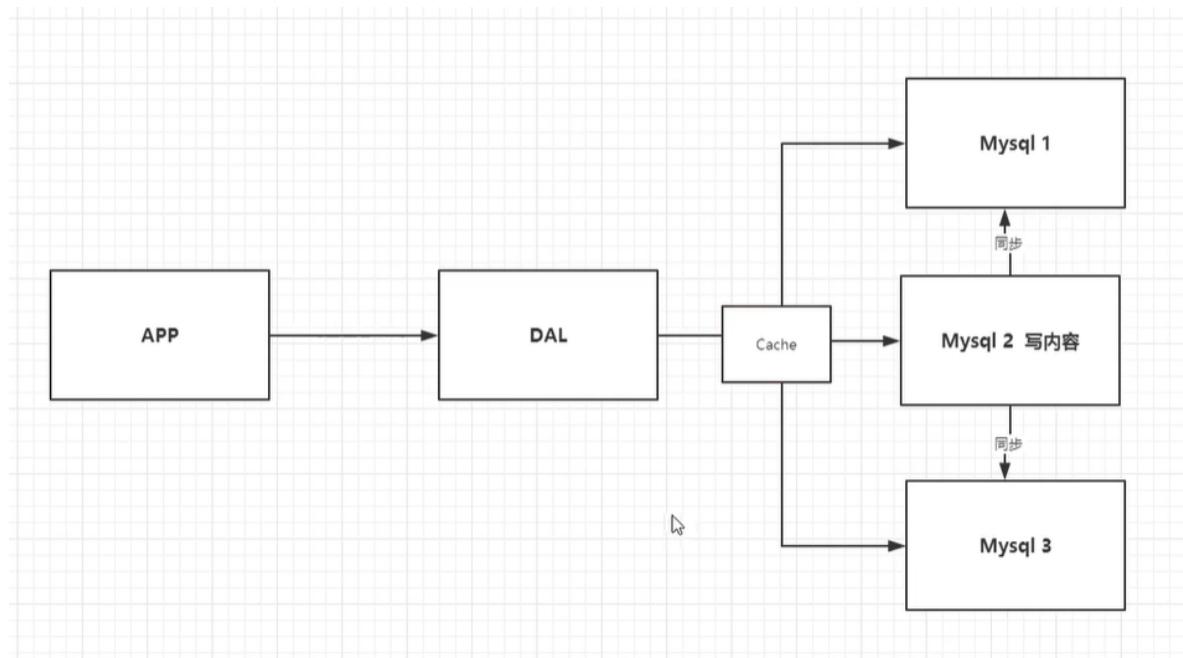


NoSQL

缓存的作用

缓存+MySQL的垂直拆分（读写分离）

网站80%的数据库操作都是在读写，所以每次去查询很麻烦，所以使用缓存来提高效率！

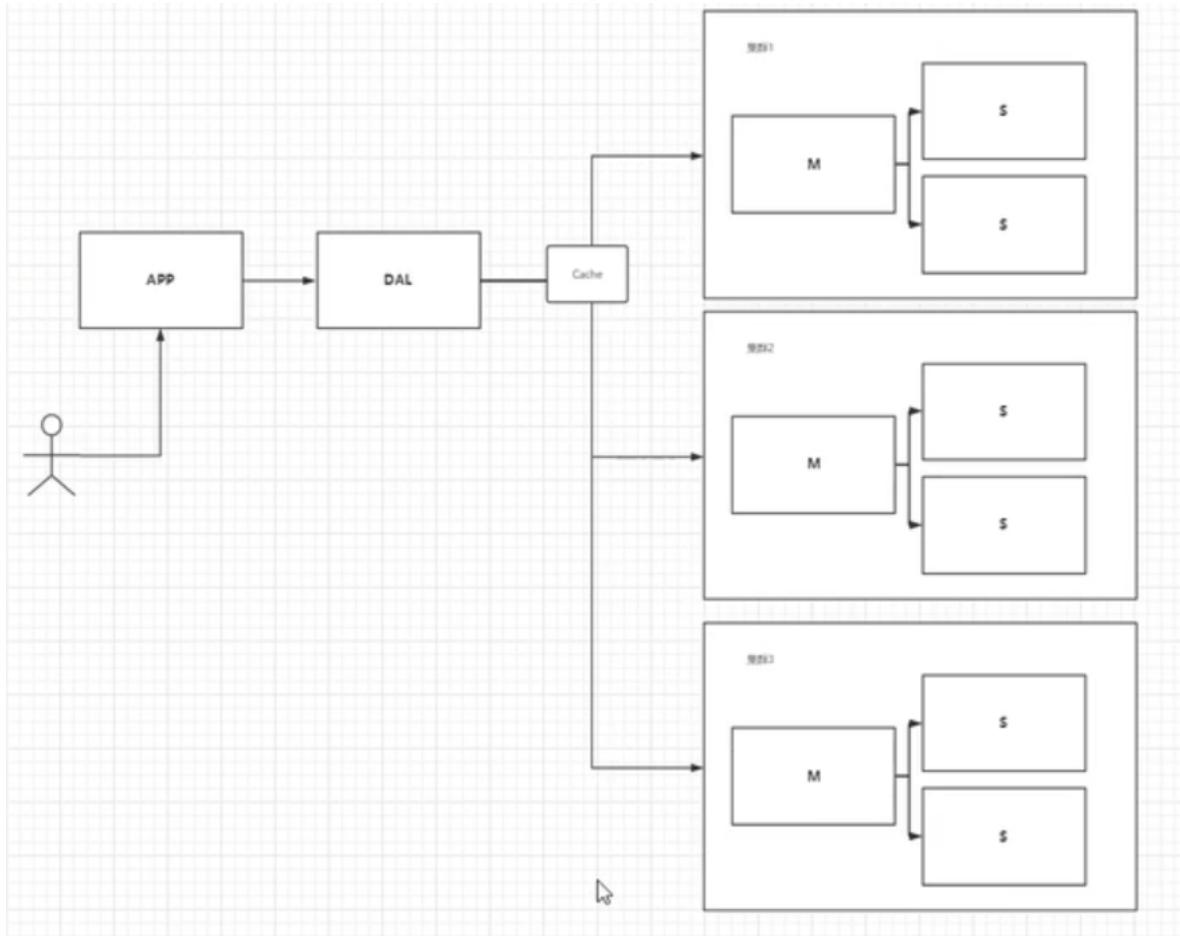


分库分表+水平拆分+MySQL集群

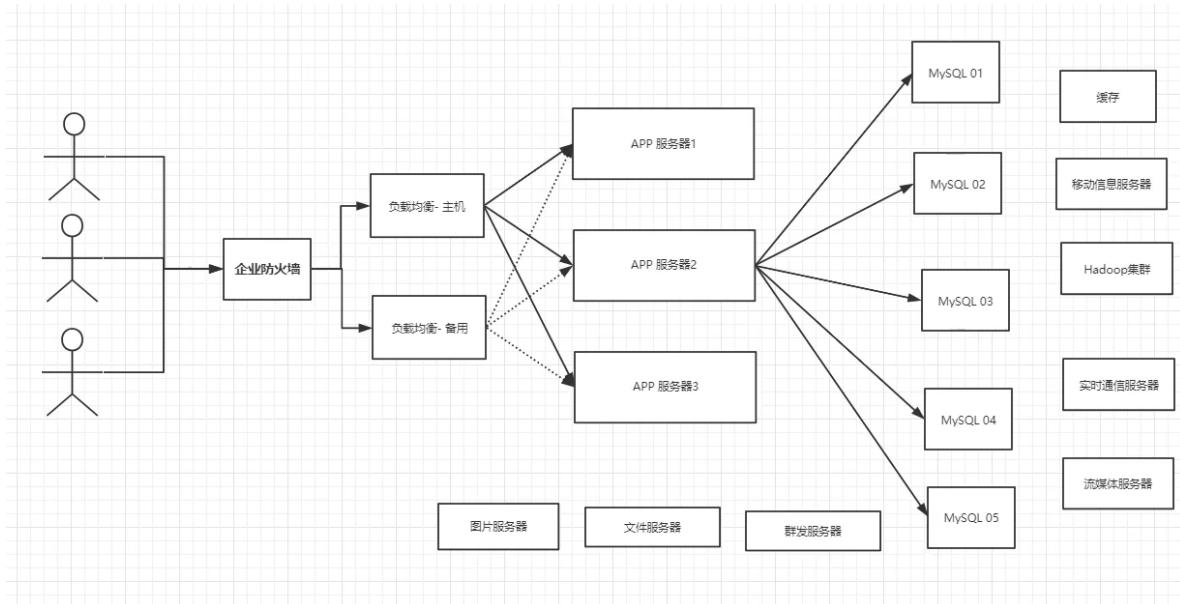
早些年MyISAM：表锁，十分影响效率！高并发问题

Innodb：行锁

最后慢慢分库分表解决写的压力！



目前互联网的基本架构模型



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!

NoSQL是什么

NoSQL = Not Only SQL

特点

1. 方便扩展（无关系的数据，很好扩展！）
2. 大数据量高性能（Redis 写 8万/s 读 11万/s，细粒度缓存，性能高）
3. 数据类型是多样型！（无需设计数据库，随取随用）

RDBMS和NoSQL

RDBMS(关系型数据库): SQL,结构化组织，数据和关系都存在单独的表中，严格一致性，事务。

NoSQL:

- 无固定的查询语言
 - 键值对存储，列存储，文档存储，图形数据库（社交）
 - 最终一致性
 - CAP定理和BASE
 - 高性能，高可用，高可扩

NoSQL的四大类

1. KV键值对: Redis
2. 文档型数据库 (bson 二进制Json)：
 - MongoDB (必须掌握)
 - 分布式文件存储的数据库，C++编写，处理大量文档
 - 介于关系型和非关系型中间的，最像关系型数据的~
 - ConthDB
3. 列存储数据库
 - HBase
 - 分布式文件系统
4. 图关系数据库
 - 存的不是图形，是关系，社交关系
 - Neo4j, InfoGrid

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的集群方案。[3]

Redis入门

Redis是啥

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作用

1. 内存存储，持久化
2. 效率高，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.

Redis特性

1. 多样数据类型
2. 持久化
3. 集群
4. 事务
5.

Windows 安装

1. <https://github.com/MicrosoftArchive/redis/releases>
2. 下载完解压即可

```
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> -
```

Linux安装

emmmmmmmmmmmmmmmmmmm

docker 安装它不香吗?

```
docker pull redis
```

当然也可以解压tar包

```
tar -zxvf tar包
```

启动

- 非docker的做法

首先需要到redis.conf里面将daemonize no 改为 daemonize yes

```
# 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
```

```
redis-server /opt/redis-5.08/redis.conf #启动服务 指定配置问题
redis-cli -p 6379 #启动客户端 可以加-h 指定主机ip
```

- docker使用数据卷指定配置文件，一键启动~

需要在主机配置对应的配置，然后才可以跟容器挂载起来。

```
do \
mkdir -p /home/ubuntu/mydata/redis/node-1/conf
touch /home/ubuntu/mydata/redis/node-1/conf/redis.conf
cat << EOF >/home/ubuntu/mydata/redis/node-1conf/redis.conf
port 6379
bind 0.0.0.0
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
cluster-announce-ip 172.38.0.11
cluster-announce-port 6379
cluster-announce-bus-port 16379
appendonly yes
EOF
done
```

```
docker run -p 6371:6379 -p 16371:16379 --name redis-1 -v /mydata/redis/node-1/data:/data -v /mydata/redis/node-1/conf/redis.conf:/etc/redis/redis.conf -d --net redis --ip 172.38.0.11 redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf
```

进入redis

```
docker exec -it 容器id redis-cli
```

执行相关命令

- ping: 测试连接
- set 【key】 【value】 设置键值对
- get 【key】 获取值
- shutdown 退出关闭redis

```
ps -ef|grep redis 查看进程是否存在 # 非docker  
docker stats #可查看容器
```

- 测试100个连接100000个请求

```
redis-benchmark -h localhost -p 6379 -c 100 -n 100000
```

- 切换数据库select

```
127.0.0.1:6379> SELECT 3  
OK  
127.0.0.1:6379[3]> DBSIZE  
(integer) 0
```

- 清除当前数据库 flushdb

```
127.0.0.1:6379[3]> set name hcode  
OK  
127.0.0.1:6379[3]> get name  
"hcode"  
127.0.0.1:6379[3]> flushdb  
OK  
127.0.0.1:6379[3]> get name  
(nil)
```

- 清除全部数据库 flushall

Redis是单线程的！Redis是基于内存操作，速度很快~，Redis的性能瓶颈跟机器的内存和网络带宽有关！跟CPU没有多大关系。

Redis是将所有数据放在内存中的，所以使用单线程取操作效率是最高的

五大数据类型

Redis-Key

```
exists key          #判断值是否存在  
expire key 10      #设置key值10s后过期  
ttl key            #查看值的缓存时间还剩下多少  
move key 2         #将key数据移到2号数据库  
type key           #查看类型
```

String

```
127.0.0.1:6379> APPEND k "xixi"  
(integer) 5  
127.0.0.1:6379> get k  
"vxixi"  
  
append key "追加的字符串" #如果当前key不存在，就相当于set key  
strlen key #获取字符串长度  
incr key #每次让key对应的值加1  
incrby key 10 #每次让key对应的值加10  
相对于 decr 用法一样，只是变成减  
decr key  
decr key 10  
getrange key 0 10 #获取前11个字符  
setrange key 1 字符串 #替换指定位置开始的字符串
```

```
setex k2 10 hello #设置值过期时间为10s  
setnx k2 xixi #不存在值就设置值 分布式锁中常常使用
```

批量操作

```
mset k1 v1 k2 v2 ... #设置值  
mget k1 k2 k3.. #获取值  
msetnx k1 v1 k2 v2 #不存在值就创建，但是这是原子性的操作，一个key失败全部失败
```

设置对象

```
mset user:1:name hcode user:1:age 12  
mget user:1:name
```

如果有key就获取值，没有就设置值value

```
getset key value
```

List

```

lpush list value      # 将值插入列表的头部，或者最左边
rpush list value      # 将值插入列表的尾部，或者最右边
lrange list 0 -1      #从左到右读取list
lpop list             #将最左边的值移除
rpop list             #将最右边的之移除
lindex list 1          #获取下标为1的值
llen list              #获取list长度
lrem list 2 value      #移除list中的2个value
ltrim list 0 1          #只截取0 1下标的值。其它在list剔除了
rpoplpush list1 list2   #移除list1最后一个元素。移到到list2中
lset list 0 newvalue    #将list中第0个值换成newvalue
linsert list before【after】 value newvalue #往value之前插入newvalue

```

Set

set的值不可重复

```

sadd set value        #添加值
smembers set           #查看set里面全部的值
sismember set value    #判断value是否存在在set里面
scard set              #获取set集合中的个数
srem set value         #移除set中的value
strandmember set 【num】 #随机获取set中的一个值，可以在后面添加个数
spop set               #随机移除一个value
smove set1 set2 value  #将set1中的value移除，然后移到set2
sdiff set1 set2        #获取set1中set2没有的值 差集
sinter set1 set2       #获取set1和set2相同的值 交集
sunion set1 set2       #获取set1和set2全部的值 并集

```

Hash

key-map 现在的值是一个map集合，跟String差不多

更适合对象的存储~

```

hset myhash k1 v1          #加入一个map
hmset myhash k1 v1 k2 v2    #批量加入map k存在就覆盖
hmget myhash k1 k2          #批量获取值
hgetall myhash             #获取全部的map
hdel myhash k1              #删除指定的map
hlen myhash                 #获取map的个数
hexists myhash k1           #判断是否存在k1的值
hvals myhash                #获取所有的value
hkeys myhash                 #获取所有的key
hincrby myhash k1 1          #给k1对应的值加1
hdecrby myhash k1 1          #给k1对应的值减1
hsetnx myhash k1 v1          #不存在就设置一个map

```

Zset

在set基础上增加了一个值

```
zadd myset 1 one 2 two #多了一个序号
zrange 0 -1           #查询全部
zrangebyscore myset -inf +inf #从序号最小到最大排序
ZREVRANGEBYLEX myset max min #返回max到min之间的降序
zrem myset value        # 删除一个value
zcard myset             # 查询set的个数
zrevrange myset 0 -1    # 倒着查出全部
zcount myset 0 2        # 查询下标0到2的值
```

三种特殊的数据类型

geospatial 地理位置

geoadd

```
127.0.0.1:6379> GEOADD china:beicity 116.40 39.90 beijing
(integer) 1
127.0.0.1:6379> GEOADD china:beicity 121.47 31.23 shanghai
(integer) 1
127.0.0.1:6379> GEOADD CHINA:beicity 114.05 22.52 shenzhen
(integer) 1
GEOADD key 经度 纬度 名称
南北两极不可添加
```

geopos

获得当前定位

```
127.0.0.1:6379> GEOPOS china:beicity beijing
1) 1) "116.39999896287918091"
   2) "39.90000009167092543"
# 获取指定城市的经纬度
```

geodist

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。

查询两地之间的距离可以指定单位

```
127.0.0.1:6379> GEODIST china:beicity beijing shanghai
"1067378.7564"
127.0.0.1:6379> GEODIST china:beicity beijing shanghai km
"1067.3788"
```

georadius

获取半径内的所有城市

```
GEORADIUS china:beicity 110 20 1000 km #获取经纬度110, 20在china: beicity这个key里面
对应1000km之内的value
withdist withcoord count 2 #参数对应为显示距离, 显示经纬度, 只显示2个
```

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"
```

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"
```

Hyperloglog

可以容错再使用

```
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 元素的基数数量 (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
```

Bitmaps

两种状态的数据可以使用这个~

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

一次性，顺序性，排他性。

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>
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
(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
- 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！

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

```
<!--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>
```

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");
        Transaction multi = jedis.multi(); // 开启事务
        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, 实例可以在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

配置文件

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！
```

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 noeviction # 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、noeviction : 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！
```

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

消息发布订阅

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>
```

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 ()

主从复制

环境配置

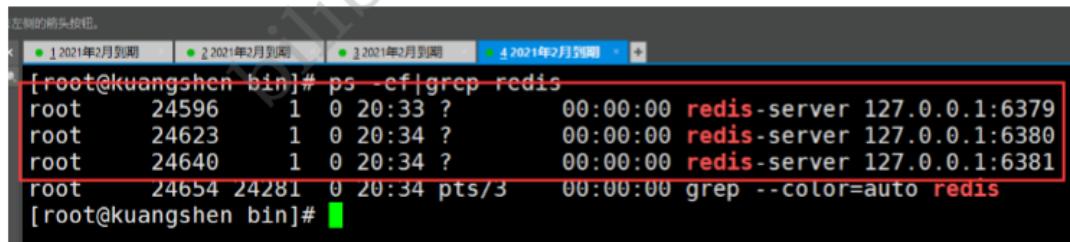
只配置从库，不用配置主库！

```
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
replog_backlog_active:0
replog_backlog_size:1048576
replog_backlog_first_byte_offset:0
replog_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！



```
[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]#
```

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 # 可以看到主机的信息
master_port:6379
```

```
master_link_status:up
master_last_io_seconds_ago:3
master_sync_in_progress:0
slave_repl_offset:14
slave_priority:100
slave_read_only:1
connected_slaves:0
master_replid:a81be8dd257636b2d3e7a9f595e69d73ff03774e
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:14
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:14

# 在主机中查看!
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:1 # 多了从机的配置
slave0:ip=127.0.0.1,port=6380,state=online,offset=42,lag=1 # 多了从机的配置
```

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

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。

测试！

我们目前的状态是一主二从！

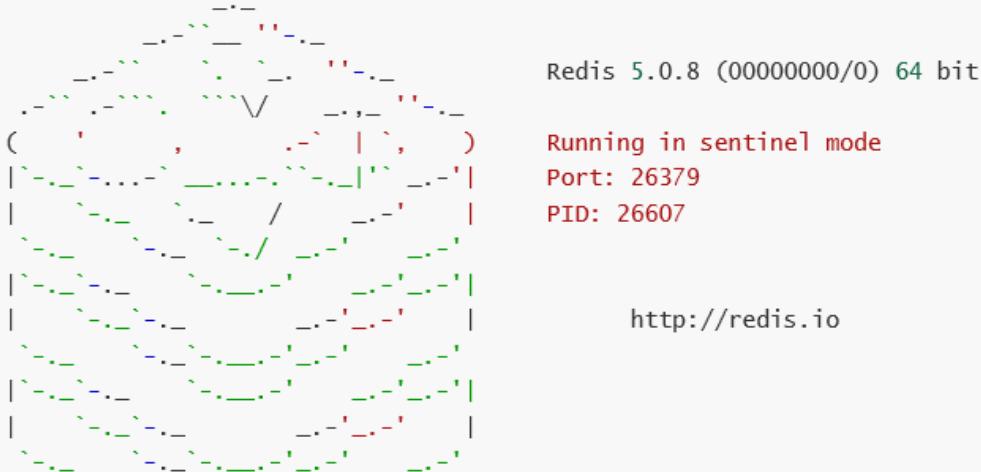
1、配置哨兵配置文件 sentinel.conf

```
# 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 # o000o000o000 Redis is starting o000o000o000
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
```



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！

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

哨兵文件的配置

```
# 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!
```

Redis缓存和雪崩
