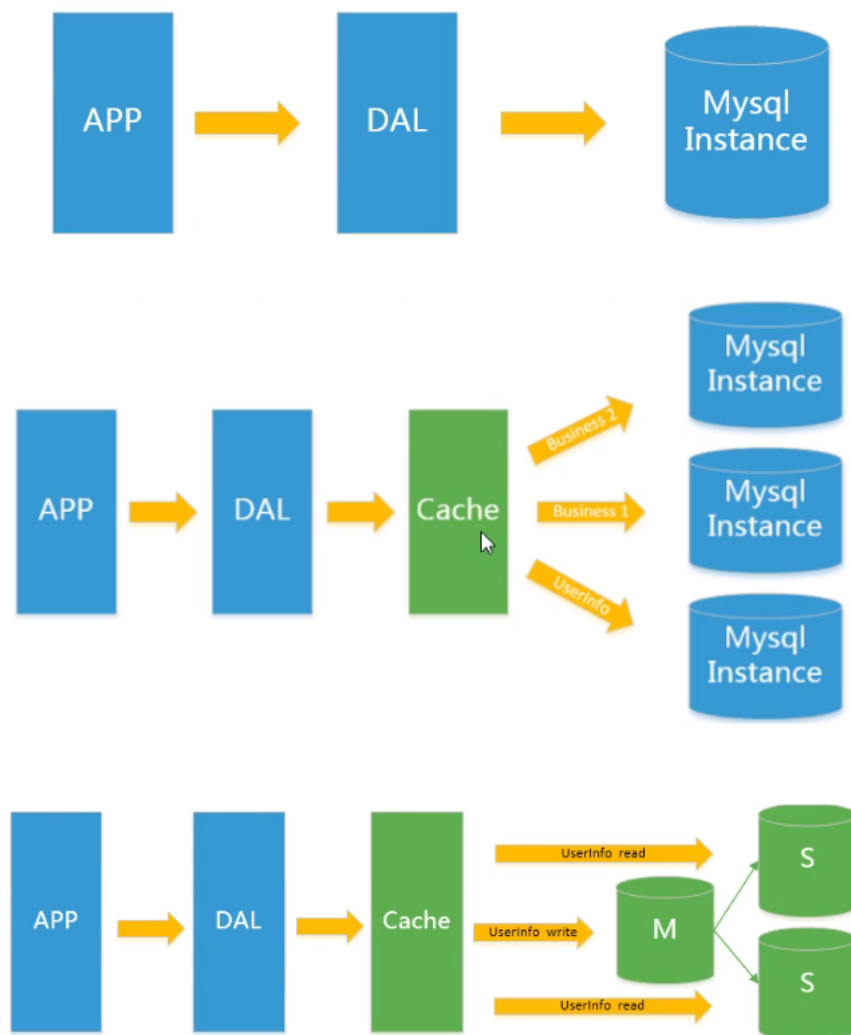


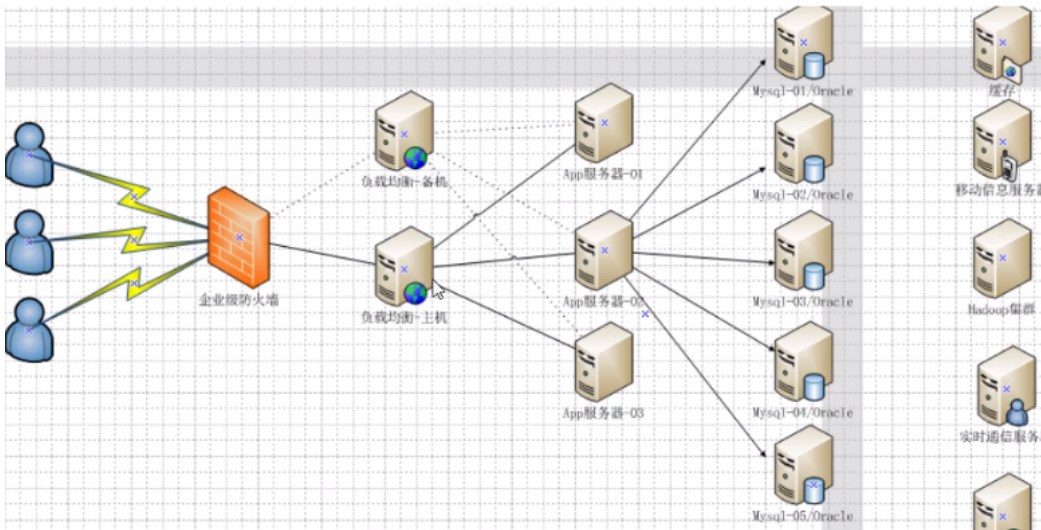
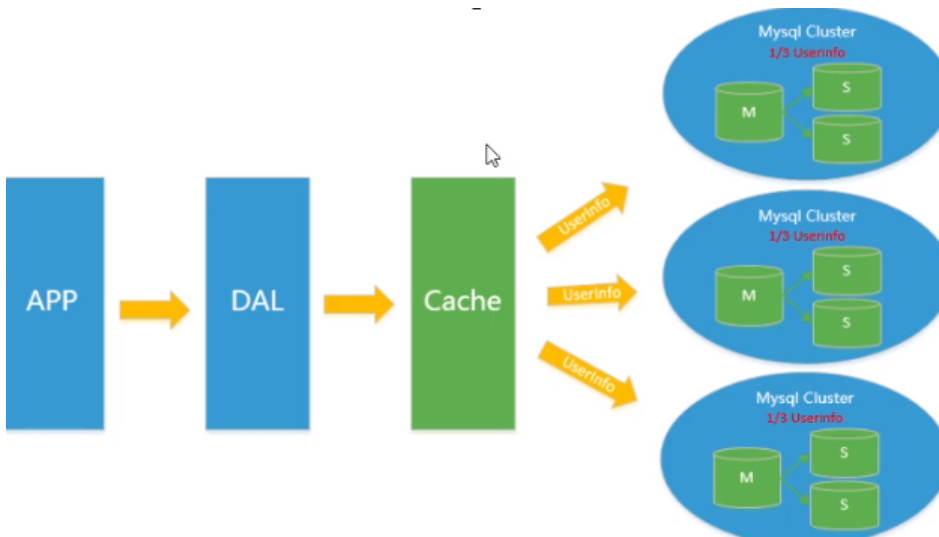
# NoSQL

---

## 1. 背景介绍

### 历史发展





## 为什么使用 NoSQL

我们对于用户数据进行挖掘，SQL数据库已经不适合这些应用了

NoSQL的发展可以很好处理这些大的数据

这一类型的数据存储不需要固定的模式，无需多于操作就可以横向拓展

## NoSQL的作用

易扩展

- 数据之间物关系，非常易于扩展

大数据量高性能

- NoSQL具有非常高的读写性能，在大数据量下同样表现优秀
- 因为它的无关系性，数据库的结构简单

## 多样灵活的数据模型

- 无需事先为存储的数据建立字段，随时可以存储自定义的数据格式

## NoSQL

- 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）
- 非结构化和不可预知的数据

## 如何使用

- Key-Value
- Cache
- Persistence

## 3V+3高

### 大数据时代的3V

- 海量Volume
- 多样Variety
- 实时Velocity

### 互联网需求的3高

- 高并发
- 高可扩展
- 高性能

## 当下NoSQL的应用

当下的应用是SQL和NoSQL一起使用

# 数据库运用-以淘宝商城为例

## 栏目

- 商品基本信息：名称、价格、出厂日期、生产厂商  
关系型数据库 MySQL
- 多文字类：商品描述、详情、评价信息  
文档数据库 MongoDB
- 商品的图片  
分布式文件系统 Hadoop HDFS
- 商品的关键字  
搜索引擎 ISearch
- 商品的波段性的热点高频信息  
Tair、Redis、Memcache
- 商品的交易、价格计算、积分累计  
微信支付、支付宝支付、第三方接口

## 企业面临的难题

- 数据类型多样性
- 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

## 解决方法

使用统一数据平台，满足各种数据库的使用调用

## 2. NoSQL简介

### BOSN

BOSN是一种类似 json 的一种二进制文件

简称 Binary JSON

它和 JSON 一样，支持内嵌的文档对象和数组对象

```
{
  "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"}}],
      }
    ]
  }
}
```

- 高并发的操作不太建议关联查询
- 互联网公司用冗余数据来避免关联查询
- 分布式事务支持不了太多并发

## NoSQL数据模型与分类

### KV键值

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

### 文档型数据库（BSON）

- CouchDB
- MongoDB: 基于分布式文件存储的数据库，介于关系数据库和非关系数据库之间

### 列族

- Cassandra
- HBase

### 图

存放朋友圈社交网络、广告推荐系统等

专注于构建关系图谱

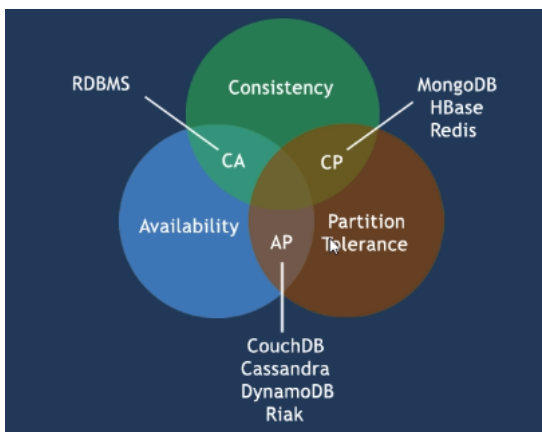
- Neo4J
- InfoGrid

## CAP

- C（Consistency）：强一致性
- A（Availability）：可用性
- P（Partition tolerance）：分区容错性

CAP理论核心是：一个分布式系统不可能同时很好的满足一致性、可用型和分区容错性三个需求，最多只能同时较好的满足两个

因此，根据CAP原理，将NoSQL数据库分成了满足CA原则、满足CP原则和满足AP原则三大类



- CA：单可扩展性不强大，传统数据库
- CP：性能不是很高，Redis，Mongodb
- AP：对一致性要求低一些，大多数网站架构的选择可以降低一些一致性的容忍度

## BASE

BASE是为了解决关系型数据库强一致性引起的可用性降低而提出的解决方案

- BA（Basically Available）：基本可用
- S（Soft state）：软状态
- E（Eventually consistent）：最终一致

通过让系统放松对某时刻一致性的要求来换取整体伸缩性和性能上的改观

## 分布式与集群简介

- **分布式**：不同的多台服务器上面部署不同的服务模块，对内通讯，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 service 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

## 3. Redis

### 基本介绍

Redis: **RE**mote **D**ictionary **S**erver（远程字典服务器）

完全开源免费，高性能的键值对分布式数据

- 支持数据持久化
- 不仅支持键值对类型，还提供列表、集合、哈希等数据结构的存储
- 支持数据备份（主从模式）

Redis是单进程的数据库，默认内置16个数据库[0, 15]

[Redis使用手册](#)

### 基本命令

#### 清空命令行

```
1 CLEAR
```

#### 切换数据库

```
1 SELECT <index>
```

查看数据库的值的数目

```
1 DBSIZE
```

清空数据库

```
1 FLUSHDB      -- 清空当前数据库的内容
2 FLUSHALL     -- 清空所有数据库
```

查看某一个键的数据类型

```
1 TYPE <key>
```

KEY

设置键值对

```
1 SET <key> <value>
```

查看所有键

```
1 KEYS *
```

根据键获取值

```
1 GET <key>
```

判断是否存在某个键

```
1 EXISTS <key>
```



将当前某一键值对移动到另一个数据库去

```
1 MOVE <key> <index>
```

查看某一个键值对的存在时间

```
1 TTL <key>
2 -- >0 表示剩余的存在时间
3 -- -1 表示永不过期
4 -- -2 表示过期无法访问（过期后将无法用 GET 访问）
```

给某一键值对指示存在时间倒计时

```
1 EXPIRE <key> <time (s)>
```

删除某一键值对

```
1 DEL <key>
```

## STRING

String 是 Redis 最基本的类型，一个 key 对应一个 value

String 是二进制安全的，意味着它可以包含任何数据，包括图片或者序列化对象

一个 Redis 中的字符串 value 最多可以是512M

向键值对的值后连接字符串

```
1 APPEND <key> <value>
2 -- 例如有键值对 <k1 - "ty">
3 -- 输入 append k1 12345
4 -- 结果为 <k1 - "ty12345">
```

## 键值对的值加一/减一操作

index 对应的 value++

```
1 INCR <key>           -- value++
2 DECR <key>            -- value--
3 INCRBY <key> <num>    -- value += num
4 DECRBY <key> <num>    -- value -= num
5 -- SET k2 1
6 -- INCR k2
7 -- GET k2 → "2"
```

## 获取指定键值对的值的部分范围

```
1 GETRANGE <key> <start> <end>
2 -- GETRANGE k1 0 -1 获得k1的value从0开始到最后(-1)的内容
3 -- GETRANGE k1 0 3 获得k1的value其中[0,1,2,3]位上的子串
```

## 对指定键值对的值在范围内插入值

```
1 SETRANGE <index> <start> <value>
2 -- SETRANGE k1 0 xxx k1的value从第0位开始添加"xxx"
```

## 创建键值对，并设定倒计时

```
1 SETEX <key> <time> <value>
```

## 在键值对不存在的情况下插入

```
1 SETNX <key> <value> -- 如果 key 不存在，那么才插入这个键值对
```

## 同时设置/获取多个键值对

```
1 MSET <key1> <value1> <key2> <value2>
2 MGET <key1> <key2> <key3>
3 MSETNX <key1> <value1> <key2> <value2>
```

# LIST

Redis 列表是简单的字符串列表，按照插入顺序排序

可以添加一个元素到列表的头部或者尾部

底层其实是一个链表

## 创建LIST

```
1  -- (从左边开始创建)
2  LPUSH <list_name> <value1> <value2> ...
3  -- (从右边开始创建) 正进正出
4  RPUSH <list_name> <value1> <value2> ...
```

## 获得LIST中指定范围的数据

```
1  LRange <list_name> <start> <end>
2  -- LRange list1 0 -1 获取 list1 从0到最后的所有数据
3
```

## 弹出LIST里面的元素

```
1  LPOP <list_name> -- 弹出左边第一个
2  RPOP <list_name> -- 弹出右边第一个
```

## 查看LIST指定index的元素

```
1  LINDEX <list_name> <index>
```


## 查看LIST长度

```
1  LLEN <list_name>
```

## 截取指定部分并再赋值给LIST

```
1 LTRIM <list_name> <start> <end>
```

```
127.0.0.1:6379> lpush list1 0 1 2 3 4 5 6 7 8 9
(integer) 10
127.0.0.1:6379> LRANGE list1 0 -1
1) "9"
2) "8"
3) "7"
4) "6"
5) "5"
6) "4"
7) "3"
8) "2"
9) "1"
10) "0"
127.0.0.1:6379> LTRIM list1 3 5
OK
127.0.0.1:6379> LRANGE list1 0 -1
1) "6"
2) "5"
3) "4"
127.0.0.1:6379>
```



## 重新设置LIST某一index上的值

```
1 LSET <list_name> <index> <value>
```

## 在LIST某个值前面/后面插入

```
1 LINSERT <list_name> BEFORE <value> <new_value> -- new_value 插入到
value 之前
2 LINSERT <list_name> AFTER <value> <new_value> -- new_value 插入到
value 之后
```

## SET

Redis 的 Set 是 String 类型的无序集合

它是通过 HashTable 实现的

## 创建SET

```
1 SADD <set_name> <value1> <value2> ... -- Set只允许不重复的值，如果输入
重复的值，则重复的不会再放进去
```

## 查看SET的内容

```
1 SMEMBERS <set_name>
```

## 获取SET里的元素个数

```
1 SCARD <set_name>
```

## 删除SET中的元素

```
1 SREM <set_name> <value>
```

## 随机从SET里面挑出指定数量的值

```
1 SRANDMEMBER <set_name> <num>
2 -- SRANDMEMBER set01 3 从set01中随机挑三个值
```

## SET里面随机出栈

从 set 中随机出去一个值

```
1 SPOP <set_name>
```

## 将一个SET中的指定值移动到另一个SET里面

```
1 SMOVE <set1> <set2> <value>
2 -- 将 set1 的一个 value 移动到 set2 中
```

## 删除SET

```
1 DEL <set_name>
```

## 差交并

```
1 SDIFF <set1> <set2> -- 在 set1 但是不在 set2 的元素
2 SINTER <set1> <set2> -- set1 和 set2 的交集
3 SUNION <set1> <set2> -- set1 和 set2 的并集
```

## HASH

是一个键值对集合

是一个 String 类型的 field 和 value 的映射表，hash 特别适合存储对象

类似于 Java 里面的 Map<String, Object>

Key-Value模式不变，但是Value是一个键值对

### 创建HASH SET

```
1 HSET <hash_set_name> <key> <value>
```

### 获取HASH SET中的值

```
1 HGET <hash_set_name> <key>
```

### HASH SET同时设置多个键值对

```
1 HMSET <hash_set_name> <key1> <value1> <key2> <value2> ...
```

### HASH SET同时获得多个键值对

```
1 HMGET <hash_set_name> <key1> <key2>
```

### HASH SET同时获得所有键值对

```
1 HGETALL <hash_set_name>
```

## 查看HASH SET大小

```
1 HLEN <hash_set_name>
```

## 查看HASH SET里面是否存在某个KEY

```
1 HEXISTS <hash_set_name> <key>
2 -- 1: 有
3 -- 0: 没有
```

## 查看HASH SET里面所有的键/值

```
1 HKEYS <hash_set_name> -- 查看所有键
2 HVALS <hash_set_name> -- 查看所有值
```

## HASH SET 指定的键对应的值数字增加

```
1 HINCRBY <hash_set_name> <key> <num> -- key 所对应的 value+=num
```

## 如果不存在，则插入键值对

```
1 HSETNX <hash_set_name> <key> <value>
```

## ZSET

在 Set 的基础上，加了一个 score 值

之前 Set 是 k1 v1, k2 v2

现在 Zset 是 k1 score1 v1, k2 score2 v2

和 Set 一样也是 String 类型的元素的集合，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 通过分数 score 来为集合中的成员从小到大排序

Zset 的成员是唯一的，但分数 score 却可以重复

## 创建ZSET

```
1 ZADD <zset_name> <score1> <value1> <score2> <value2> ...
```

## 查看ZSET里面的值

```
1 ZRANGE <zset_name> <start> <end>
2 ZRANGE <zset_name> <start> <end> WITHSCORES -- 查看所有值并且带有分数
```

## 查看ZSET指定score范围的值

```
1 ZRANGEBYSCORE <zset_name> <low_bound> <up_bound> -- 查看分数在
[low, up] 之间的所有值
```

## 移除ZSET里面的值

```
1 ZREM <zset_name> <value>
```

## 查看ZSET大小

```
1 ZCARD <zset_name>
```

## 查看ZSET里面指定score范围的元素个数

```
1 ZCOUNT <zset_name> <low_bound> <up_bound> -- 查看分数在[low, up]之间
的值的个数
```

## 查看值在ZSET里面的排名

```
1 ZRANK <zset_name> <value>
```



查看ZSET里面的值对应的score

```
1 ZSCORE <zset_name> <value>
```

## 4. Redis 持久化

### RDB（Redis DataBase）

在指定的时间间隔内将内存中的数据集快照写入磁盘

Redis会单独创建（fork）一个子进程来进行持久化，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，主进程不进行任何的IO操作

如果需要进行大规模数据的修复，且对于数据恢复的完整性不是非常敏感，则RDB比AOF要更加高效

RDB的缺点是最后一次持久化后的数据容易丢失

### FOLK

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）数值都与原进程一致，但是是一个全新的进程，并作为原进程的子进程

### RDB的触发情况

- 900s 内如果有至少 1 个 key 被改动了
- 300s 内如果有至少 10 个 key 被改动了
- 60s 内如果有至少 10000 个 key 被改动了

## 5. Redis 事务

可以一次性执行多个命令，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

按顺序串行化执行，而不会被其它命令插入，不许加塞

在一个队列中，一次性、顺序的、排他性的执行一系列命令

## 事务执行

使用MULTI命令进入Redis事务

该命令总是返回OK

此时，用户可以发出多个命令，而不是执行这些命令，Redis将它们入队。一旦调用EXEC，所有命令都将被执行。而调用DISCARD将刷新事务队列并退出事务

## 事务命令

- MULTI: 标记一个事务块的开始
- EXEC: 执行所有事务块内的命令
- DISCARD: 取消事务、放弃执行事务块内的所有命令
- WATCH <key...>: 监视一个（或多个）key，如果事务执行之前这个/这些key被其他命令所改动，那么事务将被打断

## 正常执行

```
1 MULTI -- 开启事务，会返回OK
2 之后输入的东西都会开始排队...
3 EXEC -- 会按顺序完成操作
```

## 放弃事务

```
1 MULTI -- 开启事务，会返回OK
2 之后输入的东西都会开始排队...
3 DISCARD -- 回滚
```

## 编译异常

当一条命令出现编译错误的时候（可以是语法、其它），其它的命令都将不会执行

## 运行异常

当一条命令在执行过程中因为数据格式等不支持的原因出错时，只有该命令不会被执行成功，其它依旧执行成功

- Redis不保证原子性

## Watch监控

### 关系型数据库锁

- 表锁：锁住一整张表，并发性差，准确性强
- 行锁：锁住表的具体某一行
- 列锁：锁住表的具体某一列

### 悲观锁

每次去拿数据的时候认为别人一定会修改，所以在每次拿的时候都会上锁，这样后面的人想要拿到这个数据的时候就必须排队，直到锁被释放

- 一致性好
- 性能下降

### 乐观锁

每次去拿数据的时候认为别人不会修改，所以不会上锁，但是在更新的时候会判断在此期间别人是否有更新这个数据，可以提高吞吐量

- 开放整张表允许并发修改
- 在每行后新增一个version版本号，每次commit后版本号+1
- 如果出现提交时候本次提交版本号与数据库内版本号不同，则rollback（有人相对同时却比你早提交本条记录了）

```
1 WATCH <key> -- 开始监视
2 MULTI -- 开启事务，会返回OK
3 之后输入的东西都会开始排队...
4 EXEC -- 如果<key>在此期间被改动，该事务将不能成功执行
5 UNWATCH <key> -- 取消监视
```

