数据库

SQL：结构化查询语言，用于与数据库通讯，

可以创建1.数据库2.数据库中的表3.存储过程4.图表

DBMS(数据库管理系统)：执行SQL查询的软件

SQL vs MySQL：SQL是一个产品语言，和数据库交流

MySQL不是语言，是数据库管理系统

inner Join：产出两张表同时满足的记录

left join：把左表所有查询信息列出来，而右表只列出On后面条件与左表满足 的部分，左表有右表没有的列值为空

right join：把右表所有查询信息列出来，而左表只列出On后条件与右表满 足的部分，右表有左表没有的列值为空—NULL

**MySQL join原理**—nested-loop join

1.简单nested-loop，取主表(驱动表)每一行找右表每一行去匹配，访问次数多

2.索引nested-loop要求非驱动表有索引列，驱动表找到索引后才会回表查询

3.阻塞nested-loop把驱动表和join相关的列(包括on和select的列)先缓存到join buffer中再去匹配，这样就把第一种loop的多次比较合并为一次，效率高

**MySQL中char、varchar和text类型的区别：**

char：存固定长度字符， 0-255，不够255会用空格代替，所以肯定要占一定空间

varchar：存变化长度的字符,占用实际字符空间+1,没有空间浪费,会用1-2字节存长度

text：存变化长度大数据，没有空间浪费，比前两者没有默认值

varchar和text都能存长度65500左右的字符，尽量用varchar，因为有默认值

**MySQL 执行计划：**用explain语法来进行查询分析 + navicat工具

1. SQL语句**执行顺序**/执行过程：

from--->join, on--->where--->group by, having--->select, distinct--->order by, limit

**2. SQL一次执行的过程**：

1.**发SQL语句:**客户端和MySQL服务器建立TCP连接后发送一条SQL给MySQL服务器(同 一时刻只有一方能发数据，同时，另一方要么完整地接受数据，要么直接断开连接)

2.**检查查询缓存**：MySQL检查SQL语句是不是Select型，是的话去检查查询缓存，缓存 命中就返回结果否则去数据库找

3.**生成执行计划**：MySQL服务器解析SQL+预处理(看SQL语句有没有错)，然后优化器会 预测几个执行计划(指令树)的成本(根据表和索引页数)，选成本最小的那个，不考虑缓 存，假设读取都是读磁盘

4.**执行**：MySQL遍历指令树 + 调用存储引擎API来执行查询

5.**返回结果**：MySQL把结果返回给客户端(逐步返回)—结果集每行都会用通信协议封包， 然后再通过TCP传输

**SQL注入**：把用户输入数据作为代码执行

1.条件：1.首先用户能控制输入；2.然后是原本程序要执行的代码拼接了用户输入的数据， 把数据当代码执行了

2.类型：1.没有正确过滤的字符2.数据库服务器的漏洞3.盲目SQL注入式攻击4.条件性差错5.时间延误

3.注入点类型：1. 数字型2.字符型3.搜索型

4.原因：①查询集处理，转义字符处理，多个提交的处理和错误处理不恰当；②[数据](https://link.jianshu.com/?t=https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE/5947370" \t "_blank)库配 置不安全

5.防御：

1. 用预编译的SQL语句，SQL的语意不会变化，攻击者就不能改变SQL的结构了(比如攻击者插入了类似’or ‘1’=’1的字符串，也只会把这个字符串作为username查询)

2.先把SQL语句定义在数据库里，从存储过程来防御。(存储过程中可能也存在注入问 题，应该尽量避免在存储过程中使用动态SQL语句)

3.限制数据类型，统一数据的格式

4.开发时候尽量用安全函数代替不安全函数，写安全的代码(危险函数C里的system())

5.避免高权限用户直接连接数据库

EX：

select \* from AUTH\_USER t where t.LOG\_IN\_NAME='' or 1=1 --' and  t.PASSWORD='gdd'

**SQL优化**：1.避免全表扫描，首先考虑在where和order by涉及的列上建索引

2.避免在where子句中进行null判断和!=<>，or, in, not in

3.尽可能用varchar代表char，这样能节省存储空间

**大表优化**：1.限定数据范围(禁止那些不带限制数据范围条件的查询语句)2.读写分离(主库负责写，从库负责读)3.垂直分区(根据表的相关性拆分，也就是把数据库列拆分)4.水平分区(表结构不变，把数据分散到不同的表和库，来达到分布式目的)—最好分库因为分表只是解决单一表数据多的问题，表的数据还是在同一台机器上，不能提升Mysql并发能力

**怎么知道使用/命中了索引**：在查询语句前加**Explain**，在结果里看key列



type列， 从上到下是从坏到好，一般是要大于range



1. **主键**唯一标识一条数据 不能重复，保证数据完整 一种特殊的唯一索引

**外键**用来和其他表建立联系 可以重复，是另一个表的主键

1. **索引**(B+树)：一种数据结构，能用它来加快查找数据库数据

**没有索引的话**，数据按顺序一条条在磁盘上存储(加了主键后，比如自增主键，数据在磁盘变成了平衡树结构，也就是聚簇索引(表排列顺序和索引顺序一致))

索引**优点**：1.创建唯一性索引能保证数据库表中**每一行数据的唯一性**

2.可能可以加快数据检索和**表与表连接速度**

索引**缺点**：1. 会**减慢写入速度+增删改慢**(因为每次写入时都要更新索引)

2.创建索引和维护索引要**消耗时间**，尤其数据量大的时候

3.**占**物理**空间**和数据空间(数据表)—磁盘空间

索引**使用场景/原则**：1.加在经常需要搜索的列上，不要加在写入多和读取少的列**，**因为会先访问索引, 每次写入时都要更新索引，执行速度会变慢

2.数据多+字段值相同的时候用唯一索引(支持NULL)

3.字段多+没重复的时候用复合索引(不支持NULL)

4.用唯一性索引(为经常要查询的建索引，限制索引数目)

5.尽量用最左前缀匹配原则，因为如果索引的值很长，比如text和blog类型的字段，来全文检索的话会浪费时间，如果只检索字段前面的几个字符，能提高索引检索速度

6.where group by order by的子句要建索引 count max也要

**什么时候不用：**1.要取表里所有记录2.对不是唯一而且重复的字段，比如姓名3.经常修改和删除的字段4.记录比较少的表

**索引类型：**

1. **单列索引**：只有一个字段的索引 2.**复合/联合索引**：包含两个或以上字段的索引

复合索引只要第一个列名对应后面就都能查到，中间Y不写也没事

CREATE INDEX 索引名 ON 表名(列名X, 列名Y, 列名Z);

实际生成了3个索引：

1. 单列索引(列X）2.复合索引(列X, 列Y) 3.**复合索引(列X，列Y，列Z）**

**所以执行X = 1 and Z = 3不会生效，因为最左匹配原则，先按X排序，再按Y，**

**查XZ只能用到X，不能用到Z。 执行Y = 1 and Z = 2的话YZ都没用到因为X没用**

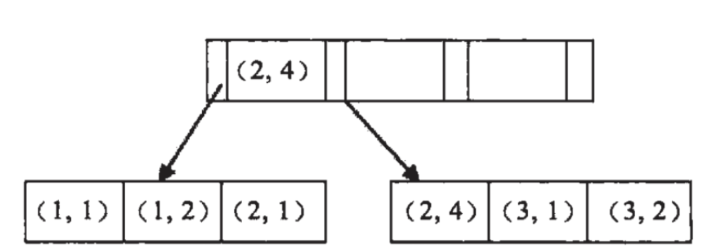
**索引（a,b,c）。 可以支持a | a,b| a,b,c**

**（1）select \* from myTest where c=4 and b=6 and a=3; where里面的条件顺序在查询之前会被mysql自动优化成abc，所以索引都用到了**

**（2）select \* from myTest where a>4 and b=7 and c=9;  
 a用到了 b和c没用，因为a用了范围所以相当于断点，之后的b，c都没有用到索引）**

**(3) where a > 0 and b = 1的过程：**先匹配a，再通过叶子节点链表取出数据过滤b

**失效是因为**b值是无序的。此时是一个无序的b+树，就找不到需要的b这个值，这种情况下就用不到索引，只能进行全表扫描

****

**最左匹配原则解释：**这里a值是有序的：112233，b是无序的：121412，所以b=2这个查询无法利用索引，因为联合索引先按照a排序，b是无序的。在a=1, b=2这个查询，在a=1的时候b是有序的，所以最左匹配原则不会停止，但是如果是a>1, b=2这种范围查询，a可以匹配上，但b不可以，因为a是一个范围，在这个范围里b是无序的。

**3.唯一索引**：表上一个或者多个字段组合起来建立的索引

性质：1.索引列的值要唯一，如果是组合索引，那么列值的组合要唯一 2.唯一索引允许有多个，一般只有一个 3.允许null

**4.主键索引**：一种特殊的唯一索引。⼀般是在建表的时候指定了主键就会创建主键索引。 和唯一索引的区别：不允许null 可有可无的

**5.聚簇索引—InnoDB用**

1.包含主键索引和对应的数据

2.主键顺序是数据的物理存储顺序，叶子节点是数据节点

3.因为真实数据的物理顺序只有一种，所以一个表最多一个聚簇索引

**缺点**：如果一个表没有聚簇索引，创建索引的时候就会对数据重新进行排序，所以它对表的修改速度比较慢，所以不建议在经常更新的列建这个索引

**优点**：性能好，因为找到第一个索引值，有连续索引值的记录也会在他后面，因为物理上顺序就是在他后面

**什么时候用**：按照表最常用的SQL查询方式来选字段作为聚簇索引

**聚簇索引默认是主键**，如果表里没有定义主键，InnoDB会选一个没有null值的列代替，如果没有这样的索引，InnoDB会隐式地定义一个主键作为聚簇索引

**聚簇索引vs唯一索引**：聚簇索引的索引值没有被要求是唯一的，就是在有聚簇索引的列上可以插多个相同的值，这些值在硬盘上的物理排序和在聚簇索引的排序一样

**非聚簇索引—MyISAM用 my zeim**

数据和索引存在不同地方，叶节点仍然是索引节点(存的是主键值)而且是有指向对应数据块的指针

**具体怎么用**：Mysiam用key\_buffer把索引先缓存到内存，通过索引访问数据的时候，在内存里直接搜索索引，然后通过索引找到磁盘里对应的数据，如果索引不在key\_buffer命中时，速度就会慢

**聚簇vs非聚簇**：聚簇索引能直接查到数据，非聚簇索引先查到主键值，再用主键值查数据。非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引一样，辅助索引叶子节点存储的不是行的物理位置，而是主键值

**覆盖索引**：不用聚簇索引就能直接查到数据，前提是符合最左侧匹配原则

例子：建立索引createindex index\_birthday on user\_info(birthday);//查询生日在1991年11月1日出生用户的用户名select user\_name from user\_info where birthday ='1991-11-1'

1.通过非聚集索引index\_birthday查找birthday等于1991-11-1主键值

2.用主键执行聚集索引找到数据（数据行）存储的位置

3.从得到的真实数据中取得user\_name字段的值

**覆盖索引**只是特定于具体select语录而言的**联合索引**。 也就是说一个联合索引对于某个select语句，通过索引可以直接获取查询结果，而不再需要回表查询，就称这个联合索引覆盖了这条select语句

**索引失效：**1.列与列对比 2.条件里有or 3.like查询以%开头(比如“%a”,"%a%"：这个时候 不是前缀法则了，它是去查找以a为结尾的字符串，这个是没有顺序的。在 没有顺序的前提下去二叉树上查找数据，肯定是查找不到的就用不到这个索引)

4.索引列没有限制not null 5.如果列类型是字符串，没有在条件中 把数据用引号引用起来

6.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

**创建索引**：1.普通索引CREATE INDEX indexName ON mytable(username(length))

ALTER TABLE tbl\_name ADD INDEX index\_name (column\_list)

2.唯一索引

CREATE UNIQUE INDEX indexName ON mytable(username(length))

3.主键索引 添加PRIMARY KEY  
ALTER TABLE `table\_name` ADD PRIMARY KEY ( `column` )

4.在创建表的时候加

**字段区分度**：为字段创建[索引](https://so.csdn.net/so/search?q=%E7%B4%A2%E5%BC%95&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_38106322/article/details/_blank)时，但是字段太长，为整个字段创建索引的话，太浪费存储空间了，所以需要计算出字段区分度，选择合适的索引长度

计算字段区分度的公式

1647354886(1)

**索引下推(ICP)**:在搜索引擎中提前判断对应的搜索条件是否满足，满足了再去回表，通过减少回表次数进而提高查询效率。

**主键随机插入的缺点：**

1. 写⼊的⽬标页可能不在内存缓存区，那么插⼊记录的时候需要先从磁盘读取⽬标页到内存中。这会导致⼤量的随机IO.如果是顺序插

⼊，由于是插⼊到上⼀个记录的后⾯，则⼤多数情况下（不需要开辟新页的情况）磁盘页是已经加载到内存了的。

2. 因为写⼊是乱序的，InnoDB可能需要不断的的做页分裂操作，以便为新的⾏分配空间。⽽页分裂会导致移动⼤量的数据，⽽且⼀次分

裂⾄少要修改三个页⽽不是⼀个页。

3.由于频繁的分页，页⾯会变得稀疏并被不规则的填充，最后会导致数据碎⽚。

**不按顺序插入会造成性能低 因为B+树中间满叶子节点所以肯定会分裂**

**MySQL和MongoDB**

**关系型数据库 vs 非关系型数据库**

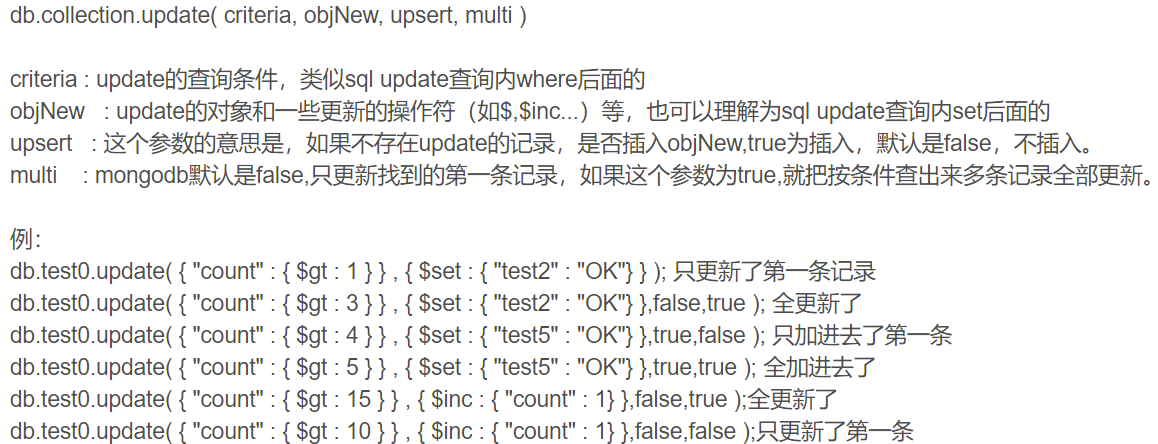
MySQL：关系型数据库 B+索引和哈希索引(memory引擎支持，但是重启后数据会丢失)

MongoDB：非关系型数据库(NoSQL), 用的bson格式--适合文档存储和查询

语句：

db.collection.find(); db.collection.distinct("name"); db.collection.find({"age": 22});

db.collection.find({ "field" : { $gte: value } } ); // greater than or equal to : field >= value



删除$pull : { field : value } } $表示自己 insert表示加

MySQL: 坏处：用表存数据在磁盘(读取比较慢) 定义表和字段才能存

好处：有Join来进行复杂查询，可以事务处理，可以来保证数据的一致性

NoSQL(redis): 坏处：没有join，不支持事务，不提供附加功能，比如报表

好处：基于文档设计，把数据放内存，比在磁盘读取快，满足高并发

即使放内存，也比SQL简单，因为它是半结构化的数据格式，MySQL的存储是经过结构化、多范式有很多复杂规则的数据，还原内存结构慢

Redis：缓存数据库，用于存储使用频繁的数据到缓存，读取速度快

而MySQL用于持久化地存数据到硬盘，速度慢

Redis是一个key-value存储系统，value有很多类型而且有add/remove, push/pop操作，都是原子性的

**数据怎么存放？**数据被放在Mysql设计的**数据页**上，数据页上数据是**一行行**的，格式是**compact**，每行都有**行描述**和**指针**去指下一行

**行式存储(MySQL) vs 列式存储(Hbase)**

1.行式存储倾向于结构固定，列式存储倾向于结构弱化

2.行式存储存储一行数据只要一个主键，列式存储要多个主键

3.行式存储要维护大量索引和物化视图，所以在时间(处理)和空间(存储)上成本都很高

列式存储只访问查询涉及的列，能很大地降低系统I/O

**Mysql存储引擎：**

**1. InnoDB**：用行锁，是默认的引擎，**支持事务**但损失效率,有乐观锁和悲观锁

不是锁记录，**是锁索引**，索引分为**主键索引和非主键索引**两种，如果一条sql 语 句操作了主键索引，Mysql 就会锁定这条主键索引；如果一条语句操作了非主 键索引，MySQL会先锁定这个非主键索引，再锁定相关的主键索引。

**如果没有索引**，InnoDB 会通过隐藏的聚簇索引来对记录加锁

如果**不通过索引条件检索数据**，InnoDB会将对表中所有数据加锁，因为没有 了索引，找到某一条记录就得扫描全表，也就得锁定表，跟表锁一样

**底层是B+树**

**2.MyISAM** :都用表锁, 更一条记录就锁整个表, 性能低并发不高, 但不会发生死锁.

每个MyISAM在磁盘上存3个文件fm(存表定义的数据)MYD(存表具体记录的数据) MYI(存索引)表类型是ISAM，**不支持事务**但存储更快(允许读写有错误数据的时候用)

3.Memory 把数据存在内存来提高数据的访问速度，每个表和一个fm文件关联

用哈希索引

**数据库事务**：DBMS执行过程中的一个**逻辑单位**，由一个数据库**操作序列**组成

**目的**：1.提供操作序列从失败到恢复正常的**方法**，同时让数据库在异常情况下仍然能**保持一致性**

1. 多个应用程序并发访问数据库的时候**提供一个隔离，**让程序的操作不会干扰到别的操作

事务**处理命令**: **初始化**(start)**回滚**(rollback)**提交**(commit)**设保存点**(savepoint identifier)

**4个特性**：

**1.原子性**(所有操作都是不可分割的，要么全部完成要么全部不执行，在执行过程 中发生错误会回滚到事务开始前的状态)

**实现：**由**回滚日志**记录事务操作来回滚，比如insert就delete

**2.一致性**(几个并行执行的事务，他们执行结果必须与按某一顺序串行执行的结果 相一致) **实现：**由原子性，隔离性，持久性实现，回滚日志

**3.隔离性**(事务的执行不受其他事务的干扰，事务执行的中间结果对其他事务必须 是透明的，也就是事务提交之前对其他事务不可见不会有多个事务并发执行 时由于交叉执行导致数据不一致)

**实现**：**悲观锁**(操作就加锁)和**乐观锁**(不同的事务可以同时看到同⼀对象(⼀ 般是数据行)的不同历史版本。如果有两个事务同时修改同⼀数据行，那么在 较晚事务提交时检测冲突。有2种实现：1.用回滚日志来获取数据行的历史 版本，2.在内存中保存同⼀数据行的多个历史版本，通过时间戳来区分)

**4.持久性**(事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失) **实现：**用重做日志redo log, 只保存事务对数据页做的修改。当一条记录需要 更新Innodb会先把这个数据页的修改写到redo log buffer中，还会在redo log记录这次操作，事务提交时再把redo log buffer刷到磁盘文件redo log中(刷盘)，这样是追加写，顺序IO速度很快, 如果MySQL宕机，重启时可以读取redo log中的数据，对数据库进行恢复。不像之前实现持久性是把事务在内存中修改的全部页面刷新到磁盘，但这些页面可能并不相邻，就会导致随机IO就很慢

redo log采用的是WAL（Write-ahead logging，预写式日志），所有修改先写入日志，再更新到Buffer Pool，保证了数据不会因MySQL宕机而丢失

**1.什么时候更新磁盘的数据页：**MySQL会根据redo log自动在系统空闲时完成更新磁盘数据页的操作, 这个过程也叫做刷盘, 会对MySQL性能有一定影响

**2.不及时刷到硬盘上会导致下次读出来的还是旧数据吗：**不会, 记到redo log buffer的时候, 这个数据页的内容已经读到内存了, 内存里面的这个数据页的内容已经被更新了, 以后都是都内存中的这个最新版本

**3.只用内存里的数据那宕机了咋办：**对数据页的修改都记录在redo log了, 宕机了根据redo log的内容恢复就好, 默认设置下事务只要提交就会持久化到磁盘文件redo log中

2、既然redo log也需要在事务提交时将日志写入磁盘，为什么它比直接将Buffer Pool中修改的数据写入磁盘(即刷脏)要快呢？  
（1）刷脏是随机IO，因为每次修改的数据位置随机，但写redo log是追加操作，属于顺序IO。  
（2）刷脏是以数据页（Page）为单位的，MySQL默认页大小是16KB，一个Page上一个小修改都要整页写入；而redo log中只包含真正需要写入的部分，无效IO大大减少。

**4种隔离级别以及InnoDB实现**

不同的锁策略(Locking Strategy)来实现不同的隔离级别。

**一．读未提交(Read Uncommitted)**

**定义：所有事务都可以读其他还没提交的事务提交的修改**

select不加锁。可能读取到不一致的数据(读脏) **性质：**并发最高，一致性最差的隔离级别

**二．串行化(Serializable)**

**定义：一个事务只能读到别的事务已经提交的内容**

select会被隐式的转化为select … in share mode.这可能导致如果有未提交的事务正在修改某些行，所有读取这些行的select都会被阻塞住。**性质：**一致性最好并发性最差的隔离级别

**大数据量，高并发量的场景下，几乎不会使用上述两种隔离级别**。

1. **可重复读(Repeated Read, RR)**

**定义：(MySQL的等级)确保一个事务的多个实例并发读取数据时候会看到相同的数据行**

InnoDB默认的隔离级别，在RR下：

1. 普通的select用快照读(一种不加锁的一致性读)，底层用MVCC实现
2. 加锁的select(select … in share mode / select … for update), update, delete等语句有两种情况： 1.**在唯一索引上使用唯一的查询条件，会使用行锁/记录锁(record lock)**，而不会锁记录 之间的间隔，也就是不会用间隙锁(gap lock)与临键锁(next-key lock)

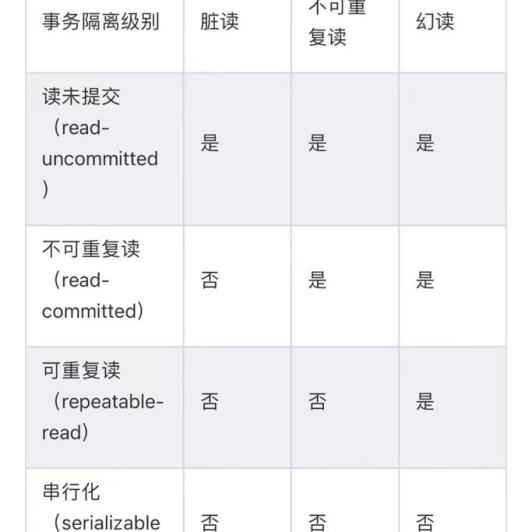
2.**范围查询条件，会使用间隙锁与临键锁**，锁住索引记录之间的范围，避免范围间插入 记录来避免产生幻行记录，尽量避免不可重复的读

**四.读已提交(Read Committed, RC)**

**定义：事务一个个排队执行，只有事务提交后其他事务才能从看数据变化，读每一行都要加锁) 执行效率很差，可能导致锁竞争**

互联网最常用的隔离级别，在RC下：1.普通读是快照读；2.加锁的select, update, delete等语句除了在外键约束检查和重复键检查时会封锁区间，其他时候都只使用行锁/记录锁；

此时，其他事务的插入依然可以执行，就可能幻读



提交读(Read Committed, RC)

**3种需要阻止的现象：**

**脏读**：一个事务**读到**别的事务修改但还**没提交的数据,** 未提交意味着这些数据可能会回滚，也就是可能最终不会存到数据库中，也就是不存在的数据

**不可重复读**：在一个事务内，最开始读到的数据和事务结束前的任意时刻读到的同一批数据出现不一致的情况, 可能期间别的事务做修改了

**幻读**：事务A在按查询条件读取某个范围的记录时，事务B又在该范围内插入或删除了新的满足条件的记录，当事务A再次按条件查询记录时， 结果集不一样，可能会产生新的满足条件的记录(幻行)

**MySQL解决幻读**：

1. **在快照读的情况下通过MVCC解决**

保存了数据资源在不同时间点上的多个快照。根据事务开始的时间不同，每个事务看到的数据快照版本是不⼀样的。用版本号实现

1. **在当前读的情况下通过next-key lock来避免幻读**

next-key lock = 记录锁 + 间隙锁(gap lock)把当前数据行和上下两行的间隙锁定

查哪些数据行就锁住这些行，保证这个范围内读取的数据是一致的，但并不 完全等于串行化的隔离级别，单纯的行锁不能解决幻读

**行锁 =** 记录锁 / 间隙锁 / next-key lock

**记录锁：**对索引记录加锁，如果事务A对id=1加行锁事务B就不能加了 **间隙锁：**采用范围条件查询数据时，InnoDB 会锁住这整个区间，比如id有1357四条 数据，查1-7，则1-7都会加锁，246没有数据是间隙。不管范围内是不是数据都 锁定这样会影响性能。索引会形成一个个**左开右闭**的一个区间

**两者使用情况：1.**只有在可重复读的隔离级别下才有间隙锁，读提交级别没有间隙锁，只 有行锁 **2.** 使用唯一索引来搜索唯一行的语句不用间隙锁定。(主键索引或者唯一 索引是否用Gap锁分情况：(1)如果where条件全部命中不用Gap锁只加行锁；(2) 如果where条件部分命中或者全不命中就会加Gap锁)

**3**. 搜索条件里有多个查询条件(即使每个列都有唯一索引)也会有间隙锁

**快照读**：读取的是记录的可见版本是一致的但有可能是历史版本，不用加锁。

简单的select操作，不加锁。实现：undolog和多版本并发控制MVCC

**当前读：**读取的是最新版本的数据并且对读的记录加锁，阻塞其他事务同时改动相 同记录。特殊的读操作(插⼊/更新/删除)，属于当前读, 也要加锁

EX：加S锁(共享锁): select \* from table where ? lock in share mode;

加X锁 (排它锁)。1.select \* from table where ? for update; (可以锁表/锁行)

2.insert into table values…;update table set ? where ?;delete from table where ?;

select \* from table where ? for update; 锁表/锁行的情况：



**MVCC**：多版本控制器，基于锁的并发控制，好处是读不加锁，读写不冲突增加系统并发性能。而且保存了数据资源在不同时间点上的多个快照。根据事务开始的时间不同，每个事务看到的数据快照版本是不⼀样的。

**InnoDB中的MVCC实现**：存储引擎全局维护⼀个系统版本号，每开启⼀个新的事务，这个系统版本号就会递增。事务开始的系统版本号会作为这个事务本身的版本号。同时给每行加两个隐藏字段，分别是这行创建和删除的版本。

**在可重复读隔离级别下，MVCC的操作：**

**insert**：InnoDB把新插的行的创建版本号设为当前系统的版本号

**select**：要满足两个条件，Innodb才会返回当前行的数据：

1.记录行创建版本号 <= 当前版本号保证在select操作之前这条记录已经存在， 事务开始之后的数据不会被读到

2.记录行的删除版本号 > 当前版本或者为空，也就是当前事务开始的时候，这条记录还没有过期。在事务开始之前就已经过期的数据行，这个事务也不会看到。大于意味着有一个并发事务把这行删除了

**delete**：把要删除的行的删除版本号设为当前系统的版本号

**update**：不执行原地update，而是转换成insert + delete。把旧行的删除版本号设为当前版本号，把新行插入然后同时把创建版本号设为当前版本号

**同时，写操作（insert、delete和update)执行时，要把系统版本号递增**

**MVCC只在READ COMMITTED和REPEATABLE READ这两个级别下⼯作**

**MVCC读操作可以分成两类：快照读 (snapshot read)与当前读 (current read)**

**数据库的锁：悲观锁**(数据库的增删改操作默认都会加排他锁，而查询不会加任何锁)

**类型**：1.读锁/共享/**乐观锁**(被锁定的对象只允许被读)有两个步骤：冲突检测和数据更新

事务A对一个数据上锁，A只能读不能写，其他事务只能给这个数据加读锁，除非A释放锁

多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新成功，失败的线程并不会被挂起，并可以再次尝试

**做法：**在表中增加一个version字段，操作前先查询当前版本信息，在数据提交时检查version字段是否被修改。

select \* from table lock in share mode

MVCC和CAS都是乐观锁的一种实现

CAS(compare and swap)：乐观锁的一种实现机制，实现线程安全

思想：包含三个操作数——内存位置（V）、预期原值（A）和新值（B）：当一个线程需要修改一个共享变量的值，完成这个操作需要先取出共享变量的值，赋给A，基于A进行计算，得到新值B，再用预期原值A和内存中的共享变量值比较，如果相同就认为其他线程没有进行修改，然后把新值写入内存

**CAS缺点**：1.CPU开销大：并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，又因为自旋的时候会一直占用CPU，如果CAS一直更新不成功就会一直占用，造成CPU的浪费

2.不能保证代码块的原子性：CAS机制所保证的只是一个变量的原子性操作，比如要保证3个变量共同进行原子性更新就要用synchronized

3.ABA问题



**ABA解决:**

1.使用版本号，也就是在变量前面追加版本号，每次变量更新时把版本号加1顺序递增，那么A-B-A就会变成1A-2B-3A。

2.从jdk1.5开始，jdk中的Atomic包里提供了一个类AtomicStampedReference来支持原子条件的更新。这个类的compareAndSet方法的作用首先检查当前引用和标志是不是等于预期的引用和标志，如果都相等，就以原子方式把这个引用和标志的值设为给定的更新值。

AtomicMarkableReference更新(对象引用, 布尔值)二元组，通过这种二元组把节点保存在链表中同时把他们标记为已删除节点

2.写锁/排他锁/**悲观锁:**认为数据在被修改时一定会存在并发问题，一个事务对一个数据加锁，其他事务不能读也不能加锁，在整个数据处理过程中将数据锁定

select \* from table for update 执行后，会在表上加行锁，到事务提交后才解除

行锁、表锁、排他锁都是悲观锁

**表级锁**：是系统开销最低但并发性最低的一个锁策略，操作对象是数据库

注意：给表加锁要获得所有涉及到表的锁 LOCK TABLE

InnoDB和MySIAM支持 不会产生死锁

**什么时候锁住整个表？** 1.磁盘满的时候2.insert时候 3.如果对用InnoDB的表使用行锁，被锁的字段不是主键，而且也没有对它建索引的话，行锁锁的是整张表

**行级锁**：操作的对象是表里的一行，开销大 并发性好 SELECT…LOCK IN

InnoDB支持

**触发条件**：InnoDB行锁通过索引上的索引项实现，**只有通过索引条件检索数据，InnoDB才会使用行级锁，否则，InnoDB将使用表锁**  
在MySQL中，行级锁不是直接锁记录，而是**锁索引**。索引分为主键索引和非主键索引两种，如果一条[sql语句](https://so.csdn.net/so/search?q=sql%E8%AF%AD%E5%8F%A5&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/hukuncd/article/details/_blank)操作了主键索引，MySQL就会锁定这条主键索引；如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引。 在UPDATE、DELETE操作时，MySQL不仅锁定WHERE条件扫描过的所有索引记录，而且会锁定相邻的键值，也就是next-key lock。  
**由于InnoDB预设是行锁，所以只有明确指定主键，MySQL才会执行Row lock (只锁住被选取的资料例) ，否则MySQL将会执行表锁 (将整个资料表单给锁住)：WHERE id<>‘3’ id like ‘3’ 会直接锁表**

**页面锁**：开销和加锁时间和锁定粒度介于表锁和行锁之间会，出现死锁，并发度也一般

**MVCC**(多版本并发控制)：处理并发 开销最大 处理能力最强

**死锁**：多个资源并发的时候产生。表级锁不会产生死锁

**死锁原因**：**1.事务对资源访问顺序的交替导致**(一个用户A 访问表A(锁住了表A)，然后又访问表B；另外一个用户B 访问表B(锁住了表B)，然后想访问表A；这样会导致用户A因为用户B已经锁住了表B，它就必须等用户B释放表B才能继续，然后用户B要等用户A释放表A才能继续，这样就产生死锁了) 解决：调整程序的逻辑

**2.并发修改同一个记录**(用户A查一条纪录，然后修改这条纪录；同时用户B也想修改这条纪录，这时用户A的事务里锁的性质从查询的共享锁想上升到独占锁，但用户B里的独占锁因为A有共享锁所以必须等A释放掉共享锁，而A由于B的独占锁不能上升独占锁也就不会去释放共享锁，这样就出现了死锁)

解决：乐观锁(实现写-写并发)和悲观锁(保证操作的独占性)

**3. 索引不当**(2个情况：1.在事务里执行一条不满足条件的语句导致了全表扫描，把行级锁上升为表级锁，然后多个这样的事务执行后，就很容易产生死锁2.表中的数据量很大而且索引建的太少或者建的不合适，这样就会经常有全表扫描，这样之后应用系统会越来越慢，最后导致死锁)

解决：SQL语句中不使用太复杂的关联多表的查询

**避免死锁**：1.以固定顺序访问表和行2. 大事务拆小3.一个事务中一次锁定所有资源4.降低隔离等级

InnoDB有自动检查死锁的功能，会自动解决

**InnoDB的锁：**

1，Record Lock：单个行上的锁

2，Gap Lock：间隙锁，锁定一个范围，但不包括记录本身

3，Next-Key Lock：1+2，锁定一个范围，而且锁定记录本身。

对于行的查询，都是用这个方法，主要是为了**解决幻读**的问题

**InnoDB的行级锁：**

共享锁(Shared Lock, S锁): 在事务需要读取一条记录时，需要先获取改记录的S锁

排他锁(Exclusive Lock, X锁): 在事务要改动一条记录时，需要先获取该记录的X锁

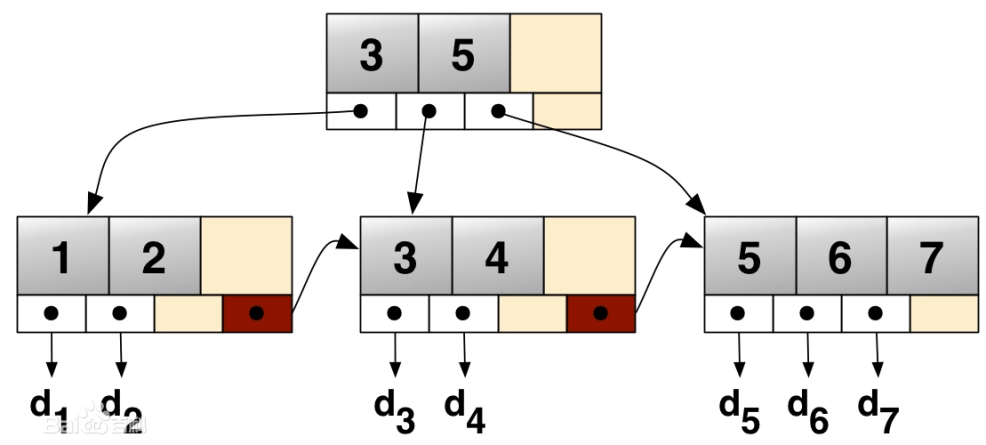
**InnoDB的锁是针对索引。对行记录加锁，如果存在加X锁，否则加 next-key lock/间隙锁**

**B+树**：层数一般控制在3-5层(IO次数就是树的高度)，主要为了避免磁盘的读取次数太多，然后它的数据都在叶子节点，并且叶子节点之间用指针连接，这样区间遍历和查询会更快，然后新的值可以插在已经有的节点里，不用改变树的高度，这样就能减少重新平衡和数据迁移的次数，而且中间节点只保存索引不保存数据所以能容纳更多节点元素

(1)每个结点至多有m个子女；

(2)除根结点外，每个结点至少有[m/2]个子女，根结点至少有两个子女

(3)有k个子女的结点必有k个关键字。



**B树**：每个节点都有key/value, 所以经常访问的节点可能离根节点更近, 访问更快

**vs B+树：每一层要递归遍历，而且相邻元素可能在内存里不相邻，缓存命中率没有B+树好，所以B+树更适合文件系统**

**缺点：**  
1、每个节点都有key和data，但是每个页的存储**空间是有限**的，如果data比较大的话会导致每个节点存储的key数量变小  
2、存储的数据量很大的时候会让深度变大，查询时候磁盘io次数就多了，这样就会影响查询性能

B+树的查找与B树不同，当索引部分某个结点的关键字与所查的关键字相等时，并不停止查找，应继续沿着这个关键字左边的指针向下，一直查到该关键字所在的叶子结点为止。 [2]

**应用：文件系统和数据库索引**

**红黑树特点**：根节点和叶子节点都是黑色，而且从任一节点到其叶节点黑节点数 量一样，二叉树查询复杂度可能是O(n)--节点都在一边

**应用：**java的treeMap实现 + IO多路复用epoll的实现 -> 支持快速增删改查

**B+/B树(**平衡多路查找树**) vs 红黑树：**

在同一个节点，B/B+树的高度会远小于红黑树，红黑树IO次数就更多了

**[红黑树](https://www.nowcoder.com/jump/super-jump/word?word=%E7%BA%A2%E9%BB%91%E6%A0%91" \t "_blank)**的索引时间复杂度稳定在O(logn)，但是读硬盘读取的时候，会导致读取的数据较多（磁盘读取的基本单位是扇区），根据空间局部性原理，需要读的数据的周围数据也可能读取到，所以使用B+树更好。

**AVL树 vs 红黑树：**控制在O(logn)，但左右子树高度差不能超过1，所以经常要调整，但红黑树在插入和删除时不会频繁破坏红黑树的规则，不用频繁调整节点位置

应用：Windows NT内核

其他数据库索引：哈希索引(查得快，只要哈希就定位了但不支持部分索引列查询)

**数据库三范式**(为了设计冗余较小、结构合理的数据库)

**1. 第一范式**：要保证所有**字段**都是**不可分解的原子值**，也就是有原子性

比如地址列是浙江省杭州市慧芝湖花园，经常访问地址列的话就要把地址这列分成省份 城市 详细地址三列

**2. 第二范式**：要保证**每列都和主键有关**，**一个表只能存一种数据**

比如分开订单编号表和商品编号表

**3.第三范式**：要保证**每列都和主键直接相关**，不是间接相关，然后是用外键连接不同表去得到别的表的信息

**Mysql事务分类：**

1.**扁平事务**(最简单但最频繁)：begin work开始到commit work/rollback work结束

操作要么都执行要么都回滚, 所以是**有原子性的应用程序的基本模块**

**缺点**：因为某些地方出错但不会导致所有操作无效的时候还是要回滚之前所有操作, 代 价太大，因为它只被隐式地设置了一个保存点

**2.带有保存点的扁平事**务：支持扁平事务的操作+允许事务执行的时候回滚到事务较早的状 态，保存点用来通知事务系统应该记住事务当前状态

**3.链事务**：是保存点事务的变种，在提交事务的时候会释放不需要的对象，把上下文传给 下个要开始的事务，这个是一个原子操作。而保存点事务的保存点在系统崩溃的时候 会丢失，系统恢复后事务要重新执行而不是从最近保存点继续执行

**区别：**保存点事务能回滚到任意保存点但链事务只能回滚到当前保存点

**4.嵌套事务**：是一个层次结构框架，有一个顶层事务控制各个层次的子事务

父事务回滚, 子事务也会被回滚—子事务具有ACI特性但没有D特性

**5.分布式事务**：是分布式环境下运行的扁平事务

InnoDB不支持分布式事务，只支持前4个，但它支持XA事务，通过XA来实现分布式事务，而且这个时候隔离等级必须是串行化

Oracle，SQL server都支持XA事务---由1.资源管理器(提供事务访问方法)(通常一个数据库就是一个资源管理器)2.事务管理器(协调各个事务)3.应用程序(定义事务边界)

**Mysql日志分类**：

1.事务日志包括(1.**重做日志**(redo log)：记录事务执行后的状态，来恢复没写进文件的但已 经执行成功的事务它更新的数据 保证事务原子性和持久性

2.**回滚日志**(undo log)：保存事务发生前的数据，之后可以用于回滚，还提 供mvcc(多版本并发控制下的读) 保证事务一致性+数据的原子性

重做日志**vs**回滚日志：回滚日志只是把数据从逻辑上恢复到事务开始前的状态而 不是从物理页面上操作实现

**2.错误日志**：记录mysql的启动和停止，还有服务器运行过程中发生的错误

**3.普通查询日志**：记录了服务器接收到每个查询和命令

**4.慢查询日志**：记录执行时间过长和没有用索引的查询语句

**5.二进制日志(binlog)**：用于记录数据库执行的写入性操作(不包括查询)信息，以二进制形式保存在磁盘中。是mysql的逻辑日志(记录的就是sql语句,物理日志记录的是数据页变更)

**使用场景**：1.主从复制：在Master端开启binlog，然后将binlog发送到各个Slave端，Slave 端重放binlog从而达到主从数据一致。

2.数据恢复：也就是通过使用mysqlbinlog工具来恢复数据。

**三种格式**：1.STATEMENT(基于SQL语句的复制,每一条会修改数据的sql语句会记录到 binlog中) 好处：不用记录每行变化,减少日志量，节约IO, 从而提高了性能

坏处：一些情况比如sleep()会导致主从数据不一致

2.ROW(基于行的复制,不记录每条sql语句的上下文信息，只记录哪条数据被修改了。

好处：不会出现某些特定情况下的存储过程、或function、或trigger的调用和 触发无法被正确复制的问题；

缺点：会产生大量的日志，尤其是alter table的时候会让日志暴涨

3.MIXED(用STATMENT和ROW两种模式的混合复制，一般的复制用 STATEMENT模式保存binlog,对于STATEMENT无法复制的操作用ROW模式保存)

**重做日志redo log vs binlog**

（1）作用不同：redo log是用于crash recovery的，保证MySQL宕机也不会影响持久性；binlog是用于point-in-time recovery的，保证服务器可以基于时间点恢复数据，此外binlog还用于主从复制。  
（2）层次不同：redo log是InnoDB存储引擎实现的，而binlog是MySQL的服务器层(可以参考文章前面对MySQL逻辑架构的介绍)实现的，同时支持InnoDB和其他存储引擎。  
（3）内容不同：redo log是物理日志，内容基于磁盘的Page；binlog的内容是二进制的，根据binlog\_format参数的不同，可能基于sql语句、基于数据本身或者二者的混合。  
（4）写入时机不同：binlog在事务提交时写入；redo log的写入时机相对多元：

前面曾提到：当事务提交时会调用fsync对redo log进行刷盘；这是默认情况下的策略，修innodb\_flush\_log\_at\_trx\_commit参数可以改变该策略，但事务的持久性将无法保证。

除了事务提交时，还有其他刷盘时机：如master thread每秒刷盘一次redo log等，这样的好处是不一定要等到commit时刷盘，commit速度大大加快。

**分布式**是指通过网络连接的几个组件，然后通过交换信息协作这样 形成的系统

**分布式的CAP定理**：Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），最多只能同时三个特性中的两个，三者不可兼得

C：更新操作成功并返回客户端后，所有节点在同一时间的数据完全一致

A：服务一直可用，而且是正常响应时间

P：分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性或可用性的服务

**集群**的话是同一种组件的多个实例形成的逻辑上的整体

**区别**：多个不同组件构成的系统就是分布式系统而不是集群

是集群不是分布式系统的情况的话，比如多个经过负载均衡的HTTP服务器，它们之间不会互相通信，如果不带上负载均衡的部分的话，一般不叫做分布式系统

**微服务**是一种架构模式，它就是把单一应用程序划分成一组小的服务，然后服务之间互相配合来给用户提供价值

每个服务运行在他们独立的进程里，服务和服务之间用比较轻量级的通信机制沟通(通常是基于 HTTP 的 RESTful API)。每个服务都围绕业务来构建，而且他们能被**独立地**部署到生产环境。然后要避免的是统一的管理，对于一个服务，要根据业务上下文，选择合适的语言和工具去构建它。

**秒杀系统**(将请求尽量拦在系统上游同时对请求进行限流和削峰)

**特点**：1.高性能：因为有大量并发读和写，所以要支持高并发访问。2.一致性：有限数量的库存同时被很多请求同时来减库存，在大并发更新的过程中要保证数据的准确性。

3.高可用：秒杀时会在一瞬间涌入大量流量，为了避免系统宕机，要做好流量控制。

**前端**

限流：用验证码来分散用户请求

禁止重复提交：一个用户秒杀后，把提交按钮置灰

动静分离：把静态数据直接缓存到离用户最近的地方，比如浏览器，服务器端的缓存

**后端**

限流：屏蔽无用的流量，允许少部分流量走后端

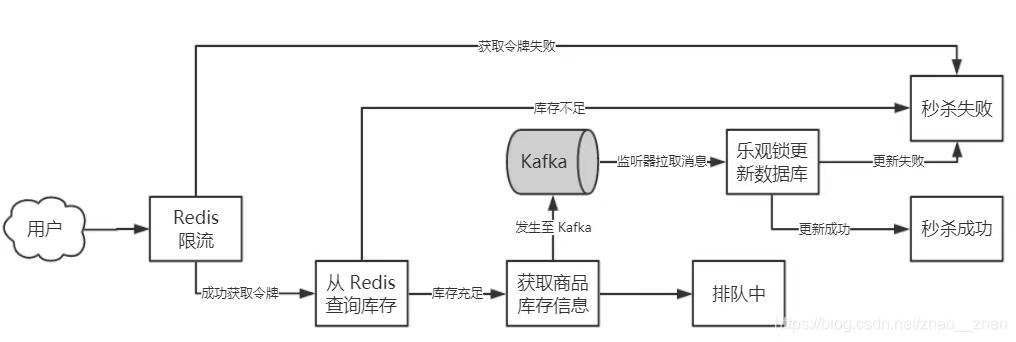
削峰：缓冲瞬时流量，尽量让服务器对资源进行平缓处理

异步：将同步请求转化为异步请求提高并发量

利用缓存：把商品信息放缓存中，减少数据库查询

负载均衡：用多个服务器并发处理请求，减少单个服务器压力

Kafka：高吞吐量的分布式发布订阅消息系统



**网页爬虫设计**：

用例:服务是抓取链接，生成包含搜索词的网页倒排索引，生成页面标题和摘要

用户输入搜索词可以看到相应结果列表

服务有高可用性

假设：搜索流量分布不均，有的搜索词热门有的冷门

用户很快能看到结果，能抓取10亿个链接，每月搜索量1000亿次

**爬虫系统URL去重：**用数组+hash函数(求出URL的哈希值再对数组长度取模)，但是hash冲突时候就会误判(url长度一样但内容不一样以为存在了),

减少哈希冲突：增长数组长度+用多个hash函数判断

布隆过滤器：判断某个元素存在的话可能不存在，但判断某个元素不存在则一定不存在

首先声明一个二进制数据，里面全是0，然后把数据进行n次运算，每次运算都会找到一个位置，就把这些位置变成1，在查询数据的时候也进行n次运算，如果一个位置为0，那么我们就认为这个数据是不存在的；

以下3个问题请求少的是影响不大，但请求多会造成服务器宕机，重启也没用

为了查询速度加快用缓存保存数据，让请求直接查缓存而不是数据库，缓存没有的话再查数据库然后写入缓存，缓存是有有效时长的，不然会一直占内存。

**缓存雪崩**：访问缓存的时候大部分缓存同时过期，请求就会打到数据库，如果请求数量足够大就会把数据库压垮

解决：1.不设置过期时间，缓存更新直接刷新 2.用集群把数据均匀分布在机器上

3.过期时间上加随机值防止缓存几种过期 4.用一些限流机制

**缓存击穿**：大量请求访问缓存里的一个热点key，但这个key刚好过期了，大量请求就会穿过缓存直接到数据库

解决：1.让热点key不设置过期时间 2.设置定时任务把将要过期的key刷新

3. 在缓存中没有数据去数据库查询时加上锁，让一个线程去查询数据库以及更新缓存，其他线程等待，这样减小数据库压力

**缓存穿透**：大量请求访问缓存和数据库都没有的key

解决：1.在缓存放查询的key并且把值设为null，这样请求就不会打到数据库

2.在请求接口处检查，对于不合法的请求直接返回

3.用布隆过滤器(只有0和1的bit数组)-- 对待过滤的数值求hash散列后可以查看这个数组中对应的位置上是否为1来进行判断过滤

后端，提供客户端的数据请求和响应，客户端跟服务端保持连接、客户端发消息，首先是发给服务端、服务端将消息入库并将消息转发给对方；添加一个好友，是通过客户端操作的，搜索好友即向服务端请求搜索、服务端在“库”里面搜索结果、返回结果给客户端

**内存和缓存：**

内存是作为CPU和硬盘间的存储支撑，缓存是cpu的一部分，在cpu里，逻辑上在内存和CPU之间。cpu读数据很快，内存就慢很多。缓存是为了解决cpu速度和内存速度差异问题的。内存里被cpu访问最频繁的数据和指令被复制到cpu的缓存里。在缓存没命中就去内存和磁盘找(数据库在磁盘里，缓存没命中的话数据库会把这些信息更新到缓存)

**缓存的使用**：

读数据：先读取缓存，若不存在则从DB中读取，并将结果写入到缓存中；下次数据读取时便可以直接从缓存中获取数据。

改数据：直接失效缓存数据，再修改DB内容

**缓存不一致问题：**

1.先删除缓存，数据库还没有更新成功，此时如果读取缓存，缓存不存在，去数据库中读取到的是旧值，就会缓存不一致

解决：延时双删--也就是为了避免更新数据库的时候，其他线程从缓存里读不到数据，就在更新完数据库后，再 Sleep 一段时间，然后再次删除缓存--sleep时间大于读写缓存时间就行

ex：线程1删了缓存然后去更新数据库，线程2发现缓存被删了就去数据库读了旧值，并且它把旧值放回到了数据库，线程1sleep时间之后再去把这个缓存删了

1. 先更新数据库还没来得及删缓存，旧值就被读了--不一致

解决：用有监听binlog(二进制日志文件)消息的消息队列(做核对的工作),也就是借助监听binlog的消息队列来删缓存，这样中间件直接帮忙做了解耦((解除2个东西互相影响的现象) 不用单独引入一个消息队列)

为什么删除缓存不是更新缓存：这样会让缓存更新次数和数据库一样多，但缓存可能只被读了很少的次数

redis数据结构：1.string(一个key对应一个value) 2.hash(一个mapmap)

3.链表(有序) 4.集合(无序) 5.zset(有序集合)