[**https://blog.csdn.net/u011863024/article/details/107476187**](https://blog.csdn.net/u011863024/article/details/107476187)

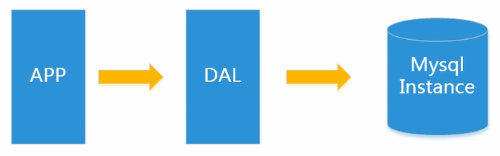
**redis 过期策略** <https://blog.csdn.net/a_bang/article/details/52986935>

# 一、NoSQL

## 1.演变过程

### 1.1单机SQL的美好时代

在90年代，一个网站的访问量一般都不大，用单个数据库完全可以轻松应付。  
在那个时候，更多的都是静态网页，动态交互类型的网站不多。



DAL dal是数据访问层的英文缩写，即为数据访问层（Data Access Layer）

上述架构下，我们来看看数据存储的瓶颈是什么?

数据量的总大小一个机器放不下时

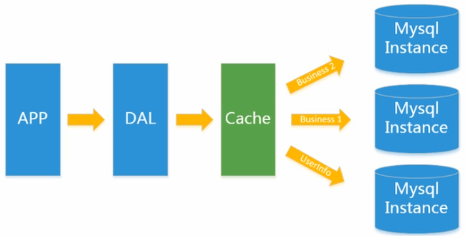
数据的索引(B+ Tree)一个机器的内存放不下时

访问量(读写混合)一个实例不能承受

如果满足了上述1or3个，进化…

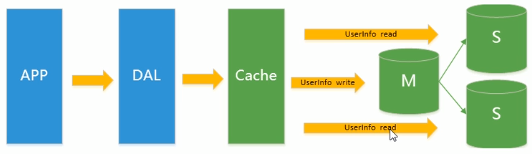
### 1.2MemCached+MySQL+垂直拆分

后来，随着访问量的上升，几乎大部分使用MySQL架构的网站在数据库上都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，但是当访问量继续增大的时候，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了了比较高的IO压力。在这个时候，Memcached就自然的成为一个非常时尚的技术产品。



### 1.3 Mysql主从读写分离

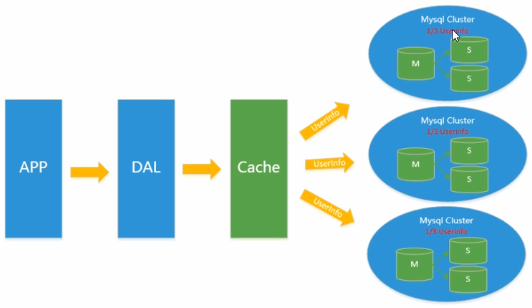
由于数据库的写入压力增加，Memcached 只能缓解数据库的读取压力。读写集中在一个数据库上让数据库不堪重负，大部分网站开始使用主从复制技术来达到读写分离，以提高读写性能和读库的可扩展性。Mysql的master-slave模 式成为这个时候的网站标配了。



### 1.4分表分库+水平拆分+mysql集群

在Memcached的高速缓存，MySQL的主从复制， 读写分离的基础之上，这时MySQL主库的写压力开始出现瓶颈，而数据量的持续猛增，由于MyISAM使用表锁，在高并发下会出现严重的锁问题，大量的高并发MySQL应用开始使用InnoDB引擎代替MyISAM。

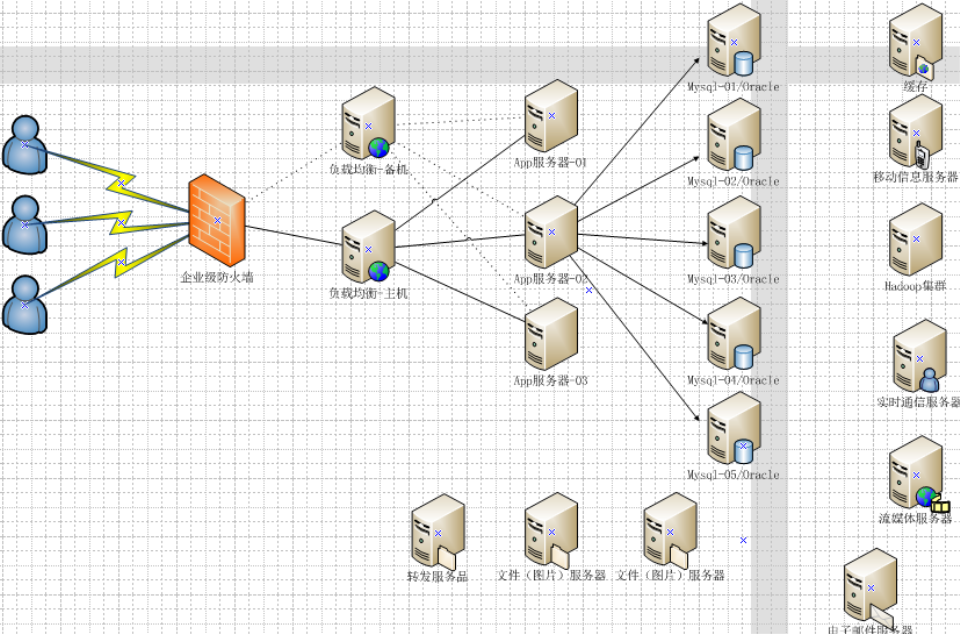
同时，开始流行使用分表分库来缓解写压力和数据增长的扩展问题。这个时候，分表分库成了一个热门技术，是面试的热门问题也是业界讨论的热门技术问题。也就在这个时候，MySQL推出了还不太稳定的表分区，这也给技术实力一般的公司带来了希望。虽然MySQL推出了MySQL Cluster集群，但性能也不能很好满足互联网的要求，只是在高可靠性上提供了非常大的保证。



### 1.5MySQL的扩展性瓶颈

MySQL数据库也经常存储一些大文本字段，导致数据库表非常的大，在做数据库恢复的时候就导致非常的慢，不容易快速恢复数据库。比如1000万4KB大小的文本就接近40GB的大小， 如果能把这些数据从MySQL省去，MySQL将变得非常的小。关系数据库很强大，但是它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，正是当前使用MySOL的开发人员面临的问题。

### 1.6今天是什么样子? ?



## 2.是什么

NoSQL(NoSQL = Not Only SQL)，意即“不仅仅是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.优点

### 易扩展

NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。数据之间无关系，这样就非常容易扩展。也无形之间，在架构的层面上带来了可扩展的能力。

### 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，每次表的更新Cache就失效，是一种大粒度的Cache，在针对web2.0的交互频繁的应用，Cache性能不高。

而NoSQL的Cache是记录级的，是一种细粒度的Cache，所以NoSQL在这个层面上来说就要性能高很多了。

### 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，增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增加字段简直就是一个噩梦。

## 4. 传统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

## 5.NoSQL数据库的四大分类



## 6.CAP

Redis: CP

**对复杂的SQL查询，特别是多表关联查询的需求**

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。

## 7.BASE

**BASE就是为了解决关系数据库强一致性引起的问题而引起的可用性降低而提出的解决方案**。

BASE其实是下面三个术语的缩写：

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**。为什么这么说呢，缘由就在于大型系统往往由于地域分布和极高性能的要求，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

# 二、Redis

## 1.概述

Redis: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。

## 2.特点

内存存储和异步持久化

支持多种数据结构的存储，包括五大基本数据结构：string、list、set、zset、hash

支持数据备份（master-slave）

## 3.五大基本类型

**String**（字符串）

* + string是redis最基本的类型，你可以理解成与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。意思是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**

**Hash**（哈希，类似java里的Map）

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

**List**（列表）

* + Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素导列表的头部（左边）或者尾部（右边）。
  + 它的底层实际是个链表

**Set**（集合）

* + Redis的Set是string类型的无序集合。它是通过HashTable实现实现的

**Zset**(sorted set：有序集合)

* + 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。

## 4.命令

### 4.1 key

|  |  |
| --- | --- |
| DEL key | 该命令用于在 key 存在时删除 key。 |
| DUMP key | 序列化给定 key ，并返回被序列化的值。 |
| EXISTS key | 检查给定 key 是否存在。 |
| EXPIRE key seconds | 为给定 key 设置过期时间，以秒计。 |
| EXPIREAT key timestamp | EXPIREAT 的作用和 EXPIRE 类似，都用于为 key 设置过期时间。  不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳(unix timestamp)。 |
| PEXPIRE key milliseconds | 设置 key 的过期时间以毫秒计。 |
| PEXPIREAT key milliseconds-timestamp | 设置 key 过期时间的时间戳(unix timestamp) 以毫秒计 |
| KEYS pattern | 查找所有符合给定模式( pattern)的 key 。 |
| MOVE key db | 将当前数据库的 key 移动到给定的数据库 db 当中。 |
| PERSIST key | 移除 key 的过期时间，key 将持久保持。 |
| PTTL key | 以毫秒为单位返回 key 的剩余的过期时间。 |
| TTL key | 以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)。 |
| RANDOMKEY | 从当前数据库中随机返回一个 key 。 |
| RENAME key newkey | 修改 key 的名称 |
| RENAMENX key newkey | 仅当 newkey 不存在时，将 key 改名为 newkey 。 |
| SCAN cursor [MATCH pattern] [COUNT count] | 迭代数据库中的数据库键。 |
| TYPE key | 返回 key 所储存的值的类型。 |

### 4.2 string

getrange k1 0 2 [0-2]

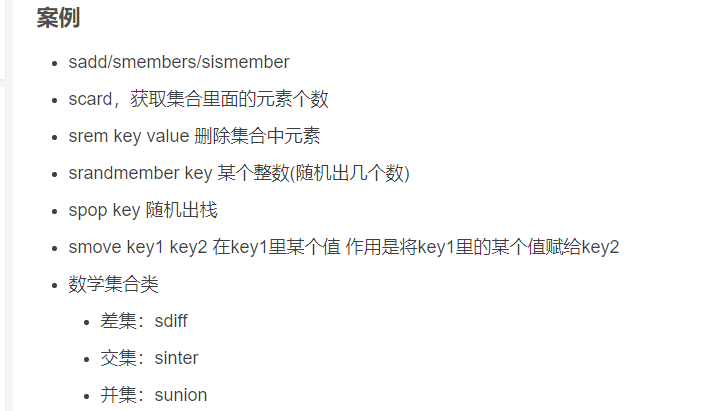
| **命令** | **描述** |
| --- | --- |
| SET key value | 设置指定 key 的值 |
| GET key | 获取指定 key 的值。 |
| GETRANGE key start end | 返回 key 中字符串值的子字符 |
| GETSET key value | 将给定 key 的值设为 value ，并返回 key 的旧值(old value)。 |
| GETBIT key offset | 对 key 所储存的字符串值，获取指定偏移量上的位(bit)。 |
| MGET key1 [key2…] | 获取所有(一个或多个)给定 key 的值。 |
| SETBIT key offset value | 对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)。 |
| SETEX key seconds value | 将值 value 关联到 key ，并将 key 的过期时间设为 seconds |
| SETNX key value | 只有在 key 不存在时设置 key 的值。 |
| SETRANGE key offset value | 用 value 参数覆写给定 key 所储存的字符串值，从偏移量 offset 开始。 |
| STRLEN key | 返回 key 所储存的字符串值的长度。 |
| MSET key value [key value …] | 同时设置一个或多个 key-value 对。 |
| MSETNX key value [key value …] | 同时设置一个或多个 key-value 对，  当且仅当所有给定 key 都不存在。 |
| PSETEX key milliseconds value | 这个命令和 SETEX 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 SETEX 命令那样，以秒为单位。 |
| INCR key | 将 key 中储存的数字值增一。 |
| INCRBY key increment | 将 key 所储存的值加上给定的增量值（increment） 。 |
| INCRBYFLOAT key increment | 将 key 所储存的值加上给定的浮点增量值（increment） 。 |
| DECR key | 将 key 中储存的数字值减一。 |
| DECRBY key decrement | key 所储存的值减去给定的减量值（decrement） 。 |
| APPEND key value | 如果 key 已经存在并且是一个字符串，  APPEND 命令将指定的 value 追加到该 key 原来值（value）的末尾。 |

### 4.3 list



### 4.4 set



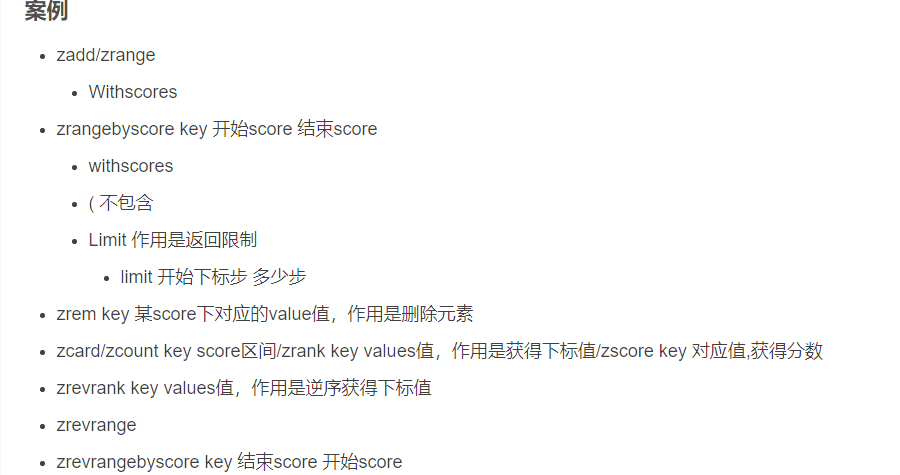


### 4.5 hash



### 4.6 zset





## 5.配置

**1. daemonize no**

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（Windows 不支持守护线程的配置为 no ）

**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**

指定 Redis 监听端口，默认端口为 6379，作者在自己的一篇博文中解释了为什么选用 6379 作为默认端口，因为 6379 在手机按键上 MERZ 对应的号码，而 MERZ 取自意大利歌女 Alessia Merz 的名字

**4. bind 127.0.0.1**

绑定的主机地址

**5. timeout 300**

当客户端闲置多长秒后关闭连接，如果指定为 0 ，表示关闭该功能

**6. loglevel notice**

指定日志记录级别，Redis 总共支持四个级别：debug、verbose、notice、warning，默认为 notice

**7. 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

**8. databases 16**

设置数据库的数量，默认数据库为0，可以使用SELECT 命令在连接上指定数据库id

**9. save <seconds> <changes>**Redis 默认配置文件中提供了三个条件：  
save 900 1  
save 300 10  
save 60 10000

分别表示 900 秒（15 分钟）内有 1 个更改，300 秒（5 分钟）内有 10 个更改以及 60 秒内有 10000 个更改。  
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

**10. rdbcompression yes**

指定存储至本地数据库时是否压缩数据，默认为 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

**11. dbfilename dump.rdb**

指定本地数据库文件名，默认值为 dump.rdb

**12. dir ./**

指定本地数据库存放目录

**13. slaveof <masterip> <masterport>**

设置当本机为 slave 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

**14. masterauth <master-password>**

当 master 服务设置了密码保护时，slav 服务连接 master 的密码

**15. requirepass foobared**

设置 Redis 连接密码，如果配置了连接密码，客户端在连接 Redis 时需要通过 AUTH 命令提供密码，默认关闭

**16. maxclients 128**

设置同一时间最大客户端连接数，默认无限制，Redis 可以同时打开的客户端连接数为 Redis 进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。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

**17. 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，达到最大内存后，Redis 会先尝试清除已到期或即将到期的 Key，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

**18. appendonly no**

指定是否在每次更新操作后进行日志记录，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

**19. appendfilename appendonly.aof**

指定更新日志文件名，默认为 appendonly.aof

**20. 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）

always：表示每次更新操作后手动调用 fsync() 将数据写到磁盘（慢，安全）

everysec：表示每秒同步一次（折中，默认值）

**21. vm-enabled no**

指定是否启用虚拟内存机制，默认值为 no，简单的介绍一下，VM 机制将数据分页存放，由 Redis 将访问量较少的页即冷数据 swap 到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析 Redis 的 VM 机制）

**22. vm-swap-file /tmp/redis.swap**

虚拟内存文件路径，默认值为 /tmp/redis.swap，不可多个 Redis 实例共享

**23. vm-max-memory 0**

将所有大于 vm-max-memory 的数据存入虚拟内存，无论 vm-max-memory 设置多小，所有索引数据都是内存存储的(Redis 的索引数据 就是 keys)，也就是说，当 vm-max-memory 设置为 0 的时候，其实是所有 value 都存在于磁盘。默认值为 0

**24. vm-page-size 32**

Redis swap文件分成了很多的 page，一个对象可以保存在多个 page 上面，但一个 page 上不能被多个对象共享，vm-page-size 是要根据存储的 数据大小来设定的，作者建议如果存储很多小对象，page 大小最好设置为 32 或者 64bytes；如果存储很大大对象，则可以使用更大的 page，如果不确定，就使用默认值

**25. vm-pages 134217728**

设置 swap 文件中的 page 数量，由于页表（一种表示页面空闲或使用的 bitmap）是在放在内存中的，，在磁盘上每 8 个 pages 将消耗 1byte 的内存。

**26. vm-max-threads 4**

设置访问swap文件的线程数,最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4

**27. glueoutputbuf yes**

设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

**28. hash-max-zipmap-entries 64**

hash-max-zipmap-value 512 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法

**29. activerehashing yes**

指定是否激活重置哈希，默认为开启（后面在介绍 Redis 的哈希算法时具体介绍）

**30. 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，而同时各个实例又拥有自己的特定配置文件

## 6.持久化

### 6.1 RDB（Redis DataBase）

#### 是什么

在指定的时间间隔将内存中的数据集快照写入磁盘，他恢复时是直接将快照文件读到内存中。

Redis会单独fork一个子进程来进行持久化。先将数据写到临时文件中，待持久化操作完成，在将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IO操作的，这确保了极高的性能。对于大规模数据的恢复，对数据的完整性不敏感，RDB的性能是优于AOF的。RDB的缺点是可能会丢失最后一次持久化的数据

Fork

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（变量、环境变量、程序计数器等） 数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程

保存的文件是dump.rdb

相关配置在配置文件的位置 - 在redis.conf搜寻### SNAPSHOTTING ###

#### 如何触发RDB快照

1.配置文件触发save 900 1 代表900s内有1个修改就会在900s后写入rdb文件中

2.save 无需等待触发时间，直接保存，但主进程阻塞，无法进行响应客户端请求

3.bgsava redis在后台异步保存，快照还可以响应客户端请求

#### 如何恢复

将备份文件移动到redis安装目录并启动服务即可

#### 优势和劣势

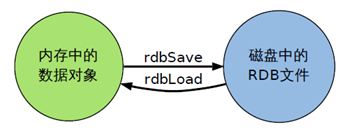
优势：

* 适合大规模的数据恢复
* 对数据完整性和一致性要求不高

劣势：

* redis在最后一个快照保存时意外崩掉，会丢失最后一次快照的所有修改
* fork时，内存的数据被克隆了一份，大致2倍的膨胀性需要考虑

#### 小结



* RDB是一个非常紧凑的文件。
* RDB在保存RDB文件时父进程唯一需要做的就是fork出一个子进程，接下来的工作全部由子进程来做，父进程不需要再做其他I0操作，所以RDB持久化方式可以最大化redis的性能。
* 与AOF相比，在恢复大的数据集的时候，RDB方式会更快一一些。
* 数据丢失风险大。
* RDB需要经常fork子进程来保存数据集到硬盘上，当数据集比较大的时候fork的过程是非常耗时的吗，可能会导致Redis在一些毫秒级不能回应客户端请求。

### 6.2 AOF（Append Olny File）

#### 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启动之初会读取该文件重新构建数据，换言之，redis 重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作

#### AOF配置

相关配置在配置文件的位置 - 在redis.conf搜寻### APPEND ONLY MODE ###

aof保存的是appendonly.aof文件（在配置文件可修改文件名）

#### AOF启动/修复/恢复

正常恢复

* + 启动：设置Yes
    - 修改默认的appendonly no，改为yes
  + 将有数据的aof文件复制一份保存到对应目录(config get dir)
  + 恢复：重启redis然后重新加载

异常恢复

* + 启动：设置Yes
    - 修改默认的appendonly no，改为yes
  + 备份被写坏的AOF文件
  + 修复：
    - **Redis-check-aof --fix进行修复**
  + 恢复：重启redis然后重新加载

#### rewrite

AOF采用文件追加方式，文件会越来越大。为避免出现此种情况，新增了重写机制， 当AOF文件的大小超过所设定的阈值时，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

**重写原理**

AOF文件持续增长而过大时，会fork出一条新进程来将文件重写(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。重写aof文件的操作，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了一个新的aof文件，这点和快照有点类似

**触发机制**

Redis会记录上次重写时的AOF大小，默认配置是当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发

#### 优势与劣势

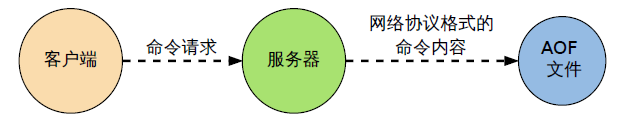
优势

* + 每修改同步：appendfsync 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
  + 每秒同步：appendfsync everysec 异步操作，每秒记录 如果一秒内宕机，有数据丢失
  + 不同步：appendfsync no 从不同步

劣势

* + 相同数据集的数据而言aof文件要远大于rdb文件，恢复速度慢于rdb
  + Aof运行效率要慢于rdb,每秒同步策略效率较好，不同步效率和rdb相同

#### 小结



* AOF文件时一个只进行追加的日志文件
* Redis可以在AOF文件体积变得过大时，自动地在后台对AOF进行重写
* AOF文件有序地保存了对数据库执行的所有写入操作，这些写入操作以Redis协议的格式保存，因此AOF文件的内容非常容易被人读懂，对文件进行分析也很轻松
* 对于相同的数据集来说，AOF文件的体积通常要大于RDB文件的体积
* 根据所使用的fsync 策略，AOF的速度可能会慢于RDB

## 7.事务

可以一次执行多个命令，本质是一组命令的集合。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。

一个队列中，一次性、顺序性、排他性的执行一系列命令。



* Watch指令，类似乐观锁，事务提交时，如果Key的值已被别的客户端改变， 比如某个list已被别的客户端push/pop过了，整个事务队列都不会被执行
* 通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化， EXEC命令执行的事务都将被放弃，同时返回Nullmulti-bulk应答以通知调用者事务执行失败

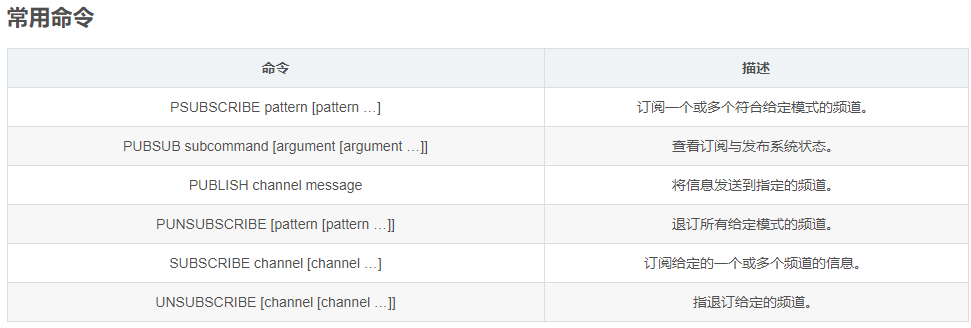
### 3特性

* **单独的隔离操作**：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
* **没有隔离级别的概念**：队列中的命令没有提交之前都不会实际的被执行，因为事务提交前任何指令都不会被实际执行， 也就不存在”事务内的查询要看到事务里的更新，在事务外查询不能看到”这个让人万分头痛的问题
* **不保证原子性**：redis同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

**不遵循传统的ACID中的AI**

## 8.发布订阅





## 9.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/slave机制。

master以写为主，slave以读为主

**作用：读写分离，容灾恢复**

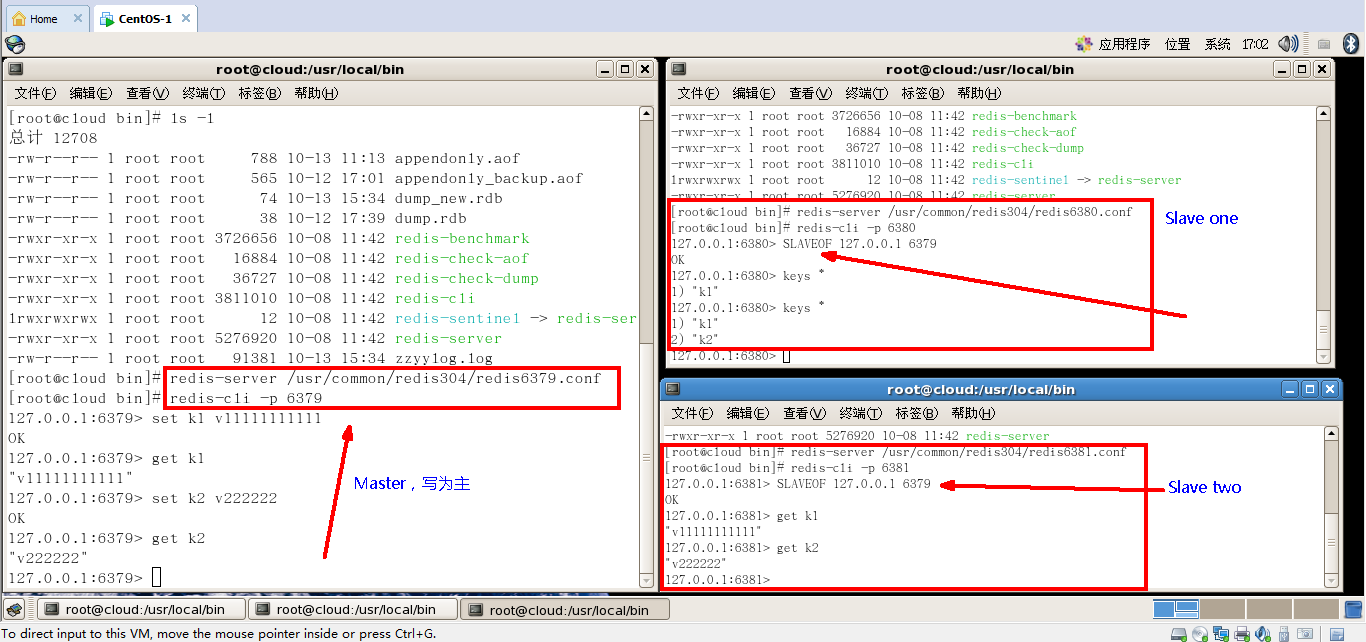
### 9.1 准备

配从（slave）不配主（master）

配置从机命令：slaveof 主库IP 主库端口

查看主从配置信息：info replication

### 9.2 一主二从



问题：

* Q1. 主机在从机还未连接时写入数据，从机能否得到数据

A：从机是从头开始复制的

* Q2：从机是否可以写

A： 不行，主写从读

* Q3：主机挂掉后，从机如何抉择，是重新选取master还是原地待命？

A：原地待命，并且主机回来后与原来一致

* Q4：从机挂掉后，能否复制主机新增的数据

A：不能，从机挂掉后，需要重新连接主机，除非将连接命令写入从机的配置文件（具体位置：redis.conf搜寻#### REPLICATION ####）

### 9.3 薪火相传

上一个Slave可以是下一个slave的Master，Slave同样可以接收其他 slaves的连接和同步请求，那么该slave作为了链条中下一个的master, 可以有效减轻master的写压力（奴隶的奴隶还是奴隶）

中途变更转向：会清除之前的数据，重新建立拷贝最新的

slaveof 新主库IP 新主库端口

### 9.4 反客为主

* SLAVEOF no one

使当前数据库停止与其他数据库的同步，转成主数据库

### 9.5 哨兵模式

一组sentinel能够监听多个master

反客为主的自动版，后台监控主机是否故障，如果故障，则从从机中通过投票选举出master

**配置sentinel**

**1.** 新建sentinel.conf

2. 配置哨兵 sentinel monitor 被监控的主机名称 主机ip 主机端口 num

num表示主机挂掉后slave需要成为master得到的投票数，

3. 启动哨兵 redis-sentinel /sentinel.conf (根据实际的目录)

Q：如果原来的master挂了，重启后回来，会不会双master冲突

A：不会，根据哨兵模式，原来的master回来后变成了slave

### 9.6 缺点

**复制延时**：由于所有的写操作都在master上进行，然后同步到slave上，同步过程存在一定延时，当系统繁忙时，延时问题会更加严重；slave机器数量的增加也会增加延时。

# 三、Redis扩展

## 1.Redis删除过期key的三种策略

### 1.1 被动删除

当一个过期key被访问时触发惰性删除策略，直接被删除

### 1.2 主动删除

redis会定期主动去删除一些已过期的key

### 1.3 主动清理

当前已用内存超过maxMemory限定时，触发主动清理策略

* volatile-lru：只对设置了过期时间的key进行LRU（默认值）
* allkeys-lru ： 删除lru算法的key
* volatile-random：随机删除即将过期key
* allkeys-random：随机删除
* volatile-ttl ： 删除即将过期的
* noeviction ： 永不过期，返回错误