧连接到主机的，但是没有写操作，这个时候，主机如果回来了，从机依旧可以直接获取到主机写的信息！

如果是使用命令行，来配置的主从，这个时候如果重启了，就会变回主机！只要变为从机，立马就会从主机中获取值！

复制原理

Slave 启动成功连接到 master 后会发送一个sync同步命令

Master 接到命令，启动后台的存盘进程，同时收集所有接收到的用于修改数据集命令，在后台进程执行完毕之后，master将传送整个数据文件到slave，并完成一次完全同步。

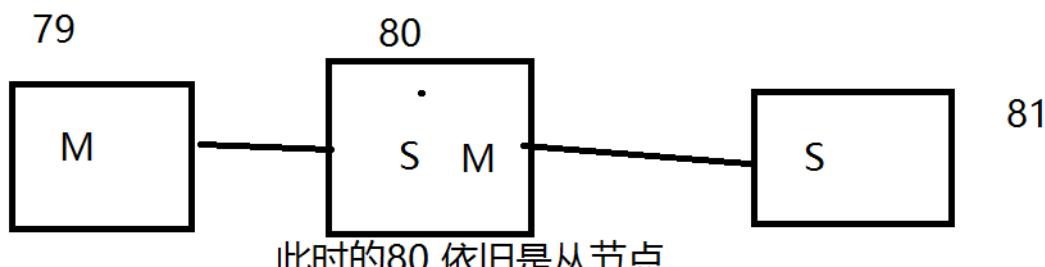
全量复制：而slave服务在接收到数据库文件数据后，将其存盘并加载到内存中。

增量复制：Master 继续将新的所有收集到的修改命令依次传给slave，完成同步

但是只要是重新连接master，一次完全同步（全量复制）将被自动执行！我们的数据一定可以在从机中看到！

层层链路

上一个M链接下一个 S！



这时候也可以完成我们的主从复制！

如果没有老大了，这个时候能不能选择一个老大出来呢？手动！

谋朝篡位

如果主机断开了连接，我们可以使用 `SLAVEOF no one` 让自己变成主机！其他的节点就可以手动连接到最新的这个主节点（手动）！如果这个时候老大修复了，那就重新连接！

哨兵模式

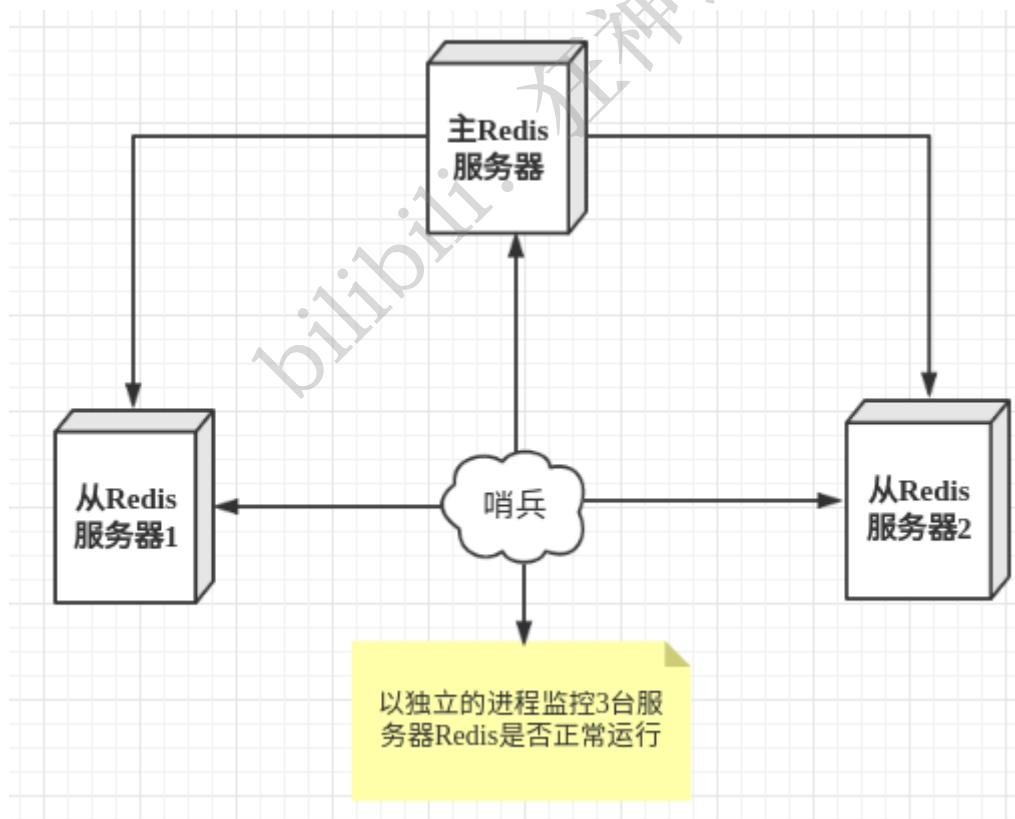
（自动选举老大的模式）

概述

主从切换技术的方法是：当主服务器宕机后，需要手动把一台从服务器切换为主服务器，这就需要人工干预，费事费力，还会造成一段时间内服务不可用。这不是一种推荐的方式，更多时候，我们优先考虑哨兵模式。Redis从2.8开始正式提供了Sentinel（哨兵）架构来解决这个问题。

谋朝篡位的自动版，能够后台监控主机是否故障，如果故障了根据投票数自动将从库转换为主库。

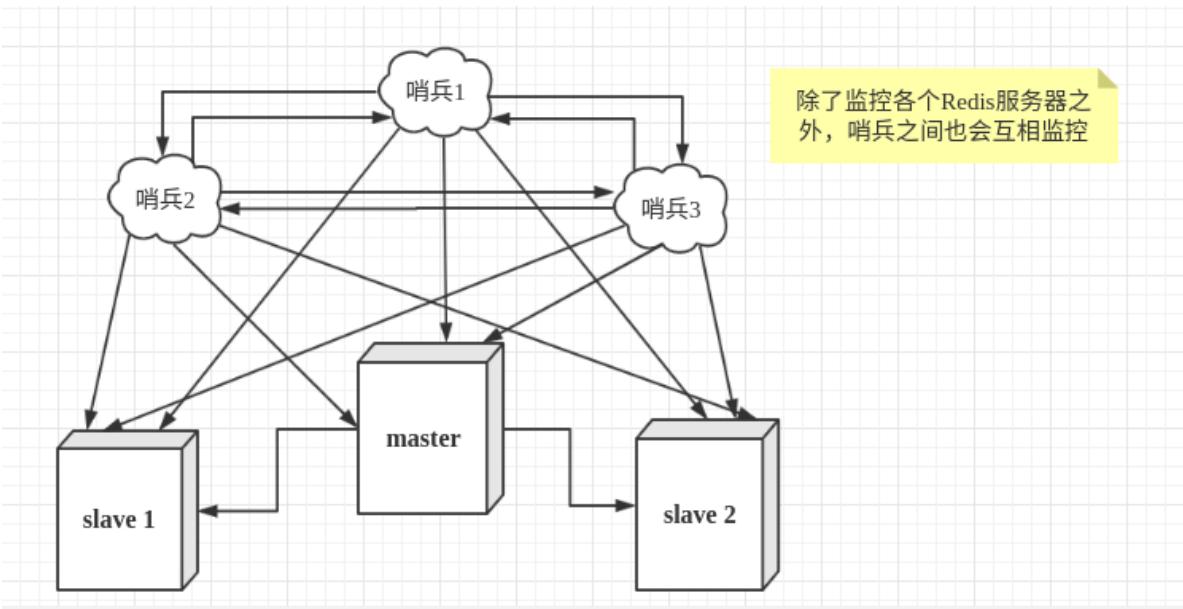
哨兵模式是一种特殊的模式，首先Redis提供了哨兵的命令，哨兵是一个独立的进程，作为进程，它会独立运行。其原理是哨兵通过发送命令，等待Redis服务器响应，从而监控运行的多个Redis实例。



这里的哨兵有两个作用

- 通过发送命令，让Redis服务器返回监控其运行状态，包括主服务器和从服务器。
- 当哨兵监测到master宕机，会自动将slave切换成master，然后通过发布订阅模式通知其他的从服务器，修改配置文件，让它们切换主机。

然而一个哨兵进程对Redis服务器进行监控，可能会出现问题，为此，我们可以使用多个哨兵进行监控。各个哨兵之间还会进行监控，这样就形成了多哨兵模式。



假设主服务器宕机，哨兵1先检测到这个结果，系统并不会马上进行failover过程，仅仅是哨兵1主观的认为为主服务器不可用，这个现象成为**主观下线**。当后面的哨兵也检测到主服务器不可用，并且数量达到一定值时，那么哨兵之间就会进行一次投票，投票的结果由一个哨兵发起，进行failover[故障转移]操作。切换成功后，就会通过发布订阅模式，让各个哨兵把自己监控的从服务器实现切换主机，这个过程称为**客观下线**。

测试！

我们目前的状态是一主二从！

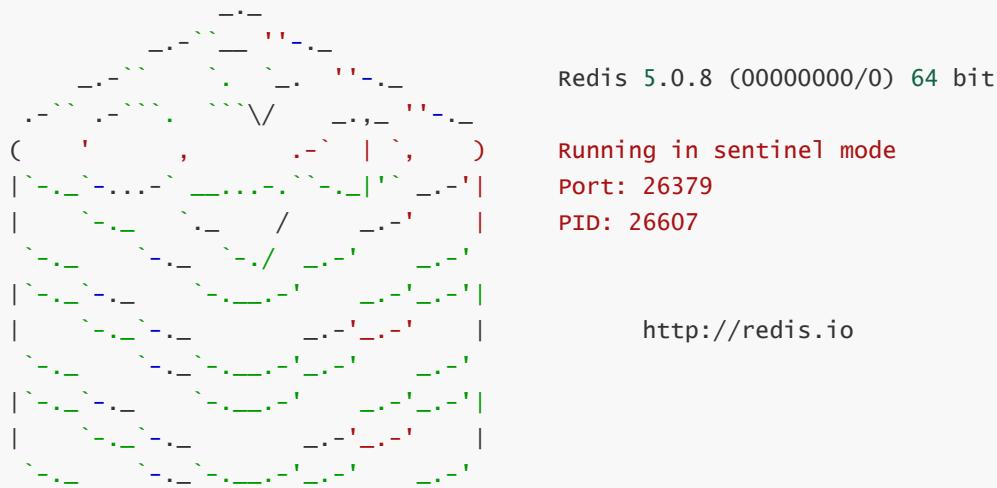
1、配置哨兵配置文件 sentinel.conf

```
# sentinel monitor 被监控的名称 host port 1
sentinel monitor myredis 127.0.0.1 6379 1
```

后面的这个数字1，代表主机挂了，slave投票看让谁接替成为主机，票数最多的，就会成为主机！

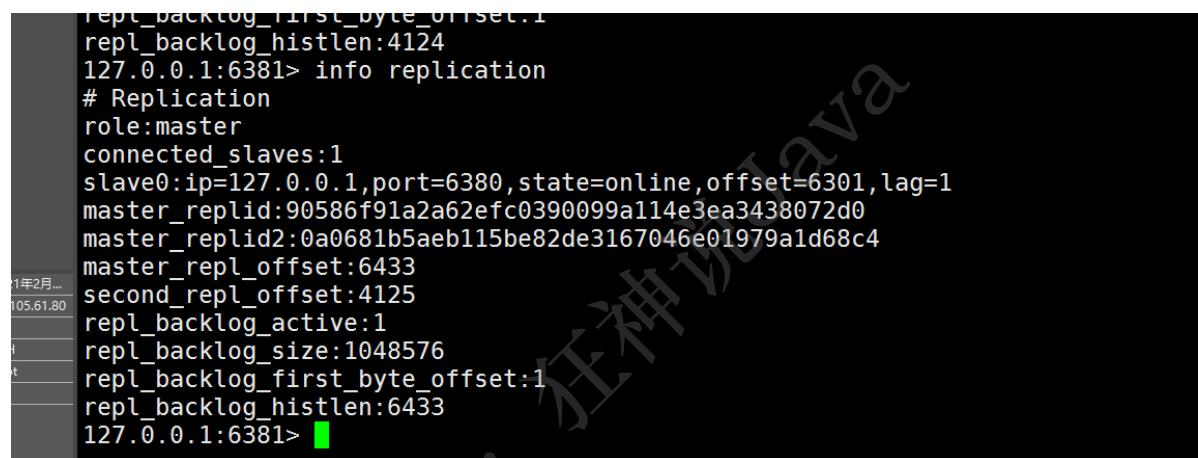
2、启动哨兵！

```
[root@kuangshen bin]# redis-sentinel kconfig/sentinel.conf
26607:x 31 Mar 2020 21:13:10.027 # 000000000000 Redis is starting 000000000000
26607:x 31 Mar 2020 21:13:10.027 # Redis version=5.0.8, bits=64,
commit=00000000, modified=0, pid=26607, just started
26607:x 31 Mar 2020 21:13:10.027 # Configuration loaded
```



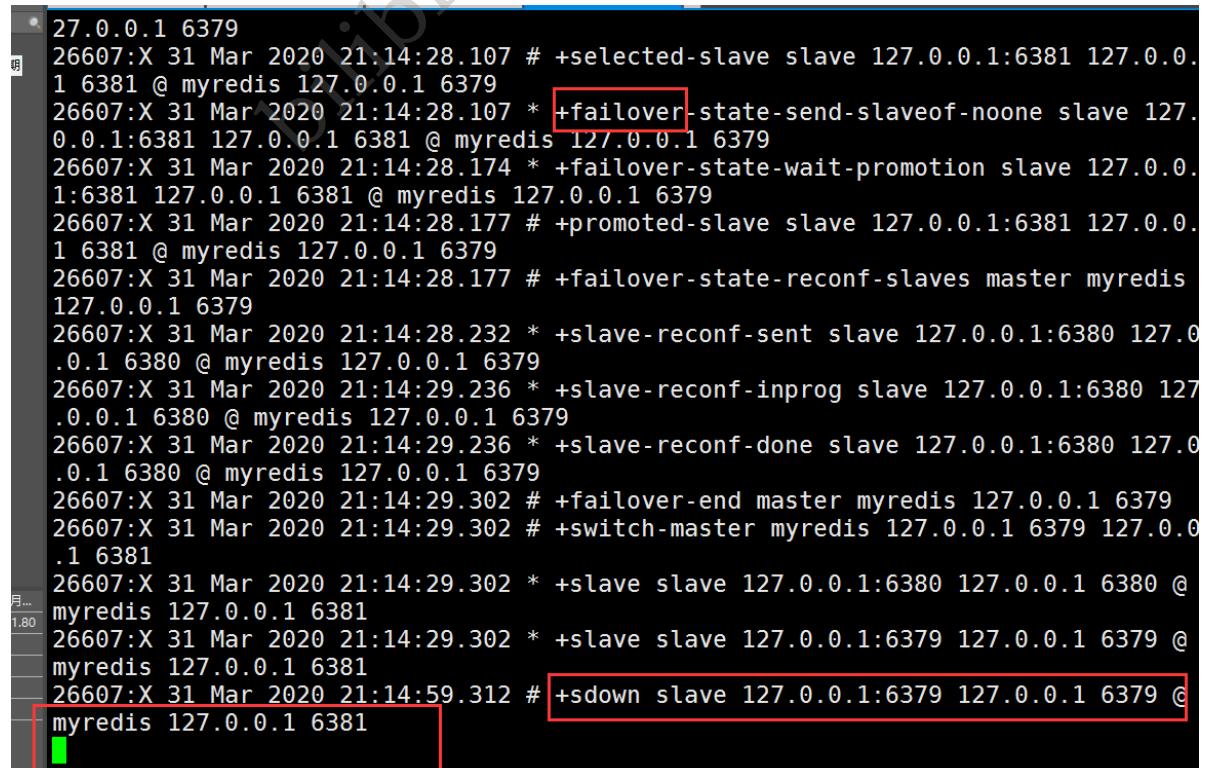
```
26607:x 31 Mar 2020 21:13:10.029 # WARNING: The TCP backlog setting of 511  
cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value  
of 128.  
26607:x 31 Mar 2020 21:13:10.031 # sentinel ID is  
4c780da7e22d2aebe3bc20c333746f202ce72996  
26607:x 31 Mar 2020 21:13:10.031 # +monitor master myredis 127.0.0.1 6379 quorum  
1  
26607:x 31 Mar 2020 21:13:10.031 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @  
myredis 127.0.0.1 6379  
26607:x 31 Mar 2020 21:13:10.033 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @  
myredis 127.0.0.1 6379
```

如果Master 节点断开了，这个时候就会从从机中随机选择一个服务器！（这里面有一个投票算法！）



```
repl_backlog_histlen:1  
repl_backlog_histlen:4124  
127.0.0.1:6381> info replication  
# Replication  
role:master  
connected_slaves:1  
slave0:ip=127.0.0.1,port=6380,state=online,offset=6301,lag=1  
master_replid:90586f91a2a62efc0390099a114e3ea3438072d0  
master_replid2:0a0681b5aeb115be82de3167046e01979a1d68c4  
master_repl_offset:6433  
second_repl_offset:4125  
repl_backlog_active:1  
repl_backlog_size:1048576  
repl_backlog_first_byte_offset:1  
repl_backlog_histlen:6433  
127.0.0.1:6381>
```

哨兵日志！



```
27.0.0.1 6379  
26607:X 31 Mar 2020 21:14:28.107 # +selected-slave slave 127.0.0.1:6381 127.0.0.  
1 6381 @ myredis 127.0.0.1 6379  
26607:X 31 Mar 2020 21:14:28.107 * +failover-state-send-slaveof-noone slave 127.  
0.0.1:6381 127.0.0.1 6381 @ myredis 127.0.0.1 6379  
26607:X 31 Mar 2020 21:14:28.174 * +failover-state-wait-promotion slave 127.0.0.  
1:6381 127.0.0.1 6381 @ myredis 127.0.0.1 6379  
26607:X 31 Mar 2020 21:14:28.177 # +promoted-slave slave 127.0.0.1:6381 127.0.0.  
1 6381 @ myredis 127.0.0.1 6379  
26607:X 31 Mar 2020 21:14:28.177 # +failover-state-reconf-slaves master myredis  
127.0.0.1 6379  
26607:X 31 Mar 2020 21:14:28.232 * +slave-reconf-sent slave 127.0.0.1:6380 127.0.  
.0.1 6380 @ myredis 127.0.0.1 6379  
26607:X 31 Mar 2020 21:14:29.236 * +slave-reconf-inprog slave 127.0.0.1:6380 127.  
.0.0.1 6380 @ myredis 127.0.0.1 6379  
26607:X 31 Mar 2020 21:14:29.236 * +slave-reconf-done slave 127.0.0.1:6380 127.0.  
.0.1 6380 @ myredis 127.0.0.1 6379  
26607:X 31 Mar 2020 21:14:29.302 # +failover-end master myredis 127.0.0.1 6379  
26607:X 31 Mar 2020 21:14:29.302 # +switch-master myredis 127.0.0.1 6379 127.0.  
.1 6381  
26607:X 31 Mar 2020 21:14:29.302 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @  
myredis 127.0.0.1 6381  
26607:X 31 Mar 2020 21:14:29.302 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @  
myredis 127.0.0.1 6381  
26607:X 31 Mar 2020 21:14:59.312 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @  
myredis 127.0.0.1 6381
```

如果主机此时回来了，只能归并到新的主机下，当做从机，这就是哨兵模式的规则！

哨兵模式

优点：

- 1、哨兵集群，基于主从复制模式，所有的主从配置优点，它全有
- 2、主从可以切换，故障可以转移，系统的可用性就会更好
- 3、哨兵模式就是主从模式的升级，手动到自动，更加健壮！

缺点：

- 1、Redis 不好啊在线扩容的，集群容量一旦到达上限，在线扩容就十分麻烦！
- 2、实现哨兵模式的配置其实是很麻烦的，里面有很多选择！

哨兵模式的全部配置！

```
# Example sentinel.conf

# 哨兵sentinel实例运行的端口 默认26379
port 26379

# 哨兵sentinel的工作目录
dir /tmp

# 哨兵sentinel监控的redis主节点的 ip port
# master-name 可以自己命名的主节点名字 只能由字母A-z、数字0-9、这三个字符".-_组成。
# quorum 配置多少个sentinel哨兵统一认为master主节点失联 那么这时客观上认为主节点失联了
# sentinel monitor <master-name> <ip> <redis-port> <quorum>
sentinel monitor mymaster 127.0.0.1 6379 2

# 当在Redis实例中开启了requirepass foobared 授权密码 这样所有连接Redis实例的客户端都要提供
# 密码
# 设置哨兵sentinel 连接主从的密码 注意必须为主从设置一样的验证密码
# sentinel auth-pass <master-name> <password>
sentinel auth-pass mymaster MySUPER--secret-0123passw0rd

# 指定多少毫秒之后 主节点没有应答哨兵sentinel 此时 哨兵主观上认为主节点下线 默认30秒
# sentinel down-after-milliseconds <master-name> <milliseconds>
sentinel down-after-milliseconds mymaster 30000

# 这个配置项指定了在发生failover主备切换时最多可以有多少个slave同时对新的master进行 同步,
# 这个数字越小, 完成failover所需的时间就越长,
# 但是如果这个数字越大, 就意味着越 多的slave因为replication而不可用。
# 可以通过将这个值设为 1 来保证每次只有一个slave 处于不能处理命令请求的状态。
# sentinel parallel-syncs <master-name> <numslaves>
sentinel parallel-syncs mymaster 1

# 故障转移的超时时间 failover-timeout 可以用在以下这些方面:
#1. 同一个sentinel对同一个master两次failover之间的间隔时间。
#2. 当一个slave从一个错误的master那里同步数据开始计算时间。直到slave被纠正为向正确的master那
里同步数据时。
#3. 当想要取消一个正在进行的failover所需要的时间。
#4. 当进行failover时, 配置所有slaves指向新的master所需的最大时间。不过, 即使过了这个超时,
# slaves依然会被正确配置为指向master, 但是就不按parallel-syncs所配置的规则来了
# 默认三分钟
# sentinel failover-timeout <master-name> <milliseconds>
```

```
sentinel failover-timeout mymaster 180000

# SCRIPTS EXECUTION

#配置当某一事件发生时所需要执行的脚本，可以通过脚本来通知管理员，例如当系统运行不正常时发邮件通知相关人员。
#对于脚本的运行结果有以下规则：
#若脚本执行后返回1，那么该脚本稍后将会被再次执行，重复次数目前默认为10
#若脚本执行后返回2，或者比2更高的一个返回值，脚本将不会重复执行。
#如果脚本在执行过程中由于收到系统中断信号被终止了，则同返回值为1时的行为相同。
#一个脚本的最大执行时间为60s，如果超过这个时间，脚本将会被一个SIGKILL信号终止，之后重新执行。

#通知型脚本：当sentinel有任何警告级别的事件发生时（比如说redis实例的主观失效和客观失效等等），将会去调用这个脚本，这时这个脚本应该通过邮件，SMS等方式去通知系统管理员关于系统不正常运行的信息。调用该脚本时，将传给脚本两个参数，一个是事件的类型，一个是事件的描述。如果sentinel.conf配置文件中配置了这个脚本路径，那么必须保证这个脚本存在于这个路径，并且是可执行的，否则sentinel无法正常启动成功。
#通知脚本
# shell编程
# sentinel notification-script <master-name> <script-path>
sentinel notification-script mymaster /var/redis/notify.sh

# 客户端重新配置主节点参数脚本
# 当一个master由于failover而发生改变时，这个脚本将会被调用，通知相关的客户端关于master地址已经发生改变的信息。
# 以下参数将会在调用脚本时传给脚本：
# <master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>
# 目前<state>总是“failover”，
# <role>是“leader”或者“observer”中的一个。
# 参数 from-ip, from-port, to-ip, to-port是用来和旧的master和新的master(即旧的slave)通信的
# 这个脚本应该是通用的，能被多次调用，不是针对性的。
# sentinel client-reconfig-script <master-name> <script-path>
sentinel client-reconfig-script mymaster /var/redis/reconfig.sh # 一般都是由运维来配置！
```

社会目前程序员饱和（初级和中级）、高级程序员重金难求！（提升自己！）

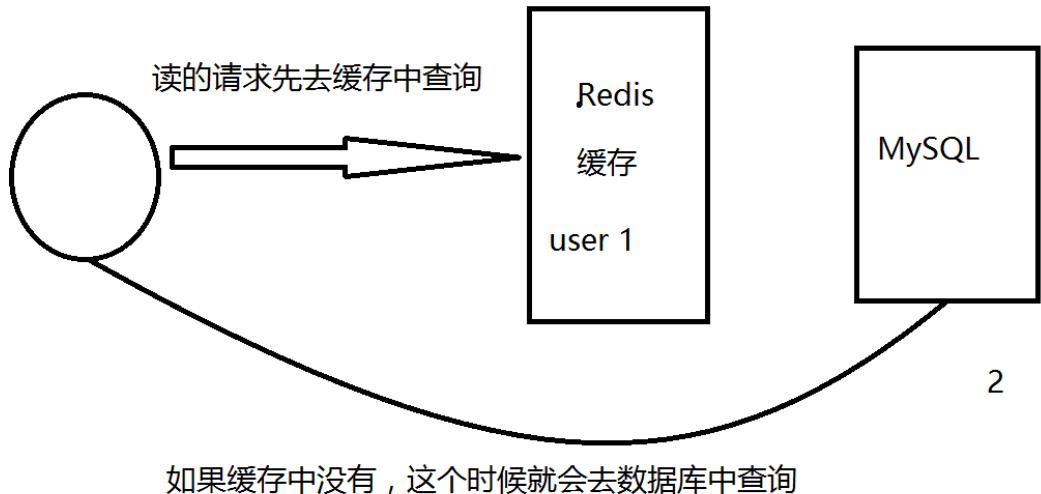
Redis缓存穿透和雪崩

服务的高可用问题！

在这里我们不会详细的区分析解决方案的底层！

Redis缓存的使用，极大的提升了应用程序的性能和效率，特别是数据查询方面。但同时，它也带来了一些问题。其中，最要害的问题，就是数据的一致性问题，从严格意义上讲，这个问题无解。如果对数据的一致性要求很高，那么就不能使用缓存。

另外的一些典型问题就是，缓存穿透、缓存雪崩和缓存击穿。目前，业界也都有比较流行的解决方案。



缓存穿透（查不到）

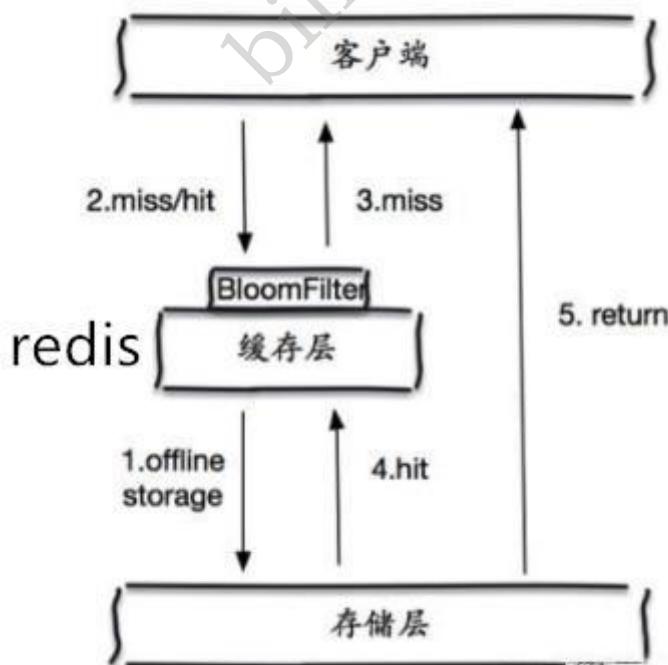
概念

缓存穿透的概念很简单，用户想要查询一个数据，发现redis内存数据库没有，也就是缓存没有命中，于是向持久层数据库查询。发现也没有，于是本次查询失败。当用户很多的时候，缓存都没有命中（秒杀！），于是都去请求了持久层数据库。这会给持久层数据库造成很大的压力，这时候就相当于出现了缓存穿透。

解决方案

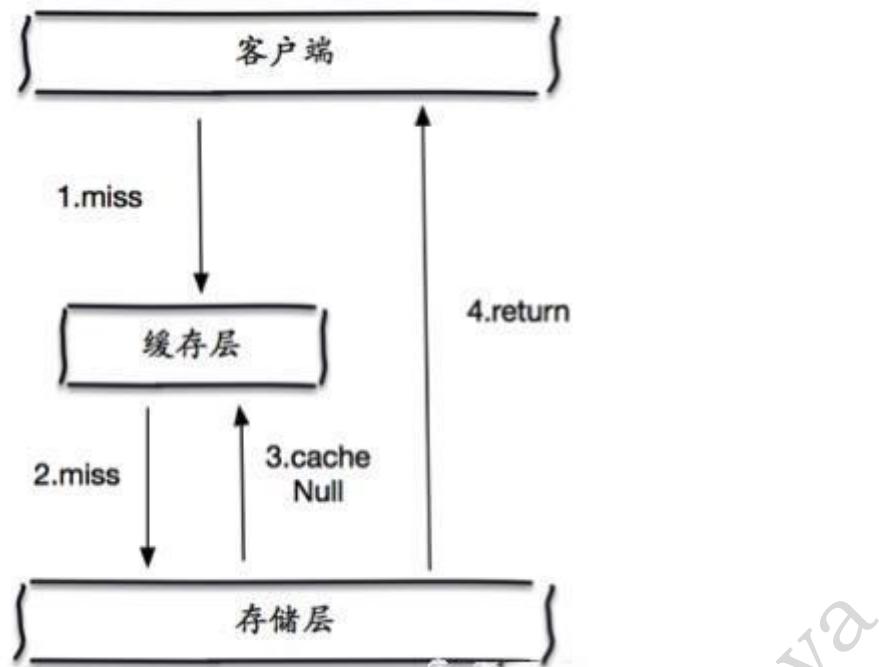
布隆过滤器

布隆过滤器是一种数据结构，对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃，从而避免了对底层存储系统的查询压力；



缓存空对象

当存储层不命中后，即使返回的空对象也将其缓存起来，同时会设置一个过期时间，之后再访问这个数据将会从缓存中获取，保护了后端数据源；



但是这种方法会存在两个问题：

- 1、如果空值能够被缓存起来，这就意味着缓存需要更多的空间存储更多的键，因为这当中可能会有很多的空值的键；
- 2、即使对空值设置了过期时间，还是会存在缓存层和存储层的数据会有一段时间窗口的不一致，这对于需要保持一致性的业务会有影响。

缓存击穿（量太大，缓存过期！）

概述

这里需要注意和缓存击穿的区别，缓存击穿，是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

当某个key在过期的瞬间，有大量的请求并发访问，这类数据一般是热点数据，由于缓存过期，会同时访问数据库来查询最新数据，并且回写缓存，会导使数据库瞬间压力过大。

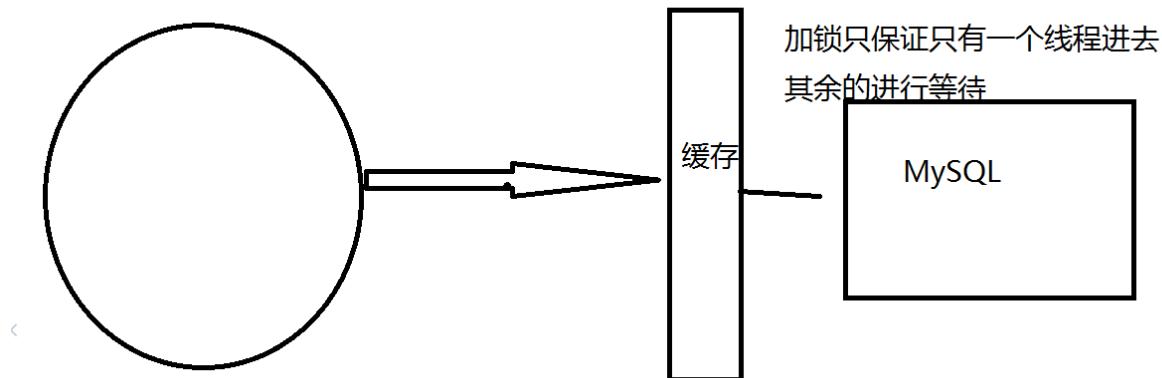
解决方案

设置热点数据永不过期

从缓存层面来看，没有设置过期时间，所以不会出现热点 key 过期后产生的问题。

加互斥锁

分布式锁：使用分布式锁，保证对于每个key同时只有一个线程去查询后端服务，其他线程没有获得分布式锁的权限，因此只需要等待即可。这种方式将高并发的压力转移到了分布式锁，因此对分布式锁的考验很大。

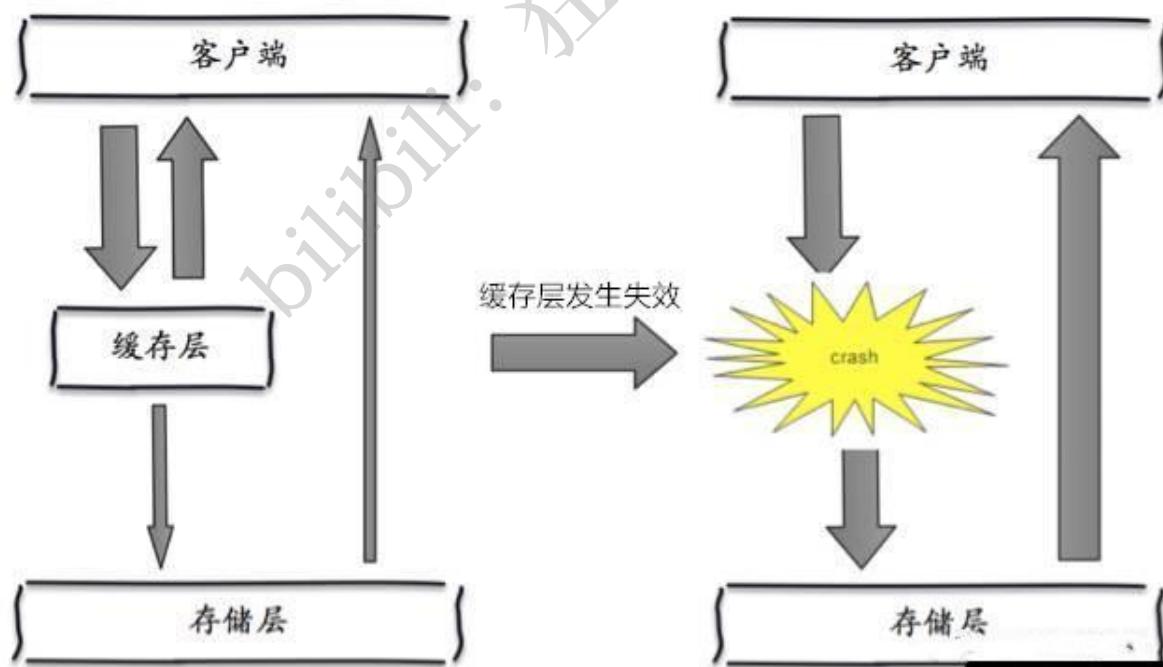


缓存雪崩

概念

缓存雪崩，是指在某一个时间段，缓存集中过期失效。Redis 容机！

产生雪崩的原因之一，比如在写本文的时候，马上就要到双十二零点，很快就会迎来一波抢购，这波商品时间比较集中的放入了缓存，假设缓存一个小时。那么到了凌晨一点钟的时候，这批商品的缓存就都过期了。而对这批商品的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。于是所有的请求都会达到存储层，存储层的调用量会暴增，造成存储层也会挂掉的情况。



其实集中过期，倒不是非常致命，比较致命的缓存雪崩，是缓存服务器某个节点宕机或断网。因为自然形成的缓存雪崩，一定是在某个时间段集中创建缓存，这个时候，数据库也是可以顶住压力的。无非就是对数据库产生周期性的压力而已。而缓存服务节点的宕机，对数据库服务器造成的影响是不可预知的，很有可能瞬间就把数据库压垮。

解决方案

redis高可用

这个思想的含义是，既然redis有可能挂掉，那我多增设几台redis，这样一台挂掉之后其他的还可以继续工作，其实就是搭建的集群。（异地多活！）

限流降级（在SpringCloud讲解过！）

这个解决方案的思想是，在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

数据预热

数据加热的含义就是在正式部署之前，我把可能的数据先预先访问一遍，这样部分可能大量访问的数据就会加载到缓存中。在即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

小结

关注公众号：狂神说，每日更新动态！

所有看我视频学习的小伙伴，分享或者写笔记的时候，可以带上我的视频连接，表示尊重！

后面的课程安排：bilibili免费直播，<https://space.bilibili.com/95256449>