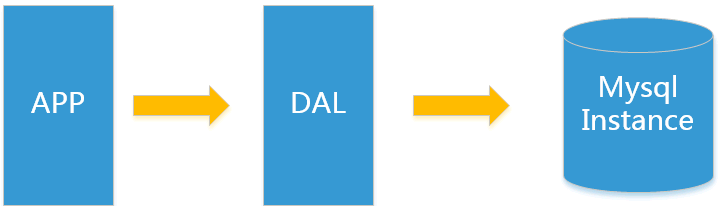
# NoSql\_Redis

## NoSql入门和概述

### 1.互联网时代背景下大机遇，为什么用nosql

#### 单机MySQL的美好年代

在90年代，大多都是静态网页，并且网站的访问量一般都不大，单个数据库足以应付。



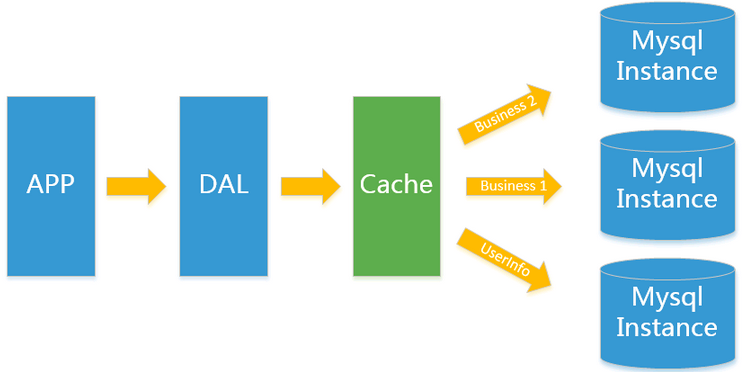
上述架构下，我们来看看数据存储的瓶颈是什么？

1.数据量的总大小 一个机器放不下

2.数据的索引（B+ Tree） 一个机器的内存放不下

3.访问量(读写混合) 一个实例不能承受

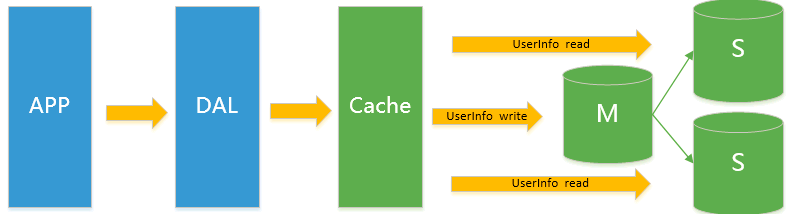
#### Memcached(缓存)+MySQL+垂直拆分

****

随着访问量的上升，数据库上开始出现了性能问题，web程序不再仅仅专注在功能上，同时也在追求性能。程序员们开始大量的使用缓存技术来缓解数据库的压力，优化数据库的结构和索引。开始比较流行的是通过文件缓存来缓解数据库压力，但多台web机器通过文件缓存不能共享，大量的小文件缓存也带来了比较高的IO压力。在这个时候，Memcached出现了。

Memcached作为一个独立的分布式的缓存服务器，为多个web服务器提供了一个共享的高性能缓存服务，在Memcached服务器上，又发展了根据hash算法来进行多台Memcached缓存服务的扩展，然后又出现了一致性hash来解决增加或减少缓存服务器导致重写hash带来的大量缓存失效的弊端

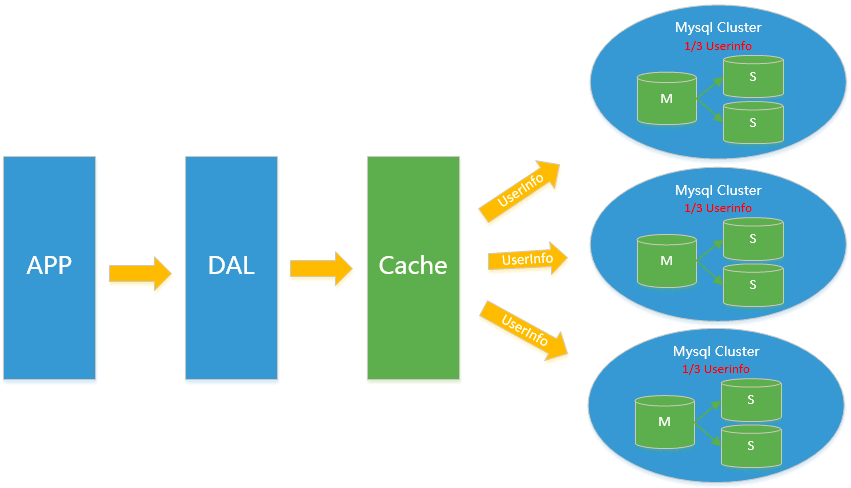
#### Mysql主从读写分离



由于数据库的写入压力增加，Memcached只能缓解数据库的读取压

力。读写集中在一个数据库上让数据库不堪重负，大部分网站开始使用主从复制技术来达到读写分离，以提高读写性能和读库的可扩展性。Mysql的master-slave模式成为这个时候的网站标配了。

#### 分表分库+水平拆分+mysql集群



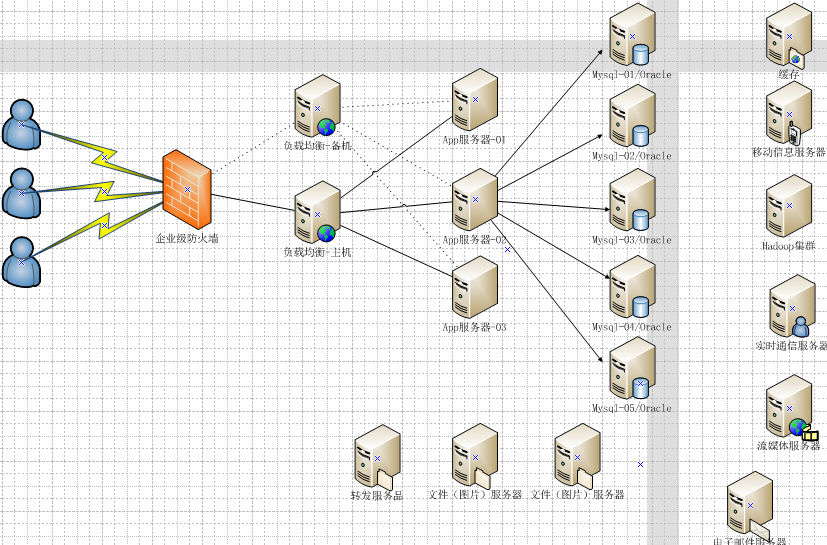
MySQL主库的写压力又开始出现问题，由于MyISAM使用表锁，在大量的高并发下会出现严重的锁问题，这时MySQL应用开始使用InnoDB引擎代替MyISAM。

同时，开始流行使用分表分库来缓解写压力和数据增长的扩展问题。这个时候，分表分库成了一个热门技术，也就在这个时候，MySQL推出了还不太稳定的表分区，这也给技术实力一般的公司带来了希望。虽然MySQL推出了MySQL Cluster集群，但性能也不能很好满足互联网的要求，只是在高可靠性上提供了非常大的保证。

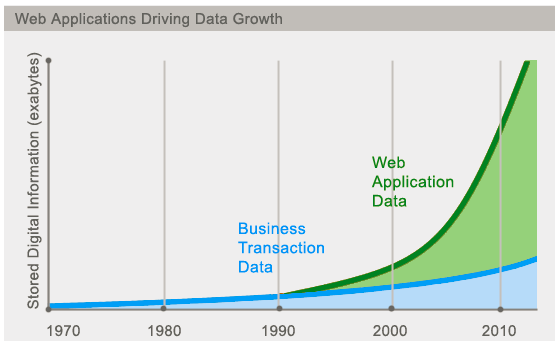
#### 5.MySQL的扩展性瓶颈

MySQL存储一些大文本字段会导致数据库表非常的大，做数据库恢复的时候就不容易快速恢复数据库。所以关系数据库并不能很好的应付所有的应用场景。MySQL的扩展性差（需要复杂的技术来实现），大数据下IO压力大，表结构更改困难。

#### 6.今天是什么样子？？



#### 7.为什么用NoSQL



今天我们可以通过第三方平台容易的访问和抓取数据。用户的个人信息，社交网络，地理位置，用户生成的数据和用户操作日志已经成倍的增加。我们如果要对这些用户数据进行挖掘，那SQL数据库已经不适合这些应用了，NoSQL数据库的发展也却能很好的处理这些大的数据。

### 2.是什么？

NoSQL(NoSQL = Not Only SQL )，泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在应付web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题，包括超大规模数据的存储。

例如谷歌或Facebook每天为他们的用户收集万亿比特的数据。这些类型的数据存储不需要固定的模式，无需多余操作就可以横向扩展。

### 3.能干什么？

#### 1.易扩充

NoSQL数据库种类繁多，但是都是去掉了关系型特性，这样就非常容易扩展，也在架构的层面上带来了可扩展的能力。

#### 2.大数据量高性能

NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。这得益于它的无关系性，数据库的结构简单。

一般MySQL使用Query Cache，每次更新表Cache就失效，性能不高，是一种大粒度的Cache，而NoSQL的Cache是细粒度的Cache，是记录级的，故能高很多

#### 3.多样灵活的数据模型

NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增加字段比较麻烦

#### 4.RDBMS vs NoSQL

RDBMS - 高度组织化结构化数据

- 结构化查询语言（SQL）

- 数据和关系都存储在单独的表中。

- 数据操纵语言，数据定义语言

- 严格的一致性

- 基础事务

NoSQL

- 代表着不仅仅是SQL

- 没有声明性查询语言

- 没有预定义的模式

- 键值对存储，列存储，文档存储，图形数据库

- 最终一致性，而非ACID属性

- 非结构化和不可预知的数据

- CAP定理

- 高性能，高可用性和可伸缩性

### 4.3V和3高

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

### 5.NoSql数据模型

传统的关系型数据库通过表来建立数据关系，而Nosql可以使用Bson

{"customer":{"id":1136, "name":"Z3","billingAddress":[{"city":"beijing"}],

"orders":[ {"id":17, "customerId":1136, "orderItems":[{"productId":27,"price":77.5,"productName":"thinking in java"}],

"shippingAddress":[{"city":"beijing"}]

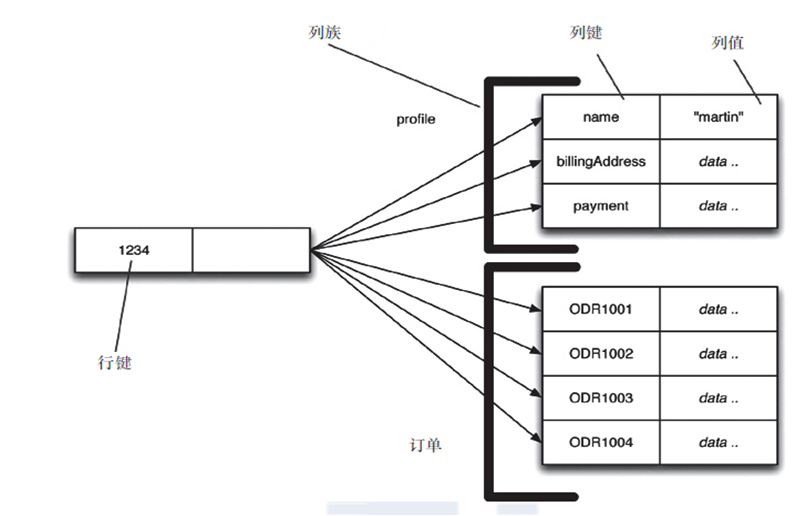
"orderPayment":[{"ccinfo":"111-222-333","txnid":"asdfadcd334",

"billingAddress":{"city":"beijing"}}]}}

#### 1.KV键值对

#### 2.Bson

#### 3.列族

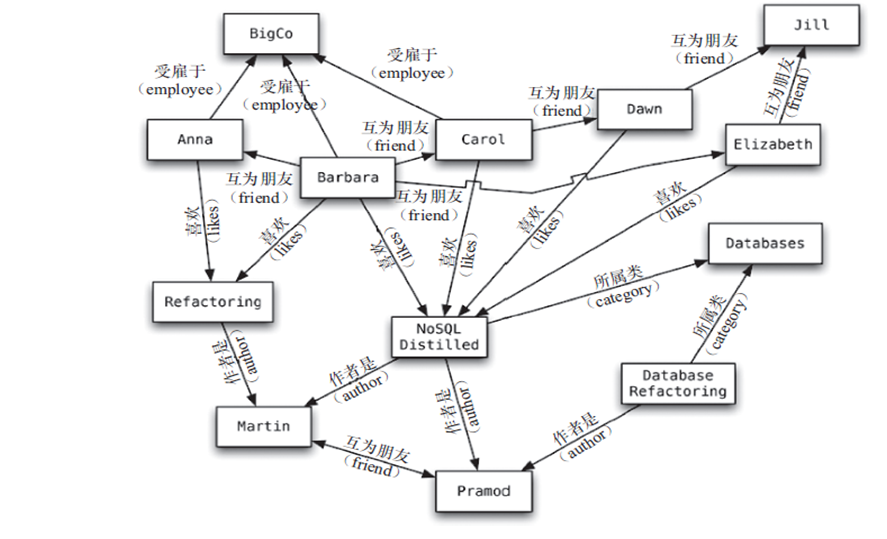


方便存储结构化和半结构化数据

方便做数据压缩，

对针对某一列或者某几列的查询有非常大的IO优势。

#### 4.图形



### 6.NoSQL数据库的四大分类

#### KV键值对

新浪：BerkeleyDB+redis 美团：redis+tair 阿里、百度：memcache+redis

#### 文档数据库，Bson格式较多

CouchDB MongoDB

MongoDB 是一个基于分布式文件存储的数据库。由 C++ 语言编写。旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。

#### 列存储数据库

Cassandra, Hbase 分布式文件系统

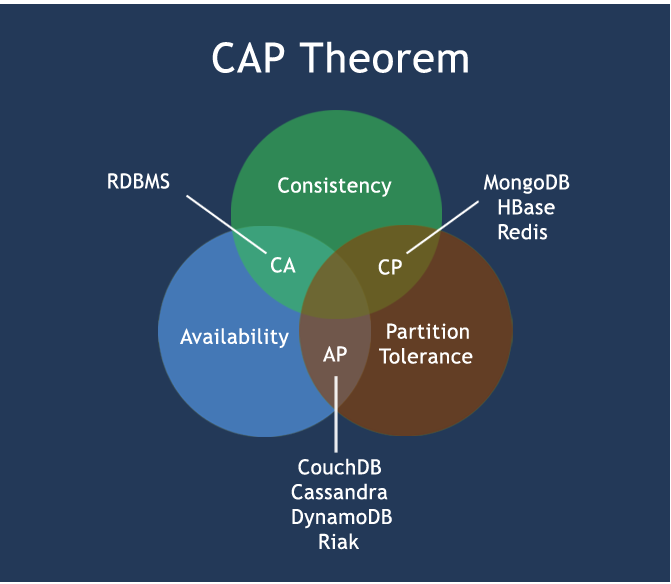
#### 图关系数据库

它不是放图形的，放的是关系，专注于构建关系图谱 Neo4J, InfoGrid

#### 对比



### 7.CAP的三进二



CAP理论的核心：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。

Consistency:强一致性 Availability：高可用性 Partition tolerance：分布式容忍性

没有NoSQL系统能同时保证三点，由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容忍性是必须要实现，分布式架构的时候必须做出取舍。所以只能在一致性和可用性之间进行权衡。多余大多数web应用并不需要强一致性。因此牺牲C换取P

CA - 单点集群，通常在可扩展性上不太强大。传统Oracle数据库通常使用。

CP - 通常性能不是特别高。大多数网站架构的选择使用。

AP - 通常可能对一致性要求低一些。Redis、Mongodb

一致性与可用性的决择 对于web2.0网站来说，关系数据库的很多主要特性往往无用武之地。

事务一致性的需求

很多web实时系统并不要求严格的数据库事务，对读一致性的要求很低， 有些场合对写一致性要求并不高。允许实现最终一致性。

写实性和读实性的需求

对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比方说 发一条消息之后，过几秒乃至十几秒之后，我的订阅者才看到这条动态是完全可以接受的。

复杂Sql的需求

任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的报表查询，特别是SNS类型的网站，从需求以及产品设计 角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大的弱化了。

### 8.BASE

BASE是为了针对关系数据库强一致性导致可用性降低而提出的解决方案。

BASE其实是三个术语的缩写：基本可用（Basically Available）、软状态（Soft state）、最终一致（Eventually consistent）

它的思想是通过让系统放松对某一时刻数据一致性的要求来换取系统整体伸缩性和性能上改观。大型系统往往由于地域分布和性能的要求，不可能采用分布式事务来完成这些指标，要想获得这些指标，我们必须采用另外一种方式来完成，这里BASE就是解决这个问题的办法

### 9.分布式和集群

分布式系统（distributed system）是由多台计算机和通信的软件组件通过计算机网络连接（本地网络或广域网）组成。分布式系统是建立在网络之上的软件系统。正是因为软件的特性，所以分布式系统具有高度的内聚性和透明性。因此，网络和分布式系统之间的区别更多的在于高层软件（特别是操作系统），而不是硬件。分布式系统可以应用在在不同的平台上如：Pc、工作站、局域网和广域网上等。

简单来讲，分布式指不同的多台服务器上面部署不同的服务模块（工程），他们之间通过Rpc/Rmi之间通信和调用，对外提供服务和组内协作。

集群指不同的多台服务器上面部署相同的服务模块，通过分布式调度软件进行统一的调度，对外提供服务和访问。

## Redis入门介绍

### 1.是什么？

Remote DIctionary Server(远程字典服务器)是完全开源免费的，用C语言编写的，遵守BSD协议，是一个高性能的(key/value)分布式内存数据库，基于内存运行并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之一,也被人们称为数据结构服务器

Redis 与其他 key - value 缓存产品有以下三个特点：

Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用

Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储

Redis支持数据的备份，即master-slave模式的数据备份

### 2.能干嘛？

内存存储和持久化：redis支持异步将内存中的数据写到硬盘上，同时不影响继续服务

取最新N个数据的操作，如：可以将最新的10条评论的ID放在Redis的List集合里面

可以设定类似于HttpSession过期时间的功能

发布、订阅消息系统

定时器、计数器

### 3.Linux版Redis安装

1.解压目录/opt下的redis-3.0.4.tar.gz（tar -zxvf redis-3.0.4.tar.gz）。进入到解压目录执行make命令，这步如果出错，需要yum install gcc-c++。再执行make install命令

2.默认安装目录：usr/local/bin

redis-benchmark:性能测试工具，可以在自己本子运行，看看自己本子性能如何

redis-check-aof：修复有问题的AOF文件，rdb和aof后面讲

redis-check-dump：修复有问题的dump.rdb文件

redis-cli：客户端，操作入口

redis-sentinel：redis集群使用

redis-server：Redis服务器启动命令

#### 3.启动和关闭

1. cp redis.conf /myredis/

2.修改拷贝文件中的daemonize 设置为 yes

3. redis-check-dump：修复有问题的dump.rdb文件

4. redis-server /myredis/redis.conf 启动

5.redis-cli –p 6379 连接，测试连同：ping，连通成功会响应pong

6. redis-cli shutdown 关闭

#### 4.其他知识点

1.单进程模型来处理客户端的请求。对读写等事件的响应是通过对epoll函数的包装来做到的。Redis的实际处理速度完全依靠主进程的执行效率

Epoll是Linux内核为处理大批量文件描述符而作了改进的epoll，是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率。

2.默认有16个数据库，类似数组下表从零开始，初始默认使用0号库

select ~ ：切换库

dbsize ：查看当前库存在key的数量

flushdb ： 清空当前库中的key

flushall ： 清空所有库中的key

3.统一密码管理，16个库都是同样密码，要么都OK要么一个也连接不上

## Redis数据类型

### 1.String

string是redis最基本的类型，可以理解成与Memcached一模一样的类型，一个key对应一个value。

string类型是二进制安全的，redis的string可以包含任何数据，比如jpg图片或者序列化的对象。

string类型是Redis最基本的数据类型，一个redis中字符串value最多可以是512M

### 2.Hash

Redis hash 是一个键值对集合，是一个string类型的field和value的映射表，类似Java里面的Map<String,Object>特别适合用于存储对象。

### 3.List

Redis列表是简单的字符串列表，按照插入顺序排序，它的底层实际是个链表，所以可以前后插入数据

### 4.Set

Redis的Set是string类型的无序集合，通过HashTable实现实现的。

### 5.Zset

Redis Zset 和 Set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数,redis正是通过分数来为集合中的成员进行从小到大的排序。Zset的成员是唯一的，但分数(score)却可以重复。

### 用法

1. key

1. keys \*

2. exists key

3. move key db

4. expire key seconds

5. ttl key -1表示永不过期，-2表示已过期

6. type key

7. del key

8. dump key 序列化，返回序列值

9. persist key 解除过期时间

10. pttl key

11. randomkey

12.rename key newkey

#### string 单值单value

1. set key kvalue
2. get key
3. append key
4. strlen key
5. incr key
6. decr key
7. incrby key increment
8. decrby key decrement
9. getrange key begin end
10. setrange key index value
11. setex key seconds value 设置时定义过期时间
12. setnx key value 不存在才设置
13. mset key1 value1 key2 value2 …
14. mget key1 key2…
15. msetnx key1 value1 key2 value2 … 一个存在，全部无法设置
16. getset key value 会返回旧值
17. incrbyfloat key value

#### list 单值多value

1. lpush key value1 value2… key不存在，则创建，存在则新增
2. rpush key value1 value2…
3. lrange key begin end
4. lpop key 左右可以操作，效率高
5. rpop key
6. lindex key index 中间也可以操作，效率低
7. llen key
8. lrem key count value 删除几个指定value值，值全移除，key消失
9. ltrim key index1 index2
10. rpoplpush key1 key2
11. lset key index value
12. linsert key before/after value newvalue
13. blpop key seconds 如果发现直接弹出，规定时间内如果没有发现就不弹出，发现后立即弹出，返回key和弹出的value
14. brpop key seconds
15. brpoplpush key1 key2 seconds
16. lpushx key value
17. rpushx key value

#### set 单值多value，value不重复

1. sadd key value1 value2…
2. scard key
3. smembers key
4. sismember key value
5. srem key value
6. srandmember key
7. spop key
8. smove key1 key2 key1’value
9. sdiff key1 key2 差集
10. sinter key1 key2 交集
11. sunion key1 key2 并集
12. sdiffstore key3 key1 key2
13. sinterstore key3 key1 key2
14. sunionstore key3 key1 key2

#### hash 单值单value，单value多键值对

1. hset key field value
2. hget key field
3. hmset key field1 value1 field2 value2…
4. hmget key field1 field2
5. hgetall key
6. hdel key field1 field2
7. hlen key
8. hexists key field
9. hkeys key
10. hvals key
11. incrby key field increment
12. hincrbyfloat key field increment
13. hsetnx key field value
14. hdel key field1 field2

#### zset 在set之前添加score指定自己的区域，指定范围时在数字前加 “(” 表示不包含

1. zadd key score1 value1 srore2 value2…
2. zrange key begin end（withscores）
3. zrangebyscore key beginscore endscore（withscores）（limit index count）
4. zrem key value1 value2…
5. zcard key 获取个数
6. zcount key minscore maxscore 获取区间中的个数
7. zrank key value 获取下标
8. zscore key value 获取score
9. zrevrank key value 获取反序下标
10. zrevrangbyscore key endscore beginscore（withscores）（limit index count） 反
11. zincrby key increment value
12. zinterstore key3 numkeys key1 key2… 并集，以value作条件
13. zremrangbyrank key beginindex endindex
14. zremrangbyscore key mixscore maxscore
15. zunionstore key3 numkeys key1 key2 并集

## redis.conf配置文件

### 单位

# 1k => 1000 bytes

# 1kb => 1024 bytes

# 1m => 1000000 bytes

# 1mb => 1024\*1024 bytes

# 1g => 1000000000 bytes

# 1gb => 1024\*1024\*1024 bytes

只支持bytes不支持bit，对大小写不敏感

### 2. INCLUDES包含

# include /path/to/local.conf 指定包含其它的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，

# include /path/to/other.conf 而同时各个实例又拥有自己的特定配置文件

### 3. GENERAL通用

1. daemonize yes Redis默认不是以守护进程的方式运行，可以通过该配置项修改，使用yes启用守护进程
2. pidfile /var/run/redis.pid 当Redis以守护进程方式运行时，Redis默认会把pid写入/var/run/redis.pid文件，可以通过pidfile指定
3. port 6379 Redis监听端口
4. tcp-backlog 511

backlog其实是一个连接队列，backlog队列总和 = 未完成三次握手队列 + 已经完成三次握手队列。

在高并发环境下你需要一个高backlog值来避免慢客户端连接问题。注意Linux内核会将这个值减小到/proc/sys/net/core/somaxconn 的值，所以需要确认增大somaxconn和tcp\_max\_syn\_backlog两个值来达到想要的效果

1. timeout 300 客户端闲置指定秒数会关闭连接
2. bind 127.0.0.1 绑定主机地址
3. tcp-keepalive 0 检测连接状态的时间间隔
4. loglevel verbose 日志级别：有debug、verbose、notice、warning，默认为verbose
5. logfile stdout 日志记录方式，默认为标准输出，如果配置Redis为守护进程方式运行，而这里又配置为日志记录方式 为标准输出，则日志将会发送给/dev/null
6. syslog-enabled no 是否输出日志到syslog中
7. syslog-ident redis 日志标识
8. syslog-facility local0 日志输出设备，可以是user、local0~7
9. databases 16 设置数据库的数量，默认数据库为0

### 4. SNAPSHOTTING快照

1. save <seconds> <changes> 指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合

如果想禁用RDB持久化的策略，只要不设置任何save指令，或者给save传入一个空字符串参数也可以

2. dbfilename dump.rdb 指定本地数据库文件名，默认值为dump.rdb，这就是RDB的保存文件

3. stop-writes-on-bgsave-error yes 出错时阻止写的操作，如果设置为null，表示在乎数据不一致或者有其他的手段发现和控制

4. rdbcompression yes 对于存储到磁盘中的快照，可以设置是否进行压缩存储。如果是的话，redis会采用redis会采用LZF算法 进行压缩，会消耗少量CPU，如果关闭会导致数据库文件变的巨大

5. rdbchecksum yes 在存储快照后，还可以让redis使用CRC64算法来进行数据校验，会这样做会增加大约10%的性能消耗

6. dir ./ 指定本地数据库存放目录

### 5. REPLICATION复制

1. slaveof <masterip> <masterport> 设置当本机为slav服务时，设置master服务的IP地址及端口，在Redis启动时，它会自动从master进行数据同步

2. masterauth <master-password> master服务设置了密码保护时，slav服务连接master的密码

3. slave-read-only no 配置从机是否只读

4. slave-priority 100 当选主机优先级，数值最低的在Master Shutdown后为新的Master，配置为0代表永远不能成为Master

### 6. SECURITY安全

config get requirepass

config set reuirepass “123456” 动态设置密码

auth 123456 如果设置了密码设置密码，在连接数据库执行任何操作之前，需要输入密码

config get dir 获取当前开启的路径，有时配置文件就写在这个路径下

requirepass xxx 设置Redis连接密码，默认关闭

### 7. LIMITS限制

maxclients 128

设置同一时间最大客户端连接数，也就是Redis进程可以打开的最大文件描述符数，默认1000，0表示无限制，当客户端连接数到达限 制时，Redis会关闭新的连接并向客户端返回max number of clients reached错误信息

maxmemory <bytes>

指定Redis最大内存限制。Redis在启动时会把数据加载到内存中，一旦到达内存使用上限，redis将会试图移除内部数据，移除规则可以 通过maxmemory-policy来指定。当此方法处理后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis新 的vm机制，会把Key存放内存，Value会存放在swap区

如果 redis无法根据移除规则来移除内存中的数据，或者设置了“不允许移除”，那么redis则会针对那些需要申请内存的指令返回错 误信息，比如SET、LPUSH等。如果你的redis是主redis（说明你的redis有 从redis），那么在设置内存使用 上限时，需要在系统中留出 一些内存空间给同步队列缓存，只有设置 “不移除”的情况下，才不用考虑这个因素

maxmemory-policy

1. volatile-lru：使用LRU算法移除key，只对设置了过期时间的键，该算法移除最近最少使用的key

2. allkeys-lru：使用LRU算法移除key

3. volatile-random：在过期集合中移除随机的key，只对设置了过期时间的键

4. allkeys-random：移除随机的key

5. volatile-ttl：移除那些TTL值最小的key，即那些最近要过期的key

6. noeviction：不进行移除，针对写操作，只是返回错误信息

maxmemory-samples

设置样本数量，LRU算法和最小TTL算法都并非是精确的算法，而是估算值，所以你可以设置样本的大小，redis默认会检查这么多个key 并选择其中LRU的那个，默认为5

### 8. APPEND ONLY MODE追加

1.appendonly yes 指定是否开启AOF，Redis在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。 因为 redis本身同步数据文件是按上面save条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为no

2. appendfilename 指定更新日志文件名，默认为appendonly.aof

3. appendfsync everysec 指定更新日志条件，共有3个可选值

no：表示等操作系统进行数据缓存同步到磁盘（快）

everysec：出厂默认推荐，异步操作，每秒记录，如果一秒内宕机，就有数据丢失 always：同步持久化，每次发生数据变更会被立即记录到磁盘 性能较差但数据完整性比较好

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

5.auto-aof-rewrite-min-size 设置重写的基准值，一个指定大小，一个指定百分比

auto-aof-rewrite-percentage

## Redis的持久化

### RDB（Redis DataBase）

#### 1.是什么？

在指定的时间间隔内将内存中的数据集快照写入磁盘，也就是Snapshot快照，它恢复时是将快照文件直接读到内存里Redis会单独 fork一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替 换上次持久化好的 文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能。

如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。RDB的缺点是最 后一次持久化后的数据可能丢失，因为在一定间隔时间做一次备份，如果出现意外就会丢失最后一次快照后的所有修改，并且fork的时 候，内存中的数据被克隆了一份，大致2倍的膨胀性需要注意（fork的作用是复制一个与当前进程一样的进程。新进程的所有数据都和 原进程一致，但是是一个全新的进程，并作为原进程的子进程）

#### 2.触发RDB快照

1. save指令，保存时其他操作会被阻塞
2. bgsave指令，Redis会在后台异步进行快照操作，快照可以响应客户端请求。可以通过lastsave，命令获取最后一次成功执行快照的时间
3. flushall和shutdown指令，但这会使快照文件为空

#### 3.恢复和修复，将备份文件移动到 redis 本地数据库存放目录并启动服务即可，修复通过redis-check-dump –fix dump.rdb命令操作

#### 4.动态所有停止RDB保存规则的方法：redis-cli config set save ""

#### 5.总结

1. RDB是一个非常紧凑的文件
2. RDB在保存是父进程只需要fork一个子进程即可，接下在的工作全由子进程完成，但父进程在这期间不能做其他IO操作，所以RDB持久化方式可以最大化redis的性能
3. 与AOF相比，在恢复大的数据集的时候RDB更快
4. 数据丢失风险高
5. RDB需要经常fork子进程来保存数据集到硬盘，当数据较大时也就非常耗时，可能导致redis在毫秒级不能响应客户请求

### AOF（Append Only File）

#### 1.是什么？

以日志的形式来记录每个写操作，将Redis执行过的所有写指令记录下来(读操作不记录)，只许追加文件但不可以改写文件，redis重启时会根据日志文件的内容将写指令从前到后重新执行一次以完成数据的恢复工作

#### 2.恢复和修复，将备份文件移动到 redis 本地数据库存放目录并启动服务即可，修复通过redis-check-aof –fix appendonly.aof命令操作

#### 3. rewrite

1.是什么？

AOF采用文件追加方式，所以文件会越来越大，为避免特殊情况发生，新增了重写机制,当AOF文件的大小超过所设定的值时，Redis就会 启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集.可以使用命令bgrewriteaof

2. 原理

AOF文件过大时，会fork出一条进程将文件重写(也是先写临时文件最后再rename)，遍历新进程的内存中数据，每条记录有一条的set语 句。其并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写到新的aof文件中，这点和快照有点类似

3. 触发机制

Redis会记录上次重写时的AOF大小，默认是当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发，可配置重写基准值

#### 4.总结

1. AOF文件是一个只进行追加的日志文件
2. Redis在AOF文件过大时，会自动在后台进行AOF重写
3. AOF文件内容有有序且为redis协议格式，方便分析
4. 相同量的内容，AOF文件大小大于RDB文件
5. 根据使用的fsyn策略，AOF速度可能慢于RDB

### 总结

1. RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储
2. AOF持久化方式记录每次对服务器写的操作，服务器重启时会重新执行这些命令来恢复原始的数据，AOF命令以redis协议追加保存每次写的操作到文件末尾，

Redis还能对AOF文件进行后台重写，使得AOF文件的体积不至于过大

1. 如果只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.
2. 同时开启两种持久化方式

这种情况下，当redis重启的时候会优先载入AOF文件来恢复原始的数据，因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整.

但也不能因为这个原因而只用AOP，因为RDB更适合用于备份数据库，快速重启，而且不会有AOF可能潜在的bug，留着作为一个万一的手段。

1. 性能建议

因为RDB文件只用作后备用途，建议只在Slave上持久化RDB文件，而且只要15分钟备份一次就够了，也就是只保留save 900 1这条规则。

如果Enalbe AOF，好处是在最恶劣情况下也只会丢失不超过两秒数据，启动脚本较简单只load自己的AOF文件就可以了。代价一是带来了持续的IO，二是AOF rewrite的最后将rewrite过程中产生的新数据写到新文件造成的阻塞几乎是不可避免的。只要硬盘许可，应该尽量减少AOF rewrite的频率，AOF重写的基础应该修改到适当的数值。

如果不Enable AOF ，仅靠Master-Slave Replication 实现高可用性也可以。能省掉一大笔IO也减少了rewrite时带来的系统波动。代价是如果Master/Slave同时倒掉，会丢失十几分钟的数据，启动脚本也要比较两个Master/Slave中的RDB文件，载入较新的那个。新浪微博就选用了这种架构

## Redis的事务

### 1.是什么？

可以在一个队列中，一次性、顺序地、排他性地执行多个命令。一个事务中的所有命令都会序列化，按顺序地串行化执行而不会被其它命令插入，不许加塞

### 怎么玩儿？

常用命令：multi 开启事务、discard 取消事务、exec 执行事务、watck key1 key2… 监控keys，如果值更改，则事务会被打断、unwatch 取消所有key的监控

#### 1.全体连坐，一条命令输入出错，全部命令都不会执行

#### 2.冤头债主，一条命令执行出错，其他命令照样执行，所以redis不保证原子性

#### 3.watch监控

悲观锁和乐观锁

悲观锁：我很悲观，我在操作数据库时别人也肯定要操作，为了不让别人影响到我，所以我要上锁。

乐观锁：我很乐观，我认为别人不会影响到我需要操作的数据，我不上锁。但我在更新时会先判断是否有人修改我操作的数据，再决定是否提交我的操作。

传统的关系型数据库用到了很多这种锁机制，都是在做操作之前先上锁。乐观锁适用于多读的应用类型，这样可以提高吞吐量。

Watch指令类似乐观锁，在开启事务前调用，事务提交时，如果任何Key被别修改，整个事务会被放弃，同时返回Nullmulti-bulk以通知调用者事务执行失败

### 3.3个阶段

#### 1.开启：以multi开启事务

#### 2.入队：将多个命令放入队列，它们只等待被执行，根本不会去尝试去执行，因此redis不存在事务隔离界别一说

#### 3.执行：以exec执行事务

## 发布和订阅

发布和订阅是进程间的一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

psubscribe prattern1 prattern2 订阅指定频道，支持通配符

pulish parttrn message 指定频道发布消息

···

## Redis的复制(Master/Slave)

### 1.是什么？

主从复制，主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制，Master以写为主，Slave只能读

### 2.能干嘛？ 1.读写分离 2.容灾恢复

### 3.怎么玩儿？

需要命令：info replicaiton：查看本机消息 2.slaveof ip port：认主机，甘愿为从机

修改配置文件细节操作

1. 拷贝多个redis.conf文件

2. 修改文件内容：1. daemonize 2.文件名称 3.port 4.log文件名称 5.RDB文件名称

#### 1.一主二仆

主机日志： 7559:M 29 Jul 19:30:21.696 \* Synchronization with slave 127.0.0.1:6381 succeeded

7559:M 29 Jul 19:30:21.696 \* Synchronization with slave 127.0.0.1:6380 succeeded

从机日志： 7547:S 29 Jul 19:30:21.592 \* Connecting to MASTER 127.0.0.1:6379

7547:S 29 Jul 19:30:21.593 \* MASTER <-> SLAVE sync started

7547:S 29 Jul 19:30:21.600 \* Non blocking connect for SYNC fired the event.

7547:S 29 Jul 19:30:21.602 \* Master replied to PING, replication can continue...

7547:S 29 Jul 19:30:21.605 \* Trying a partial resynchronization (request cf51cf2002dbffe9712bf93a62350a28c4f9dbde:541).

7547:S 29 Jul 19:30:21.609 \* Full resync from master: c96373d257462f4c1cfac4d95002e1c91ec0449e:1

7547:S 29 Jul 19:30:21.609 \* Discarding previously cached master state.

7547:S 29 Jul 19:30:21.697 \* MASTER <-> SLAVE sync: receiving 41 bytes from master

7547:S 29 Jul 19:30:21.697 \* MASTER <-> SLAVE sync: Flushing old data

7547:S 29 Jul 19:30:21.697 \* MASTER <-> SLAVE sync: Loading DB in memory

7547:S 29 Jul 19:30:21.697 \* MASTER <-> SLAVE sync: Finished with success

实际总结：1.当从机执行slaveof ip port命令后，会得到主机内存中的所有数据

2.主机可读可写，从机只能读（也可以进行配置，使其可写）

3.主机shutdown后，从机原地待命，主机回来，一切照旧

4.从机shutdown后，再次连接升级为master，与主机切断关系，但这个问题可以通过配置解决

#### 2.薪火相传

Slave可以是做另一个Slave的从机，同样可以接收其他Slave的内存数据，但Master只有一个，所以第二个主机任然不能写数据

#### 3.反客为主

从机执行slaveof no one之后，就会变成主机，如果同时又另一个Savle认其做主机的话，其就是反客为主。

4.总结复制原理

1. Slave启动成功连接到Master后会发送一个sync命令，执行成功后立即同步主机数据，并且配从不配主
2. Master接到命令后启动后台的存盘进程，将收集所有接收到的用于修改数据集命令，在后台进程执行完毕后，Master将整个数据文件传送给Slave，以完成一次完全同步
3. 全量复制：Save服务在接收到数据库文件数据后，将其存盘并加载到内存中。
4. 增量复制：Master会继续将新收集到的修改命令依次传给Save，完成同步
5. 缺点：由于所有的写操作都是先在Master上操作，然后同步更新到Slave上，所以从Master同步到Slave机器有一定的延迟，当系统 很繁忙的时候，延迟问题会更加严重，Slave机器数量的增加也会使这个问题更加严重。

### 4. 哨兵模式(sentinel)

#### 1.是什么？

反客为主的自动版，通过后台监控主机，出现故障后根据投票从从机中选择新主机，并且原主机回来自动变成新主机的从机

#### 2.怎么玩儿？

1. 在redis.conf文件目录下新建sentinel.conf文件，写入内容sentinel monitor 主机名称(任意) 主机ip 主机port 1，1表示投票

2. 启动redis-sentinel /myredis/sentinel.conf

3. 一组sentinel能同时监控多个Master

## Java客户端Jedis

### 1.测试连通性

#### 1.放行ip端口

-A INPUT -m state --state NEW -m tcp -p tcp --dport 8080 -j ACCEPT

-A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT

-A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT

#### 2.导入jar包

需要Jedis和Commons-pool

#### 3.代码

System.out.println(new Jedis("192.168.188.130",6379).ping());//PONG

### 2.5+1

public class Demo {

public static void main(String[] args) {

Jedis jedis = new Jedis("192.168.188.130", 6379);

// key

Set<String> keys = jedis.keys("\*");

for (Iterator iterator = keys.iterator(); iterator.hasNext();) {

System.out.println((String) iterator.next());

}

System.out.println(jedis.exists("k2"));

System.out.println(jedis.ttl("k1"));

// String

jedis.append("k1","myreids");

System.out.println(jedis.get("k1"));

jedis.set("k4", "k4\_redis");

jedis.mset("str1", "v1", "str2", "v2", "str3", "v3");

System.out.println(jedis.mget("str1", "str2", "str3"));

// list

jedis.lpush("mylist","v1","v2","v3","v4","v5");

List<String> list = jedis.lrange("mylist", 0, -1);

for (String element : list) {

System.out.println(element);

}

// set

jedis.sadd("orders", "jd001");

jedis.sadd("orders", "jd002", "jd003");

Set<String> set1 = jedis.smembers("orders");

for (Iterator iterator = set1.iterator(); iterator.hasNext();) {

System.out.println((String) iterator.next());

}

jedis.srem("orders", "jd002");

System.out.println(jedis.smembers("orders").size());

// hash

jedis.hset("hash1", "userName", "lisi");

System.out.println(jedis.hget("hash1", "userName"));

Map<String, String> map = new HashMap<String, String>();

map.put("telphone", "13811814763");

map.put("address", "atguigu");

map.put("email", "abc@163.com");

jedis.hmset("hash2", map);

List<String> result = jedis.hmget("hash2", "telphone", "email");

for (String element : result) {

System.out.println(element);

}

// zset

jedis.zadd("zset01", 60d, "v1");

jedis.zadd("zset01", 70d, "v2");

jedis.zadd("zset01", 80d, "v3");

jedis.zadd("zset01", 90d, "v4");

Set<String> s1 = jedis.zrangeByScore("zset01", "(60", "(90");

for (Iterator iterator = s1.iterator(); iterator.hasNext();) {

System.out.println((String) iterator.next());

}

}

}

### 3.事务

public boolean transMethod() {

Jedis jedis = new Jedis("192.168.188.130", 6379);

int balance;

int debt;

int amtToSubtract = 10;

jedis.watch("balance");

balance = Integer.parseInt(jedis.get("balance"));

if (balance < amtToSubtract) {

jedis.unwatch();

System.out.println("modify");

return false;

} else {

Transaction transaction = jedis.multi();

transaction.decrBy("balance", amtToSubtract);

transaction.incrBy("debt", amtToSubtract);

transaction.exec();

System.out.println(Integer.parseInt(jedis.get("balance")));

System.out.println(Integer.parseInt(jedis.get("debt")));

return true;

}

}

### 4.主从复制

public static void main(String[] args) throws InterruptedException {

Jedis jedis\_M = new Jedis("127.0.0.1",6379);

Jedis jedis\_S = new Jedis("127.0.0.1",6380);

jedis\_S.slaveof("127.0.0.1",6379);

jedis\_M.set("k6","v6");

Thread.sleep(500); //内存中的数据读取极快，需要使线程休眠

System.out.println(jedis\_S.get("k6"));

}

### 5. JedisPool

public class JedisPoolUtile {

private static volatile JedisPool jedisPool = null;// 被volatile修饰的变量不会被本地线程缓存，对该变量的读写都是直接操作共享内存。

private JedisPoolUtile() {}

public static JedisPool getJedisPoolInstance() {

if (null == jedisPool) {

synchronized (JedisPoolUtile.class) {

if (null == jedisPool) {

JedisPoolConfig poolConfig = new JedisPoolConfig();

poolConfig.setMaxTotal(1000);

poolConfig.setMaxIdle(32);

poolConfig.setMaxWaitMillis(100 \* 1000);

poolConfig.setTestOnBorrow(true);

jedisPool = new JedisPool(poolConfig, "192.168.188.130");

}

}

}

return jedisPool;

}

public static void release(JedisPool jedisPool, Jedis jedis) {

if (null != jedis) {

jedisPool.returnResourceObject(jedis);

}

}

}

public class Test01 {

@Test

public void fun() {

JedisPool jedisPool = JedisPoolUtil.getJedisPoolInstance();

Jedis jedis = null;

try {

jedis = jedisPool.getResource();

jedis.set("k18", "v183");

} catch (Exception e) {

e.printStackTrace();

} finally {

JedisPoolUtil.release(jedisPool, jedis);

}

}

}