Redis 笔记:

一. NoSql 入门和概述:

一.

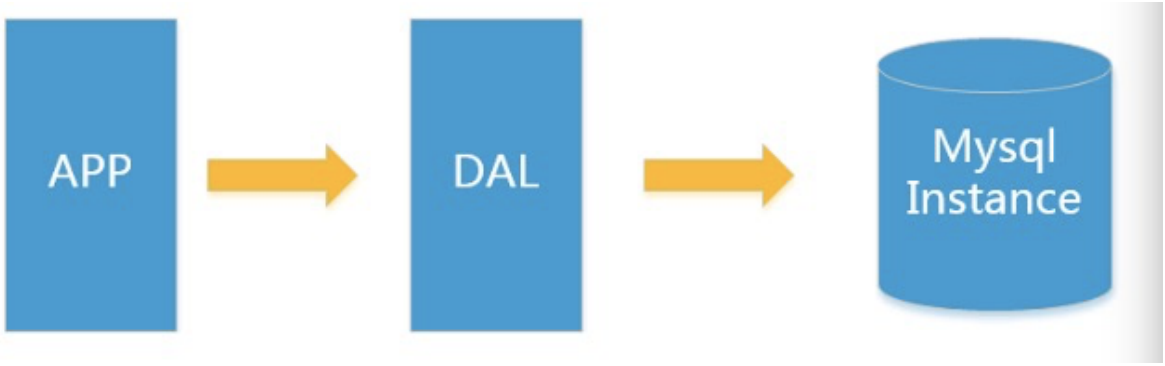
- 单机MySQL的瓶颈:

  -> 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） 一个实例不能承受时

  -> 架构图:



- Memcacher（缓存）+ MySQL + 垂直拆分

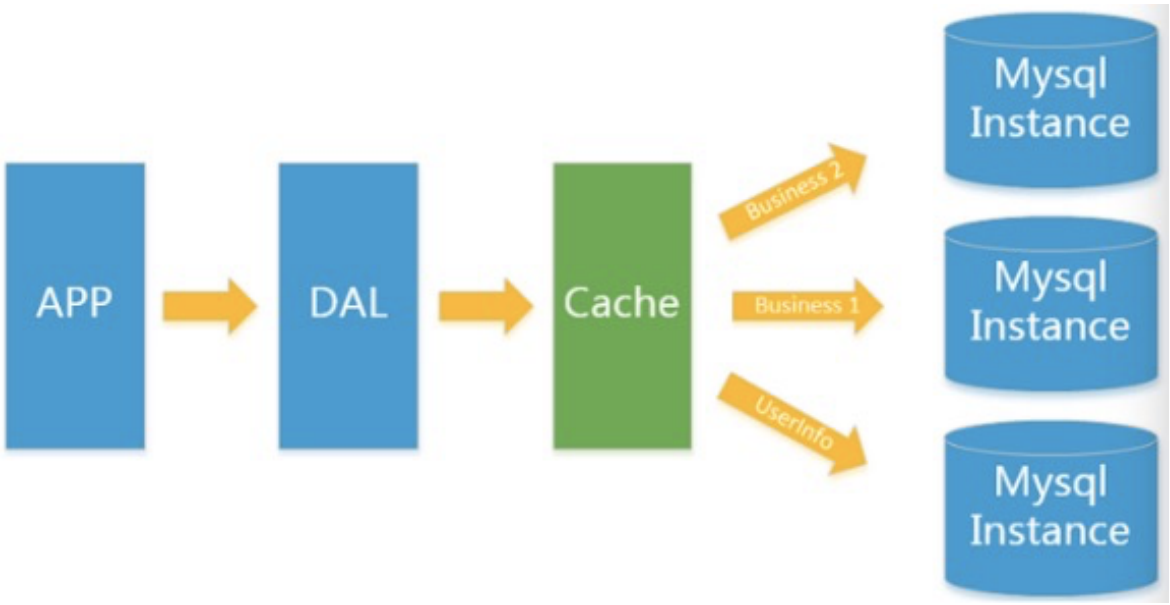
  -> 通过缓存来缓解数据库压力。

  -> 缺点:

    - 当访问量继续增大时，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比较高的 IO 压力。

  -> 架构图:



- MySQL 主从读写分离:

-> Memcached 只能缓解数据库的读取压力。

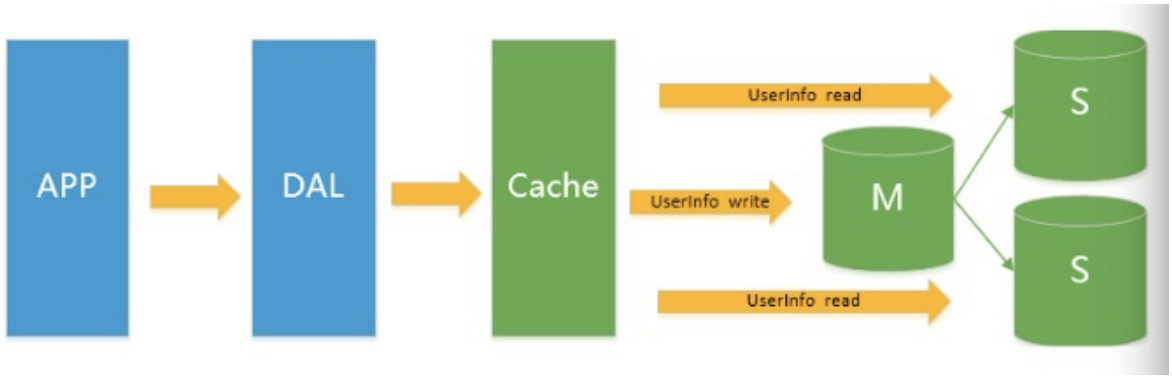
-> 读写集中导致数据库不堪重负。

-> 开始使用主从复制技术来达到读写分离。

-> 此时网站标配模式:

- MySQL 的 master-slave 模式

-> 架构图:



- 分表分库 + 水平拆分 + mysql 集群

-> 在 Memcached 的高速缓存，MySQL 主从复制，读写分离 的基础上

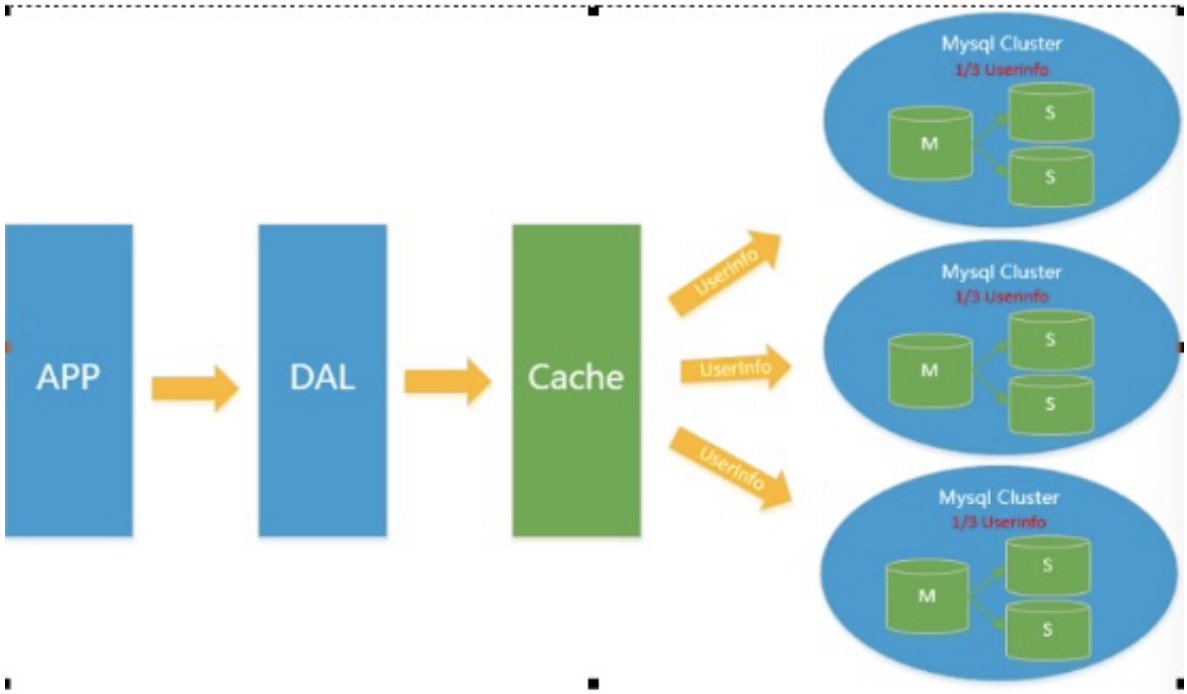
MySQL 主库的写压力出现瓶颈。

-> 数据量的持续猛增，由于 MyISAM（数据库引擎）使用表锁，在高并发下

出现严重的锁问题，开始使用 InnoDB 引擎代替 MyISAM

-> MySQL 推出了 MySQL Cluster 集群（性能不足，高可靠性）

-> 架构图:



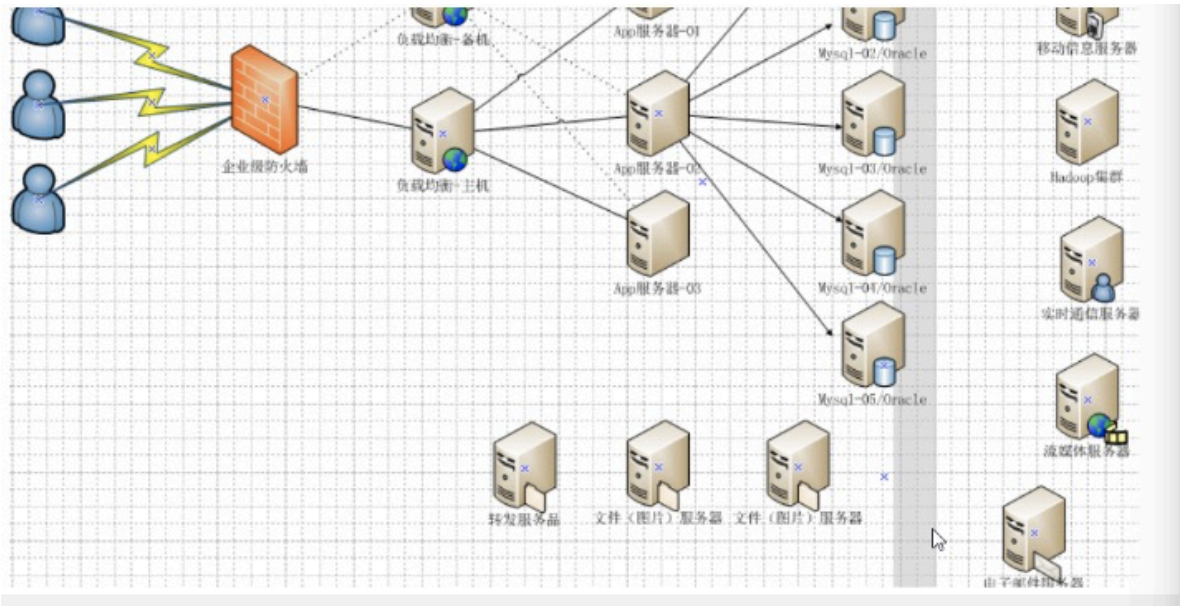
- MySQL 的扩展性瓶颈:

-> 存储大文本字段，做数据库恢复的时候非常慢。

-> 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。

- 今天是什么样子?



- 为什么要用 NoSQL?

-> 社交网络的兴起，导致数据的成倍增加。

-> 对这些用户数据进行挖掘，SQL 数据库已经不适用了。

-> NoSQL 数据库的发展能很好的处理这些大的数据。

二.

- NoSQL 是什么?

-> Not Only SQL （不仅仅是 SQL）

-> 泛指非关系型的数据库

-> 传统关系数据库对 超大规模和高并发的 SNS 类型的 web2.0 纯动态网站力不从心

-> NoSQL 就是为了解决大规模数据集合多重数据种类带来的挑战。

-> 这些类型的数据存储不需要固定模式，无需多余操作就可横向扩展。

三.

- NoSQL 能干嘛?

-> 易扩展

- 共同特点: 去掉关系数据库的关系型特性

-> 大数据量高性能

- NoSQL 读写性能非常高，得益于它的无关系性，数据库的结构简单

- MySQL 使用 Query Cache ，每次表的更新 Cache 就失效，大粒度级。

- NoSQL 的Cache 是记录级的，细粒度。

-> 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，随身可以存储自定义的数据格式。

- 关系数据库里，增删字段是非常麻烦的。

-> 传统 RDBMS（关系数据库） 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

-> CAP 定理。

-> 高性能，高可用性和可伸缩性。

四.

- 去哪下?

-> Redis

-> Memcache

-> Mongdb

五.

- 怎么玩?

-> KV

-> Cache

-> Persistence

-> ...

二. 3V + 3高

一.

- 大数据时代的 3V

-> 海量 Volume

-> 多样 Variety

-> 实时 Velocity

二.

- 互联网需求的 3高

-> 高并发

-> 高可扩

-> 高性能

三. 当下的 NoSQL 经典应用:

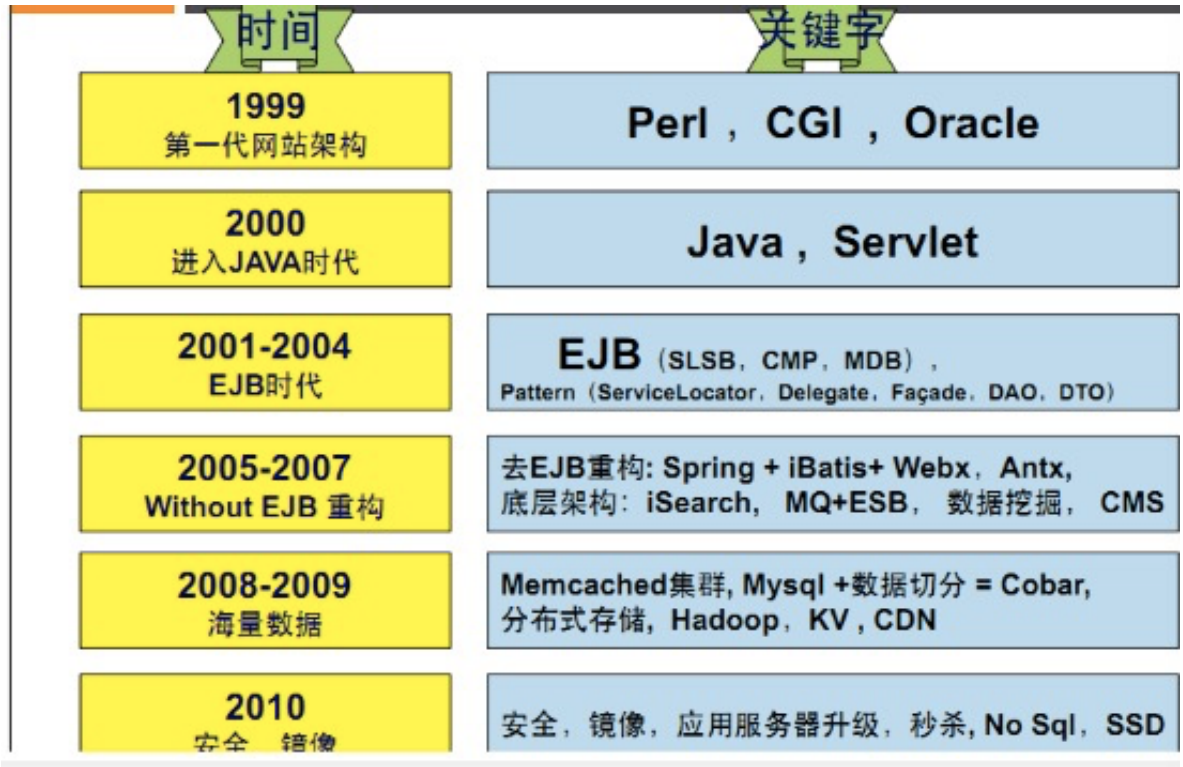
一.

- 当下的应用是 sql 和 nosql 一起使用

二.

- 阿里巴巴中文站商品信息如何存放:

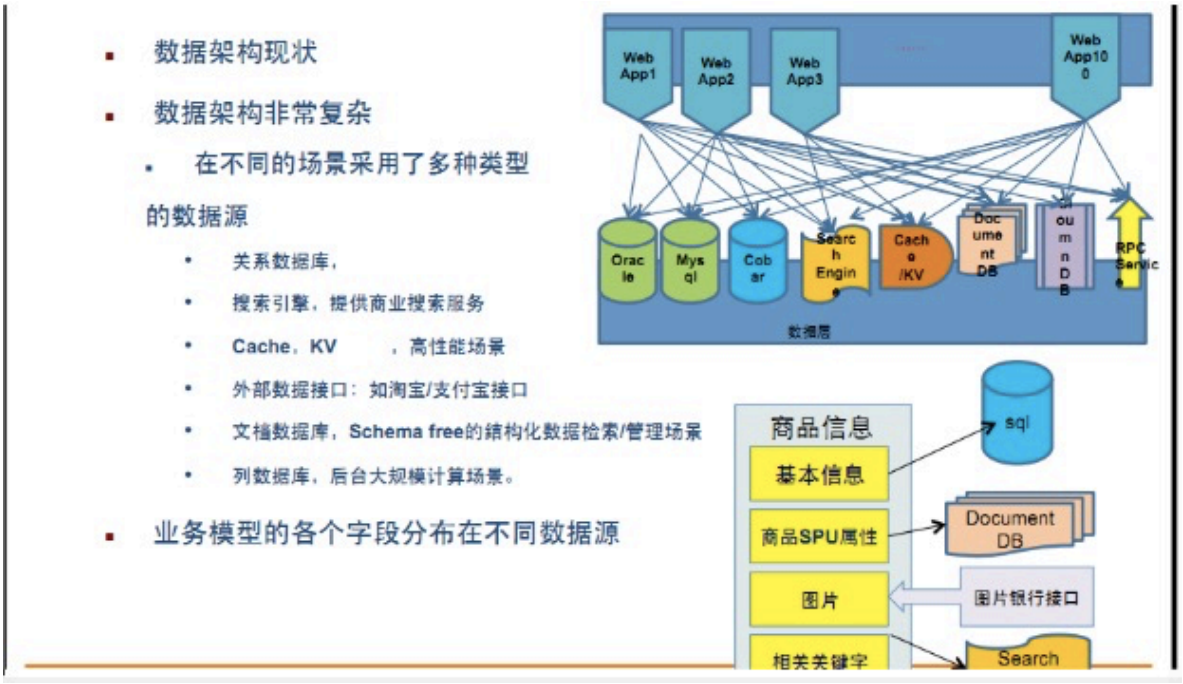
-> 架构发展过程:



- 2011年: 第五代网站架构

-> 敏捷，开放，体验

- 多数据源多数据类型的存储问题:



- 阿里的 MySQL 是经过内部改装的。

- 去IOE 化:

    -> 2008年，王坚（前微软亚洲研究院常务副院长）加盟阿里提出。

    -> 去除 IBM 小型机 、 Oracle 数据库 、EMC 存储设备

- 总结大型互联网应用（大数据 、高并发 、 多样数据类型）的难点和解决方案

    -> 难点:

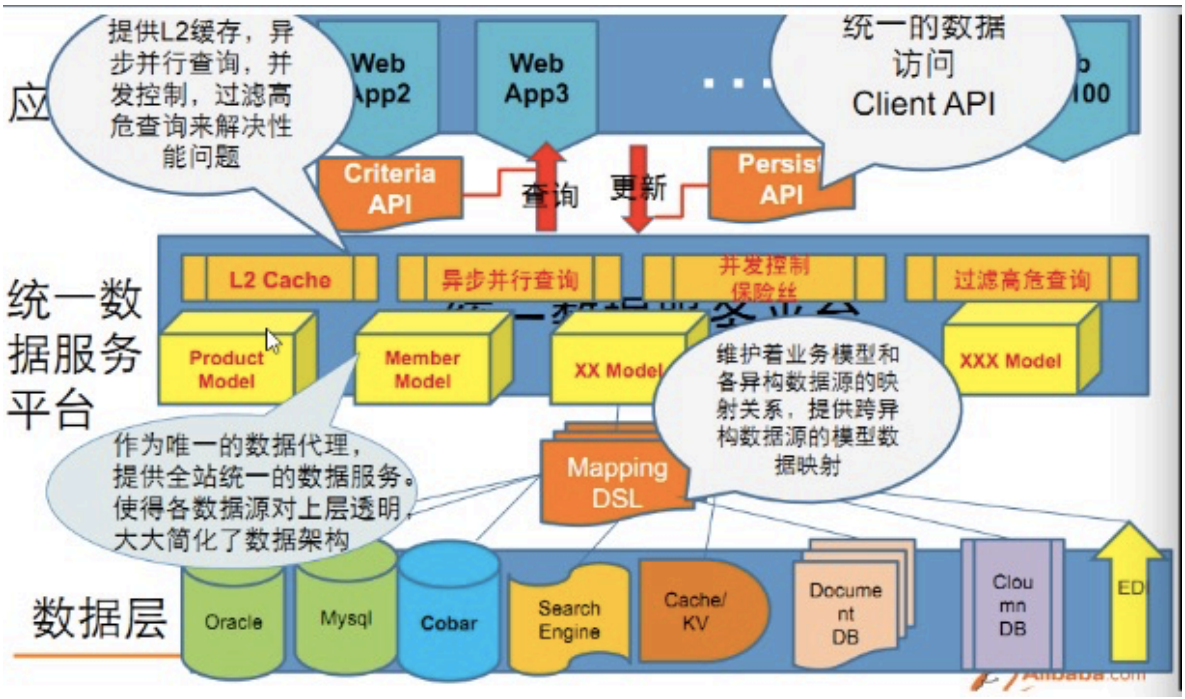
        -> 数据类型多样化

        -> 数据源多样性和变化重构

        -> 数据源改造而数据服务平台不需要大面积重构

    -> 解决方案:

        -> UDSL（统一数据服务平台）



四. NoSQL 数据模型简介:

一.

- 传统的关系型数据库:

-> ER 图（1:1 / 1:N / N:N，主外键等常见）

- NoSQL 的设计:

-> 什么是 BSON?

- BSON() 是一种类 JSON 的一种二进制形式的存储格式，简称 Binary JSON

- 和 JSON 一样，支持内嵌的文档对象和数组对象

-> 为什么上述情况可以用 聚合模型 来处理?

- 高并发的操作是不太建议有关联查询的，互联网公司用冗余数据来避免。

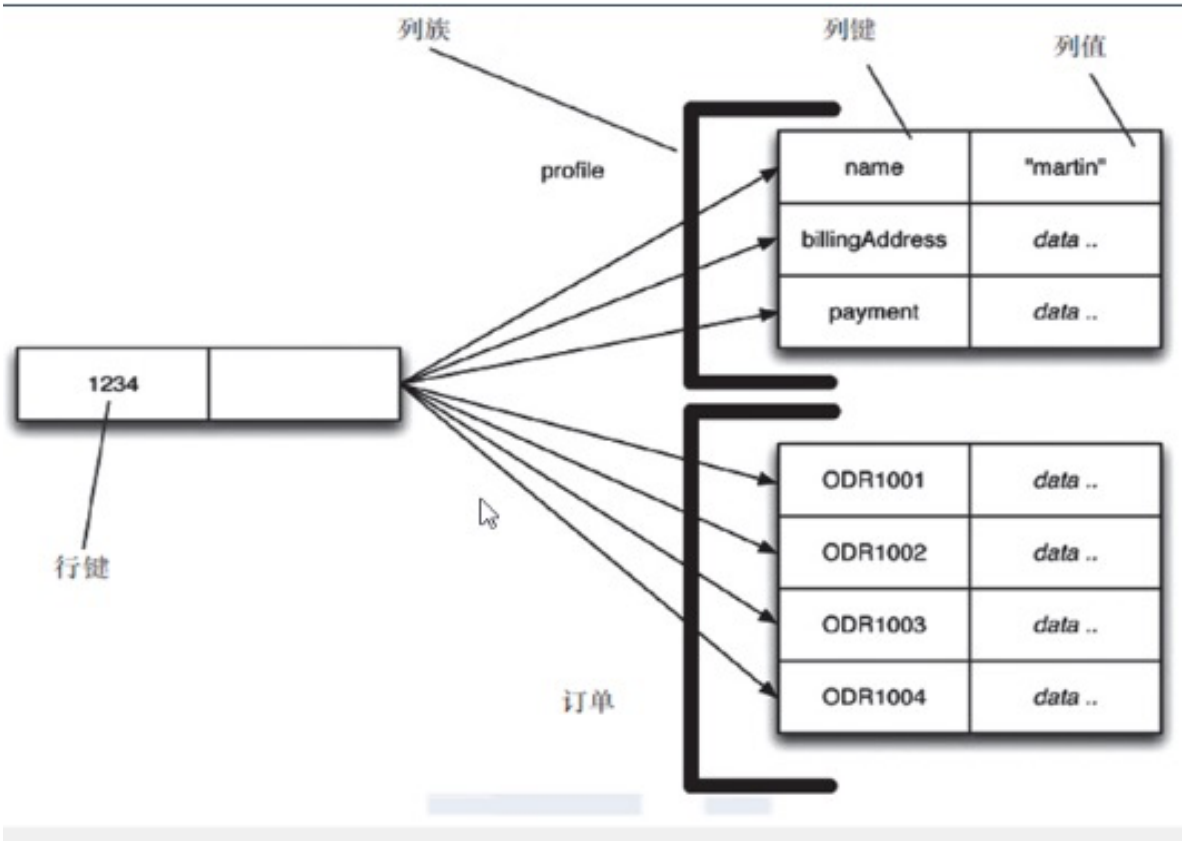
- 分布式事务是支持不了太多的并发的。

-> 聚合模型:

- KV 键值对

- BSON

- 列族



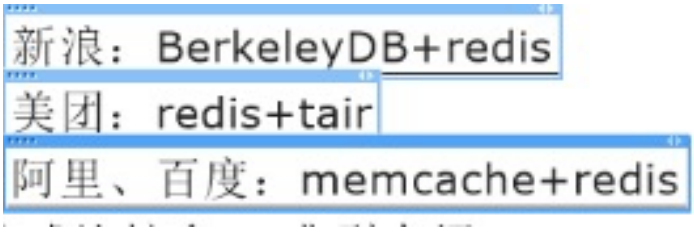
- 图形

五. NoSQL 数据库的四大分类:

一.

- KV 键值:

  ->



二.

- 文档型数据库（主BSON）：

  -> CouchDB

  -> MongoDB

- 基于 分布式文件存储的数据库，由 C++ 编写。

- 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。

三.

- 列存储数据库:

  -> Cassandra ， HBase

  -> 分布式文件系统

四.

- 图关系数据:

  -> 专注于构建关系图谱

  -> Neo4J ， InfoGrid

五.

- 四者对比:

  ->

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]

六: 在分布式数据库中 CAP 原理 CAP + BASE

一.

- 传统的 ACID 分别是什么?

  -> A（Atomicity）原子性

  -> C（Consistency）一致性

  -> I（Isolation）独立性

  -> D（Durability）持久性

二.

- CAP:

  -> C（Consistency) 强一致性

  -> A（Avaliability）可用性

  -> P（Partition tolerance）分区容错性

三.

- CAP 的 3 进 2

  -> 分区容忍性是我们必须要实现的。

  -> CA 传统关系型（Oracle）数据库

  -> AP 大多数网站架构的选择

‘

  -> CP: Redis 、Mongodb

  -> 分布式架构的时候必须做出取舍

四.

-经典 CAP 图

  -> CAP 理论核心:

    - 一个分布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

    - 最多满足两个

  -> CA 原则

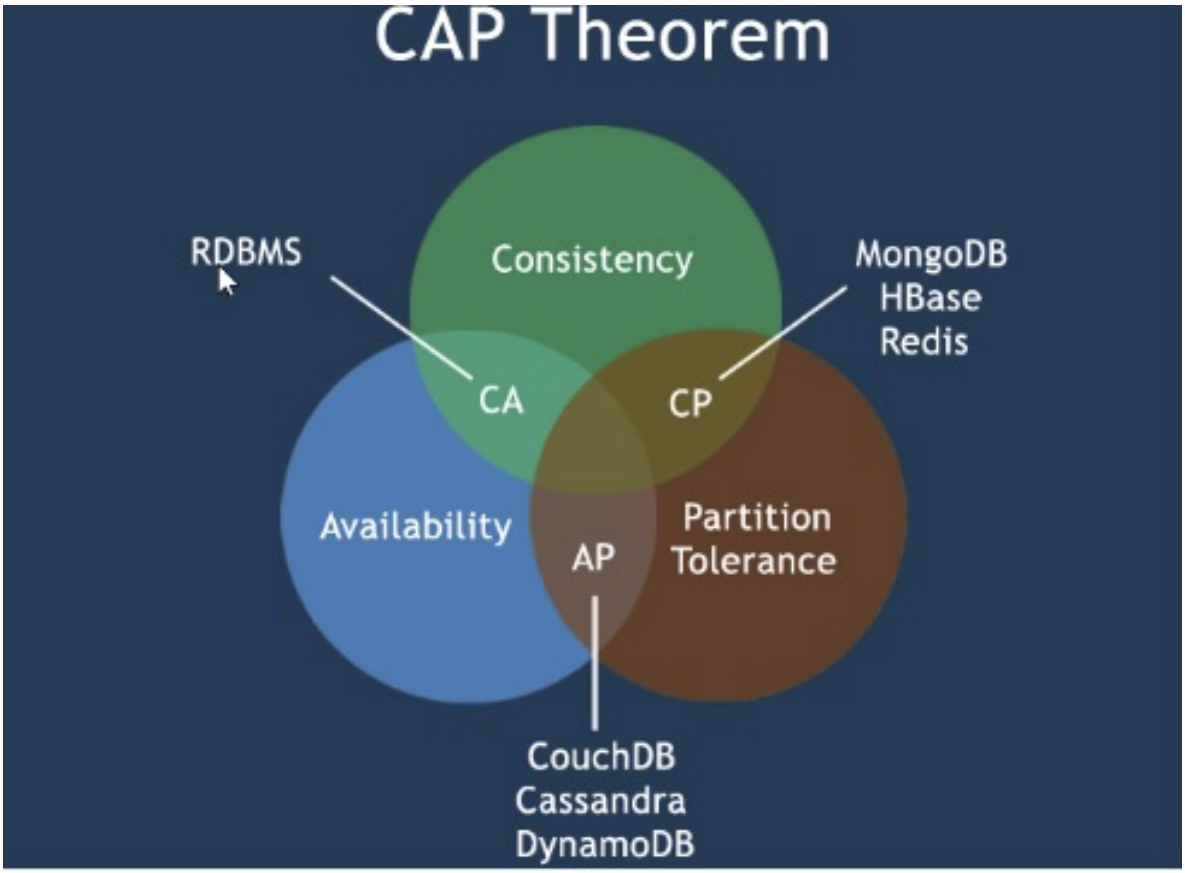
    - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。

  -> CP 原则

    - 满足一致性，分区容忍性的系统，通常性能不是特别高。

  -> AP 原则

- 满足可用性，分区容忍性的系统，通常对一致性要求较低



五.

- BASE

-> 为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。

-> 三个术语的缩写:

- 基本可用: Basically Available

- 软状态: Soft state

- 最终一致: Eventually consistent

六.

- 分布式 + 集群简介

-> 分布式系统（distributed system）

- 多台计算机和通信的软件组件 通过计算机网络连接组成。

- 是建立在网络之上的软件系统。

- 软件特性，分布式系统具有高度的内聚性和透明性。

- 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）

- 分布式系统可以运用在不同的平台（PC、工作站、局域网、广域网等）

-> 分布式:

- 不同多台服务器上，部署不同的服务模块（工程）。

- 通过 Rpc/Rmi 之间通信和调用，对外提供服务和组内协作。

-> 集群:

- 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。

七. Redis 入门概述:

一.

- 是什么?

-> REmote DIctionary Server（远程字典服务器）

-> 开源免费，用 C 语言编写，遵守 BSD 协议。

-> 高性能的 键值 分布式内存数据库，基于内存运行

-> 是当前最热门的NoSQL 数据库之一，被称为数据结构服务器。

-> 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 支持 Key-Value 、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。

二.

- 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

-> 模拟类似于 HttpSession 这种需要设定过期时间的功能。

-> 发布、订阅消息系统

-> 定时器、计数器

三.

- 去哪下?

四.

- 怎么玩?

-> 数据类型、基本操作和配置

-> 持久化和赋值， RDB / AOF

-> 事务的控制

-> 复制

五.

- 安装 redis

-> 远程传输工具上传 .tar.gz 文件，解压，并安装。

-> 如果make 命令报错，则需安装 gcc

-> make install

-> cd /usr/local/sbin

> redis-server /myredis/redis.conf ：启动服务

-> redis-cli -p 6379 : 进入端口

- gcc :

-> Linux 下的一个编译程序，是 C 程序的编译工具。

-> GUN 计划提供的编译器家族。

-> 支持 C，C++，Objecttive-C，Fortran，Java 和 Ada

-> yum install gcc- c++

六.

- Redis 启动后杂项基础知识讲解

-> 单进程:

- 通过对 epoll 函数的包装，实现单进程模型来处理客户端读写事件的请求响应。

- Epoll 是 Linux 内核为处理大批量文件描述符而作了改进的 epoll。

- 是Liux 下多路复用 IO 接口 select/poll 的增强版本。

- 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。

-> 默认16个数据库，类似数组下标从 0 开始，初始默认使用零号库。

-> Select 命令切换数据库

-> DBSIZE : 查看当前数据库的 key 的数量

-> keys * : 显示当前库所有 Key ， 支持占位符风格 ， 例如 keys k?

-> FLUSHDB : 清空当前库

-> FLUSHALL : 清楚所有库

-> 统一密码管理，16个库都是一个密码。

-> Redis 索引都是从 0 开始。

-> 默认端口是 6379 ，（merz : 作者纪念一个意大利的女歌手。）

八. Redis 的数据类型:

一.

- Redis 的五大数据类型 :

-> String : 字符串

- String 是 redis 最基本的类型，和 Memcached 一模一样的类型，一个 Key 对应一个 Value

- 二进制安全，可以包含任何数据，比如 jpg 图片或者序列化的对象。

- 一个 String 中 value 最多可以是 512 M

-> Hash : 哈希

- 一个键值对集合

- 是 String 类型的 field 和 value 的映射表，hash 特别适用于存储对象

- 类似 Java 里的Map<String,Object>

-> List : 列表

- 简单的字符串列表，按照插入顺序排序。

- 可以添加元素到头部或尾部。

- 底层实际是个链表 LinkdList

-> Set : 集合

- 是 String 类型的无序集合。

- 通过 HashTable 实现的。

-> Zset（sorted set）：有序集合

- 和Set 一样也是 String 类型元素的集合，且不允许重复的成员

- Zset 每个元素都会关联一个 double 类型的分数

- 通过分数来为集合中的成员进行从小到大的排序，成员唯一，分数(score)可以重复。

二.

- 常用操作:

-> Redis 键(key) :

- keys *

- exists key : 判断某个 key 是否存在

- move key db ： 移动到其他库

- expire key 秒钟 : 为给定的 key 设置过期时间

- ttl key : 查看还有多少秒过期，-1 表示永不过期，-2表示已过期。

- type key : 查看你的 key 是什么类型

-> Redis 字符(String):

- set/get/del/append/strlen

- Incr/decr/incrby/decrby： 递增和递减，一定要是数字

- getrange/setrange： 取得下标区间的值，从下标开始设置

- setex(set with expire) 键秒值 / setnx(set if not exit)：setex k1 10 v1

- mset/mget/msetnx： more set，同时搞多个东西

- getset (先get再set)

-> Redis 列表(List):

- lpush : 左压栈 | rpush: 右压栈 | lrange : 遍历List

- lpop : 左出栈 | rpop : 右出栈

- lindex : 按照索引下标获得元素（从上到下）

- llen : List 的长度

- lrem key : 删 N 个value

- ltrim key : 开始index，结束index，截取指定范围的值后再赋值给 key

- rpoplpush 源列表 目的列表 : 右出栈，左压栈。

- lset key index value : 根据下标的值

- linsert key before 值 : 在什么之前插入。 After： 在什么之后。

-> Redis 集合(Set):

- sadd : 往一个 Set 里添值，smembers : 查看一个 Set 里的值，sismember : 检查集合里是否有

- scard : 获取集合里面的元素个数

- srem key value : 删除集合中的元素

- srandmember key : 随机出几个数

- spop key : 随机出栈

- smove key1 key2 key1中的某个值 : 两个集合间移动值

- 数字集合类 :

-> sdiff : 差集 : 两个集合相差的部分

-> sinter : 交集 : 两个集合共同拥有的部分

-> sunion : 并集 : 两个集合合并起来

-> Redis 哈希(Hash　非常重要!)

- hset : 设置一个哈希 <key , <key,value>>
- hget : 获取一个哈希 key 中 key
- hmset : 设置多个哈希
- hmget : 获取多个哈希
- hgetall : 获取所有哈希
- hdel : 删除某个哈希
- hlen : 获取某个哈希中 几个 key 中 Key

-> Zset 集合(sorted set) :

- zadd : 设置一个 zset 集合
- zrange : 获取一个 zset 集合 [withscores] : 带分数
- zrangebyscore key 开始score 结束 score : 获取 zset 区间的值
    -> withscores : 带分数
    -> (不包含
    -> limit 开始下标，偏移量: 返回限制
- zrem Key 某 score 下对应的 value值，作用是删除元素
- zcard key score : 返回分数对应值的个数
    -> zcount key score区间 : 返回分数区间对应值的个数
    -> zrank key values值 : 取得下标
    -> zscore key 对应值 : 获得分数
- zrevrank key values 值 : 逆序获得下标
- zrevrange : 逆序获得 zset

- zrevrangebyscore key 分数区间 : 逆序获得分数区间的值

配置文件的介绍:

Redis 的持久化:

- RDB : Redis DataBase

-> 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

- 行话讲的 Snapshot 快照，恢复时是将快照文件直接读到内存里。

-> 具体细节:

- Redis 会单独创建（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，再替换上次持久化好的文件。

- 整个过程，主进程是不进行任何 IO 操作的，确保了极高的性能。

- 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。缺点是可能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）数值都原进程保持一致，
但是是一个全新的进程，作为原进程的子进程。

-> RDB 保存的是 dump.rdb 文件。

-> sava 秒钟 写操作次数

- RDB 是整个内存的压缩过的 Snapshot ， RDB 的数据结构，可以配置符合的快照触发条件。

- 默认:

-> 1 分钟改了 1万次

-> 5分钟改了10次

-> 15分钟改了1次

- 禁用:

-> 如果想禁用 RDB 持久化的策略，只要不设置任何 sava 指令，

或者给 save 传入一个空字符串参数即可。

-> Stop-writers-on-bgsave-error :

- 默认为 yes ， 后台执行保存备份出错时，前台停止写入操作。

- 如果设置为 No ， 可能会导致数据的不一致

-> rdbcompression :

- 对于存储到磁盘中的快照，可以设置是否进行存储压缩。

- 默认为 true ， 采用 LZF 算法进行压缩。

-> rdbchecksum :

- 存储快照后，还可以让 redis 使用 CRC64 算法来进行数据校验。

-> dbfilename :

- 默认为 dump.rdb

-> dir :

- 目录

-> 如何触发 RDB 快照:

- 配置文件中默认的快照配置

-> 冷拷贝到备机上。

- 命令 save 或者是 bgsave

-> save 只管快照备份，阻塞其他操作。

-> bgsave : 异步保存，快照同时还可以响应客户端请求，

通过 lastsave 命令获取最后一次成功执行快照的时间

- 执行 flushall 命令，但是空的，没有意义。

-> 如何恢复:

- 将备份文件(dump.rdb) 移动到 redis 安装目录并启动服务即可。

- config get dir 获取目录。

-> 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。

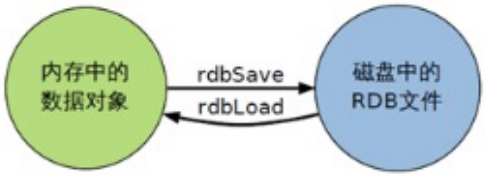
-> 劣势:

- redis 意外挂掉的话，就会丢失最后一次快照后的所有修改。

- Fork 的时候，内存中的数据都被克隆了一次，大致2倍的膨胀性需要考虑。

-> 如何停止:

- 动态停止所有RDB 保存规则的方法 : redis-cli config set save “”



- 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IO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,可能会导致Redis在一些毫秒级不能相应客户端请求

- AOF : Append Only File

-> 是什么?

- 以日志的形式来记录每个写操作，读操作不记录。

- 只许追加文件，但不可以改写文件。

- 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 保存的是 appendonly.aof 文件

-> 配置位置:

- APPEND ONLY MODE 追加 ， 设置成 yes

- appendonly : 默认为no， yes 打开 AOF备份

- appendfilename : AOF 文件的默认名。

- Appendfsync : 三种策略:

  -> Always : 同步持久化，性能差数据完整性较好。

  -> Everysec : 默认，异步操作，每秒记录。

-> No : 不记录

- No-appendfysnc-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
- Auto-aof-rewrite-min-size : 设置重写的基准值
- Auto-aof-rewrite-percentage : 设置重写的基准值

-> AOF 启动/修复/恢复

- 启动就是上面的配置
- 修复 redis-check-aof --fix appendonly.aof
- 恢复:
  -> 正常恢复:
    - 将有数据的 AOF 文件复制一份保存到对应目录 (config get dir)
    - 重启 redis 然后重新加载。
  -> 异常恢复:
    - 修复 redis-check-aof --fix appendonly.aof
    - 重启 redis 然后重新加载。

-> Rewrite

- 是什么 :
  -> AOF 采用文件追加方式，文件会越来越大。
  -> 避免此种情况，新增重写机制。
  -> 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
而是将这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。

类似快照。

- 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 触发

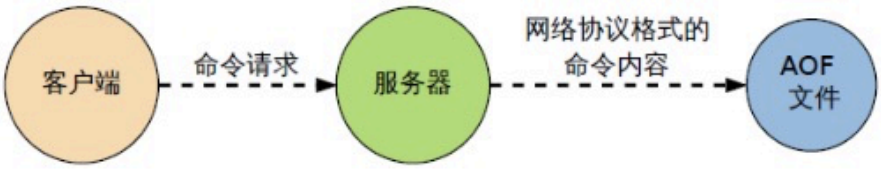
-> 优势:

- 同步性能比 RDB 高，确保数据完整性。

-> 劣势:

- 相同数据集的数据而言，AOF 文件要远大于 rdb 文件，恢复速度慢与 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



- 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

Redis 的事务:

- 是什么?

-> 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他命令插入，不加塞。

- 能干嘛?

-> 一个队列中，一次性、顺序性、排他性的执行一系列命令。

- 怎么玩?

-> 常用命令:

- DISCARD : 取消事务，放弃执行事务块内的所有命令。

- EXEC : 执行所有事务块内的命令。

- MULTI : 标记一个事务块的开始

- UNWATCH : 取消 WATCH 命令对所有 key 的监视。

- WATCH key [key ...] : 监视一个或多个 Key
如果在事务执行之前这个 key 被其他

命令所改动，那么事务被打断。

-> Case1 : 正常执行

-> Case2 : 放弃事务

-> Case3 : 全体连坐　　：编译直接报错

-> Case4 : 冤头债主　　：运行时异常

-> Case5 : watch 监控。

- 悲观锁 : 表锁，性能极差，安全性很高。

- 乐观锁 : 类似SVN，在表中添加版本号，想修改必须先同步到最新版本。

- CAS（Check And Set）：

- watch 指令类似乐观锁，事务提交时，如果 key 的值已经被其他线程改变，
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 发生变化，EXEC 命令执行事务失败，返回 Nil

- 3阶段:

-> 开启 : 以 MULIT 开始一个事务

-> 入队 : 将多个命令入队到事务中，不会立即执行。

-> 执行 : EXEC 命令触发事务

- 3特性:

-> 单独的隔离操作 :

- 事务中的所有命令都会序列化、按顺序的执行。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，都不会被执行。

- 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”问题。

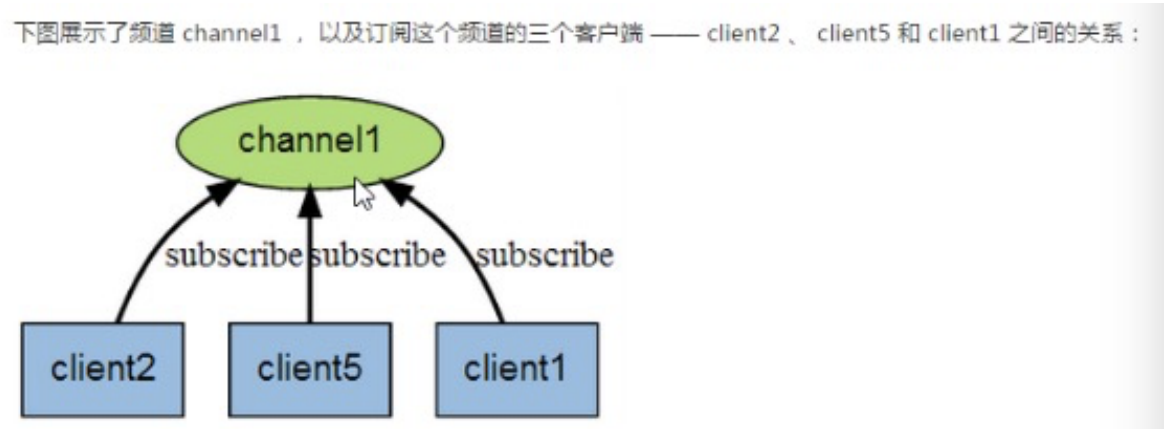
-> 不保证原子性：

- 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。

Redis 的发布订阅:

- 是什么?

-> 进程间的一种消息通信模式：发送者（pub）发送消息，订阅者（sub）接受消息



- 命令:

命令

序号	命令及描述
1	PSUBSCRIBE pattern [pattern ...] 订阅一个或多个符合给定模式的频道。
2	PUBSUB subcommand [argument [argument ...]] 查看订阅与发布系统状态。
3	PUBLISH channel message 将信息发送到指定的频道。
4	PUNSUBSCRIBE [pattern [pattern ...]] 退订所有给定模式的频道。
5	SUBSCRIBE channel [channel ...] 订阅给定的一个或多个频道的信息。
6	UNSUBSCRIBE [channel [channel ...]] 指退订给定的频道。

- Redis 的复制（Master/Slave）：

-> 是什么?

- 主从复制。主机更新数据后根据配置和策略，自动同步到备机。

- Master 以写为主，Slave 以读为主

-> 能干嘛?

- 读写分离：

- 容灾恢复：

-> 怎么玩?

- 配从库不配主库

- 从库配置：slaveof 主库IP 主库端口

-> 每次与 master 断开之后，都需要重新连接，除非配置进 redis.conf 文件

-> Info replication

- 修改配置文件细节操作

-> 拷贝多个 redis.conf 文件

-> 开启 daemonize yes

-> Pid 文件名字

-> 指定端口

-> Log 文件名字

-> dump.rdb 名字

- 常用 3 招 :

-> 一主二仆

-> 薪火相传

-> 链式主从，去中心化

-> 中途变更转向 : 清除之前的数据，拷贝最新的。

-> Slaveof 新主库IP ， 新主库端口

-> 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。

-> 复制原理:

- Slave 启动成功连接到 master 后会发送一个 sync 命令

- Master 接到命令启动后台的存盘进程，同时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 将传送整个数据文件到 slave，以完成一次命令同步。

- 全量复制 : slave 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。

- 增量复制 : Master 继续将新的所有收集到的修改命令依次传递给 slave ，完成同步。

- 只要重新连接到 master ，一次完全同步将被自动执行。

-> 哨兵模式（sentinel）：

- 是什么?

-> 反客为主的自动版，能够后台监控主机是否故障，true 就根据投票数自动将从库转换为主库

- 怎么玩（使用步骤）？

-> 调整结构

-> 自定义的 /myredis 目录下新建 sentinel.conf 文件，名字不能错。

-> 配置哨兵，填写内容

- sentinel monitor : 被监控的主机名字(自己起名字), 127.0.0.1 6379 1

- 上面最后一个数字 1， 表示主机挂掉后 salve 投票数。

- 启动哨兵

-> redis-sentinel /myredis/sentinel.conf

- 正常主从演示

- 原有的 master 挂了

- 投票新选

- 重新主从继续开工，info replication 查查看

- 问题 : 如果之前的 master 重启回来，会不会双 master 冲突

-> 成为新 master 的 slave

- 一组sentinel 能同时监控多个 Master

-> 复制缺点：

- 复制延时：

-> 由于所有的写操作都是现在 Master 上操作，然后同步更新到 Slave 上，
所有从 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。