面试题by txh

Mysql

- 数据库的三个基础范式

  ```
  1NF 每一列属性都是不可再分的属性，属性要具有原子性
  2NF 列之间要有一个主键，每个属性都要与主键相关，不能有部分依赖
  3NF 列之间不能有传递依赖，每个属性直接依赖于主键
  ```

- mysql的引擎

  ```
  innodb支持事务，mysiam不支持事务；innodb支持外键，mysiam不支持；innodb是聚集索引，mysiam是非聚集索引；mysiam支持全文索引，存储了表的行数；innodb最小的锁粒度是行锁，mysiam是表锁；

  - MyISAM存储: 如果表对事务要求不高，同时是以查询和添加为主的，我们考虑使用myisam存储引擎,比如bbs 中的 发帖表，回复表，还有批量添加MyISAM效率高
  - INNODB 存储: 对事务要求高，保存的数据都是重要数据，我们建议使用INNODB,比如订单表，账号表。
  - memory：数据放在内存，减少io，适合io量很大的场景
  ```

- MyISam和innodb的底层索引结构

  ```
  MyISAM索引文件和数据文件是分离的(非聚集)，索引的叶节点存放的是对应索引在文件系统中的数据地址编码，比如说查找id=49的元素时，是先索引树查询到49对应的数据文件地址，然后再拿着地址在数据文件中取出对应的数据，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据

  InnoDB使用的是聚簇索引，将主键组织到一棵B+树中，而行数据就储存在叶子节点上，若使用"where id = 14"这样的条件查找主键，则按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据。若对Name列进行条件搜索，则需要两个步骤：第一步在辅助索引B+树中检索Name，到达其叶子节点获取对应的主键。第二步使用主键在主索引B+树种再执行一次B+树检索
  ```

- 说下innodb事务隔离的级别

  ```
  ACID:原子性、一致性、隔离性、持久性

  并发问题：

  丢失修改：后修改了覆盖了之前的修改

  读脏数据：t1修改后t2读,t1又回滚了

  不可重复读：t2读取两次，其间t1修改，那么t2两次读取的不一样

  幻读：t2读取某个范围，期间t1插入insert或者删除delete，那么t2两次读取的不一样，和不可重复读区别在于insert，与加锁不一样
  ```

- 乐观锁和悲观锁：

悲观锁的理念：一致性锁定锁，依靠数据库的锁机制

乐观锁：版本记录version号的机制，即MVCC

- 简述一下读锁S和写锁X：

读锁。若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

写锁。若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。这保证了其他事务在T释放A上的锁之前不能再读取和修改A。

- mysql索引的数据结构，联合索引的数据结构

B+树，为什么用B+树？

联合索引：最左前缀原则，联合索引的字段从左到右匹配。

假设你在表的state、city和zip数据列上建立了复合索引。索引中的数据行按照state/city/zip次序排列，先判断最前的一个，从左到右排序判断建立B+树

- 查询缓存，解析 优化 执行

网络

- OSI七层模型

```
应用层 HTTP HTTPS dhcp
表示层
会话层
传输层 TCP UDP
网络层 IP...
数据链路层
物理层
```

- TCP、UDP的区别

TCP面向连接，UDP面向无连接

TCP面向字节流，UDP面向报文

TCP可靠，因为有拥塞控制和丢包重传机制，而且是点对点传输，全双工的可靠信道，UDP不可靠，支持交互通信

- TCP的三次握手

1：客户端发送SYN
2：服务端发送SYN=1,ACK=1
3：客户端发送ACK=1

为啥两次不行？因为客户端知道他到服务端的连接已成功，但是服务端并不知晓连接是否成功，就会出现很多问题。比如说已失效的连接请求报文段又传送到了服务端，因而产生错误的新连接。

- 四次挥手

> 1：客户端发送FIN
>
> 2：服务端发送ACK=1
>
> 3：服务端发送FIN=1，ACK=1
>
> 4：客户端发送ACK=1
>
> 为啥三次不行？因为客户端肯定要知道服务端已经断开连接了啊...
> 服务端需要close_wait。服务器要知道客户端有没有断开连接，如果服务器的FIN+ACK信息客户端没有收到，客户端会一直处在等待状态。

- 为什么最后客户端要等待2MSL的时间

> 1 保证客户端发送的最后一个ACK报文能到达服务器
>
> 2 防止已经失效的连接请求又传过来，让本连接持续的时间内产生的所有报文都去掉

- TCP的流量控制、拥塞控制

> 流量控制：接收端会把缓冲区信息告诉发送端 置0后窗口探测
> 拥塞控制：慢启动，拥塞窗口 阈值

- TCP粘包问题

> 由于TCP的Nagle算法（即把较小包合成数据块再封包）的原因。接收缓冲区中后一包数据的头紧接着前一包的尾。
> 写入的数据大于缓冲区的大小会发生拆包，小于会发生粘包
> 不及时读取缓冲区的，也会发生粘包现象。
>
> 如何解决？
> 1：数据包中包含数据包长度信息
> 2：每个数据包封装为固定长度
> 3：设置特殊符号作为边界

- cookie和session：

> cookie的数据存放在客户端的浏览器上，session的数据存放在服务器上。
>
> 这也导致cookie不安全
>
> session会保存在服务器上，导致占用服务器的性能
>
> cookie保存的数据不能超过4K

- http和https：

1、https协议需要到CA申请证书，一般免费证书较少，因而需要一定费用。

2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl/tls加密传输协议。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL/TLS+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全

这两个最主要的区别就是http是超文本传输协议，是明文传输的，https是ssl/tls+http构建的加密传输。
https更安全，但是HTTPS连接缓存不如HTTP高效，流量成本高。

- get和post

get的请求放在url里，post 放在requestbody里，这导致了get安全，post不咋安全
get传输的数据量小，post大
get效率比post好

- http1.0和http1.1

http1.1比1.0增加的：长连接，新增了range头域支持只发送head和请求部分内容，增加了host处理
2.0：多路复用，服务器推送，头信息和数据体都是二进制，头信息压缩

Java

- java三大特性

继承、封装、多态
多态如何实现？
继承&接口

- Comparable&Comparator

comparable是内部比较器，comparator是外部比较器
Comparable 是一个对象支持自比较，所需要实现的接口
comparator是一个外部比较器，当这个对象不支持自比较或者自比较函数不能满足你的要求时，你可以写一个比较器来完成两个对象之间大小的比较

- java方法传参、深浅拷贝

java只有值传递，没有引用传递。
浅拷贝：只是引用传递
深拷贝：重写clone方法，super.clone

- 重载和重写

```
overload/override
同名函数不同参数
子类中和父类相同的名字、返回类型和参数表
```

- java中“==”和equals的区别

```
== 如果是基本数据类型则对值进行比较，对象则比较地址
equals方法继承object类，有一些类重写了方法，变为类型一致以及内容一致则返回true，比如string/date/file/包装类
如果equals为true，那么hashcode必相等；hashcode不等，则equals必不等
```

- object类的方法

```
clone
getclass 反射获取运行时类型
toString
finalize 释放资源
equals
hashcode: 用于hash查找，可以减少在查找中使用equals的次数
wait: 线程等待该对象的锁
notify: 唤醒某个线程
notifyAll: 唤醒在该对象上等待的所有线程
```

- 抽象类和接口

```
抽象类是为了继承而存在，方法可以为抽象或具体的，接口更加抽象一点，方法只能为抽象的
区别：
语法上：
抽象类方法可以具体或抽象，接口只能抽象
接口中成员变量只能是public static final，抽象类可以是各种类型（不能是private）
接口中不能有静态方法，抽象类可以
设计层面：
类是“是不是”设计，接口是“有没有”设计，接口是一种行为规范，抽象类则更是一种模板式的设计
```

- HashMaP

```
  1:存储结构：数组+链表，1.8后加入红黑树。实例化hashmap时，系统会创建一个entry数组，长度为capacity。数组存放的是entry对象（1.8后改名Node），entry对象可以带一个引用变量指向下一个元素，也就是一个entry链，用于解决hash冲突。
  2：put过程。1：对key对象求hashcode(h^(h>>>16))计算下标 2：如果表空，调用resize()。 3：如果key equals则替换旧值。如果没有碰撞则放入bucket中，如果碰撞了就链接到后面，如果链表长度超了8就转成红黑树。4：如果capacity满了则resize()扩容
  3：hash函数实现
  h = key.hashCode()
  hash = (h^(h>>>16))
  (n-1)&hash
  4:什么时候扩容，扩容机制
  ++size>loadfactor*capacity时扩容
  如果旧数组没有初始化则初始化
  如果初始化了则扩容成两倍，分配一个新数组（2倍长），然后重新进行运算复制到新数组里
  5：解决hash冲突：
  hash地址一样，key不一样，链地址法，开放定址法，rehash
  6：并发问题
  hashmap线程不安全，两个线程可以同时put，可能导致一个线程put的数据被覆盖，不可见
```

7：红黑树
根与叶子为黑色，每个红色节点的孩子都是黑色，任意节点到所有叶子节点的路径的黑色节点数目相同（保证平衡）
和AVL树的比较：AVL树完全平衡，红黑树部分平衡，但是红黑树插入/删除只需O(1)
8：LinkedHashMap多维护一个双向链表
9：TreeMap底层是红黑树，和hashmap的区别：
treemap是排序的，TreeMap的底层使用了红黑树来实现，像TreeMap对象中放入一个key-value键值对时，就会生成一个Entry对象，这个对象就是红黑树的一个节点，其实这个和HashMap是一样的，一个Entry对象作为一个节点，只是这些节点存放的方式不同。
存放每一个Entry对象时都会按照key键的大小按照二叉树的规范进行存放，所以TreeMap中的数据是按照key从小到大排序的。
10：concurrenthashmap 线程安全
concurrent和hashtable hashtable锁住整个表
ConcurrentHashMap将Hash表默认分为16个bucket（每一个bucket可以被看作是一个Hashtable），大部分操作都没有用到锁，而对应的put、remove等操作也只需要锁住当前线程需要用到的bucket，而不需要锁住整个数据。重入锁。

11：新特性
在1.8之后，在数组+链表+红黑树来实现hashmap，当碰撞的元素个数大于8时 & 总容量大于64，会有红黑树的引入
除了添加之后，效率都比链表高，1.8之后链表新进元素加到末尾
ConcurrentHashMap (锁分段机制)，concurrentLevel,jdk1.8采用CAS算法(无锁算法，不再使用锁分段)，数组+链表中也引入了红黑树的使用

- Java的io和nio

io基于流，nio基于缓冲区

- ArrayList和LinkedList

ArrayList数组，LinkedList链表
这导致了getset和插入删除的不同
ArrayList扩容机制：
无参数构造方法创建 ArrayList 时，实际上初始化赋值的是一个空数组。当真正对数组进行添加元素操作时，才真正分配容量。即向数组中添加第一个元素时，数组容量扩为10
//添加元素之前，先调用ensureCapacityInternal方法
//再调用 ensureExplicitCapacity() 方法
//最后grow()扩容成1.5倍

JVM

- 运行时数据区

经典，自己背
取消永久代，方法存放于元空间(Metaspace)，元空间仍然与堆不相连，但与堆共享物理内存，逻辑上可认为在堆中 ，但是实际上我们说的堆指的是用于存放java对象的那些空间。元空间并不在虚拟机中，而是使用本地内存。

- GC

首先，堆和方法区需要gc，主要是堆GC
GC常用算法有：标记-清除算法，标记-压缩算法，复制算法，分代收集算法。

目前主流的JVM（HotSpot）采用的是分代收集算法。
分代收集算法

现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代(Young)和老年代(Tenure)。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用复制算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用标记-整理 或者 标记-清除。

- GC收集器

```
serial
serial-old
parnew serial的多线程版本
parallel
p-old

CMS
老年代
标记-清除
并发+并行，减少停顿
缺点：空间碎片

G1
老年代+新生代
标记-整理为主
把堆内存分成大小固定的独立区域，跟踪区域的垃圾堆积程度，维护一个优先列表，优先回收垃圾最多的区域
```

- 守护线程

```
守护线程与普通线程的唯一区别是：
当JVM中所有的线程都是守护线程的时候，JVM就可以退出了；
如果还有一个或以上的非守护线程则不会退出
```

Java多线程

- 线程池

```
为什么要有线程池？防止过多的创建/销毁线程的操作，在线程池的编程模式下，线程池在拿到任务后，就在内部寻找是否有空闲的线程，如果有，则将任务交给某个空闲的线程。
 Executors.newCacheThreadPool()：可缓存线程池
 Executors.newFixedThreadPool(int n)：创建一个可重用固定个数的线程池，共享队列
 Executors.newScheduledThreadPool(int n)：创建一个定长线程池，支持定时及周期性任务执行
 Executors.newSingleThreadExecutor()：创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务

 这些其实就是ThreadPoolExecutor

 缓冲队列BlockingQueue：
```

BlockingQueue是双缓冲队列。BlockingQueue内部使用两条队列，允许两个线程同时向队列一个存储，一个取出操作。在保证并发安全的同时，提高了队列的存取效率。
ThreadPoolExecutor:自定义的线程池
int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue
corePoolSize- 池中所保存的线程数
maximumPoolSize-池中允许的最大线程数
keepAliveTime - 当线程数大于核心时，多于的空闲线程最多存活时间
unit - keepAliveTime 参数的时间单位
workQueue - 当线程数目超过核心线程数时用于保存任务的队列
threadFactory - 执行程序创建新线程时使用的工厂
handler - 阻塞队列已满且线程数达到最大值时所采取的饱和策略

workqueue最好用new ArrayBlockingQueue<>(512)来实现，避免OOM

- 线程方法

run()和start()
start才是真正开启线程的方法，run是运行线程类中的run方法

- 线程状态

开始new，终止terminated，
就绪runnable，运行running，阻塞blocked
running->runnable
wait() sleep() 等待i/o
wait是Object的方法，释放了锁，要配合notify使用，一般用于同步
sleep是Thread的方法，一般是暂停线程使用

- 实现并发的方式

synchronized
volatile

- synchronize

synchronized 关键字来保证一次只有一个线程在执行代码块

底层实现原理：
方法级的同步：用一个acc_synchronized标志区分方法是否同步方法。
代码块的同步：monitorenter/monitorexit两个字节码指令，位于开始和结束的位置。

- volatile

Volatile 可见性 防止指令重排。
Volatile 关键字保证任何线程在读取Volatile修饰的变量的时候，读取的都是这个变量的最新数据。
Volatile强制线程从主存中读取数据

- threadlocal

  threadlocal而是一个线程内部的存储类，可以在指定线程内存储数据，数据存储以后，只有指定线程可以得到存储数据
  每个线程持有一个ThreadLocalMap对象

  跟synchronize的对比：
  Synchronized是通过线程等待，牺牲时间来解决访问冲突
  ThreadLocal是通过每个线程单独一份存储空间，牺牲空间来解决冲突，并且相比于Synchronized，ThreadLocal具有线程隔离的效果，只有在线程内才能获取到对应的值，线程外则不能访问到想要的值

- Java锁机制

  偏移锁：markword中存储thread id。
  撤销操作：revoke，不是把它撤销，而是升级为轻量级锁
  轻量级锁：超过一个线程竞争一个对象时，会变成轻量级锁，在当前线程的栈帧中创建用于存储锁记录的空间，并且将对象头的mark word复制。然后使用CAS操作将对象头中的mark word替换为指向锁记录的指针，失败则先自旋
  重量级锁：依靠操作系统的互斥量mutex进行，需要操作系统从用户态和内核态转换
  根据竞争的激烈程度进行机制的切换

  自旋锁：
  阻塞锁：改变了线程的运行状态
  可重入锁：不只是reentrantlock

分布式&并发

- 消息队列

  通过异步处理提高系统性能
  降低耦合（解耦）

- CAS原理

  分布式系统只能满足一致性、可用性、分区容错性其中的两个。

操作系统

- 进程调度

```
FCFS 先来先服务
SJF 短作业优先
SPF 短进程优先

最短剩余时间优先（对应抢占）
高响应比优先 等待时间 / 服务时间 + 1

时间片轮转 该算法简单有效，常用于分时系统，但不利于I/O频繁的而紧凑，由于这种进程用不完一个时间片，就因为等待I/O操作而被阻塞，当I/O操作结束后，只能插入到就绪队列的末尾，等待下一轮调度
```

- 进程通信

```
管道
共享内存
消息队列
信号量
信号
socket
```

- 内存管理

```
段 每一段有逻辑信息，但是大小不固定
页 无逻辑信息
段页 先段再页，段间有关页无关
```

- 虚拟内存

```
虚拟内存的重要意义是它定义了一个连续的虚拟地址空间，并且 把内存扩展到硬盘空间
基于局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其他部分留在外存，就可以启动程序执行。由于外存往往比内存大很多，所以我们运行的软件的内存大小实际上是可以比计算机系统实际的内存大小大的。在程序执行过程中，当所访问的信息不在内存时，由操作系统将所需要的部分调入内存，然后继续执行程序。
```

- 页面置换

```
  FIFO
  OPT
  LRU
  CLOCK：把所有页面都保存在一个类似时钟的环形链表，一个指针指向最老的页面。如果发生中断就首先检查表指针指向的页面，如果R位是0就换出，否则置0再寻找下一位。
```

- 磁盘调度：

FCFS：先来先服务
SSTF：最短寻道时间优先
SCAN：电梯调度，不只考虑磁道距离，也考虑磁头的移动方向，就像电梯一样，先考虑方向，下一个访问对象是该方向最近的，直到无更外的磁道访问
CSCAN：单向移动，不做返回移动，因为一般情况下磁头访问过的磁道请求密度较低，另一边的磁道可能密度较高

- 死锁

互斥：某种资源一次只允许一个进程访问
占有且等待：占有，同时还有资源未得到满足
不可抢占：
循环等待：

如何避免？最好的是在使用前进行判断，只允许不会产生死锁的进程额外申请资源。
银行家算法，实时统计

- 进程，程序，线程

1：程序只是一组指令的有序集合，是一个静态的实体。进程是操作系统分配资源的基本单位，是一个动态的实体，反应了程序运行的动态过程。进程和程序也不是一一对应的。
2：进程是一个独立单位，是系统进行资源分配和调度的基本单位，线程是包含在进程之中的，是进程中的实际执行流，是程序执行的最小单位。
3：进程拥有自己的地址空间！！有一块独立的内存区域！！

- 进程拥有的内存，进程的内存管理

一、在操作系统中，系统会给每个进程分配虚拟地址，虚拟地址的大小与处理器的位数有关，如32位处理器进程可分配4GB的虚拟内存供程序正常运行。
这4GB的虚拟内存，存储单元从地址0开始进行排序，此地址为虚拟地址。
此虚拟地址可分为：
1、栈区（stack）：由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。线程也有自己维护的栈。
2、堆区（heap） ：程序动态申请的空间，由程序释放或其他方式释放，若没有释放，可能导致内存泄露。
3、全局区（静态区）（static）：全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后有系统释放
4、文字常量区：常量字符串就是放在这里的。 程序结束后由系统释放
5、程序代码区：存放函数体的二进制代码
堆自己可以分配回收，栈程序。栈效率应该更高，因为不需要动态分配。

- 五种IO模型

阻塞：
传统模型了，用户线程发请求，看看是否就绪，如果没有就绪就一直阻塞，用户线程交出CPU
非阻塞：
用户线程需要不断地询问内核数据是否就绪，也就说非阻塞IO不会交出CPU，而会一直占用CPU
信号驱动：
用户线程发起一个IO请求操作，会给对应的socket注册一个信号函数，然后用户线程会继续执行，当内核数据就绪时会发送一个信号给用户线程，用户线程接收到信号之后，便在信号函数中调用IO读写操作来进行实际的IO请求操作。这个一般用于UDP中
多路复用：
一旦就绪就通知进程进行相应的读写，Java NIO实际上就是多路复用IO。在Java NIO中，是通过selector.select()去查询每个通道是否有到达事件，如果没有事件，则一直阻塞在那里
异步：
异步IO模型才是最理想的IO模型，在异步IO模型中，当用户线程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个asynchronous read之后，它会立刻返回，说明read请求已经成功发起了，因此不会对用户线程产生任何block
Java 7中，提供了Asynchronous IO。简称AIO

- Java io模型

Java 中的 BIO、NIO和 AIO 理解为是 Java 语言对操作系统的各种 IO 模型的封装。程序员在使用这些 API 的时候，不需要关心操作系统层面的知识，也不需要根据不同操作系统编写不同的代码。只需要使用Java的API就可以了。
BIO：同步阻塞IO，请求-应答模型，IO调用时需要等待其IO完成。采用线程池和任务队列可以实现一种叫做伪异步的 I/O 通信框架
NIO：new IO，非阻塞，NIO 包含下面几个核心的组件：

Channel(通道)
Buffer(缓冲区)
Selector(选择器)
通常来说NIO中的所有IO都是从 Channel（通道） 开始的。

从通道进行数据读取 ：创建一个缓冲区，然后请求通道读取数据。
从通道进行数据写入 ：创建一个缓冲区，填充数据，并要求通道写入数据。
AIO:JDK7提供的异步非阻塞IO，

- IO多路复用

通过一种机制（系统内核缓冲I/O数据），让单个进程可以监视多个文件描述符，一旦某个描述符就绪（一般是读就绪或写就绪），能够通知程序进行相应的读写操作
首先，操作系统等待内核处理的时候，首先要等到数据传到kernel内核space，然后kernel内核区域将数据复制到user space（理解为进程或者线程的缓冲区）
select:用单个进程处理io请求，程序呼叫select->select寻找文件描述符fd->复制回用户态
poll:基本类似select，是链式的（类似链表）
缺点：fd太大时由于需要复制，开销太大了
epoll:取消了轮询确认fd的机制，通过epoll_wait等待队列中有没有就绪的fd，每个fd指定了一个回调函数调用，不需要每次都从用户空间将fd_set复制到内核kernel

Linux

- linux文件

```
linux秉承：一切都是文件
```

- linux基本操作

```
cat/tail/more/less 查看文件
tail -f 动态监控
chmod 授权 rwx 先用户 再用户组 再所有用户
查看进程ps -ef 和 ps -aux
kill -9/kill -15
查看当前系统的网卡信息：ifconfig
查看与某台机器的连接情况：ping
查看当前系统的端口使用：netstat -an
```

- 孤儿进程和僵尸进程

```
进程exit时不占空间，但是占用ID，但是父进程会为他收尸
一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作
```

- Socket传输

```
客户端：socket()创建socket->connect()连接开启三次握手建立连接
服务端：socket()创建->bind()绑定端口->listen()监听，调用io多路复用中的select函数，查询socket描述符是否处于就绪状态，若非就绪则过一段时间再调用
```