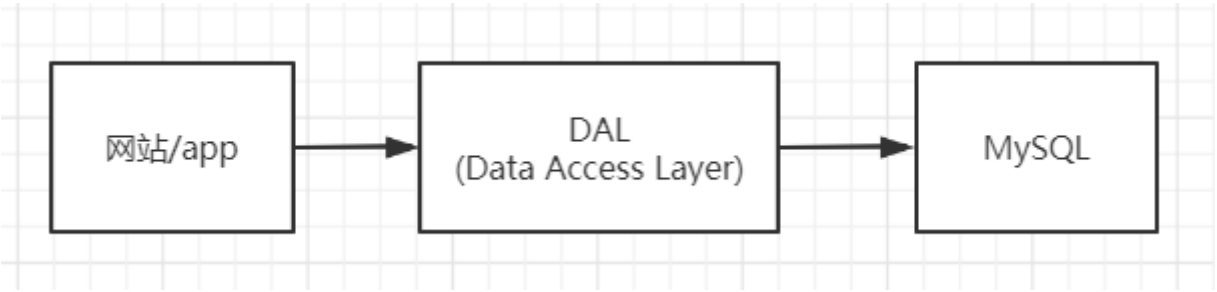
NoSQL

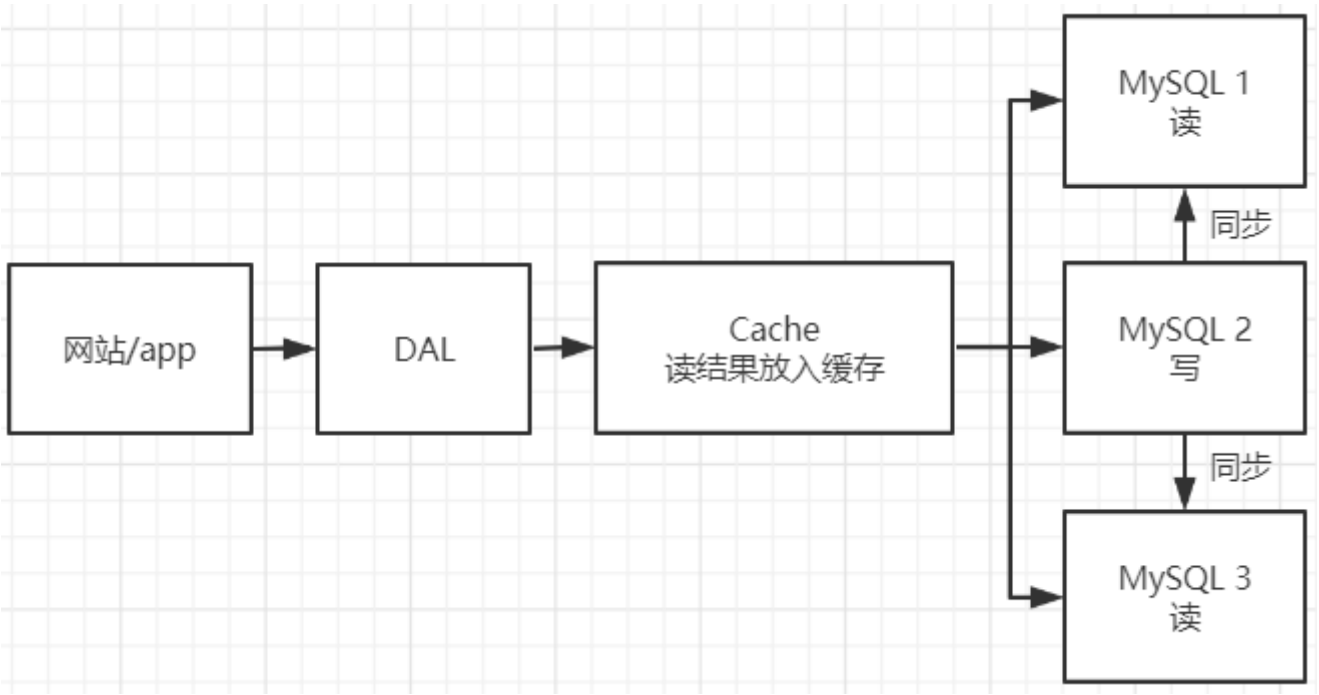
1）NoSql发展历史

1. 早期单机MySQL年代



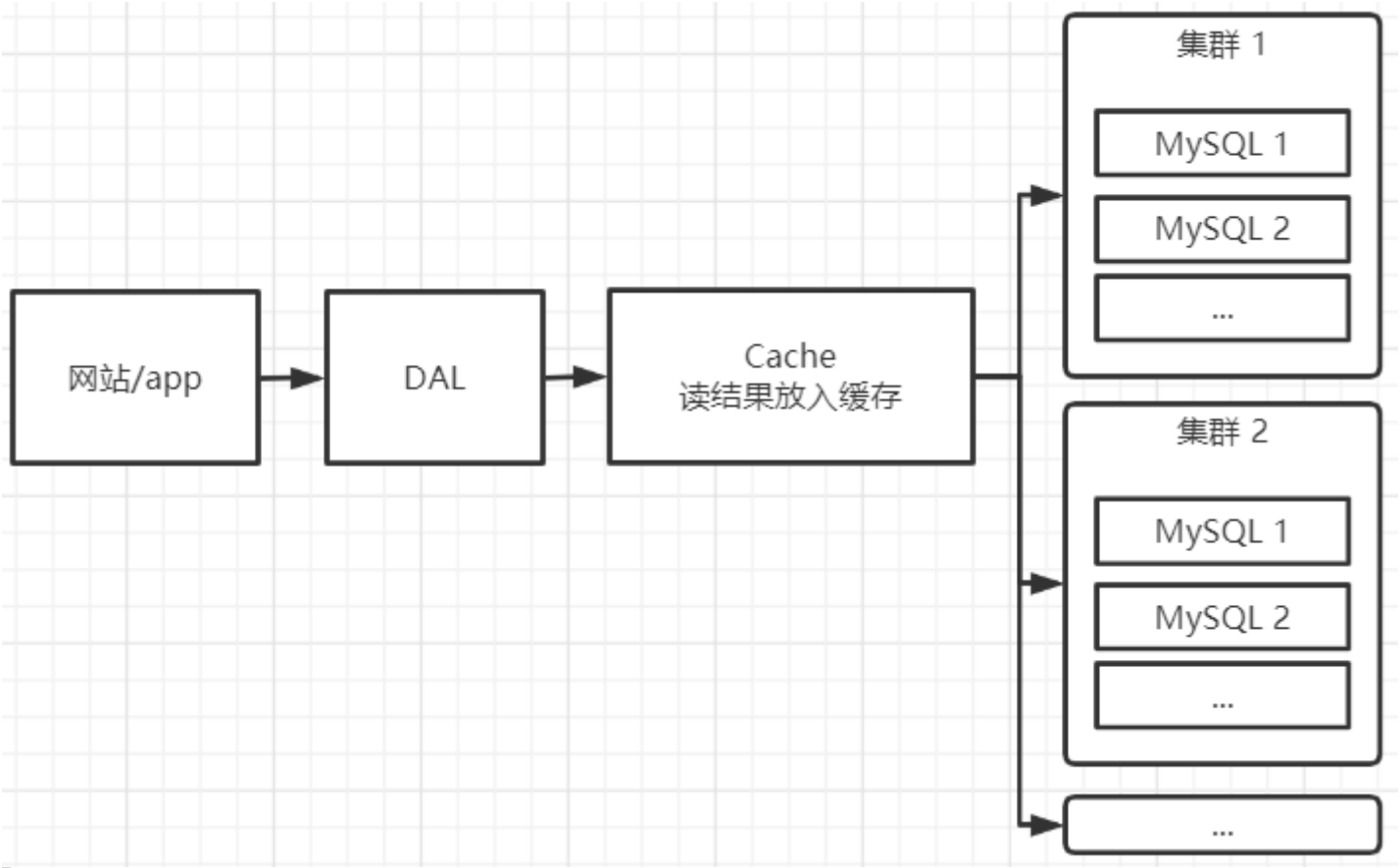
> 情况：早年访问量小，单数据库完全足够支撑，且使用的大多是静态网页，服务器压力小。
>
> 瓶颈：数据量大的时候单数据库放不下；数据索引太大，一个机器内存放不下；访问量(读写混合)，一个服务器承受不了

2. Memcache + MySQL + 垂直拆分(读写分离)年代



> 发展：优化数据结构和索引—>文件缓存(IO)—>Memcached
>
> 情况：遵循八二原则，大多数请求都是读取数据，若重复读取同一个数据都从数据库拿，会造成较大压力，因此引入缓存。写数据库向其他数据库进行同步。

3. 分库分表 + 水平拆分 + 集群年代



> 发展：MyISAM表锁，高并发效率低，Innodb行锁
>
> 情况：开始使用分库分表解决写的压力，以及推出MySQL集群

4. 现今最近年代

> 情况：数据量爆发式增长，单纯使用MySQL来持久化数据的方式已不可行，开始引入NoSQL。NoSQL在大数据环境下发展十分迅速，其中Redis发展最快。

2）NoSQL介绍

1. NoSQL是什么

> NoSQL = Not Only SQL，指非关系型数据库。适用于一些类似于个人信息、社交网络、地理位置等不需要固定的格式，且需要横向扩展的数据。

2. NoSQL特点

> 方便扩展，数据直接没有关系
>
> 大数据量高性能，Redis读11W/s，写8.1W/s。NoSQL的缓存记录是细粒度的缓存，性能比较高

> 数据类型多样性，无需实现设计数据库

3. NoSQL与传统RDBMS

RDBMS	NoSQL
结构化组织	无结构
SQL	无固定查询语言
数据和关系存在表中	键值对存储、列存储、文档存储、图形数据库
严格一致性	最终一致性

4. 常见NoSQL数据库

> Memcache：一般不持久化，仅支持简单的Key-Value模式，作为辅助缓存数据库
>
> Redis：支持持久化，主要用作备份恢复，除KV外还支持多种数据结构，作为辅助缓存数据库
>
> MongoDB：文档型数据库，支持二进制数据和大型对象

5. 大数据时代的3V和3高

> 3V：Volume海量、Variety多样、Velocity实时
>
> 3高：高并发、高可拓、高性能

6. 阿里架构演进分析

> 商品基本信息：名称、价格、商家信息等，MySQL即可解决
>
> 商品描述、评论：长文本、MongoDB
>
> 图片：分布式文件系统FastDFS，如淘宝TFS，谷歌GFS，Hadoop HDFS
>
> 商品关键字搜索：搜索引擎solr，ElasticSearch，阿里ISearch
>
> 商品热门的波段信息：内存数据库，如Redis，Tair，MemCache
>
> 商品交易，外部支付接口：第三方应用
>
> 总结：大型互联网应用问题
>
> - 数据类型太多
> - 数据源繁多，经常重构
> - 数据要改造可能涉及大面级改造

> 解决方法：使用UDSL统一数据服务平台来作为中间平台连接多种数据库和应用，并提供统一API，减少开发难度

3）NoSQL四大分类

1. KV键值对：常见Redis
2. 文档型数据库(bson格式，和json类型，不过使用二进制)：常见MangoDB，一个基于分布式文件存储的数据库
3. 列存储数据库：如HBase，分布式文件数据库
4. 图关系数据库：存拓扑关系，如Neo4j，InfoGrid

Redis基础介绍

1）概述

> Redis(Remote Dictionary Server)，远程字典服务。是一种支持网络，基于内存并可持久化的日志型、KV数据库，并提供多种语言的API，其操作都是原子性的。
>
> Redis会周期性的将更新的数据写入磁盘或者吧修改操作写入追加的记录文件，并且在此基础上实现主从同步。可配合数据库做高速缓存。

2）Linux下Redis相关指令与配置

> 修改配置文件 redis.conf的daemonize=yes，实现后台启动
>
> redis-server 启动服务器 redis-cli -p 端口 启动客户端
>
> 在客户端使用shutdown关闭

3）Redis性能测试工具benchmark

> 如测试 100个并发连接 100000请求(其余参数看Redis文档)
>
> redis-benchmark -h 127.0.0.1 -p 6379 -c 100 -n 100000

4）Redis单线程

> Redis是单线程的，它基于内存操作，CPU并不是其瓶颈。Redis的瓶颈在于机器内存和网络带宽，因此用单线程足以实现高效性能。

Memcache	Redis
多线程+锁	单线程+多路IO复用
仅支持Key-Value	除了KV，还支持多种数据结构
不支持持久化	支持持久化

5）Redis基本知识

> Redis默认有16个数据库，默认使用第0号，统一密码管理，所有库同样密码

```
# 切换数据库
127.0.0.1:6379> select 2
OK
# 查看数据库大小
127.0.0.1:6379[2]> dbsize
(integer) 0
# 移动到某个库，1表示移动成功
127.0.0.1:6379[2]> move name 1
(integer) 1
# 清除当前数据库
127.0.0.1:6379[2]> flushdb
OK
# 清除所有数据库
127.0.0.1:6379[2]> flushall
OK
```

Redis数据类型及相关操作命令

1）Redis数据类型

> String，List，Set，ZSet，Hash，特殊类型bitmaps，hyperloglos，geospatial

2）Redis键相关操作

```
# 查看键，keys后面接正则表达式
127.0.0.1:6379[2]> keys *
1) "age"
2) "name"
# 查看键是否存在，返回1表示存在，0则不存在
127.0.0.1:6379[2]> exists name
(integer) 1
# 查看键类型
127.0.0.1:6379[2]> type name
string
# 直接删除键值，1表示删除成功，0表示不存在
127.0.0.1:6379[2]> del name
(integer) 1
# 非阻塞删除，1表示删除成功，0表示不存在(先从keys里删除，真正的删除在后续异步操作)
127.0.0.1:6379> unlink name
(integer) 1
# 设置过期时间(单位s)，1表示设置成功
127.0.0.1:6379> expire name 10
(integer) 1
# 设置值的时候顺便设置过期时间
127.0.0.1:6379> setex name 10 JIA
OK
# 查看还有多久过期，单位为s，-1表示永不过期，-2表示已过期
127.0.0.1:6379> ttl name
(integer) 5
```

3）Redis数据类型之String

> String是二进制安全的，即可以包含任何数据，如jpg图片或序列化的对象，最大value512M。String的内部数据结构为简单动态字符串Simple Dynamic String，SDS。相当于Java的ArrayList，采用预分配来减少内存的频繁分配。内容小于1M时扩容为2倍，超过1M时每次加1M。

```
# 新建键值对或者覆盖，Redis允许key中带:
127.0.0.1:6379> set name JIA
OK
# 不存在才新建，1表示不存在，新建；0表示已存在，没新建也不覆盖值
127.0.0.1:6379> setnx name JIA
(integer) 1
# 获取值
127.0.0.1:6379> get name
"JIA"
# 获取后再设置
127.0.0.1:6379> getset name ZURI
"JIA"
# 批量设置
127.0.0.1:6379> mset name JIA name1 ZURI
OK
# 批量获取
127.0.0.1:6379> mget name name1
1) "JIA"
2) "ZURI"
# 批量不存在才设置，若一个失败全部失败，结果同setnx
127.0.0.1:6379> msetnx name JIA name1 ZURI
(integer) 0

# 指定范围查看值，注意这里包括第"3"个字符
127.0.0.1:6379> getrange name1 0 3
"ZURI"
# 从指定下标开始修改，超出修改内容的部分不变。返回修改后长度。
127.0.0.1:6379[2]> setrange name 3 ZURI
(integer) 7
# 查看值长度
127.0.0.1:6379> strlen name
(integer) 3
# 追加，若key不存在则相当于新建，返回值长度
127.0.0.1:6379> append name 1234
(integer) 7

# 自增，若key不存在则新建从0开始自增，返回修改后值
127.0.0.1:6379> incr count
(integer) 1
# 自减，若key不存在则新建从0开始自减，返回修改后值
127.0.0.1:6379> decr count
(integer) -1
# 增加，若key不存在则新建从0开始增加，返回修改后值
127.0.0.1:6379> incrby count 10
(integer) 10
# 减少，若key不存在则新建从0开始减少，返回修改后值
127.0.0.1:6379> decrby count 10
(integer) -10
```

4）Redis数据类型之List

> List为简单的String列表，可以按照插入顺序排序，分为左右两边，相当于头部和尾部。List的底层实际上是个双向列表，对两端的操作性能较高，但通过索引取值性能较低。
>
> List的数据结构为快速链表quickList，在元素较少的时候使用一块连续内存，即ziplist，多个ziplist用双向指针结合起来成为quicklist。而不是每个元素都是用双向指针。

```
# 选择从左放入+取值(lpush+lrange)相当于栈，从右放入+取值(rpush+lrange)，相当于队列
# 从左放入元素，无对应键则新建，返回当前长度
127.0.0.1:6379> lpush list 1
(integer) 1
# 从右放入元素，无对应键则新建，返回当前长度
127.0.0.1:6379> rpush list 2
(integer) 2
# 按照范围[0，-1]从左读取元素，注意顺序
127.0.0.1:6379> lrange list 0 -1
1) "1"
2) "2"
# 取出最左边的元素
127.0.0.1:6379> lpop list
"1"
# 取出最右边的元素
127.0.0.1:6379> rpop list
"2"
# 取出最右边的元素，并新建到另一个列表中
127.0.0.1:6379> rpoplpush list newlist
"JIA"
# 在指定值左边before或右边after插入新值,返回列表长度。如world左边和hello右边
127.0.0.1:6379> linsert list before world new
(integer) 3
127.0.0.1:6379>  linsert list after hello the
(integer) 4

# 按下标索引从左边取值
127.0.0.1:6379> lindex list 0
"2"
# 查看列表长度
127.0.0.1:6379> llen list
(integer) 2
# 从左开始删除指定数目和value的元素，返回删除数量
127.0.0.1:6379> lrem list 1 JIA
(integer) 1
# 指定下标修改值，前提列表存在，且下标元素存在
127.0.0.1:6379> lset list 0 jia3
OK
```

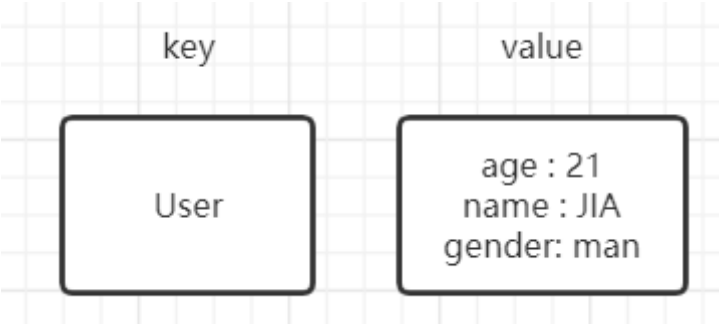
5）Redis数据类型之Set

> Set与List区别在于Set可自动排重，但无序。不过set可以查看某个成员是否在set内，而list做不到。Set的底层是一个value为null的hash表。

```
# 添加元素，已存在元素被忽略，返回添加成功的元素个数
127.0.0.1:6379> sadd set 1 2 3
(integer) 3
# 取出集合所有元素
127.0.0.1:6379> smembers set
1) "1"
2) "2"
3) "3"
# 判断某个值是否存在，1表示有，0表示没有
127.0.0.1:6379> sismember set 1
(integer) 1
# 获取集合元素个数
127.0.0.1:6379> scard set
(integer) 3
# 删除元素，1表示删除成功，0表示没删除，即没有这个元素
127.0.0.1:6379> srem set 1
(integer) 1
# 随机弹出一个值
127.0.0.1:6379> spop set
"4"
# 随机取出几个值，但不会弹出
127.0.0.1:6379> srandmember set 2
1) "1"
2) "3"
# 移动元素到另一个集合，1表示移动成功
127.0.0.1:6379> smove set set1 1
(integer) 1
# 取多个集合交集，只填一个相当于smembers
127.0.0.1:6379> sinter set set1 set2
1) "2"
2) "3"
# 取并集
127.0.0.1:6379> sunion set set1 set2
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
# 取差集
127.0.0.1:6379> sdiff set set1 set2
1) "1"
2) "4"
```

6）Redis数据类型之Hash

> hash是一个键值对集合，相当于Map<String, Object>，适合用于存储对象，value使用fileid和filedValue存储。hash对应的数据结构有两种，field-value少的时候用ziplist，否则用hashtable

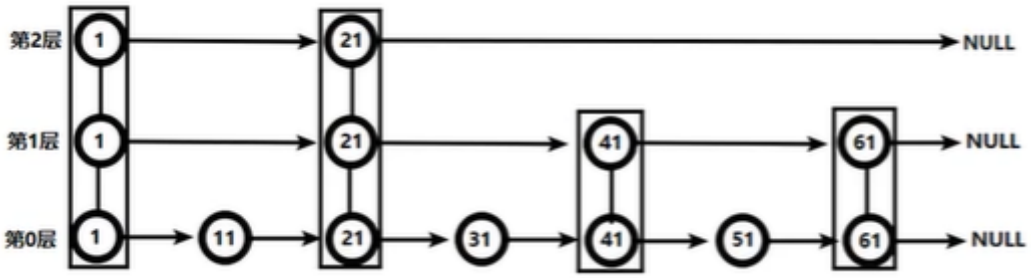



```
# 给键值对集合添加字段，返回添加的字段个数
127.0.0.1:6379> hset user:001 age 21 name JIA
(integer) 2
# 字段不存在时才添加，只允许添加1个字段，返回1表示添加成功
127.0.0.1:6379> hsetnx user:001 age 21
(integer) 0
# 获取键值对集合的指定字段的值
127.0.0.1:6379> hget user:001 age
"21"
# 判断某个字段是否存在，1表示存在
127.0.0.1:6379> hexists user:001 name
(integer) 1
# 查询所有字段名称
127.0.0.1:6379> hkeys user:001
1) "age"
2) "name"
3) "gender"
# 查询所有字段的值
127.0.0.1:6379> hvals user:001
1) "21"
2) "JIA"
3) "man"
# 指定字段值增加，返回增加后数值
127.0.0.1:6379> hincrby user:001 age 10
(integer) 31
# 删除指定字段，1表示删除成功
127.0.0.1:6379> hdel user:001 name
(integer) 1
```

7）Redis数据类型之ZSet

> ZSet是一个有序Set。集合中每个成员都关联一个score，被用于从最低分到最高分进行排序，成员唯一，但是score可以重复。

> ZSet类似于Map<String,Double>，每个元素有自己的权重。它的底层包括两个数据结构，一个是hash，用于关联value和score，保障value唯一，可通过value获取score。另一个是跳跃表，用于给元素value排序，提供根据score范围获取元素列表。




```
# 添加评分和成员，返回添加个数
127.0.0.1:6379> zadd zset 100 ZURI 99 JIA
(integer) 2
# 从集合中取值，按排序从最小开始取出第0到第1个元素，0 -1表示所有元素
127.0.0.1:6379> zrange zset 0 1
1) "JIA"
2) "ZURI"
# 取值并输出评分
127.0.0.1:6379> zrange zset 0 -1 withscores
1) "JIA"
2) "99"
3) "ZURI"
4) "100"
# 从集合中取值，从大到小排序取出第0到第9个元素
127.0.0.1:6379> zrevrange zset 0 9
1) "JIA"
2) "ZURI"
# 指定评分范围取值，参数为min max
127.0.0.1:6379> zrangebyscore zset 99 100
1) "JIA"
2) "ZURI"
# 指定评分范围从大到小取出值，参数为max min
127.0.0.1:6379> zrevrangebyscore zset 100 99
1) "ZURI"
2) "JIA"
# 增加指定值的评分，返回增加后评分
127.0.0.1:6379> zincrby zset 100 JIA
"199"
# 删除元素，1表示删除成功
127.0.0.1:6379> zrem zset JIA
(integer) 1
# 统计分数范围内的元素个数
127.0.0.1:6379> zcount zset 100 200
(integer) 2
```

```
# 查询排名，从小到大排名，从0开始
127.0.0.1:6379> zrank zset JIA
(integer) 1
```

8）Redis特殊数据类型之Bitmaps

> Bitmaps本身不算一种新的数据类型，它是对String（Key-Value）的一种位的表示，允许位操作。相当于一个以0和1作为元素的数组，下标也称为偏移量。

```
# 设置指定下标的值，成功返回0。例如可用于通过0，1表示是否用户当天登录，如20号用户今天登录
# 如果初始化设置偏移量太大，执行会很慢，因为需要定义前面几位全为0
127.0.0.1:6379> setbit user:20210705 20 1
(integer) 0
# 获取指定偏移量的值
127.0.0.1:6379> getbit user:20210705 20
(integer) 1
# 统计位为1的数量，可指定byte位置范围。
# 注意8个bit为一个byte，这里按照byte范围来查找
127.0.0.1:6379> bitcount user:20210705
(integer) 2
127.0.0.1:6379> bitcount user:20210705 0 -1
(integer) 2
# 复合操作，与and、或or、非not、异或xor，将结果保存在指定新bitmaps，如result
 127.0.0.1:6379> bitop and result user:20210705 user:20210706
(integer) 1
```

9）Redis特殊数据类型之HyperLogLog

> 常遇到需要统计独立访客、独立IP数、搜索记录数等需要去重和统计的情况，这类求不重复个数元素个人的问题称为基数问题。
>
> 常见解决基数问题的方案有：
>
> (1)使用MySQL，通过distinct count来去重并统计
>
> (2)使用Redis的hash、set、bitmaps等来处理
>
> 但由于数据量不断扩大，会导致占用空间越来越大，因此采用HyperLogLog来降低精度来平衡空间，每个HyperLogLog只需12KB内存，即可计算出2^64不同元素基数。
>
> HyperLogLog有点像Set，但它只保存基数，而不保存元素本身，因此不会占用太大内存。

```
# 添加元素，若成功引起近似基数变化，返回1，否则返回0
127.0.0.1:6379> pfadd language Chinese
(integer) 1
# 统计近似元素个数
127.0.0.1:6379> pfcount language
(integer) 3
# 将多个HLL合并后结果存在新的HLL中
127.0.0.1:6379> pfmerge result language language1
OK
```

10）Redis特殊数据类型之Geospatial

> Geospatial为二维坐标，redis基于该类型提供了经纬度设置，查询，距离查询等常见操作

```
# 增加地理位置到集合中，参数为经度，纬度，返回添加元素个数
# 经度范围为-180~180，纬度范围为-85.05~85.05，两极无法直接添加
127.0.0.1:6379> geoadd china 116.20 39.56 beijing 120.52 30.40 shanghai
(integer) 2
# 取出集合的元素位置
127.0.0.1:6379> geopos china shanghai
1) 1) "120.52000075578689575"
   2) "30.39999952668997452"
# 取两地直线距离，可选单位m、km、mi(英里)、ft(英尺)
127.0.0.1:6379> geodist china shanghai beijing km
"1091.8689"
# 指定经纬度，找出半径内所有元素，可选单位和上面一致
127.0.0.1:6379> georadius china 110 30 2000 km
1) "shanghai"
2) "beijing"
```

Redis配置文件

> 默认缩写配置：1k => 1000 bytes，1kb => 1024 bytes，1m => 1000000 bytes，1mb => 1024*1024 bytes*，*1g => 1000000000 bytes*，*1gb => 1024*1024*1024 bytes，表示可以用1k、5g等直接表示。
>
> include：可添加如include /path/to/local.conf，include /path/to/other.conf 同时使用多个配置文件
>
> network：bing=127.0.0.1 表示只接受127.0.0.1的主机访问，注释或删除后允许别的主机访问
>
> protected-mode：yes 表示不支持远程访问，改成yes支持远程访问

> tcp-backlog：设置的是一个连接队列，backlog=未完成三次握手队列+完成三次握手队列，高并发情况下需要把这个值改大点才能接收多个并发。但Linux内核会将它减小到/proc/sys/net/core/somaxconn的值(128)，所以需要同时增大/proc/sys/net/core/somaxconn和/proc/sys/net/piv4/tcp_max_syn_backlog来达到效果timeout：连接超时时间，0表示永不超时
>
> tcp-keepalive：检测存活时间间隔，每经过这段时间检测一次该连接是否还在操作，没有的话就断开连接
>
> daemonize：yes表示后台启动，no表示前台启动
>
> pid_file：redis的启动进程号存放文件
>
> loglevel：日志级别，有debug，verbose，notice(默认)，warning
>
> database：默认16个数据库
>
> requirepass：是否需要密码，密码为什么
>
> maxclients：最大客户端数量
>
> maxmemory和maxmemory-policy：最大内存和数据移除规则

Redis的发布/订阅

> 消息并不会持久化，后面订阅的收到不到之前的信息。

```
# 订阅
127.0.0.1:6379> subscribe mychannel
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "mychannel"
3) (integer) 1
# 发布
127.0.0.1:6379> publish mychannel hello
(integer) 1
# 收到的信息格式
127.0.0.1:6379> subscribe mychannel
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "mychannel"
3) (integer) 1
1) "message"
2) "mychannel"
3) "hello"
```

Redis的操作之Jedis

1）Jedis依赖

```
<dependency>
        <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version></version>
</dependency>
```

2）准备工作

> 修改Redis的配置文件，将bind绑定可访问的主机IP地址加上测试主机IP或者注释掉。同事将本地访问保护protected-mode设置为no。

```
# 同时服务器对外开放6379端口
# 查看端口是否开发
[root@Master bin]# firewall-cmd --query-port=6379/tcp
no
# 添加指定端口开放
[root@Master bin]# firewall-cmd --add-port=6379/tcp --permanent
success
# 重载防火墙
[root@Master bin]# firewall-cmd --reload
success
```

3）API基本操作与命令行一致

```
Jedis jedis = new Jedis("192.168.100.100", 6379);
// 其他操作与命令行基本一致，如Jedis.ping/get/set/sadd/zadd/hset/pfadd等
```

4）Spring boot整合Redis

4.1）依赖

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<!-- redis常用连接池 -->
<dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-pool2</artifactId>
        <version></version>
</dependency>
```

4.2）配置信息

```
# redis服务器地址
spring.redis.host=192.168.100.100
# redis端口
spring.redis.port=6379
# redis数据库索引
spring.redis.database=0
# redis连接超时时间(毫秒)
spring.redis.timeout=6000
# redis连接池最大连接数，负数表示无限制
spring.redis.lettuce.pool.max-active=20
# redis最大阻塞等待时间，负数表示无限制
spring.redis.lettuce.pool.max-wait=-1
# reids连接池最大空闲连接数，idle 闲置的
spring.redis.lettuce.pool.max-idle=5
# redis连接池最小空闲连接数
spring.redis.lettuce.pool.min-idle=0
```

4.3）添加Redis配置类

```
@EnableCaching
@Configuration
public class RedisConfigs extends CachingConfigurerSupport {

    @Bean
    public RedisTemplate<String, Object> redisTemplate
                                                    (RedisConnectionFactory factory){
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        RedisSerializer<String> redisSerializer = new StringRedisSerializer();
        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer
                                        = new Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jackson2JsonRedisSerializer.setObjectMapper(om);
        template.setConnectionFactory(factory);
        // key序列化
        template.setKeySerializer(redisSerializer);
        // value序列化
        template.setValueSerializer(jackson2JsonRedisSerializer);
        // value hashmap序列化
        template.setHashValueSerializer(jackson2JsonRedisSerializer);
        return template;
    }

    @Bean
    public CacheManager cacheManager(RedisConnectionFactory factory){
        RedisSerializer<String> redisSerializer = new StringRedisSerializer();
        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer
                                        = new Jackson2JsonRedisSerializer(Object.class);
        // 解决查询缓存转化异常问题
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jackson2JsonRedisSerializer.setObjectMapper(om);
        // 配置序列化，解决乱码问题，过期时间600
        RedisCacheConfiguration config
           = RedisCacheConfiguration.defaultCacheConfig()
                .entryTtl(Duration.ofSeconds(600))
                                .serializeKeysWith(
RedisSerializationContext.SerializationPair.fromSerializer(redisSerializer))                    .
RedisSerializationContext.SerializationPair.fromSerializer(jackson2JsonRedisSerializer))
                .disableCachingNullValues();

        RedisCacheManager cacheManager = RedisCacheManager.builder(factory)
                .cacheDefaults(config)
                .build();
        return cacheManager;
    }

}
```

4.4）使用

```
@Autowired
private RedisTemplate redisTemplate;

redisTemplate.opsForValue().set("name", "JIA");
Object name = redisTemplate.opsForValue().get("name");
```

5）Redis事务

5.1）操作

> 通过Multi开始组队，依次放入命令，后使用Exec按顺序执行队列的命令，Discard可以放弃组队。

```
# 正常组队，执行
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set name JIA
QUEUED
127.0.0.1:6379(TX)> set name2 ZURI
QUEUED
127.0.0.1:6379(TX)> exec
1) OK
2) OK
# 正常组队，放弃组队
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set name JIA
QUEUED
127.0.0.1:6379(TX)> set name2 ZURI
QUEUED
127.0.0.1:6379(TX)> discard
OK
# 组队失败，全部放弃
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set name JIA
QUEUED
127.0.0.1:6379(TX)> set name2
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379(TX)> exec
(error) EXECABORT Transaction discarded because of previous errors.
```

```
# 组队正常，极个别队列元素出错，其他的会正常运行
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set name JIA
QUEUED
127.0.0.1:6379(TX)> incr name
QUEUED
127.0.0.1:6379(TX)> exec
1) OK
2) (error) ERR value is not an integer or out of range
```

5.2）加锁

> 悲观锁：每次操作前都认为数据可能被修改，因此对该数据进行加锁防止并发。
>
> 乐观锁：假设数据一般不会造成冲突，当进行更新提交时才检验是否起冲突。常用版本号检测冲突（Compare and Swap）

```
# 监视，在执行事务前监视
# 机器1(先)，执行前监视name，版本号一致执行成功
127.0.0.1:6379> watch name
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set name ZURI
QUEUED
127.0.0.1:6379(TX)> exec
1) OK
# 机器2(后)，执行前监视name，版本号已被改动，执行失败
127.0.0.1:6379> watch name
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set name JIA2
QUEUED
127.0.0.1:6379(TX)> exec
(nil)
```

5.3）Redis事务是不保证原子性的

5.4）Redis事务案例之秒杀

> linux实现并发请求
>
> 工具：yum install -y httpd-tools

> 命令：ab -n 1000 -c 100 http://192.168.100.100:8080/test/test2

```
List<Integer> nums = redisTemplate.opsForSet().randomMembers("nums", 4);
StringBuilder id = new StringBuilder();
nums.stream().forEach(num -> id.append(num));
// 事务
redisTemplate.execute(new SessionCallback() {
        @Override
        public Object execute(RedisOperations redisOperations) throws
                                                                DataAccessException {
        // 这里用的乐观锁，可能造成库存问题，即同一时间只有一个成功，但库存还有
                if (redisOperations.opsForSet().isMember("order", id.toString())){
                System.out.println("已经参与过");
            return null;
        }
        redisOperations.watch("product:001");
        Integer productCount
            = (Integer) redisOperations.opsForValue().get("product:001");
        if (productCount <= 0){
            System.out.println("商品已经被抢完");
            return null;
        }
        redisOperations.multi();
        redisOperations.opsForSet().add("order", id.toString());
        redisOperations.opsForValue().decrement("product:001");
        List<Object> result = redisOperations.exec();
        if (result != null){
            System.out.println(id + "用户抢到了商品");
        }
        return null;
    }
});
```

> 为了解决库存问题，即这类并发任务导致的问题。引入Lua脚本，搭配2.6以上的Redis组成任务队列，解决争抢问题。即利用了Redis的单线程性，形成串行。

6）Redis持久化

> Redis提供了两种不同的持久化方式，即RDB(Redis DataBase)、AOF(Append Of File)。
>
> RDB：备份执行时，单独创建一个子进程fork来执行持久化， 将数据写入一个临时文件中，等持久化完成后，再用临时文件替换上次持久化好的文件。若需要大规模恢复，且完整性要求不高，则RDB效率较高，但RDB的缺点是最后一次持久化的数据可能丢失(即最后的数据不满足快照生产条件，如只变化了5个key)。

- 在指定时间间隔内将内存中的数据集快照写入磁盘，到时候可以直接将快照文件读到内存中

- 配置文件中dbfilename dump.rdb指的是快照的名称，save 36 10指的是36秒内至少10个key发送改变都会写到快照去(默认配置)。而假设36秒内有12个Key发生了变化，则前10个先被写持久化，后面的重新开始计时。
- 配置中save为阻塞保存，阻塞其他全部操作；bgsave为异步，快照同时响应请求。
- stop-writes-on-bgsave-error yes 无法写入磁盘时关闭Redis的写操作
- rdbcompression yes 压缩快照文件
- rdbchecksum yes 检查数据完整性
- 恢复备份：redis启动之后会默认使用dump.rdb

> AOF：以日志的形式来记录每个写操作，将Redis执行过的所有写指令记录下来，只追加文件，而不改写文件。

- aof默认不开启，appendonly no改为yes，若AOF和RDB同时开启优先使用AOF
- redis-check-aof-fix appendonly.aof可以修复异常的aof文件
- appendfsync always/everysec/no 同步频率(每个写记录/每秒/不主动同步，交给操作系统)
- rewrite AOF文件持续增大时，会fork一个新进程来将文件重写(重写命令)

7）主从配置

```
# 在从机上加入服从于主机，从机只可以做读操作
127.0.0.1:6379> slaveof 192.168.100.100 6379
OK
# 查看主机状态，主服务器宕机后还是主服务器，从不会变主
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:192.168.100.100
master_port:6379
master_link_status:down
master_last_io_seconds_ago:-1
master_sync_in_progress:0
slave_repl_offset:0
master_link_down_since_seconds:-1
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:1a2e5e2feba31a36510c506eb8418d423cf8858f
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
```

```
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

> 哨兵模式启动redis-sentinel sentinel.conf。在主从都启动后再启动哨兵，此时如果主机宕机了，选为哨兵的那个可以当选主机。宕机的老主机重连后变成从机。