Redis

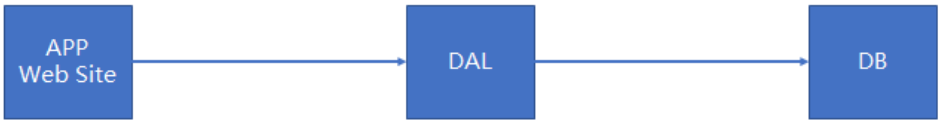
一、NoSQL概述

1、为什么要用NoSQL

- **单机数据库时代**

> 单机时代网站的访问量较小，单个数据库足以支撑网站运行。
>
> 但是随着时代发展，网站可能会面临如下问题：
>
> 1、数据量增加 2、访问量增加 3、数据库索引增加，导致服务器内存不足

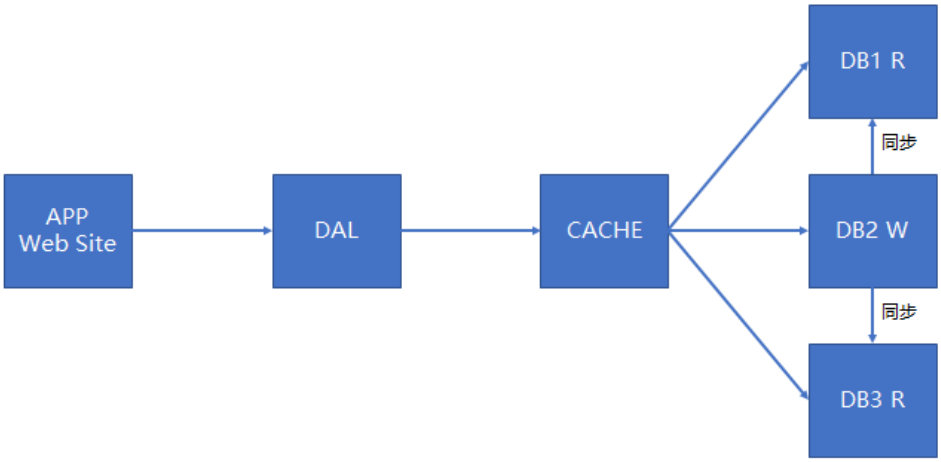
单机数据库



- **缓存 + 数据库 + 垂直拆分（读写分离）**

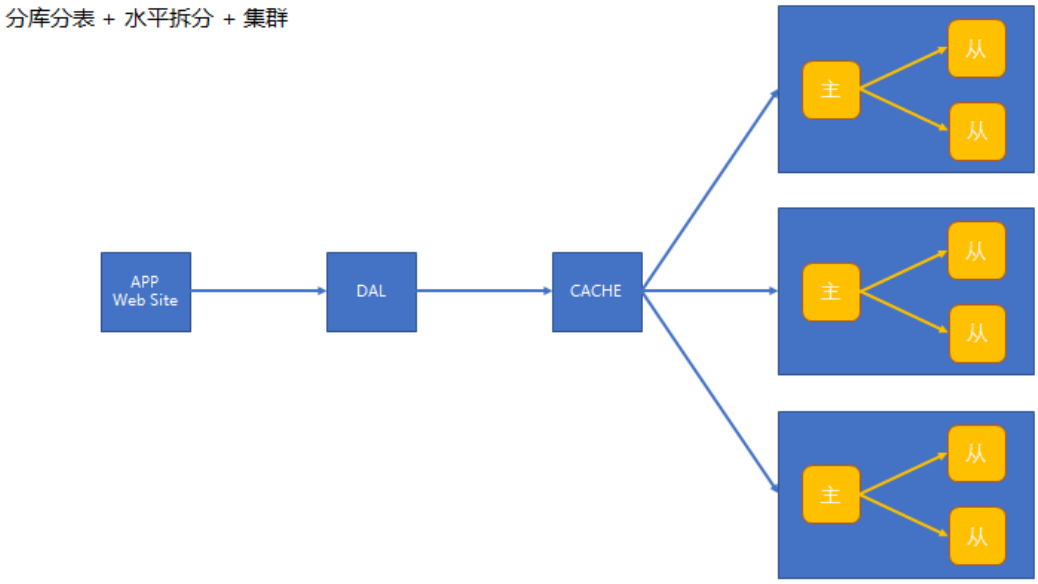
> 垂直拆分：把一个DB 拆分成多个，每一个都是一样的（结构和数据）。
>
> 分库分表：把查询操作和增删操作分开，查询是一个DB，增删改是一个DB，一般读写的需求量比较大，所以读库比较多。将数据拆分到不同的读库中，每个读库都存储一部分，查询的时候可以直接从指定的库做查询。
>
> 网站大部分需求是读取数据，每次读取数据都去数据库时数据库的压力会非常大，也非常麻烦。为了减轻服务器压力，可以采用**缓存的形式**提高效率。

缓存 + 数据库 + 垂直拆分



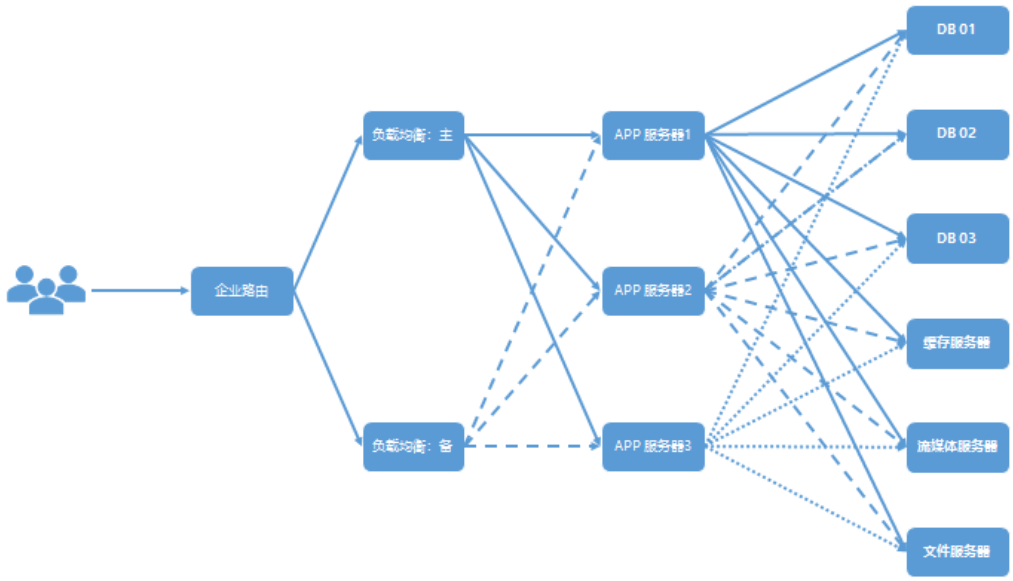
- **分库分表 + 水平拆分 + 集群**

> 当主从复制读写分离的这样的结构也扛不住压力时，可以把这样的架构复制多分，形成多节点，称为集群



- **大数据时代**

> 音乐、热搜、抖音排行、定位、热点数据、秒杀、访问量等数据量非常大，而且要求**响应速度快**，普通的关系型数据库已经不能满足这样的需求，如果将这些数据存储到数据库中，对数据库的压力是非常大的。如果能有专门的数据库存储这些数据，对原来的数据库压力会减少很多。



2、什么是NoSQL

> NoSQL：Not Only SQL，不仅仅是SQL，泛指非关系型数据库；
>
> 很多类型的数据不需要固定格式（行列）存储，不需要多余的操作就可以横向扩展的。`Map<String, Object>`
>
> NoSQL特点：
>
> 1、方便扩展 2、高性能（ Redis 读的速度是 110000次/s（11 W次/s）,写的速度是81000次/s（8.1W 次/s）） 3、数据类型多样化（不局限于关系型数据库有固定的数据类型，随去随用）

关系型数据库和非关系型数据库区别：

关系型数据库：

- 采用表格式结构化组织，数据单一
- SQL语言通用（MySQL、Oracle、SQL Server）
- 事务（ACID）

非关系型数据库

- 多样化存储架构（K-V、列、图形等）
- 高性能、高可用、高可扩
- 最终一致性

```
扩展了解：大数据时代 3V + 3高

海量Volume、多样Variety、实时Velocity
高并发、高可扩、高性能
```

二、Redis概述

（一）什么是Redis

> 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。
>
> Redis是一个开源（BSD许可），内存存储的数据结构服务器，可用作**数据库**，**高速缓存**和**消息队列代理**。它支持字符串、哈希表、列表、集合、有序集合，位图，hyperloglogs等数据类型。内置复制、Lua脚本、LRU收回、事务以及不同级别磁盘持久化功能，同时通过Redis Sentinel提供高可用，通过Redis Cluster提供自动分区。
>
> https://www.redis.net.cn/（中文翻译版） https://redis.io/（官网）

> 关系型数据库：MySQL、Oracle、SQL Server
>
> 非关系型数据库：Redis、Memercash、Hadoop、MongoDB
>
> NoSQL (Not only SQL)：数据结构式键值对的形式。

> **Redis 作用**
>
> 1、内存存储、持久化 2、效率高 3、发布订阅系统 4、地图信息分析 5、计时器、计数器（订阅量、点赞量）

（二）Redis安装

1、gcc基础环境安装

> 下图所示需要先安装 gcc 环境

```
echo LDFLAGS= >> .make-settings
echo REDIS_CFLAGS= >> .make-settings
echo REDIS_LDFLAGS= >> .make-settings
echo PREV_FINAL_CFLAGS=-pedantic -DREDIS_STATIC='' -std=c99 -Wall -W -Wno-missing-field-initializers -O2 -g -ggdb   -I../deps/hiredis -I../deps/linenoise -I../deps/lua/src -I../deps/hdr_histogram -DUSE_JEMALLOC -I../deps/jemalloc/include >> .make-settings
echo PREV_FINAL_LDFLAGS=  -g -ggdb -rdynamic >> .make-settings
(cd ../deps && make hiredis linenoise lua hdr_histogram jemalloc)
make[2]: 进入目录“/opt/redis-6.2.5/deps”
(cd hiredis && make clean) > /dev/null || true
(cd linenoise && make clean) > /dev/null || true
(cd lua && make clean) > /dev/null || true
(cd jemalloc && [ -f Makefile ] && make distclean) > /dev/null || true
(cd hdr_histogram && make clean) > /dev/null || true
(rm -f .make-*)
(echo "" > .make-cflags)
(echo "" > .make-ldflags)
MAKE hiredis
cd hiredis && make static
make[3]: 进入目录“/opt/redis-6.2.5/deps/hiredis”
cc -std=c99 -pedantic -c -O3 -fPIC   -Wall -W -Wstrict-prototypes -Wwrite-strings -Wno-missing-field-initializers -g -ggdb alloc.c
make[3]: cc：命令未找到
make[3]: *** [alloc.o] 错误 127
make[3]: 离开目录“/opt/redis-6.2.5/deps/hiredis”
make[2]: *** [hiredis] 错误 2
make[2]: 离开目录“/opt/redis-6.2.5/deps”
make[1]: [persist-settings] 错误 2 (忽略)
    CC adlist.o
/bin/sh: cc: 未找到命令
make[1]: *** [adlist.o] 错误 127
make[1]: 离开目录“/opt/redis-6.2.5/src”
make: *** [all] 错误 2
[root@localhost redis-6.2.5]#
```

确认基础环境，通过命令 `gcc -v` 查看，下图是没有环境的提示

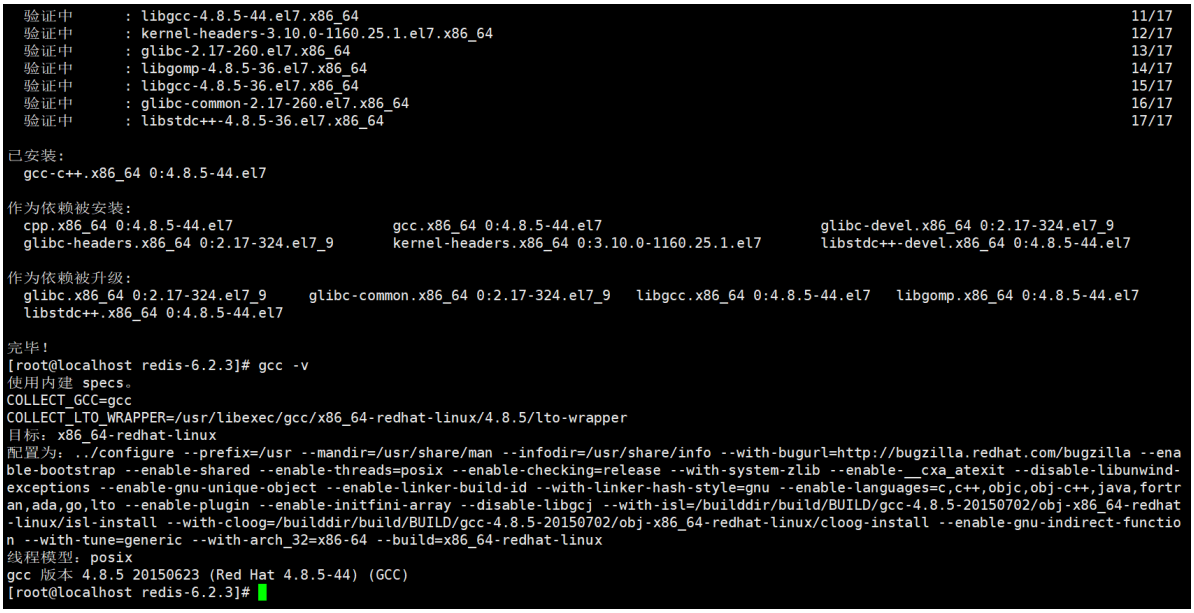
```
[root@localhost redis-6.2.3]# gcc -v
bash: gcc: 未找到命令...
[root@localhost redis-6.2.3]#
```

- 在线安装（方便）

输入命令：`yum install gcc-c++`

`rm -f /var/run/yum.pid`

```
[root@localhost redis-6.2.3]# yum install gcc-c++
已加载插件：fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirrors.aliyun.com
 * extras: mirrors.ustc.edu.cn
 * updates: mirrors.aliyun.com
base                                                                                                | 3.6 kB  00:00:00
extras                                                                                              | 2.9 kB  00:00:00
updates                                                                                             | 2.9 kB  00:00:00
(1/2): extras/7/x86_64/primary_db                                                                   | 236 kB  00:00:01
(2/2): updates/7/x86_64/primary_db                                                                  | 8.0 MB  00:00:16
正在解决依赖关系
--> 正在检查事务
---> 软件包 gcc-c++.x86_64.0.4.8.5-44.el7 将被 安装
--> 正在处理依赖关系 libstdc++-devel = 4.8.5-44.el7，它被软件包 gcc-c++-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 libstdc++ = 4.8.5-44.el7，它被软件包 gcc-c++-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 gcc = 4.8.5-44.el7，它被软件包 gcc-c++-4.8.5-44.el7.x86_64 需要
--> 正在检查事务
---> 软件包 gcc.x86_64.0.4.8.5-44.el7 将被 安装
--> 正在处理依赖关系 libgomp = 4.8.5-44.el7，它被软件包 gcc-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 cpp = 4.8.5-44.el7，它被软件包 gcc-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 libgcc >= 4.8.5-44.el7，它被软件包 gcc-4.8.5-44.el7.x86_64 需要
--> 正在处理依赖关系 glibc-devel >= 2.2.90-12，它被软件包 gcc-4.8.5-44.el7.x86_64 需要
---> 软件包 libstdc++.x86_64.0.4.8.5-36.el7 将被 升级
---> 软件包 libstdc++.x86_64.0.4.8.5-44.el7 将被 更新
---> 软件包 libstdc++-devel.x86_64.0.4.8.5-44.el7 将被 安装
--> 正在检查事务
---> 软件包 cpp.x86_64.0.4.8.5-44.el7 将被 安装
---> 软件包 glibc-devel.x86_64.0.2.17-324.el7_9 将被 安装
--> 正在处理依赖关系 glibc-headers = 2.17-324.el7_9，它被软件包 glibc-devel-2.17-324.el7_9.x86_64 需要
--> 正在处理依赖关系 glibc = 2.17-324.el7_9，它被软件包 glibc-devel-2.17-324.el7_9.x86_64 需要
--> 正在处理依赖关系 glibc-headers，它被软件包 glibc-devel-2.17-324.el7_9.x86_64 需要
---> 软件包 libgcc.x86_64.0.4.8.5-36.el7 将被 升级
---> 软件包 libgcc.x86_64.0.4.8.5-44.el7 将被 更新
```



- 离线安装

将Linux的系统文件加载到光驱，切换到目录：`cd /run/media/root/CentOS 7 x86_64/Packages/`，将一下命令挨个执行即可，注意不同的系统镜像版本不同。

```
rpm -ivh cpp-4.8.5-11.el7.x86_64.rpm
rpm -ivh kernel-headers-3.10.0-514.el7.x86_64.rpm
rpm -ivh glibc-headers-2.17-157.el7.x86_64.rpm
rpm -ivh glibc-devel-2.17-157.el7.x86_64.rpm
rpm -ivh libgomp-4.8.5-11.el7.x86_64.rpm
rpm -ivh gcc-4.8.5-11.el7.x86_64.rpm
```

2、安装`rpm -ivh cpp-4.8.5-11.el7.x86_64.rpm`报错解决放案：

```
输入命令：rpm -ivh libmpc-1.0.1-3.el7.x86_64.rpm
```

2、Redis安装

1）下载Redis安装包，上传Llinux中

2）解压安装包：`tar -zxvf redis-x.x.x.tar.gz`

3）进入redis解压后的目录：`cd redis-x.x.x`

```
[root@localhost redis-6.2.3]# ll
总用量 228
-rw-rw-r--.  1 root root 26882 5月   4 03:57 00-RELEASENOTES
-rw-rw-r--.  1 root root    51 5月   4 03:57 BUGS
-rw-rw-r--.  1 root root  5026 5月   4 03:57 CONDUCT
-rw-rw-r--.  1 root root  3384 5月   4 03:57 CONTRIBUTING
-rw-rw-r--.  1 root root  1487 5月   4 03:57 COPYING
drwxrwxr-x.  7 root root   145 5月   4 03:57 deps
-rw-rw-r--.  1 root root    11 5月   4 03:57 INSTALL
-rw-rw-r--.  1 root root   151 5月   4 03:57 Makefile
-rw-rw-r--.  1 root root  6888 5月   4 03:57 MANIFESTO
-rw-rw-r--.  1 root root 21567 5月   4 03:57 README.md
-rw-rw-r--.  1 root root 93724 5月   4 03:57 redis.conf
-rwxrwxr-x.  1 root root   275 5月   4 03:57 runtest
-rwxrwxr-x.  1 root root   279 5月   4 03:57 runtest-cluster
-rwxrwxr-x.  1 root root  1046 5月   4 03:57 runtest-moduleapi
-rwxrwxr-x.  1 root root   281 5月   4 03:57 runtest-sentinel
-rw-rw-r--.  1 root root 13768 5月   4 03:57 sentinel.conf
drwxrwxr-x.  3 root root  4096 5月   4 03:57 src
drwxrwxr-x. 11 root root   182 5月   4 03:57 tests
-rw-rw-r--.  1 root root  3055 5月   4 03:57 TLS.md
drwxrwxr-x.  9 root root  4096 5月   4 03:57 utils
[root@localhost redis-6.2.3]#
```

4） 输入 `make` 命令构建 Redis，如果构建过程遇到问题需要二次构建需要输入 `make distclean` 清除缓存信息。

```
    CC childinfo.o
    CC defrag.o
    CC siphash.o
    CC rax.o
    CC t_stream.o
    CC listpack.o
    CC localtime.o
    CC lolwut.o
    CC lolwut5.o
    CC lolwut6.o
    CC acl.o
    CC gopher.o
    CC tracking.o
    CC connection.o
    CC tls.o
    CC sha256.o
    CC timeout.o
    CC setcpuaffinity.o
    CC monotonic.o
    CC mt19937-64.o
    LINK redis-server
    INSTALL redis-sentinel
    CC redis-cli.o
    CC cli_common.o
    LINK redis-cli
    CC redis-benchmark.o
    LINK redis-benchmark
    INSTALL redis-check-rdb
    INSTALL redis-check-aof

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

make[1]: 离开目录“/opt/redis-6.2.3/src”
[root@localhost redis-6.2.3]#
```

5） 输入 `make install` 命令，将 redis 的命令安装到 Linux 中的环境中

```
[root@localhost redis-6.2.3]# make install
cd src && make install
make[1]: 进入目录“/opt/redis-6.2.3/src”
    CC Makefile.dep
make[1]: 离开目录“/opt/redis-6.2.3/src”
make[1]: 进入目录“/opt/redis-6.2.3/src”

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

    INSTALL redis-server
    INSTALL redis-benchmark
    INSTALL redis-cli
make[1]: 离开目录“/opt/redis-6.2.3/src”
[root@localhost redis-6.2.3]#
```

Redis默认的安装目录：`/usr/local/bin` -> `windows [C:\Program Files]`

```
[root@localhost bin]# ll
总用量 18872
-rwxr-xr-x. 1 root root 4829480 5月   8 15:15 redis-benchmark
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 5002896 5月   8 15:15 redis-cli
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 9484232 5月   8 15:15 redis-server
[root@localhost bin]#
```

```
redis-benchmark：测试 redis 的性能，一定要启动服务
redis-cli：客户端，用于连接 Redis 服务器
redis-server：服务端，用于启动 Redis 服务
redis-sentinel：哨兵命令
```

6）启动 Redis 的时候需要使用 `redis.conf` ，为了不改变原来的配置，将原配置文件（`/opt/redis-6.2.5/redis.conf`）备份一份。启动时使用备份的配置文件。

```
[root@localhost bin]# ll
总用量 18872
-rwxr-xr-x. 1 root root 4829480 5月   8 15:15 redis-benchmark
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 5002896 5月   8 15:15 redis-cli
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 9484232 5月   8 15:15 redis-server
[root@localhost bin]# cp /opt/redis-6.2.3/redis.conf redis.conf
[root@localhost bin]# ll
总用量 18964
-rwxr-xr-x. 1 root root 4829480 5月   8 15:15 redis-benchmark
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 5002896 5月   8 15:15 redis-cli
-rw-r--r--. 1 root root   93724 5月   8 15:27 redis.conf
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 9484232 5月   8 15:15 redis-server
[root@localhost bin]#
```

7）Redis 的服务默认不支持后台启动

```
[root@localhost bin]# redis-server redis.conf
18404:C 31 Aug 2021 15:29:27.873 # o000o000o000o Redis is starting o000o000o000o
18404:C 31 Aug 2021 15:29:27.873 # Redis version=6.2.5, bits=64, commit=00000000, modified=0, pid=18404, just started18404:C 31 Aug 202
1 15:29:27.873 # Configuration loaded
18404:M 31 Aug 2021 15:29:27.875 * Increased maximum number of open files to 10032 (it was originally set to 1024).
18404:M 31 Aug 2021 15:29:27.875 * monotonic clock: POSIX clock_gettime
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 6.2.5 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 18404
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           https://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'

18404:M 31 Aug 2021 15:29:27.880 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is s
et to the lower value of 128.
18404:M 31 Aug 2021 15:29:27.880 # Server initialized
18404:M 31 Aug 2021 15:29:27.880 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix t
his issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for thi
s to take effect.
```

修改`redis.conf`配置文件299行左右，将`daemonize no`修改为`daemonize yes`，

```
# tls-session-caching no

# Change the default number of TLS sessions cached. A zero value sets the cache
# to unlimited size. The default size is 20480.
#
# tls-session-cache-size 5000

# Change the default timeout of cached TLS sessions. The default timeout is 300
# seconds.
#
# tls-session-cache-timeout 60

################################# 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.
# When Redis is supervised by upstart or systemd, this parameter has no impact.
# daemonize no
daemonize yes

# If you run Redis from upstart or systemd, Redis can interact with your
# supervision tree. Options:
#   supervised no      - no supervision interaction
#   supervised upstart - signal upstart by putting Redis into SIGSTOP mode
#                        requires "expect stop" in your upstart job config
#   supervised systemd - signal systemd by writing READY=1 to $NOTIFY_SOCKET
#                        on startup, and updating Redis status on a regular
#                        basis.
#   supervised auto    - detect upstart or systemd method based on
#                        UPSTART_JOB or NOTIFY_SOCKET environment variables
# Note: these supervision methods only signal "process is ready."
#       They do not enable continuous pings back to your supervisor.
#
                                                                258,13        11%
```

8）启动Reids服务

```
# redis-server redis 配文件路径
redis-server redis.conf
redis-server /opt/xxx/.../redis.conf
```

```
[root@localhost bin]# ll
总用量 18964
-rwxr-xr-x. 1 root root 4829480 5月   8 15:15 redis-benchmark
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 5002896 5月   8 15:15 redis-cli
-rw-r--r--. 1 root root   93740 5月   8 15:34 redis.conf
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 9484232 5月   8 15:15 redis-server
[root@localhost bin]# redis-server redis.conf
[root@localhost bin]#
```

9）连接Redis

```
# redis-cli 参数

# 参数含义
-h <主机ip>，默认是127.0.0.1
-p <端口>，默认是6379
-a <密码>，如果redis加锁，需要传递密码
--help，显示帮助信息

# eg:
redis-cli # 不添加参数默认连接本机的服务器
redis-cli -h ip -p 6379

测试连接，输入ping，返回PONG表示连接成功！
```

```
[root@localhost bin]# ll
总用量 18964
-rwxr-xr-x. 1 root root 4829480 5月   8 15:15 redis-benchmark
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 5002896 5月   8 15:15 redis-cli
-rw-r--r--. 1 root root   93740 5月   8 15:34 redis.conf
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 9484232 5月   8 15:15 redis-server
[root@localhost bin]# redis-server redis.conf
[root@localhost bin]# redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
```

10）查看Redis的进程

```
ps -aux | grep redis
ps -ef | grep redis
```

```
[root@localhost ~]# ps -ef|grep redis
root      14590      1  0 15:36 ?        00:00:00 redis-server 127.0.0.1:6379
root      14905   9070  0 15:38 pts/1    00:00:00 redis-cli
root      14993  14952  0 15:42 pts/0    00:00:00 grep --color=auto redis
[root@localhost ~]#
```

11）连接上之后可以通过`shutdown`关闭Redis服务，然后再输入`exit`退出 Redis 命令行。

```
[root@localhost bin]# ll
总用量 18964
-rwxr-xr-x. 1 root root 4829480 5月   8 15:15 redis-benchmark
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 5002896 5月   8 15:15 redis-cli
-rw-r--r--. 1 root root   93740 5月   8 15:34 redis.conf
lrwxrwxrwx. 1 root root      12 5月   8 15:15 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 9484232 5月   8 15:15 redis-server
[root@localhost bin]# redis-server redis.conf
[root@localhost bin]# redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> shutdown
not connected> exit
[root@localhost bin]#
```

```
[root@localhost ~]# ps -ef|grep redis
root      15003  14952  0 15:43 pts/0    00:00:00 grep --color=auto redis
[root@localhost ~]#
```

（三）Redis前导信息

1、性能测试

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	

序号	选项	描述	默认值
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 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。	

```
[root@localhost bin]# redis-benchmark
====== PING_INLINE ======
  100000 requests completed in 1.83 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1
  host configuration "save": 3600 1 300 100 60 10000
  host configuration "appendonly": no
  multi-thread: no

Latency by percentile distribution:
0.000% <= 0.343 milliseconds (cumulative count 34)
50.000% <= 0.631 milliseconds (cumulative count 51520)
75.000% <= 0.751 milliseconds (cumulative count 75122)
87.500% <= 0.823 milliseconds (cumulative count 88499)
93.750% <= 0.895 milliseconds (cumulative count 93824)
96.875% <= 1.119 milliseconds (cumulative count 96880)
98.438% <= 1.423 milliseconds (cumulative count 98446)
99.219% <= 1.703 milliseconds (cumulative count 99225)
99.609% <= 1.871 milliseconds (cumulative count 99616)
99.805% <= 2.183 milliseconds (cumulative count 99805)
99.902% <= 2.823 milliseconds (cumulative count 99904)
99.951% <= 3.119 milliseconds (cumulative count 99952)
99.976% <= 3.311 milliseconds (cumulative count 99977)
99.988% <= 3.383 milliseconds (cumulative count 99988)
99.994% <= 3.455 milliseconds (cumulative count 99994)
99.997% <= 3.599 milliseconds (cumulative count 99997)
99.998% <= 3.623 milliseconds (cumulative count 99999)
99.999% <= 3.655 milliseconds (cumulative count 100000)
100.000% <= 3.655 milliseconds (cumulative count 100000)
```

2、redis的基本信息

- **Redis端口号：**6379
  - MySQL：3306；Oracle：1521；Tomcat：8080；浏览器：80
- **Redis数据库个数：**有16个数据库，数据库的ID从0开始
- Redis 是单线程的，Redis是基于内存操作，CPU的不决定Redis的性能，带宽和内存是决定 Redis 性能的关键

(四）配置Reids远程连接

1、Linux 防火墙放行 Redis 端口

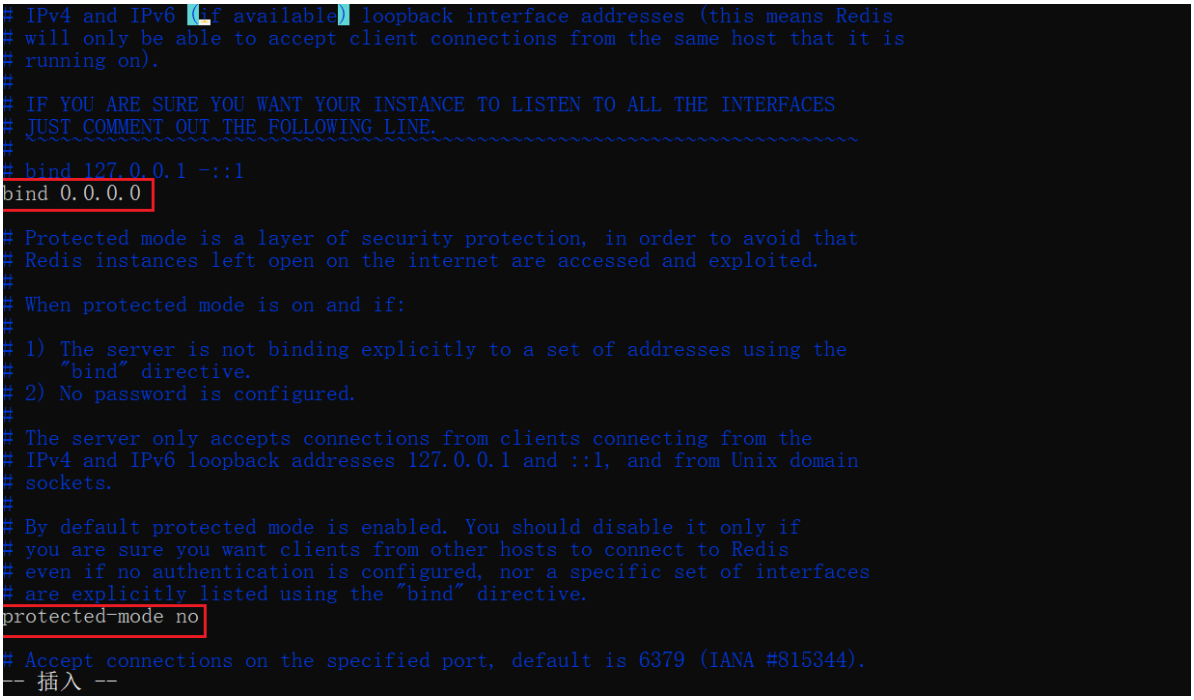
- 防火墙中添加 Redis 端口：`firewall-cmd --zone=public --add-port=6379/tcp --permanent`
- 刷新防火墙规则：`firewall-cmd --reload`
- 验证端口,查询防火墙开放端口：`firewall-cmd --zone=public --list-port`

2、修改Redis配置文件

- 关闭 redis 服务

```
# 关闭 redis 服务
shutdown
```

- 修改redis.conf配置文件




```
bind 127.0.0.1 -> bind 0.0.0.0              # 127.0.0.1仅支持本地连接，修改为0.0.0.0 可以支持远程连接
protected-mode yes -> protected-mode no     # 保护模式修改为no
```

- 测试连接：`redis-cli -h IP -p port`

```
 redis-cli -h 192.168.88.88 -p 6379
```

三、Redis-key操作

> Redis 是类似于 Map 类型的 K-V 键值对形式的存储方式
>
> 数据库一般都是增删改查：对 Key 的操作也是增删改查

```
# 删
del key                         # 删除key，对应的数据也被删除
move key db                     # 将当前的 key 移动到其他库（Reids 默认有16个库，0-15号库）
flushdb                         # 清空当前库的所有 key
flushall                        # 清空所有库的 key

# 改
expire key                      # 设置 key 的存活时间，存活时间以秒为单位。默认是永久存活。

# 查
```

```
exists key                          # 检查 key 是否存在，返回 1 表示存在，返回 0 表示不存在
ttl key                             # 查询 key 的剩余存活时间，-1：永久存在；-2：消亡
type key                            # 查询 key 对应值的数据类型
keys *                              # 查询当前库中所有的 key ，*代表通配符，所有的意思
dbsize                              # 查询当前库中 key 的个数
```

四、Redis 五大数据类型

（一）String(字符串)

> 类似 Java 中的 String 类型

```
# 增
set key value                   # 设置一个key
mset k1 v1 k2 v2 ...            # 同时设置多个值 moreSet

# 删
expire key                      # 设置key的存活时间为 -2 标识删除
del key                         # 删除key，对应的数据也被删除

# 改
append key                      # 拼接字符，返回拼接后的字符长度，如果当前key不存在，就创建一个key
getset key value                # 给个新的，返回旧的值

incr key                        # 自增，要求必须是数值，每次加1
decr key                        # 自减，要求必须是数值，每次减1
incrby key increment            # 设置增加的增量，要求必须是数值，increment 是增量。
decrby key increment            # 设置减少的增量，要求必须是数值，increment 是增量。

# 查
get key                         # 获取key对应的值
strlen key                      # 返回字符的长度（StringLength）
getrange key start end          # 截取并返回字符串，start、end是开始和结束下标，包头包尾。如果end是-1，返回从start开始到字符结尾
mget k1 k2 k3 k4 ...            # 同时获取多个值
```

（二）List(列表)

> 类似 Java 中的 List 集合，值可以重复，不会自动排序

```
# 增
lpush key value1 value2                         # 将一个或多个值插入到列表头部
rpush key value1 value2                         # 在列表中添加一个或多个值
linsert key before|after value newValue         # 在列表的指定元素前或者后插入新元素

# 删
lpop key [count]                # 删除List左边的第n个元素，返回删掉的值（List第一个元素）
rpop key [count]                # 删除List右边的第n个元素，返回删掉的值（List最后一个元素）
lrem key count value            # 移除列表n个数指定的元素
```

```
ltrim key start stop            # 对一个列表进行修剪(trim)，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。

# 改
lset key index value            # 通过索引设置列表元素的值，索引下标从0开始

# 查
lrange key start stop           # 获取指定范围内的元素， stop为 -1 时获取从 start 开始所有的值
lindex key index                # 获取索引位置的值，索引下标从0开始
llen key                        # 获取列表长度（元素个数）
```

(三) Hash(哈希)

> 类似于 Java 中的 Map，通过 Key-Value 存储

```
# 增
hset key field value            # 将哈希表 key 中的字段 field 的值设为 value

# 删
hdel key field1 [field2]        # 删除一个或多个哈希表字段，返回0表示没有指定的元素。

# 改
hset key field value            # 将哈希表 key 中的字段 field 的值设为 value 。
hmset key field1 v1 field2 v2   # 设置多个属性
hincrby key field increment     # 给key中指定的field自增，可以设置 increment 增量

# 查
hget key field                  # 获取指定key中的field属性的值
hgetall key                     # 获取指定key中所有的field以及value
hmget key field1 [field2]       # 获取指定key中n个field属性的值
```

(四) Set(集合)

> 类似于 Java 中的 TreeSet，默认按照字典顺序排序，升序，元素不可重复

```
# 增
sadd key v1 v2                  # 添加元素

# 删
spop key [count]                # 随机删除n个，并返回删除的值，count表示删除的个数

# 改
没有顺序没有位置，所以不能修改

# 查
scard key                       # 获取集合的成员数
smembers key                    # 查询key中所有的值
srandmember key count           # 返回集合中一个或多个随机数
```

（五）ZSet(有序集合)

> 在 Set 的基础上添加了一支持自定义排序的分值。

```
# 增
zadd key score v1                           # 添加一个值
zadd key score1 v1 score2 v2                # 添加多个值

# 删
zrem key member                             # 删除集合中指定的成员

# 改
zincrby key increment member                # 有序集合中对指定成员的分数(排名)加上增量 increment（正负均可）

# 查
zrange key start end                        # 返回有序集中指定区间内的成员，通过索引，分数从高到低(从小到大)
zrevrange key start end                     # 返回有序集中指定区间内的成员，通过索引，分数从高到低(从大到小)
zrangebyscore key min max                   # 通过分数返回有序集合指定区间内的成员，不带分数
zrangebyscore key min max withscores        # 通过分数返回有序集合指定区间内的成员，带分数
zcard key                                   # 获取集合中成员个数
zcount key min max                          # 计算在有序集合中指定区间分数的成员数
```

五、Redis持久化

> Redis 是基于**内存**的非关系型数据库，当机器断电或者进程关闭时内存中的数据是及其容易丢失的，所以为了将内存中的数据保存起来，Redis 提供了持久化功能，分别是 RDB 和 AOF 两种方案。

（一）RDB

> RDB（Rdies DataBase）持久化是指**在指定的时间间隔内**将内存中的数据集 快照 写入磁盘，也是**默认的持久化方式**，这种方式是就是将内存中数据以快照的方式写入到二进制文件中，默认的文件名为 **dump.rdb** 。
>
> Redis会单独创建（fork）一个子进程来进行持久化，会将数据写入到临时文件中，等待持久化结束之后，再用临时文件替换之前持久化的文件。整个过程，主进程不进行任何IO操作，确保了极高的性能。

Redis RDB配置:

```
################################ SNAPSHOTTING  ################################

# Save the DB to disk.
#
# save <seconds> <changes>
#
# Redis will save the DB if both the given number of seconds and the given
# number of write operations against the DB occurred.
```

```
#
# Snapshotting can be completely disabled with a single empty string argument
# as in following example:
# 关闭 RDB 持久化方式
# save ""
#
# Unless specified otherwise, by default Redis will save the DB:
#   * After 3600 seconds (an hour) if at least 1 key changed
#   * After 300 seconds (5 minutes) if at least 100 keys changed
#   * After 60 seconds if at least 10000 keys changed
#
# You can set these explicitly by uncommenting the three following lines.

# Redis默认持久化规则
# save 3600 1
# save 300 100
# save 60 10000

# By default Redis will stop accepting writes if RDB snapshots are enabled
# (at least one save point) and the latest background save failed.
# This will make the user aware (in a hard way) that data is not persisting
# on disk properly, otherwise chances are that no one will notice and some
# disaster will happen.
#
# If the background saving process will start working again Redis will
# automatically allow writes again.
#
# However if you have setup your proper monitoring of the Redis server
# and persistence, you may want to disable this feature so that Redis will
# continue to work as usual even if there are problems with disk,
# permissions, and so forth.

# 在异步保存发生错误的时候停止写入
stop-writes-on-bgsave-error yes

# Compress string objects using LZF when dump .rdb databases?
# By default compression is enabled as it's almost always a win.
# If you want to save some CPU in the saving child set it to 'no' but
# the dataset will likely be bigger if you have compressible values or keys.

# 指定存储至本地数据库时是否压缩数据，默认是yes，redis采用LZF压缩，如果为了节省CPU时间
# 可以关闭该选项，但会导致数据库文件变的巨大
rdbcompression yes
# Since version 5 of RDB a CRC64 checksum is placed at the end of the file.
# This makes the format more resistant to corruption but there is a performance
# hit to pay (around 10%) when saving and loading RDB files, so you can disable it
# for maximum performances.
#
# RDB files created with checksum disabled have a checksum of zero that will
# tell the loading code to skip the check.
```

```
59
60  # 对rdb文件进行校验
61  rdbchecksum yes
62
63  # Enables or disables full sanitation checks for ziplist and listpack etc
    when
64  # loading an RDB or RESTORE payload. This reduces the chances of a
    assertion or
65  # crash later on while processing commands.
66  # Options:
67  #   no         - Never perform full sanitation
68  #   yes        - Always perform full sanitation
69  #   clients    - Perform full sanitation only for user connections.
70  #                Excludes: RDB files, RESTORE commands received from the
    master
71  #                connection, and client connections which have the
72  #                skip-sanitize-payload ACL flag.
73  # The default should be 'clients' but since it currently affects cluster
74  # resharding via MIGRATE, it is temporarily set to 'no' by default.
75  #
76  # sanitize-dump-payload no
77
78  # The filename where to dump the DB
79
80  # Reids默认rdb文件
81  dbfilename dump.rdb
82
83  # Remove RDB files used by replication in instances without persistence
84  # enabled. By default this option is disabled, however there are
    environments
85  # where for regulations or other security concerns, RDB files persisted on
86  # disk by masters in order to feed replicas, or stored on disk by replicas
87  # in order to load them for the initial synchronization, should be deleted
88  # ASAP. Note that this option ONLY WORKS in instances that have both AOF
89  # and RDB persistence disabled, otherwise is completely ignored.
90  #
91  # An alternative (and sometimes better) way to obtain the same effect is
92  # to use diskless replication on both master and replicas instances.
    However
93  # in the case of replicas, diskless is not always an option.
94  rdb-del-sync-files no
95
96  # The working directory.
97  #
98  # The DB will be written inside this directory, with the filename specified
99  # above using the 'dbfilename' configuration directive.
100 #
101 # The Append Only File will also be created inside this directory.
102 #
103 # Note that you must specify a directory here, not a file name.
104
105 # 持久化文件存放的位置，在配置文件所在目录下
106 dir ./
```

重要配置摘要：

```
# 关闭 RDB 持久化方式
# save ''

# Redis 默认持久化规则
# save 秒 操作的键的个数
# save 3600 1
# save 300 100
# save 60 10000

# Reids 默认 rdb 文件，可以自定义名称
dbfilename dump.rdb

# 持久化文件存放的位置，在配置文件所在目录下
dir ./
```

RDB 触发快照条件：

1. 在指定的时间间隔内，执行指定次数的写操作
2. 执行save（阻塞， 只管保存快照，其他的等待） 或者是bgsave （异步）命令
3. 执行flushall 命令，清空数据库所有数据。
4. 执行shutdown 命令，保证服务器正常关闭且不丢失任何数据。

RDB 恢复：

> 将 dump.rdb 文件拷贝到 redis 的配置文件所在的目录的目录下，重启 redis 服务即可。
>
> **在实际开发中，一般会考虑到物理机硬盘损坏情况，选择备份dump.rdb 。**

RDB 优缺点：

优点：

1. 适合大规模的数据恢复。
2. 如果业务对数据完整性和一致性要求不高，RDB是很好的选择。

缺点：

1. 数据的完整性和一致性不高，因为RDB可能在最后一次备份时宕机了。
2. 备份时占用内存，因为Redis 在备份时会独立创建一个子进程，将数据写入到一个临时文件（此时内存中的数据是原来的两倍），最后再将临时文件替换之前的备份文件。

（二）AOF

> Redis 默认不开启AOF（Append Only File）。它的出现是为了弥补RDB的不足（数据的不一致性），所以它采用日志的形式来记录每个**写操作**，并**追加**到文件中。Redis 重启的会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。默认的文件名为**appendonly.aof**

Redis AOF配置：

```
############################## 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 https://redis.io/topics/persistence for more information.

# AOF模式默认no关闭状态，需要开启时设置为yes
appendonly no

# The name of the append only file (default: "appendonly.aof")

# AOF默认保存文件
appendfilename "appendonly.aof"

# The fsync() call tells the Operating System to actually write data on
disk
# instead of waiting for more data in the output buffer. Some OS will
really flush
# data on disk, some other OS will just try to do it ASAP.
#
# Redis supports three different modes:
#
# no: don't fsync, just let the OS flush the data when it wants. Faster.
# always: fsync after every write to the append only log. Slow, Safest.
# everysec: fsync only one time every second. Compromise.
#
# The default is "everysec", as that's usually the right compromise between
# speed and data safety. It's up to you to understand if you can relax this
to
# "no" that will let the operating system flush the output buffer when
# it wants, for better performances (but if you can live with the idea of
# some data loss consider the default persistence mode that's
snapshotting),
# or on the contrary, use "always" that's very slow but a bit safer than
# everysec.
#
# More details please check the following article:
# http://antirez.com/post/redis-persistence-demystified.html
#
# If unsure, use "everysec".

# 总是写入aof文件，并完成磁盘同步
# appendfsync always
# 每秒钟写一次
appendfsync everysec
# 不主动进行同步操作，而是交由操作系统来做（每30秒一次）
```

```
# appendfsync no

# When the AOF fsync policy is set to always or everysec, and a background
# saving process (a background save or AOF log background rewriting) is
# performing a lot of I/O against the disk, in some Linux configurations
# Redis may block too long on the fsync() call. Note that there is no fix
for
# this currently, as even performing fsync in a different thread will block
# our synchronous write(2) call.
#
# In order to mitigate this problem it's possible to use the following
option
# that will prevent fsync() from being called in the main process while a
# BGSAVE or BGREWRITEAOF is in progress.
#
# This means that while another child is saving, the durability of Redis is
# the same as "appendfsync none". In practical terms, this means that it is
# possible to lose up to 30 seconds of log in the worst scenario (with the
# default Linux settings).
#
# If you have latency problems turn this to "yes". Otherwise leave it as
# "no" that is the safest pick from the point of view of durability.

# 重写时是否可以运用Appendfsync，用默认no即可，保证数据安全性。
no-appendfsync-on-rewrite no

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

# 设置重写的增长比例，指当前aof文件比上次重写的增长比例大小
auto-aof-rewrite-percentage 100
# 设置重写的最小大小，小于配置大小时不重写
auto-aof-rewrite-min-size 64mb

# An AOF file may be found to be truncated at the end during the Redis
# startup process, when the AOF data gets loaded back into memory.
# This may happen when the system where Redis is running
# crashes, especially when an ext4 filesystem is mounted without the
# data=ordered option (however this can't happen when Redis itself
# crashes or aborts but the operating system still works correctly).
#
```

```
# Redis can either exit with an error when this happens, or load as much
# data as possible (the default now) and start if the AOF file is found
# to be truncated at the end. The following option controls this behavior.
#
# If aof-load-truncated is set to yes, a truncated AOF file is loaded and
# the Redis server starts emitting a log to inform the user of the event.
# Otherwise if the option is set to no, the server aborts with an error
# and refuses to start. When the option is set to no, the user requires
# to fix the AOF file using the "redis-check-aof" utility before to restart
# the server.
#
# Note that if the AOF file will be found to be corrupted in the middle
# the server will still exit with an error. This option only applies when
# Redis will try to read more data from the AOF file but not enough bytes
# will be found.
aof-load-truncated yes

# When rewriting the AOF file, Redis is able to use an RDB preamble in the
# AOF file for faster rewrites and recoveries. When this option is turned
# on the rewritten AOF file is composed of two different stanzas:
#
#   [RDB file][AOF tail]
#
# When loading, Redis recognizes that the AOF file starts with the "REDIS"
# string and loads the prefixed RDB file, then continues loading the AOF
# tail.
aof-use-rdb-preamble yes
```

重要配置摘要：

```
# AOF模式默认no关闭状态，需要开启时设置为 yes
appendonly no

# AOF默认保存文件，可以自定义名称
appendfilename "appendonly.aof"

# appendfsync 配置
# 总是写入aof文件，并完成磁盘同步
# appendfsync always
# 每秒钟写一次
appendfsync everysec
# 不主动进行同步操作，而是交由操作系统来做（每30秒一次）
# appendfsync no
```

AOF优缺点：

优点：

1. 每次写操作都是对AOF文件的同步，文件的完整性高；
2. 默认配置每秒同步一次，数据完整性高，最多会丢失1s的数据；

缺点：

1. 相同的数据采集AOF文件要比RDB的文件大得多，修复会比较慢；
2. AOF的执行效率比RDB要低，所以 Redis 默认关闭 AOF；

六、Redis 集群

（一）概念

> Reids 集群，是指将一台Redis服务器的数据，复制到其他Redis服务器。前者称为主节点（Master、leader），后者成为从节点（slave、follower）；
>
> 数据的复制是单向的，只能从主节点到从节点；（80%的需求是以读为主，所以主节点写；从节点：读；）
>
> 默认情况下，每一台Redis服务器都是作为主节点；且主节点可以有多个从节点，但是一个从节点只能有一个主节点。

集群作用

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. 高可用基石：除了上述作用以外，主从复制还是哨兵模式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。

（二）环境配置

主从复制只配置从节点，不配置主节点

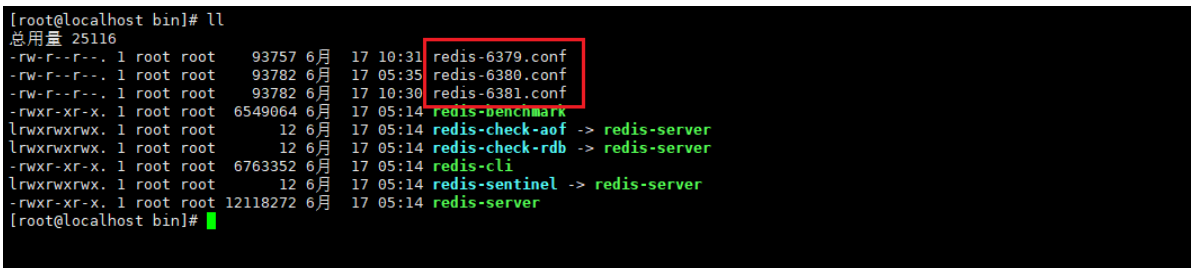
> 1、多个系统分别装Redis
>
> 2、Redis启动和端口挂钩，在一个其上开不同的端口
>
> 6379：主
>
> 6380、6380：从

查看配置信息

```
127.0.0.1:6379> info replication
# Replication
role:master                  # 角色，当前是主库
connected_slaves:2           # 连接的从库有几个，以下是从节点信息
slave0:ip=127.0.0.1,port=6381,state=online,offset=4249,lag=0
slave1:ip=127.0.0.1,port=6380,state=online,offset=4249,lag=0
```

修改配置文件

1、将默认redis配置文件复制



- 主：redis-6379.conf
- 从1：redis-6380.conf
- 从2：redis-6381.conf

2、修改主从配置

- 端口号 `port 6379`
- 日志文件 `logfile "log-6379.log"`
- RDB/AOF文件名 `dbfilename dump-6379.rdb`
- pid文件`pidfile /var/run/redis_6379.pid`

3、修改完成之后启动服务，通过`ps -ef | grep redis`查看启动进程

```
[root@localhost redis-conf]# redis-server redis-6379.conf
[root@localhost redis-conf]# redis-server redis-6381.conf
[root@localhost redis-conf]# redis-server redis-6380.conf
[root@localhost redis-conf]# ps -ef | grep redis
root      14206      1  0 17:49 ?        00:00:00 redis-server 127.0.0.1:6379
root      14212      1  0 17:49 ?        00:00:00 redis-server 127.0.0.1:6381
root      14218      1  0 17:49 ?        00:00:00 redis-server 127.0.0.1:6380
root      14224  11494  0 17:49 pts/0    00:00:00 grep --color=auto redis
```

配置从节点

1、redis配置文件配置主节点

```
replicaof <masterip> <masterport>
```

```
# Note that you must specify a directory here, not a file name.
dir ./

################################## REPLICATION #################################

# Master-Replica replication. Use replicaof to make a Redis instance a copy of
# another Redis server. A few things to understand ASAP about Redis replication.
#
#   +------------------+      +---------------+
#   |      Master      | ---> |    Replica    |
#   | (receive writes) |      |  (exact copy) |
#   +------------------+      +---------------+
#
# 1) Redis replication is asynchronous, but you can configure a master to
#    stop accepting writes if it appears to be not connected with at least
#    a given number of replicas.
# 2) Redis replicas are able to perform a partial resynchronization with the
#    master if the replication link is lost for a relatively small amount of
#    time. You may want to configure the replication backlog size (see the next
#    sections of this file) with a sensible value depending on your needs.
# 3) Replication is automatic and does not need user intervention. After a
#    network partition replicas automatically try to reconnect to masters
#    and resynchronize with them.
#
# replicaof <masterip> <masterport>
replicaof 127.0.0.1 6379

# If the master is password protected (using the "requirepass" configuration
# directive below) it is possible to tell the replica to authenticate before
# starting the replication synchronization process, otherwise the master will
# refuse the replica request.
#
# masterauth <master-password>
#
# However this is not enough if you are using Redis ACLs (for Redis version
# 6 or greater), and the default user is not capable of running the PSYNC
# command and/or other commands needed for replication. In this case it's
# better to configure a special user to use with replication, and specify the
# masteruser configuration as such:
#
# masteruser <username>
                                                              489,1         22%
```

2、启动命令

```
redis-server --replicaof <masterip> <masterport>
```

3、redis-cli 从节点客户端命令配置主节点信息

```
slaveof <masterip> <masterport>
slaveof 127.0.0.1 6379
```

4、删除所有子节点

```
slaveof no one
```

5、查看节点的信息

```
info replication
```

注意：

1、主节点可以读写、从节点只能读取；

```
127.0.0.1:6380> keys *
1) "k1"
127.0.0.1:6380> get k1
"v1"
127.0.0.1:6380> set k2 v2
(error) READONLY You can't write against a read only replica.
127.0.0.1:6380>
```

2、通过客户端命令配置的从机，在重启之后会变成主机；只要变成从机，会自动从主机中获取数据；

（三）复制原理

> 从机连接到主机后会发送一个同步命令，主机接收到同步命令，启动后台的存盘进程，同时收集所有接受到的用于修改数据集命令，在后台进程执行完毕后，主机将传送整个数据到从机，完成一次完全同步；

> **全量复制：**用于初次复制或其它无法进行部分复制的情况，将主节点中的所有数据都发送给从节点。当数据量过大的时候，会造成很大的网络开销。

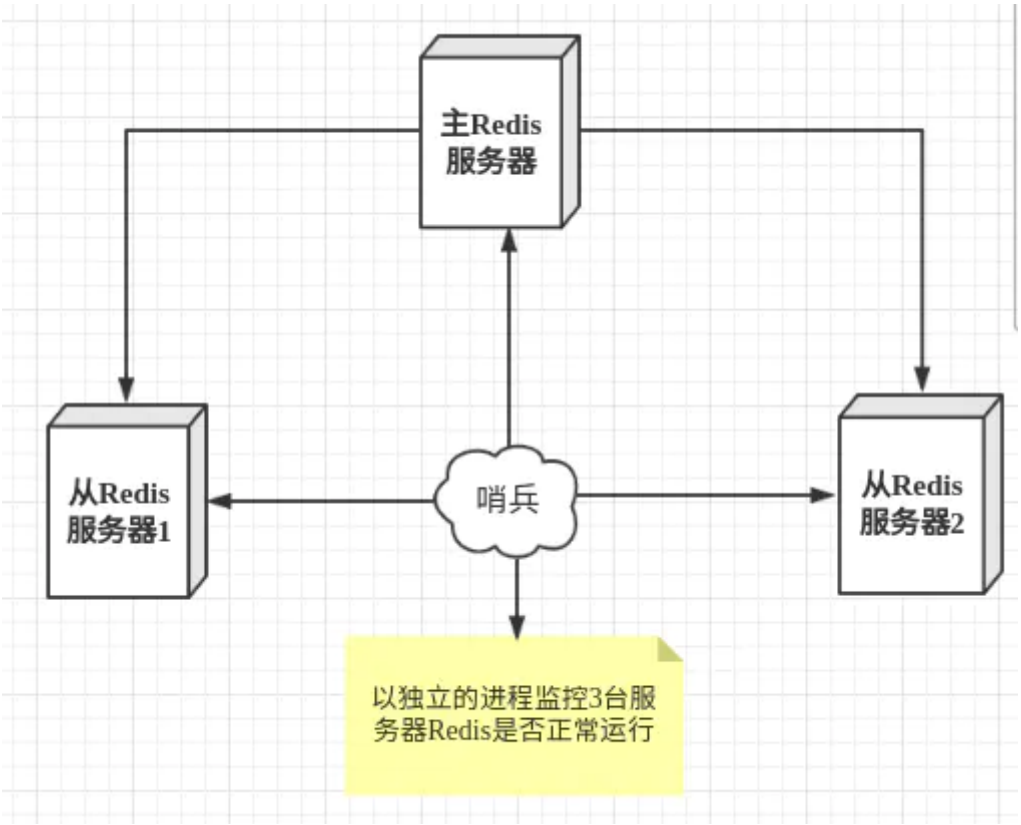
> **增量复制：**用于处理在主从复制中因网络闪退等原因造成数据丢失场景，当从节点再次连上主节点，如果条件允许，主节点会补发丢失数据给从节点，因为补发的数据远远小于全量数据，可以有效避免全量复制的过高开销。如果网络中断时间过长，造成主节点没有能够完整地保存中断期间执行的写命令，则无法进行部分复制，仍使用全量复制 。

七、Redis哨兵模式

（一）哨兵模式概念

哨兵模式概念：

> 哨兵模式是一种特殊的模式，Redis 提供了哨兵的命令（redis-sentinel），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。**主要用于集群环境下。

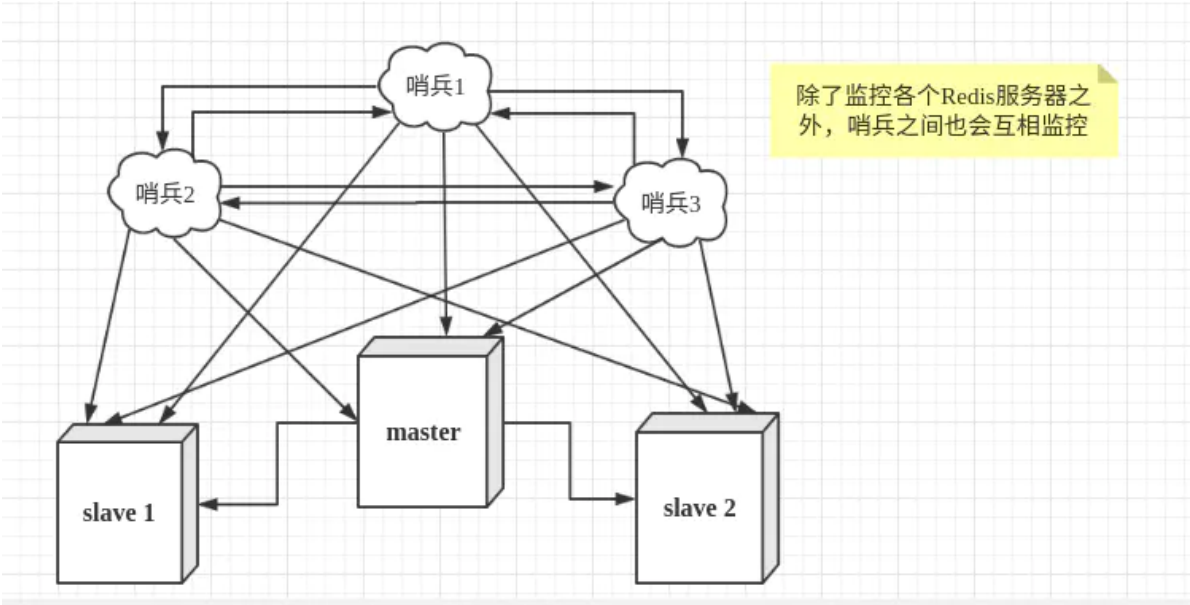


哨兵模式作用：

- 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 主节点（master） 宕机，会自动将 子节点（slave）切换成主节点（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。

多哨兵模式：

> 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。



> 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故障转移（failover）过程，仅仅是哨兵1主观的认为主服务器不可用，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，进行故障转移（failover）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。**

（二）哨兵模式配置

> redis 安装目录下 sentinel.conf 文件用于配置哨兵模式

```
总用量 236
-rw-rw-r--.  1 root root 28118 6月   1 10:03 00-RELEASENOTES
-rw-rw-r--.  1 root root    51 6月   1 10:03 BUGS
-rw-rw-r--.  1 root root  5026 6月   1 10:03 CONDUCT
-rw-rw-r--.  1 root root  3384 6月   1 10:03 CONTRIBUTING
-rw-rw-r--.  1 root root  1487 6月   1 10:03 COPYING
drwxrwxr-x.  7 root root   213 6月  17 05:12 deps
-rw-rw-r--.  1 root root    11 6月   1 10:03 INSTALL
-rw-rw-r--.  1 root root   151 6月   1 10:03 Makefile
-rw-rw-r--.  1 root root  6888 6月   1 10:03 MANIFESTO
-rw-rw-r--.  1 root root 21567 6月   1 10:03 README.md
-rw-rw-r--.  1 root root 93724 6月   1 10:03 redis.conf
-rwxrwxr-x.  1 root root   275 6月   1 10:03 runtest
-rwxrwxr-x.  1 root root   279 6月   1 10:03 runtest-cluster
-rwxrwxr-x.  1 root root  1046 6月   1 10:03 runtest-moduleapi
-rwxrwxr-x.  1 root root   281 6月   1 10:03 runtest-sentinel
-rw-rw-r--.  1 root root 13768 6月   1 10:03 sentinel.conf
drwxrwxr-x.  3 root root 12288 6月  17 05:14 src
drwxrwxr-x. 11 root root   182 6月   1 10:03 tests
-rw-rw-r--.  1 root root  3055 6月   1 10:03 TLS.md
drwxrwxr-x.  9 root root  4096 6月   1 10:03 utils
```

基础配置：

```
# sentinel monitor <master name> <ip> <redis port> <quorum>
# sentinel monitor 哨兵监视
# 告诉sentinel去监听地址为ip:port的一个 master，
# <master name>可以自定义，master-name只能包含英文字母，数字，和“.-_”这三个字符
# <quorum>是一个数字，指明当有多少个sentinel（哨兵）认为一个master（主机）失效时，master（主机）才算真正失效。
# <ip>需要注意的是master-ip 要写真实的ip地址而不要用回环地址（127.0.0.1）。
sentinel monitor mymaster 127.0.0.1 6379 2
```

```
# 启动哨兵模式
redis-sentinel sentinel.conf
```

```
[root@localhost bin]# redis-sentinel sentinel.conf
2438:X 21 Jun 2021 10:17:05.066 # o000o000o000o Redis is starting o000o000o000o
2438:X 21 Jun 2021 10:17:05.066 # Redis version=6.2.4, bits=64, commit=00000000, modified=0, pid=2438, just started
2438:X 21 Jun 2021 10:17:05.066 # Configuration loaded
2438:X 21 Jun 2021 10:17:05.067 * Increased maximum number of open files to 10032 (it was originally set to 1024).
2438:X 21 Jun 2021 10:17:05.067 * monotonic clock: POSIX clock_gettime
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 6.2.4 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in sentinel mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 26379
      |    `-._   `._    /     _.-'    |     PID: 2438
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |          https://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'

2438:X 21 Jun 2021 10:17:05.067 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
2438:X 21 Jun 2021 10:17:05.067 # Sentinel ID is 78c55b9952e00d1e9a33ed804936fee77d29038c
2438:X 21 Jun 2021 10:17:05.067 # +monitor master mymaster 127.0.0.1 6379 quorum 2
2438:X 21 Jun 2021 10:17:15.131 * +convert-to-slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
2438:X 21 Jun 2021 10:17:15.131 * +convert-to-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
```

实验步骤：

1、启动三个redis服务器，并查看主从关系

```
6379：主节点
6380：从节点1
6381：从节点2

下图中可以看到启动了三个进程，连接后可以分别看到当前数据库的信息
```

```
[root@localhost bin]# ps -ef | grep redis
root        3226       1  0 11:14 ?        00:00:00 redis-server 127.0.0.1:6379
root        3234       1  0 11:14 ?        00:00:00 redis-server 127.0.0.1:6380
root        3250       1  0 11:14 ?        00:00:00 redis-server 127.0.0.1:6381
root        3257    2503  0 11:14 pts/3    00:00:00 redis-cli -p 6379
root        3258    2168  0 11:14 pts/1    00:00:00 redis-cli -p 6380
root        3259    2181  0 11:15 pts/2    00:00:00 redis-cli -p 6381
root        3272    2169  0 11:15 pts/0    00:00:00 grep --color=auto redis
[root@localhost bin]#
```

```
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=252,lag=0
slave1:ip=127.0.0.1,port=6381,state=online,offset=252,lag=0
master_failover_state:no-failover
master_replid:878bc89797523447f78f757db16f7a3672317bc7
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:252
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:239
repl_backlog_histlen:14
127.0.0.1:6379>
```

```
127.0.0.1:6380> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:7
master_sync_in_progress:0
slave_repl_offset:294
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:878bc89797523447f78f757db16f7a3672317bc7
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:294
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:239
repl_backlog_histlen:56
127.0.0.1:6380>
```

```
127.0.0.1:6381> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:2
master_sync_in_progress:0
slave_repl_offset:322
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:878bc89797523447f78f757db16f7a3672317bc7
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:322
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:253
repl_backlog_histlen:70
127.0.0.1:6381>
```

2、启动哨兵进程，查看日志信息中的主从关系

> 哨兵模式启动，可以看到一个主节点和两个从节点的信息

```
[root@localhost bin]# redis-sentinel sentinel.conf
4778:X 21 Jun 2021 13:48:03.445 # o000o000o000o Redis is starting o000o000o000o
4778:X 21 Jun 2021 13:48:03.445 # Redis version=6.2.4, bits=64, commit=00000000, modified=0, pid=4778, just started
4778:X 21 Jun 2021 13:48:03.445 # Configuration loaded
4778:X 21 Jun 2021 13:48:03.446 * Increased maximum number of open files to 10032 (it was originally set to 1024).
4778:X 21 Jun 2021 13:48:03.446 * monotonic clock: POSIX clock_gettime
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 6.2.4 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in sentinel mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 26379
      |    `-._   `._    /     _.-'    |     PID: 4778
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           https://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'

4778:X 21 Jun 2021 13:48:03.447 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
4778:X 21 Jun 2021 13:48:03.449 # Sentinel ID is 48c9a7d48d8018a8bffdcb1cff70bc66a386632c
4778:X 21 Jun 2021 13:48:03.449 # +monitor master mymaster 127.0.0.1 6379 quorum 1
4778:X 21 Jun 2021 13:48:03.450 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:48:03.451 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
```

3、关闭6379主节点，查看哨兵日志和从节点信息

> 关闭主节点，等待哨兵执行投票机制后，将从库6380切换为主库，原6379主库变为从库，原6381依旧是从库不发生改变；

```
                https://redis.io

4778:X 21 Jun 2021 13:48:03.447 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
4778:X 21 Jun 2021 13:48:03.449 # Sentinel ID is 48c9a7d48d8018a8bffdcb1cff70bc66a386632c
4778:X 21 Jun 2021 13:48:03.449 # +monitor master mymaster 127.0.0.1 6379 quorum 1
4778:X 21 Jun 2021 13:48:03.450 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:48:03.451 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
^[OQ^H^H^H^H^H^H^H4778:X 21 Jun 2021 13:49:16.403 # +sdown master mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:16.403 # +odown master mymaster 127.0.0.1 6379 #quorum 1/1
4778:X 21 Jun 2021 13:49:16.403 # +new-epoch 1
4778:X 21 Jun 2021 13:49:16.403 # +try-failover master mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:16.405 # +vote-for-leader 48c9a7d48d8018a8bffdcb1cff70bc66a386632c 1
4778:X 21 Jun 2021 13:49:16.405 # +elected-leader master mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:16.405 # +failover-state-select-slave master mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:16.508 # +selected-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:16.508 * +failover-state-send-slaveof-noone slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:16.592 * +failover-state-wait-promotion slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:16.598 # +promoted-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:16.599 # +failover-state-reconf-slaves master mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:16.678 * +slave-reconf-sent slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:17.637 * +slave-reconf-inprog slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:17.637 * +slave-reconf-done slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:17.716 # +failover-end master mymaster 127.0.0.1 6379
4778:X 21 Jun 2021 13:49:17.716 # +switch-master mymaster 127.0.0.1 6379 127.0.0.1 6380
4778:X 21 Jun 2021 13:49:17.717 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6380
4778:X 21 Jun 2021 13:49:17.717 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6380
4778:X 21 Jun 2021 13:49:47.771 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6380
```

```
127.0.0.1:6380> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6381,state=online,offset=99561,lag=0
master_failover_state:no-failover
master_replid:1eca4c28f4c380c4fb9475449cdce63f55f2aaea
master_replid2:a4b151e19bcd1fed6f749e63061b069b5a6f83c7
master_repl_offset:99561
second_repl_offset:93708
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:90885
repl_backlog_histlen:8677
127.0.0.1:6380>
```

```
127.0.0.1:6381> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6380
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_repl_offset:100373
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:1eca4c28f4c380c4fb9475449cdce63f55f2aaea
master_replid2:a4b151e19bcd1fed6f749e63061b069b5a6f83c7
master_repl_offset:100373
second_repl_offset:93708
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:90885
repl_backlog_histlen:9489
127.0.0.1:6381>
```

4、重新启动6379主节点，查看哨兵日志和从节点信息

1. 重启6379节点，查看信息展示当前节点为从节点；

```
[root@localhost bin]# redis-server redis-6379.conf
[root@localhost bin]# redis-cli -p 6379
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6380
master_link_status:up
master_last_io_seconds_ago:0
master_sync_in_progress:0
slave_repl_offset:106218
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:1eca4c28f4c380c4fb9475449cdce63f55f2aaea
master_replid2:a4b151e19bcd1fed6f749e63061b069b5a6f83c7
master_repl_offset:106218
second_repl_offset:93708
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:93708
repl_backlog_histlen:12511
127.0.0.1:6379>
```

配置详解：

```
# Example sentinel.conf

# *** IMPORTANT ***
#
# 默认情况下，Sentinel将无法从不同于的接口访问localhost，或者使用“bind”指令绑定到网络
# 接口，或通过“protected mode no”禁用保护模式将其添加到此配置文件。
# 在执行此操作之前，请确保实例受到外部保护通过防火墙或其他方式。
```

```
# 例如，您可以使用以下选项之一：
#
# bind 127.0.0.1 192.168.1.1

# 受保护模式
# protected-mode no

# 哨兵运行端口
# port <sentinel-port>
port 26379

# 默认情况下，Redis Sentinel不作为守护进程运行。如果需要，请使用“是”。
# 注意，Redis会在/var/run/Redis-sentinel.pid中写入一个pid文件。
daemonize no

# 运行守护模式时，Redis Sentinel在默认情况下为/var/run/redis-sentinel.pid。可以指定自定义pid文件
pidfile /var/run/redis-sentinel.pid

# 指定日志文件名。空字符串也可以用来强制
# Sentinel登录标准输出。请注意，如果您使用标准
# 日志输出但是daemonize，日志将被发送到/dev/null
logfile ""

# sentinel announce-ip <ip>
# sentinel announce-port <port>

# 上述两个配置指令在以下环境中很有用：
# 由于NAT，Sentinel可以通过非本地地址从外部访问。当提供announce-ip时，Sentinel将在通信中声明指定的IP地址，而不是像通常那样自动检测本地地址。
# 类似地，当提供了announce端口且该端口有效且非零时，Sentinel将宣布指定的TCP端口。
# 这两个选项不需要一起使用，如果只需要使用announceip前提是，Sentinel将宣布指定的IP和服务器端口由“端口”选项指定。如果只提供了公告端口，
# 则Sentinel将宣布自动检测到的本地IP和指定的端口。

# 例如：
# sentinel announce-ip 1.2.3.4

# dir <working-directory>
# 每个长时间运行的进程都应该有一个定义良好的工作目录。对于Redis Sentinel来说，启动时chdir到/tmp是最简单的事情使流程不干扰管理任务
dir /tmp

# sentinel monitor <master name> <ip> <redis port> <quorum>

# 告诉sentinel去监听地址为ip:port的一个master，这里的master-name可以自定义，quorum是一个数字，指明当有多少个sentinel认为一个master失效时，
# master才算真正失效。master-name只能包含英文字母，数字，和“.-_”这三个字符需要注意的是master-ip 要写真实的ip地址而不要用回环地址（127.0.0.1）。
sentinel monitor mymaster 127.0.0.1 6379 2

# sentinel auth-pass <master-name> <password>

# 设置用于向主副本和副本进行身份验证的密码。如果Redis实例中设置了要监视的密码，则非常有用。
```

```
# 请注意，主密码也用于副本，因此不是可以在主机和副本实例中设置不同的密码
# 如果您希望能够使用Sentinel监视这些实例。但是，您可以在不启用身份验证的情况下使用Redis实例
# 与需要身份验证的Redis实例混合（只要对于所有需要密码的实例，密码设置都是相同的）
# AUTH命令在具有身份验证的Redis实例中无效已关闭。
#
# 例如:
# sentinel auth-pass mymaster MySUPER--secret-0123passw0rd

# sentinel auth-user <master-name> <username>
#
# 这有助于对具有 ACL 功能的实例进行身份验证，即运行 Redis 6.0 或更高版本。
# 当仅提供 auth-pass 时，Sentinel 实例将使用旧的“AUTH <pass>”方法向 Redis 进行身份验证。
# 当还提供用户名时，它将使用“AUTH <user> <pass>”。 在 Redis 服务器端，ACL 只提供对 Sentinel 实例的最小访问，
# 应该按照以下几行进行配置:
#
#     user sentinel-user >somepassword +client +subscribe +publish \
#                        +ping +info +multi +slaveof +config +client +exec on

# sentinel down-after-milliseconds <master-name> <milliseconds>
#
# 主节点或副本在指定时间内没有回复PING，便认为该节点为主观下线 S_DOWN 状态。
# 默认是30秒。
sentinel down-after-milliseconds mymaster 30000

# 重要提示：从 Redis 6.2 开始，Sentinel 模式支持 ACL 功能，请参阅 Redis 网站 https://redis.io/topics/acl 了解更多详细信息。

# Sentinel 的 ACL 用户按以下格式定义:
#
#   user <username> ... acl rules ...
#
# 例如:
#
#   user worker +@admin +@connection ~* on >ffa9203c493aa99
#
# 有关 ACL 配置的更多信息，请参阅 Redis 网站 https://redis.io/topics/acl 和 redis 服务器配置模板 redis.conf。
# ACL LOG
#
# ACL日志跟踪失败的命令和关联的身份验证事件使用ACL。ACL日志有助于对阻止的失败命令进行故障排除
# 通过ACL。ACL日志存储在内存中。您可以使用ACL日志重置。在下面定义ACL日志的最大条目长度。
acllog-max-len 128

# 使用外部ACL文件
# 可以使用仅列出用户的独立文件。这两种方法不能混用:
# 如果在此处配置用户，同时激活外部ACL文件，服务器将拒绝启动。
# 外部ACL用户文件的格式与redis.conf中用于描述用户的格式。
#
# aclfile /etc/redis/sentinel-users.acl
```

```
103
104 # requirepass <password>
105 #
106 # 您可以将 Sentinel 本身配置为需要密码，但是这样做时 Sentinel 将尝试使用相同的密码对所有其他 Sentinel 进行身份验证。
107 # 因此，您需要使用相同的“requirepass”密码配置给定组中的所有 Sentinel。
108 #
109 # 重要提示：从 Redis 6.2 开始，“requirepass”是 ACL 系统之上的兼容层。 选项效果只是为默认用户设置密码。
110 # 客户端仍然会像往常一样使用 AUTH <password> 进行身份验证，或者更明确地使用 AUTH default <password> 如果他们遵循新协议：两者都可以工作。
111 # 建议新配置文件对传入连接（通过 ACL）和传出连接（viasentinel-user 和 sentinel-pass）使用单独的身份验证控制
112 # requirepass 与 aclfile 选项和 ACL LOAD 命令不兼容，这些将导致 requirepass 被忽略。
113
114 # sentinel sentinel-user <username>
115 # 您可以将 Sentinel 配置为使用特定用户名与其他 Sentinel 进行身份验证。
116
117 # sentinel sentinel-pass <password>
118 #
119 # Sentinel 与其他 Sentinel 进行身份验证的密码。 如果 sentinel-user 未配置，Sentinel 将使用带有 sentinel-pass 的“默认”用户进行身份验证。
120
121 # sentinel parallel-syncs <master-name> <numreplicas>
122 #
123 # 在故障转移期间，我们可以重新配置多少副本以同时指向新副本。 如果您使用副本来提供查询服务，请使用较小的数字，
124 # 以避免在与主服务器执行同步时几乎同时无法访问所有副本。
125 sentinel parallel-syncs mymaster 1
126
127 # sentinel failover-timeout <master-name> <milliseconds>
128 #
129 # 以毫秒为单位指定故障转移超时。 它以多种方式使用：
130 # - 在上一次故障转移后重新启动故障转移所需的时间是已经由给定的 Sentinel 尝试对抗同一个 master，是两个次故障转移超时。
131
132 # -副本复制到错误的主节点所需的时间到 Sentinel 当前配置，强制复制使用正确的主人，正是故障转移超时（从Sentinel 检测到错误配置的那一刻）。
133
134 # -取消已经在进行中的故障转移所需的时间没有产生任何配置更改（SLAVEOF NO ONE 还没有被提升的副本确认）。
135
136 # -正在进行的故障转移等待所有副本完成的最长时间重新配置为新主服务器的副本。 然而即使在这个时间之后无论如何，副本将由哨兵重新配置，
137 # 但不会指定的精确并行同步进程。
138 #
139 # Default is 3 minutes.
140 sentinel failover-timeout mymaster 180000
141
142 # 脚本执行
143 #
144 # sentinel notification-script 和 sentinel reconfig-script 用于配置被调用以通知系统管理员或在故障转移后重新配置客户端的脚本。
145 # 脚本按照以下错误处理规则执行：
```

```
#
# 如果脚本以“1”退出，则稍后重试执行（最多当前设置为 10 的最大次数）。
#
# 如果脚本以“2”（或更高的值）退出，则脚本执行没有重试。
#
# 如果脚本因为收到信号而终止，则行为与退出代码 1 相同。
#
# 一个脚本的最长运行时间为 60 秒。 达到此限制后，脚本以 SIGKILL 终止并重试执行。

# 通知脚本
#
# sentinel notification-script <master-name> <script-path>
# 为在警告级别生成的任何哨兵事件调用指定的通知脚本（例如 -sdown、-odown 等）。
# 此脚本应通过电子邮件、短信或任何其他消息系统通知系统管理员，被监控的设备有问题
# Redis 系统。
#
# 脚本调用时只有两个参数：第一个是事件类型，第二个是事件描述。
#
# 如果提供此选项，脚本必须存在并且是可执行的，以便哨兵启动。
#
# Example:
#
# sentinel notification-script mymaster /var/redis/notify.sh

# 客户端重新配置脚本
#
# sentinel client-reconfig-script <master-name> <script-path>

# 当 master 由于故障转移而改变时，可以调用脚本来执行特定于应用程序的任务，以通知客户端配置已更改并且 master 位于不同的地址。
#
# 以下参数传递给脚本：
#
# <master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>
#
# <state> 目前总是 "failover"
# <role> 或者是 "leader" or "observer"
#
# 参数 from-ip, from-port, to-ip, to-port 用于传达 master 的旧地址和所选副本（现在是 master）的新地址。
#
# 此脚本应能抵抗多次调用。
#
# Example:
#
# sentinel client-reconfig-script mymaster /var/redis/reconfig.sh

# 安全
# 默认情况下，SENTINEL SET 将无法在运行时更改通知脚本和客户端重新配置脚本。
# 这避免了客户端可以将脚本设置为任何内容并触发故障转移以使程序执行的微不足道的安全问题。

sentinel deny-scripts-reconfig yes

# Redis 命令重命名
```

```
# 有时 Redis 服务器有某些命令，这些命令是 Sentinel 正常工作所需的，重命名为不可猜测的字符串。
# 在提供 Redis 即服务且不希望客户在管理控制台之外重新配置实例的提供商的上下文中，CONFIG 和 SLAVEOF 通常就是这种情况。

# 在这种情况下，可以告诉 Sentinel 使用不同的命令名称而不是普通的命令名称。
# 例如，如果主“mymaster”和关联的副本将“CONFIG”都重命名为“GUESSME”，我可以使用：
# SENTINEL rename-command mymaster CONFIG GUESSME 设置好这样的配置后，Sentinel 每次使用CONFIG都会请改用 GUESSME。
# 请注意，实际上没有必要遵守命令情况，因此在上面的示例中编写“config guessme”是相同的。
# SENTINEL SET 也可用于在运行时执行此配置。为了将命令设置回其原始名称（撤消重命名），可以将命令重命名为自身：
# SENTINEL 重命名命令 mymaster CONFIG CONFIG

# 主机名支持
#
# 通常 Sentinel 仅使用 IP 地址，并且需要 SENTINEL MONITOR 指定 IP 地址。 此外，它需要 Redis replica-announce-ip 关键字来仅指定 IP 地址。
#
# 您可以通过启用解析主机名来启用主机名支持。 请注意，您必须确保您的 DNS 配置正确，并且 DNS 解析不会引入很长的延迟。
#
SENTINEL resolve-hostnames no

# 当启用解析主机名时，Sentinel仍然使用IP地址向用户公开实例、配置文件等要在公布时保留主机名，请启用下面的“公布主机名”。
SENTINEL announce-hostnames no
```

八、Redis 内存问题

（一）内存穿透

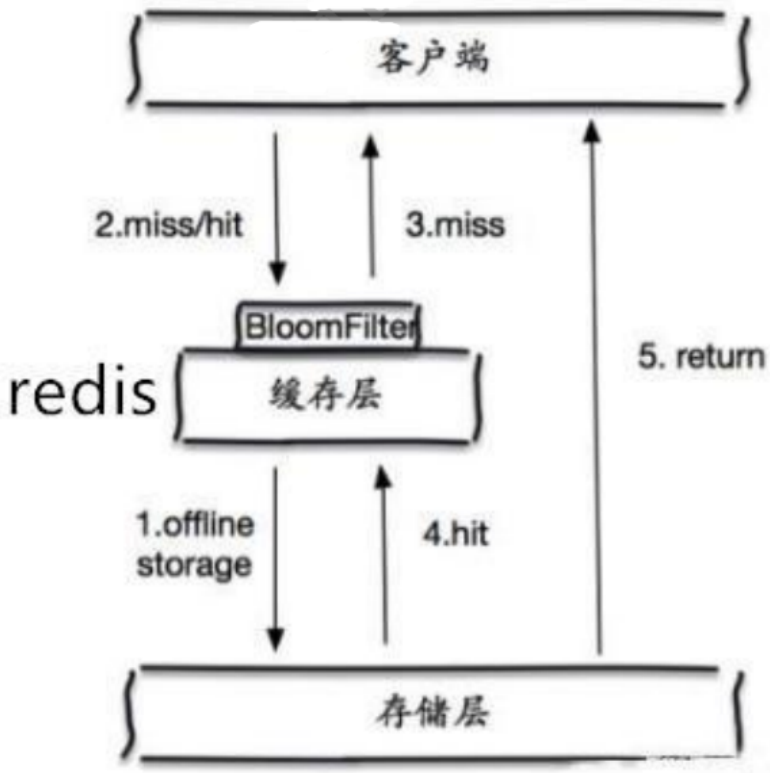
概念：

> 内存穿透是指查询一个**不存在的数据**。由于缓存不命中，并且出于容错考虑，如果从数据库查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，失去了缓存的意义。

解决方案：

- **布隆过滤器**

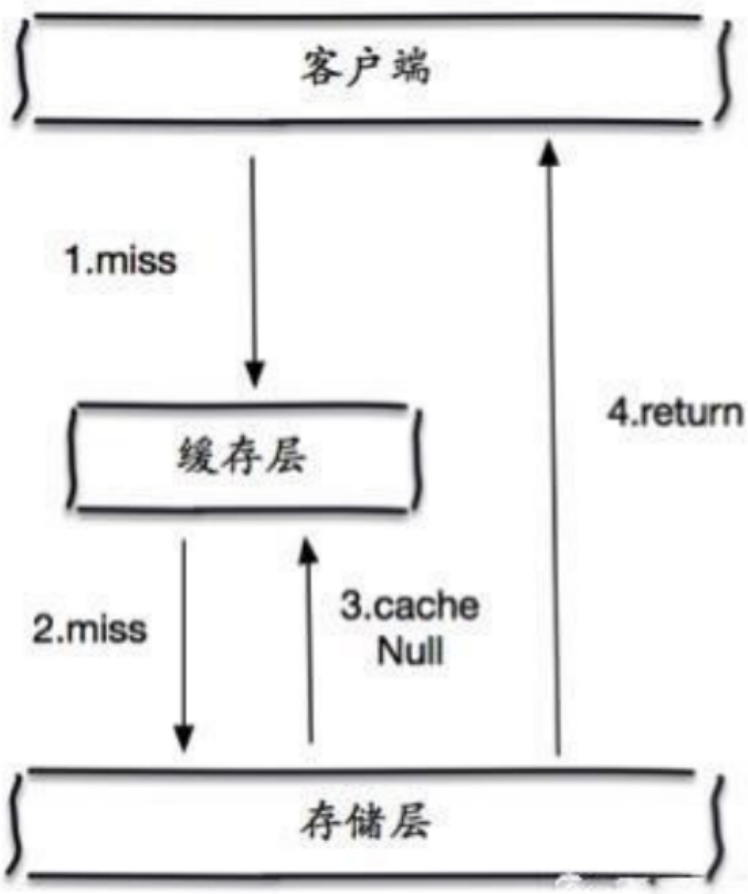
> 布隆过滤器是一种数据结构，对所有可能查询的参数以 hash 形式存储，在控制层先进行校验，不符合则丢弃，从而避免了对底层存储系统的查询压力；



缓存空对象

- **缓存空对象**

> 当存储层查不到，即使是空值，我们也将其存储起来并且在Redis中设置一个过期时间，之后再访问这个数据将会从Redis中访问，保护了持久层的数据库！



缓存空对象可能带来的问题：

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。

（二）内存击穿

概念

> 是指一个**非常热点的key（查询频次非常高的键）**，在不停的扛着大并发，当这个 key 失效时，一瞬间大量的请求冲到持久层的数据库中，就像在一堵墙上某个点凿开了一个洞！

解决方案：

- 设置热点key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

> 在查询持久层数据库时，保证了只有一个线程能够进行持久层数据查询，其他的线程让它睡眠几百毫秒，等待第一个线程查询完会回写到Redis缓存当中，剩下的线程可以正常查询Redis缓存，就不存在大量请求去冲击持久层数据库了！

（三）内存雪崩

概念

> 在某一个时间段，**缓存的key大量集中同时过期了**，所有的请求全部冲到持久层数据库上，导致持久层数据库挂掉！
>
> 例如：双十一零点抢购，这波商品比较集中的放在缓存，设置了失效时间为1个小时，那么到了零点，这批缓存全部失效了，而大量的请求过来时，全部冲过了缓存，冲到了持久层数据库！

解决方案

- 搭建Redis/数据库集群

> 既然Redis/数据库有可能挂掉，那我多增设几台Redis/数据库，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，我先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** 。

九、Redis 安全

> 我们可以通过 redis 的配置文件设置密码参数，这样客户端连接到 redis 服务就需要密码验证，这样可以让你的 redis 服务更安全。

- 使用`CONFIG get requirepass`命令查看是否设置了密码验证，默认情况下 requirepass 参数是空的，这就意味着你无需通过密码验证就可以连接到 redis 服务。

```
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) ""
```

- 方式一：使用`CONFIG set requirepass "密码"`命令来设置密码。设置密码后，客户端连接 redis 服务就需要密码验证，否则无法执行命令。

```
127.0.0.1:6379> CONFIG set requirepass "123456"
OK
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) "123456"
```

- 方式二：配置文件配置密码，修改默认`# requirepass foobared`配置，将`#`去掉，`foobared`改为自己的密码即可。

```
requirepass foobared
```

- 使用`AUTH password`命令验证密码

```
127.0.0.1:6379> AUTH "123456"
OK
127.0.0.1:6379> SET mykey "Hello Password"
OK
127.0.0.1:6379> GET mykey
"Hello Password"
```

十、SpringBoot整合Redis（Jedis）

> SpringBoot 连接 Reids 出现连接异常时，检查如下配置：
>
> 1、Redis 的远程配置是否开启：`bind 0.0.0.0`
>
> 2、Redis 的保护模式是否关闭：`protected-mode no`
>
> 3、Redis 端口是否在防火墙中放行（测试时也可以关闭防火墙）：

```
# 开启6379端口命令，注意命令中间的空格
firewall-cmd --zone=public --add-port=6379/tcp --permanent
# 重新加载防火墙
firewall-cmd --reload
```

```
# 查看防火墙状态
systemctl status firewalld
# 关闭防火墙
systemctl stop firewalld
# 开启防火墙
systemctl start firewalld
```

Spring Boot 2.x

1）导包

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>2.7.2</version>
</dependency>
```

2）配置redis连接

```
spring:
  # Redis 配置
  redis:
    host: 192.168.88.88 # 默认是localhost
    port: 6379 # 默认是6379
    password: xxx # Redis 配置有密码需要配置此项
```

3）引入模板

```
@Autowired
StringRedisTemplate stringRedisTemplate;

@Autowired
RedisTemplate redisTemplate;

opsForValue ：对应 String（字符串）
opsForZSet：对应 ZSet（有序集合）
opsForHash：对应 Hash（哈希）
opsForList：对应 List（列表）
opsForSet：对应 Set（集合）
opsForGeo：对应 GEO（地理位置）
```

4）测试

```
import com.soft.entity.Student;
import com.soft.service.StudentService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.ListOperations;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.ValueOperations;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

@Controller
@RequestMapping("/stu")
public class StudentController {
    @Autowired
    private StudentService studentService;

    // 导入 Redis 模板
    @Autowired
    private RedisTemplate redisTemplate;

    /**
     * 测试保存 List 类型
     * @return
     */
    @GetMapping("/list")
    @ResponseBody
    public int list(){

        // Redis List 类型
        ListOperations listOperations = redisTemplate.opsForList();
        // Key 集合
        List<String> keyList = new ArrayList<>();
        keyList.add("stuList");
        // 判断 Key 是否存在
```

```
        Long aLong = redisTemplate.countExistingKeys(keyList);

        long start = System.currentTimeMillis();

        List<Student> list = null;
        // 等于 0 说名没有结果集，没有结果走数据库查询
        if (aLong == 0){
            // 调用 业务层 查询数据库
            list = studentService.list();
            System.out.println("走数据库！");

            // 将结果放到缓存中
            listOperations.leftPush("stuList", list);
            // 这是过期时间为1分钟
            redisTemplate.expire("stuList", 30, TimeUnit.SECONDS);
        // 走缓存查询
        } else {
            list = listOperations.range("stuList", 0, -1);
            System.out.println("走数缓存！");
        }
        long end = System.currentTimeMillis();

        System.out.println(end - start);

        return list.size();
    }

    /**
     * 测试 String 类型（添加）
     * @return
     */
    @GetMapping("/set")
    @ResponseBody
    public String setKey(){
        ValueOperations valueOperations = redisTemplate.opsForValue();
        valueOperations.set("name", "Jack");
        valueOperations.set("age", 18);
        valueOperations.set("class", 307);
        valueOperations.set("sex", "男");
        return "OK";
    }

    /**
     * 测试 String 类型（获取）
     * @param key
     * @return
     */
    @GetMapping("/get")
    @ResponseBody
    public Object getKey(String key){
        ValueOperations valueOperations = redisTemplate.opsForValue();
        return valueOperations.get(key);
    }
}

```

5）工具类

```
public class RedisUtils {
    @Autowired
    private RedisTemplate redisTemplate;
    /**
     * 写入缓存
     * @param key
     * @param value
     * @return
     */
    public boolean set(final String key, Object value) {
        boolean result = false;
        try {
            ValueOperations<Serializable, Object> operations = redisTemplate.opsForValue();
            operations.set(key, value);
            result = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }
    /**
     * 写入缓存设置时效时间
     * @param key
     * @param value
     * @return
     */
    public boolean set(final String key, Object value, Long expireTime ,TimeUnit timeUnit) {
        boolean result = false;
        try {
            ValueOperations<Serializable, Object> operations = redisTemplate.opsForValue();
            operations.set(key, value);
            redisTemplate.expire(key, expireTime, timeUnit);
            result = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }
    /**
     * 批量删除对应的value
     * @param keys
     */
    public void remove(final String... keys) {
        for (String key : keys) {
            remove(key);
        }
    }
    /**
     * 批量删除key
     * @param pattern
```

```
     */
    public void removePattern(final String pattern) {
        Set<Serializable> keys = redisTemplate.keys(pattern);
        if (keys.size() > 0){
            redisTemplate.delete(keys);
        }
    }
    /**
     * 删除对应的value
     * @param key
     */
    public void remove(final String key) {
        if (exists(key)) {
            redisTemplate.delete(key);
        }
    }
    /**
     * 判断缓存中是否有对应的value
     * @param key
     * @return
     */
    public boolean exists(final String key) {
        return redisTemplate.hasKey(key);
    }
    /**
     * 读取缓存
     * @param key
     * @return
     */
    public Object get(final String key) {
        Object result = null;
        ValueOperations<Serializable, Object> operations =
redisTemplate.opsForValue();
        result = operations.get(key);
        return result;
    }
    /**
     * 哈希 添加
     * @param key
     * @param hashKey
     * @param value
     */
    public void hmSet(String key, Object hashKey, Object value){
        HashOperations<String, Object, Object> hash =
redisTemplate.opsForHash();
        hash.put(key,hashKey,value);
    }
    /**
     * 哈希获取数据
     * @param key
     * @param hashKey
     * @return
     */
    public Object hmGet(String key, Object hashKey){
```

```
        HashOperations<String, Object, Object>  hash = redisTemplate.opsForHash();
        return hash.get(key,hashKey);
    }
    /**
     * 列表添加
     * @param k
     * @param v
     */
    public void lPush(String k,Object v){
        ListOperations<String, Object> list = redisTemplate.opsForList();
        list.rightPush(k,v);
    }
    /**
     * 列表获取
     * @param k
     * @param l
     * @param l1
     * @return
     */
    public List<Object> lRange(String k, long l, long l1){
        ListOperations<String, Object> list = redisTemplate.opsForList();
        return list.range(k,l,l1);
    }
    /**
     * 集合添加
     * @param key
     * @param value
     */
    public void add(String key,Object value){
        SetOperations<String, Object> set = redisTemplate.opsForSet();
        set.add(key,value);
    }
    /**
     * 集合获取
     * @param key
     * @return
     */
    public Set<Object> setMembers(String key){
        SetOperations<String, Object> set = redisTemplate.opsForSet();
        return set.members(key);
    }
    /**
     * 有序集合添加
     * @param key
     * @param value
     * @param scoure
     */
    public void zAdd(String key,Object value,double scoure){
        ZSetOperations<String, Object> zset = redisTemplate.opsForZSet();
        zset.add(key,value,scoure);
    }
    /**
     * 有序集合获取
     * @param key
```

```
157         * @param scoure
158         * @param scoure1
159         * @return
160         */
161        public Set<Object> rangeByScore(String key,double scoure,double
    scoure1){
162            ZSetOperations<String, Object> zset = redisTemplate.opsForZSet();
163            return zset.rangeByScore(key, scoure, scoure1);
164        }
165    }
```