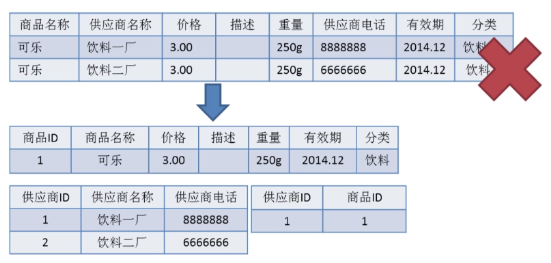
# 数据库

## 范式

1. 1NF
   1. 表中的属性/字段/列不能再被分割。
   2. 是关系型数据库的最基本要求
   3. 举例：



1. 2NF
   1. 在INF基础上，要求所有非主键都依赖于主键
   2. 个人认为，意义在于明晰每张表“是做什么的”。
   3. 举例：



1. 3NF
   1. 3NF 在 2NF 的基础之上，消除了非主属性对于码的传递函数依赖。
   2. 意义：进一步原子化。利于查询效率，但是设计困难。
   3. 举例：在关系 R(学号 , 姓名, 系名，系主任)中，学号 → 系名，系名 → 系主任。应该将（系名，系主任）拆分出一张新表。

## 存储过程

1. 是什么：sql语句的集合
2. 好：快-因为预编译；方便调用。
3. 坏：难以调试-因为写死了；没有一致性；占用数据库资源
4. 实际应用：阿里巴巴java开发手册禁用。

## drop、delete 与 truncate 区别？

1. Drop
   1. 扔到垃圾桶。全删了，表结构和数据全删了。
   2. drop table 表名
2. Truncate
   1. 截断、砍树。保留表结构。下次插入从id=1又长起来了。
   2. truncate table 表名
3. Delete
   1. 看起来和truncate本质一样。只是可以按条件截断。
   2. 区别：delete删除的数据可以回滚恢复，而truncate不可以
      * 仅将记录的delete falg设置为1，记录本身并没有被删除，即记录还存在B+树中。真正的删除操作其实被“延时”，最终在purge操作中完成。
   3. delete from 表名 where 列名=值

## 分库分表

1. 从垂直方向上——针对业务
   1. 分表：
      1. why：单表中的字段太多
      2. how：拆分出核心字段和非核心字段
      3. e.g.： 用户基本信息表/用户扩展表；订单表/订单详情表
   2. 分库：
      1. Why：功能多、系统各种关联、luojifuza
      2. How：按业务功能，按不同领域分库
      3. E.g. : 用户、产品、物流、订单相关的表，从原来一个数据库中，拆分成单独的用户库、产品库、物流库和订单库，一共四个数据库
2. 从水平方向上——针对数据
   1. 主从读写分离：
      1. Why：一般业务，读的请求很多，写的请求很少
      2. How：1. 所有写请求都给主库，主库写完后异步同步给从库。读请求则即时从从库中读到。2.如果主库挂了，升级从库给主库，又写又读。3. 不够，可以一主二从，一主多从。
   2. 分库：
      1. Why：1. 数据库连接资源不足 2.磁盘IO性能瓶颈
      2. How：经典方案-主从读写分离
      3. 场景：用户并发量大，但是需要保存的数据量小
      4. Eg：一个账号，接入多个具体的游戏厂商提供的游戏界面。(每个库单表，表不太一样。登录接口传入游戏id包含在库名中)
   3. 分表：
      1. Why：单表数据量太大，sql查询即使有索引也很慢
      2. How：
      3. 场景：用户并发不大，但是需要保存的数据条数很多
      4. Eg：游戏中，会员获得积分的方式有十数种，为了记录用户关于积分相关的数据量。
   4. 分库分表：
      1. Why：分库+分表的why
      2. How：用户请求来时，先根据用户id路由到其中一个用户库，然后再定位到某张表。(其中，路由算法有：根据id取模；给id指定区间范围；一致性哈希算法)
      3. 场景：并发量大，而且需要保存的数据量也大。

## 事务

1. 事务

1. ACID：**只有保证了事务的持久性、原子性、隔离性之后，一致性才能得到保障。也就是说 A、I、D 是手段，C 是目的！**

2. 单条sql语句数据库自动作为事务执行——隐式事务；

多条sql用begin…commit显式声明——显式事务

2. **隔离级别**

Read Uncommitted：读到未提交的即为“脏读”

Read Committed: 长查询和短修改并行，两次查询读到的数据不一致，即为“不可重复读”

\*因为Read Committed级别每次select得到最新行快照，而Repeatable Read读事务开始的行快照

Repeatable Read: 长查询和短插入并行，查同一批数据，第一次读不到，第二次居然能读到，即为“幻读”

\* InnnoDB存储引擎在Repeatable Read级别，使用Next-Key Locking机制，解决了幻读。即InnnoDB在Repeatable Read级别已经能完全保证事务的隔离性要求，即达到SQL标准的SERIALIZABLE隔离级别

Oracle及其他数据库在Serializable级别才解决幻读Phantom Probem

Serializable: 串行。避免了以上三个问题，但是效率低。

\*innodb效率其实差不多

## 关系数据库

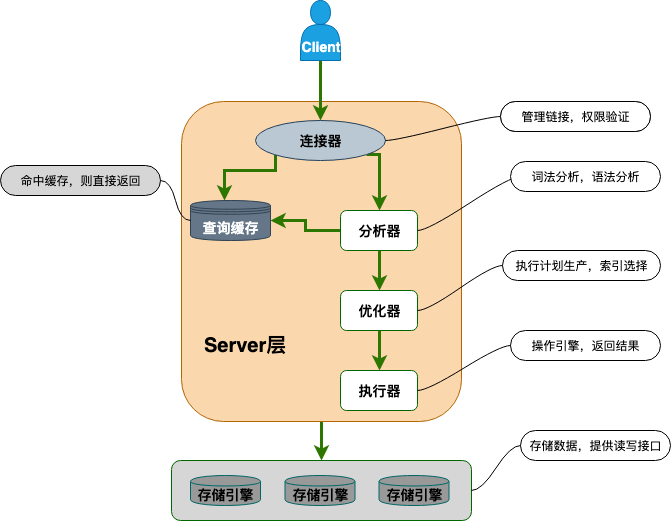
层次数据库、网状数据库、关系数据库

建立在关系模型（关系有：一对一、一对多、多对一）基础上的数据库

## MVCC

# MySQL

## MySQL数据库和InnoDB存储引擎的区别



【注意】1. MySQL采用插件式存储引擎

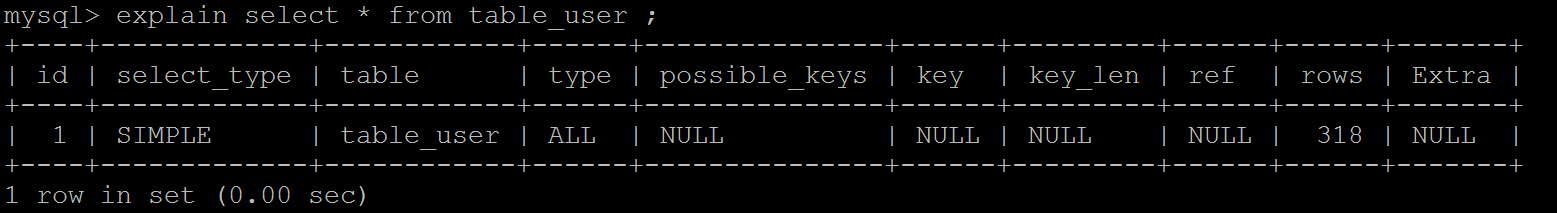
2. 存储引擎是基于表的，而不是数据库。

## B+ Tree

## 连接查询和子查询

1. 连接查询
   1. 内连接：inner join
   2. 外连接：left/right/outer join
2. 区别与选择：
   1. 子查询会多次遍历所有数据，连接查询只会遍历一次
   2. 一般情况下，数据量较大情况下，都选用连接。除非子查询只有一条记录且唯一。

## Explain命令



## Varchar和char的区别

**1. char的长度是不可变的，而varchar的长度是可变的。**

定义一个char[10]和varchar[10]。

如果存进去的是‘tao’,那么char所占的长度依然为3，除了字符‘tao’外，后面跟7个空格，varchar就立马把长度变为3了，取数据的时候，char类型的要用trim()去掉多余的空格，而varchar是不需要的。

**2. char的存取速度还是要比varchar要快得多，因为其长度固定，方便程序的存储与查找。**

char也为此付出的是空间的代价，因为其长度固定，所以难免会有多余的空格占位符占据空间，可谓是**以空间换取时间**效率。

varchar是以空间效率为首位。

**3. char的存储方式是：对英文字符（ASCII）占用1个字节，对一个汉字占用两个字节。** **varchar的存储方式是：对每个英文字符占用2个字节，汉字也占用2个字节。**

**两者的存储数据都非unicode的字符数据。**

**nchar和nvarchar是存储的unicode字符串数据**

## mysql模糊查询索引失效问题及解决方案

.不在索引列上做任何操作（计算、函数、（自动or手动）类型转换），会导致索引失效而转向全表扫描

2.尽量使用覆盖索引（只访问索引的查询（索引列和查询咧一致）），减少select\*

3.mysql在使用不等于（！=、<>）或like的左模糊查询的时候无法使用索引会导致全表扫描

4.IS NULL 和 is not null 也无法使用索引

5.字符串不加单引号索引失效，因为这里有一个隐式的类型的转换操作，更严重会导致行锁变表锁，降低sql效率

6.少用or ,用它来连接时索引会失效

7.%不能在左边

## SQL优化

1. 尽量避免使用子查询 ->多次查表，改用join连接
2. 用IN来代替OR -> IN被优化，在B+树内部排序
3. 禁止不必要的order by排序
4. 避免随机读取数据 ->无法用到索引
5. 将多次插入换成批量插入
6. 只返回必要的列，少用select \* -> 增加不必要的消耗（cpu、io、内存带宽）
7. 尽量使用数字型字段，而非字符型 -> 只需要比较一次，字符需要逐一比较

## 索引优化

1. **最佳左前缀法则 -> 最查询的放最左**
2. **不在索引列上做任何操作 ->会导致索引失效**
3. **把> < <>放在查询的最右侧**
4. **尽量使用覆盖索引，即，不用select \***
5. **like 以通配符开头(%abc..)时，mysql索引失效会变成全表扫描的操作。**
6. **少用or**
7. **在组合/联合索引中，将有区分度的索引放在前面**
8. **使用前缀索引，**
   1. 比如有一个varchar(255)的列，如果该列在前10个或20个字符内，可以做到既使前缀索引的区分度接近全列索引，那么就不要对整个列进行索引。为了减少key\_len，可以考虑创建前缀索引，即指定一个前缀长度，可以使用count(distinct leftIndex(列名, 索引长度))/count(\*) 来计算前缀索引的区分度。

# Redis

<https://juejin.cn/post/6844904017387077640>

## Redis 缓存雪崩、击穿、穿透

<https://segmentfault.com/a/1190000022029639>

雪崩：某一秒key集体失效，请求全部落到了数据库，数据库挂了 ->失效时间加个随机值

击穿：是指一个Key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个Key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库 -> 1. 设置热点数据永不过期 2.互斥锁

穿透：缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id值为 -1 的数据或 id 为特别大不存在的数据。（这样的用户很可能是攻击者） -> 1. 在接口层增加校验 2. 布隆过滤器（判断这个key在数据库存在与否）

Ps. 一般避免以上情况发生我们从三个时间段去分析下：

* 事前：**Redis** 高可用，主从+哨兵，**Redis cluster**，避免全盘崩溃。
* 事中：本地 **ehcache** 缓存 + **Hystrix** 限流+降级，避免**MySQL** 被打死。
* 事后：**Redis** 持久化 **RDB**+**AOF**，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据。

# 缓存

## 缓存cache vs 缓冲buffer

1、**Buffer**（缓冲区）是系统两端处理**速度平衡**（从长时间尺度上看）时使用的。它的引入是为了减小短期内突发I/O的影响，起到**流量整形**的作用。比如生产者——消费者问题，他们产生和消耗资源的速度大体接近，加一个buffer可以抵消掉资源刚产生/消耗时的突然变化。  
2、**Cache**（缓存）则是系统两端处理**速度不匹配**时的一种**折衷策略**。因为CPU和memory之间的速度差异越来越大，所以人们充分利用数据的局部性（locality）特征，通过使用存储系统分级（memory hierarchy）的策略来减小这种差异带来的影响。  
3、假定以后存储器访问变得跟CPU做计算一样快，cache就可以消失，但是buffer依然存在。比如从网络上下载东西，瞬时速率可能会有较大变化，但从长期来看却是稳定的，这样就能通过引入一个buffer使得OS接收数据的速率更稳定，进一步减少对磁盘的伤害。  
4、TLB（Translation Lookaside Buffer，翻译后备缓冲器）名字起错了，其实它是一个cache

## 什么是缓存？与缓存的应用场景

缓存就是cache，数据交换的中间层，空间换时间，减少与数据库/磁盘/网络的IO

**文件缓存**：文件缓存是把一些需要高速存取的变量缓存在内存中。模板引擎用的就是文件缓存机制，把动态代码编译成静态文件放入硬盘，不用每次访问都编译，直接读出即可。

**浏览器缓存**：浏览器缓存根据一套与服务器约定的规则进行工作，在同一个会话过程中会检查一次并确定缓存的副本足够新。如果在浏览过程中前进或后退时访问到同一个图片，这些图片可以从浏览器缓存中调出而即时显示。

**数据库缓存**：常用的缓存方案有memcached、redis等。把经常需要从数据库查询的数据、或经常更新的数据放入到缓存中，这样下次查询时，直接从缓存直接返回，减轻数据库压力，提升数据库性能。

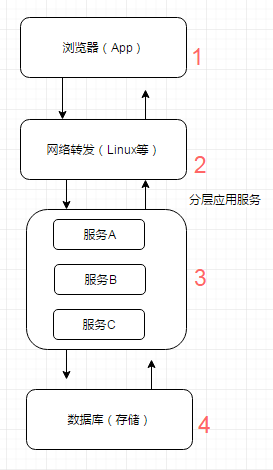
**Web应用层缓存**:应用层缓存指的是从代码层面上，通过代码逻辑和缓存策略，实现对数据、页面、图片等资源的缓存，可以根据实际情况选择将数据存在文件系统或者内存中，减少数据库查询或者读写瓶颈，提高响应效率。

**服务器缓存**：包括**代理服务器缓存**和**CDN缓存**。  
**代理服务器缓存**：代理服务器是浏览器和源服务器之间的中间服务器，浏览器先向这个中间服务器发起Web请求，经过处理后(比如权限验证，缓存匹配等)，再将请求转发到源服务器。  
代理服务器缓存的运作原理跟浏览器的运作原理差不多，只是规模更大。可以把它理解为一个共享缓存，不只为一个用户服务，一般为大量用户提供服务，因此在减少响应时间和带宽使用方面很有效，同一个副本会被重用多次。  
**CDN缓存**：也叫网关缓存、反向代理缓存。CDN缓存一般是由网站管理员自己部署，为了让他们的网站更容易扩展并获得更好的性能。  
浏览器先向CDN网关发起Web请求，网关服务器后面对应着一台或多台负载均衡源服务器，会根据它们的负载请求，动态将请求转发到合适的源服务器上。  
虽然这种架构负载均衡源服务器之间的缓存没法共享，但却拥有更好的处扩展性。从浏览器角度来看，整个CDN就是一个源服务器。

## 缓存雪崩、击穿、穿透问题->见redis

## 美团技术团队《缓存那些事》笔记

缓存的使用可以出现在1～4的各个环节中，每个环节的缓存方案与使用各有特点。



**缓存的特征**：

命中率

最大空间

清空策略（FIFO\LRU\LFU）

缓存介质（内存、硬盘、数据库）

**缓存分类和应用场景：**本地缓存+分布式缓存

本地缓存：

指的是在应用中的缓存组件，其最大的优点是应用和cache是在同一个进程内部，没有多余的网络开销；缺点是和应用耦合，多个应用程序无法直接的共享缓存，浪费内存。

a. 编程直接实现

成员变量或局部变量实现

静态变量是实现->**单例**

b. Ehcache：纯java开源缓存框架，Hibernate就集成了它

分布式缓存：

指的是与应用分离的缓存组件或服务，其最大的优点是自身就是一个独立的应用，与本地应用隔离，多个应用可直接的共享缓存。

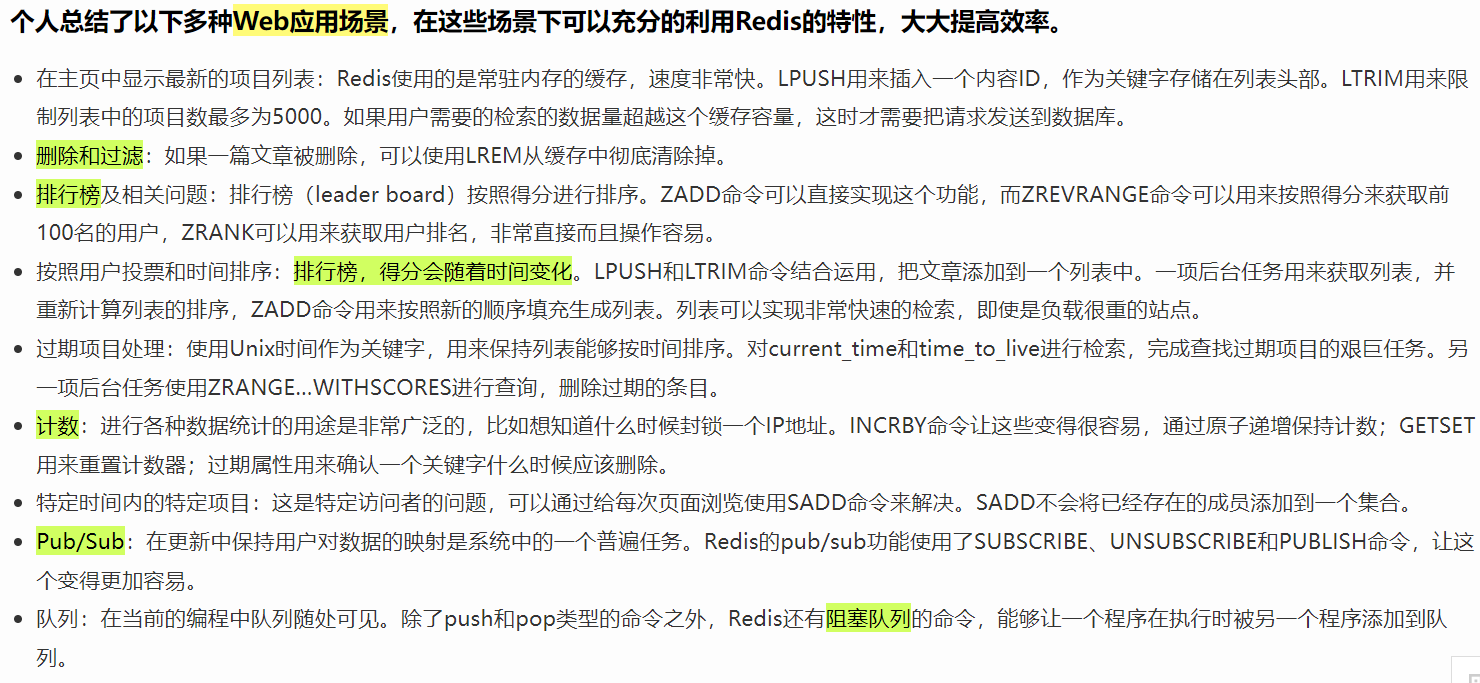
a. **memcached缓存**

它本身其实不提供分布式解决方案。在服务端，memcached集群环境实际就是一个个memcached服务器的堆积；cache的分布式主要是在客户端实现，通过客户端的路由处理来达到分布式解决方案的目的。（路由策略：一致性hash算法）

b. **Redis缓存**

非关系型数据库。键值对存储在内存，可以持久化到硬盘。（RDB快照+aof语句追加文件）利用复制特性来扩展读，客户端切片来扩展写。

与memcached客户端支持分布式方案不同，Redis更倾向于在服务端构建分布式存储（集群）



Ps. 注解使用缓存

AOP动态代理。Spring Cache主要使用三个注释标签，即@Cacheable、@CachePut和@CacheEvict