面试题汇总

https://how2j.cn/k/j2se-interview/j2se-interview-java/624.html?p=136499#step1940 面试题地址

浏览器中输入URL以后发生了什么

1. 浏览器想DNS服务器发送域名，请求该域名对应的IP地址是多少
2. DNS服务器查到IP后返回IP
3. 浏览器拿到IP地址后，根据端口号与服务器进行tcp连接
4. 浏览器向服务端发送请求，请求需要的文件信息
5. 服务器处理这个请求，找寻资源文件等
6. 向浏览器发送响应
7. 双方中某一方关闭连接
8. 浏览器解析文件并渲染

TCP四次挥手期间状态

1. 第一次挥手 A端发起断开连接FIN，进入FIN_WAIT1状态
2. 第二次挥手 B端接收到FIN后，发送ACK确认，此时B端进入CLOSE_WAIT状态
3. 第三次挥手 B端这边报文发送完，发送FIN给A端，表示要关闭连接，进入LAST_ACK状态
4. 第四次挥手 A端收到FIN，发送ACK到B端，进入TIME_WAIT状态

HTTPS详解

为什么mysql索引用B+树而不是其他树

浏览器与DNS之间交互用什么协议

UDP协议

最左匹配原则

> 作用于组合索引，建立组合索引时的顺序，mysql会根据最左边的这一列进行组合索引的排序，也就是最左边的一列是有序的，当我们进行查询时，如果查询的是根据最左侧的列等于某个值，那么就会使用到组合索引，如果根据的是右面的列进行索引，那么就会对索引全局进行扫描，寻找符合条件的数据，也就是index类型，如果有最左列的，那么使用索引，type为ref，但是ref类型也必须是A = ? and B = ?，如果 A > ?， 那么类型为range， 也就是where子句后面有一个是判断大小的操作，类型就会是range

explain 关键字查询 各种状态码

> 1. ALL 查询条件为没有索引的字段；当表中有索引，但是查询条件没有where时，也是ALL全表扫描
> 2. index 索引列进行order by操作时，它的这个状态也是全局扫描，不过扫描的是索引
> 3. range 当条件列中有索引列，并且采用的是> < 或者 in等操作，就是range，指的是有范围的索引查询
> 4. ref 这个是不是主键并且不为唯一的索引，进行查询时，会进行rec，因为组合索引和普通索引无法确定是否有重复元素，所以查到一个之后，还需要在索引附近查找是否还有该值
> 5. const 当为唯一索引或者主键索引时，查询条件为等于某值时，因为唯一，所以可以常数阶查询

类加载过程

1. 加载

> 1. 通过一个类的全限定名获取定义此类的二进制字节流；
> 2. jvm将字节流所代表的的静态存储结构转化为发放区的运行时数据结构；
> 3. 建立Class对象，作为方法区中这些数据的访问入口

2. 连接

> 1. 验证：
>   i. 验证文件格式，是否为class格式
>   ii. 元数据验证，比如验证这个类是否有父类，定义的字段名是否和父类冲突
>   iii. 字节码验证：验证语义是否正确，确保语义是和发达的，符合逻辑的，不会有危害虚拟机的情况
>   iv. 符号解析验证：主要对类自身以外的信息进行验证，目的是确保解析动作能够顺利完成
> 2. 准备： 先给类变量分配内存，之后给类的静态变量赋值，赋的是初始值
> 3. 解析： 给符号引用转为直接引用
>   i. 什么是符号引用呢，符号引用是用一串符号来描述引用的目标
>   ii. 直接引用

3. 初始化

Get与Post的区别

1. Get在http协议标准中用于查询，Post用于更新
2. Get请求时通过URL直接请求数据，数据信息可以直接在URL中看到；POST请求是放在请求头中的
3. 说GET方法请求有长度，其实是不对的，因为http协议并没有规定URL长度，只是不同的浏览器有不同的URL规定的长度;POST受限于服务器处理能力
4. 在我的使用中，在查询信息时使用GET，在增删改的时候使用POST

JVM详解

1. 内存分布： JVM内存中分为 程序计数器、栈、堆、方法区、本地方法栈

i. 堆：线程公有，里面存放对象实例，对象实例又指向方法区中的类模板
ii. 栈：存放变量、数据表、方法出口等，当递归没有合理控制时，就会占用过得的栈，所以出现栈内存溢出的错误
iii. 方法区：方法区中存放类模板信息、常量、类变量（静态变量）

java语言和Go语言的区别

http结构

1. 请求报文： 请求行，请求头，请求体
   i. 请求行：方法：GET/POST URI DEMO/DEMO/ 协议版本：HTTP/1.1
   ii. 请求头：Host：域名 Referer：告诉服务器该网页是从哪个页面链接过来的 User-agent：告诉服务器这个请求的发送者是谁 accept：表示发送端希望接收的数据类型 Accept-Language：请求端可以接收的语言类型 Accept-Encoding：表示浏览器可以接受的编码类型 Connection ： 是否保持长连接
   iii. 请求体：一般POST请求中有请求体，用来向服务端进行参数传递
2. 响应报文： 响应行，响应头，响应体
   i. 响应行： 版本：HTTP/1.1 状态码：200 原因短语：表示成功或者失败
   ii. 响应头：date：此次响应的时间 content-type：媒体格式类型 etag：第一次服务器响应给界面，之后再要进行请求，请求头里就会带上If-None-Match字段，服务器进行判断，如果这两个字段一样，就返回304，让浏览器从缓存中读取 content-length 返回的文件长度
   iii. 服务端响应给客户端的文件、数据等

HTTPS建立握手过程，涉及到哪些算法

建立握手过程：

1. 浏览器先向服务器发送自己支持的加密规则以及一套hash算法
2. 网站从中选出一组规则，并将自己的身份信息以证书的形式响应回去。证书里包含了网站地址，网站的加密公钥以及证书的颁发机构等信息
3. 浏览器获得网站的证书后，进行验证
   i. 验证证书的合法性（颁发的机构是否合法，证书中的网址是否正确），如果证书正确，则地址栏会出现一个小锁一样的图标，如果不正确，会提醒用户是否信任该证书
   ii. 如果证书合法，或者用户选择信任证书，那么浏览器会生成一串随机数，并且用网站的公钥进行加密，作为密码
   iii. 还需要一个握手信息，使用约定好的hash计算握手信息，并且用随机数来加密握手信息，将加密后的随机数和握手信息一起发送给网站
4. 网站接收到浏览器的消息后
   i. 使用自己的私钥将信息解密，得到随机数密码，用密码解密握手信息
   ii. 使用密码加密自己的握手信息，并且返回给浏览器
5. 浏览器接受握手信息后，解密握手信息并且计算hash，如果与服务端发来的hash一致，那么此时握手结束，之后所有的通信将基于之前的随机数密码以及对称加密算法进行

java堆和栈和方法区详谈

堆： 堆是jvm内存结构中的一部分，它是线程共享的，并且存放的是对象的实例，所以全局变量存储在堆中，在类的实例中。由于其是线程共享的，所以需要依赖于垃圾回收算法来保证堆内存不会溢出。堆中的对象被栈中的变量指向，它里面有方法之类的又指向方法区中的方法之类。
栈： 栈是java中线程私有的部分，每开辟一个线程就开辟一个栈。每调用一个方法，就开辟一个栈帧，里面存放局部变量，操作数栈，方法出口等。容易造成栈内存溢出的原因是失败的递归，递归不正确会造成一直在栈中开辟空间，造成栈内存溢出
方法区：方法区存放常量，静态变量，静态方法，类的信息（字段信息，方法信息，类名）以及编译器编译后的代码等 方法区中的回收规则： 1) 该类已经没有任何实例 2) 该类的类加载器已被回收 3) 代码中已经没有反射该类

IO机制

文件访问机制：读和写分别对应Read()和Write()两个系统调用，必须使用操作系统提供的接口。操作系统会将数据进行缓存。当读文件时，操作系统先检索是否有缓存，如果有缓存，就将缓存返回给进程，然后缓存到缓存中。写入的方式是将文件写入到缓存中，这时候进程就算是写完了，至于什么时候写到磁盘中由操作系统决定

union和union all的区别

union all直接返回结果集，不进行去重
union 将两个查询的结果集进行去重合并

重写重载的区别

重写：发生在继承中，将父类的同名方法在本类中再实现一次，方法名和参数列表必须相同，子类方法不能声明比父类更高的权限修饰符，不能声明非父类方法抛出异常子类之外的异常
重载: 同一个类中两个同名的方法，参数列表必须不同，返回值可以不同，抛出异常可以不同

B+树与B树的区别

B树：因为B树每个节点可以存储多个值，所以他的深度更浅，它的节点数更少。每次从磁盘中添加一个节点到内存中，就会有一次IO，所以B树进行IO的次数更少。从磁盘中读取数据到内存中一般是按页读取的，一页读取一个节点，而一个节点只存储一个数据会很耗时，所以B树比二叉搜索树和红黑树都好
B树的缺点：B树有着存储数据的问题，如果他将每行数据存储到节点的值中而又要保证每个结点接近每页的大小，那么节点的能存储元素的数量就变少了，这样就无法减少IO读取耗时了。而且当我们要查询范围的时候，想要在B树内进行范围查询，需要找到查询范围，再遍历树的上界和下界，比较费时
B+树： B+树每个非叶子节点是不存储value的，只存储key值，key值存放子节点的最小值或者最大值。叶子节点中存放值。B+树的叶子节点形成了一个有序的链表，索引到

为什么mysql要使用B+树做索引?

B+树因为非叶子节点中不含有数据，所以单一的分支结点中存储更多的元素，所以相较于B-树更能减少IO次数
而且由于B+树分支结点中不存在数据，所以它必须要遍历到叶子结点，所以它的查询更稳定
B+树在范围查询时表现比B-树要好

https://blog.csdn.net/qq_26222859/article/details/80631121

Redis的数据结构？缓存击穿是什么，怎么避免

怎么选择string还是hash存储对象？
当一次要获取对象所有元素时，使用string，只需要查询一次，而HGet需要进行多次查询；当不需要获取所有元素时 string类型：用来记录库存
hash：存储用户对象信息，
list：是简单的字符串链表
set：无序的字符串链表，我用其来实现好友列表，记录用户的好友们。
zset：有序的字符串链表 插入时需要给一个分数，用来进行排序

https://baijiahao.baidu.com/s?id=1660009541007805174&wfr=spider&for=pc

乐观锁和悲观锁

乐观锁：

数据库范式

如果数据规模增大几个数量级，可以有什么优化?

如何去设计缓存

linux的buffer内存和cache内存

ConnectTimeOut ReadTimeout异常的区别？Unknown host是什么原因导致的?

微服务是什么？微服务的粒度？怎么去划分?

G1和CMS分别是什么，区别?

mysql搜索引擎

Redis缓存击穿，缓存穿透，雪崩

算法

1. 无序列表求中位数 建立大顶堆和小顶堆，遍历数组，添加进堆的顺序是，先将元素进一个堆，如果进大顶堆，那么将大顶堆最大的数进小顶堆，如果进小顶堆，那么小顶堆最小的数进大顶堆，之后小顶堆和大顶堆顶的数就是中位数
2. 两个有序列表求中位数 添加到一个一个总数组中，之后求总数组的中位数即可，添加时注意有序添加

3. 计算字符串的最大前缀 给几个字符串 leetCode leetFine lee 求最大前缀 这个题思路是，先取第一个字符串，之后从第二个字符串开始，调用indexOf方法，如果不包含，那么将第一个字符串最后一个字节去掉，之后继续while循环，循环完毕后，就是最大前缀了
4. 计算一个字符串中是否包含另一个字符串的排列 滑动窗口，先遍历子串大小，之后把每个i位置对应的字符转成int，然后母串也转换，之后判断，如果两个数组相等，那么说明包含，如果不等，遍历母串，窗口向后滑动一格，将当前的i元素--，不自减的话会造成永远判定不对
5. 无重复字符的最大子串 这个题，也采用滑动窗口，不过窗口是变化的，一开始时窗口宽度为0，判断右侧新增的元素在不在窗口里重复，如果重复，那么将窗口左侧向右移动，继续判断是否有重复，有重复继续移动，没有重复后窗口向右移动，直到右侧到达字符串最右端
6. 字符串相乘 声明一个长度为两个字符串长度相加的整型数组，之后两个字符串每个字符两两相乘，第i位与第j位相乘记录在第 i+j+1位置上

分布式锁

https://www.jianshu.com/p/bc8dd3c0db22

mysql二级索引

聚簇索引

innoDB引擎的主键索引：叶子节点上存储的是数据行

二级索引

innoDB的二级索引，叶子节点上存储的是记录的主键

结论

1. 主键索引的效率比二级索引要高，因为二级索引先要找到索引位置之后再去找主键索引
2. 主键占用内存小的话，二级索引占用的空间也会表笑

myisam引擎

主键索引和二级索引都是存放的地址

rpc、MQ面试题，Redis如何保持缓存同步 IaaS,Saas,PaaS sql注入，分库分表，数据迁移

快手Java面经

https://zhuanlan.zhihu.com/p/86776589
https://cloud.tencent.com/developer/article/1353557

TCP与UDP详解

https://blog.csdn.net/hansionz/article/details/86435127

技术栈 java go 前端 bootstrap 业务开发 大数据 产品开发

网络

TCP的三次握手和四次挥手

三次握手

> 1. 请求方向另一方发送SYN以及序号J，进入SYN-SENT状态
> 2. 对端接收到消息后进入SYN-RCVD阶段，返回一个SYN以及序号K，以及确认ACK，J+1
> 3. 请求方发送ACK确认序号K+1，至此建立连接

四次挥手

> 1. 请求方发起FIN以及序号M，此时请求方进入FIN-WAIT1阶段
> 2. 响应方接收请求，进入CLOSE-WAIT阶段，并且向请求方发送ACK M+1通知请求方已经收到了请求，请求方进入FIN-WAIT2阶段
> 3. 当响应方发送完数据后，向请求方发送FIN以及序号N，进入LAST-ACK阶段
> 4. 请求方接收到FIN后进入TIME-WAIT阶段，向响应方回复ACK N+1，等待2MSL之后连接关闭

为什么要等待2MSL呢

1. 确保正确关闭连接，避免ACK未发送到另一端的情况，如果等待2个最大存活时间，那么即使ACK未发送到对端，也可以有时间接收对端的重发的FIN
2. 经过2MSL后，本次连接中的所有数据都已经在网络中消失，不会干扰到下一次连接

为什么需要三次握手

> 1. 为了建立有效的连接，因为如果是两次连接，那么无法确认响应方的数据序号，第一步发起方发送SYN以及SEQ，第二步响应方发送ACK以及SYN以及SEQ，连接建立，那么响应方可以确定发起方的序号，发起方无法确定响应方的序号
> 2. 防止网络上滞留的信息来发起请求，如果是两次握手，网络上滞留了一个SYN，那么接收方收到后，即开始发送SYN以及ACK，但是浏览器并不会理会，这时接收方以为建立连接了，开始等候消息，则造成了资源浪费
> 3. 为了提高效率所以不四次握手，第二步中发送SYN以及发送确认ACK是可以合并到一步的

三次握手传递的数据

1. 客户端发起连接请求，发送一个SYN=1以及一个seq=x
2. 服务端接收请求，发送一个SYN=1,ACK=1,ack=x+1,seq=y
3. 客户端收到确认，发送ACK=1,seq=x+1，ack=y+1

HTTP与HTTPS

HTTPS建立连接的过程

> 1. 首先建立TCP连接，详细参考上方TCP三次握手
> 2. 建立TCP连接后，浏览器向服务器发送自己支持的加密规则
> 3. 服务器从浏览器支持的加密规则中选取一个
> 4. 服务器将选好的加密规则和算法以及自己的公钥以证书形式返回给浏览器
> 5. 浏览器根据根证书验证证书
> 6. 浏览器生成对称秘钥，根据证书中的公钥进行加密，发送给服务器
> 7. 服务器通过公钥解密秘钥，使用秘钥加密数据
> 8. 浏览器根据对称秘钥解密数据，建立SSL连接

https采用什么加密

> 在验证公钥以及生成对称秘钥之前，采用非对称加密
> 在验证秘钥后采用对称加密

TCP如何保证数据可靠性

1. 校验和

2. 确认应答机制

当一方接收到另一方的消息后，会发送一个ACK，为下一段要接收的序列号，作用是通知对端已经接受了消息，可以发送下一段消息了

3. 超时重传

当A端发送一条数据包后，如果在一定时间内未收到确认，则A端会重发该数。
这里涉及到两个情况
如果是没有发送到B端，那么就重发，
如果发送到B端，但是B端没有响应，那么就重发，此时数据会有重复，交给B端去去重

4. 连接管理

即需要经过三次握手来建立连接以及四次挥手来断开连接

5. 流量控制

如果接收方的缓冲区满了，发送方继续发送数据就会导致接收方接收不到，导致超时未发送确认，导致超时重传
TCP在接收方接收数据后，返回给发送方自己的缓冲区大小，当缓冲区为0时，发送方就不再发送数据，但是会定时发送一个探测报文，用来探测接收方是否可以接收报文。

6. 拥塞控制

虽然有了滑动窗口，但是我们一开始不能发送过多数据，因为不知道网络的拥塞情况
所以TCP采用 **慢启动**机制，**快重传**以及**快恢复**机制。

慢启动

设置一个拥塞窗口，初始设置为1，取拥塞窗口与接收端缓冲区的较小值。
在启动初期以指数形式增加，每收到一个ACK，则将拥塞窗口乘以二。当达到慢启动的阈值后，进入线性增加，每次增加一。当达到网络拥塞后，将阈值设置为当前拥塞窗口的一半，并且启动的拥塞窗口数设置为1。

- cwnd = 1
- cwnd = 2 * cwnd
- 超过阈值之后
- cwnd++
- 网络拥塞后
- sshthresh = cwnd / 2
- cwnd = 1

快重传

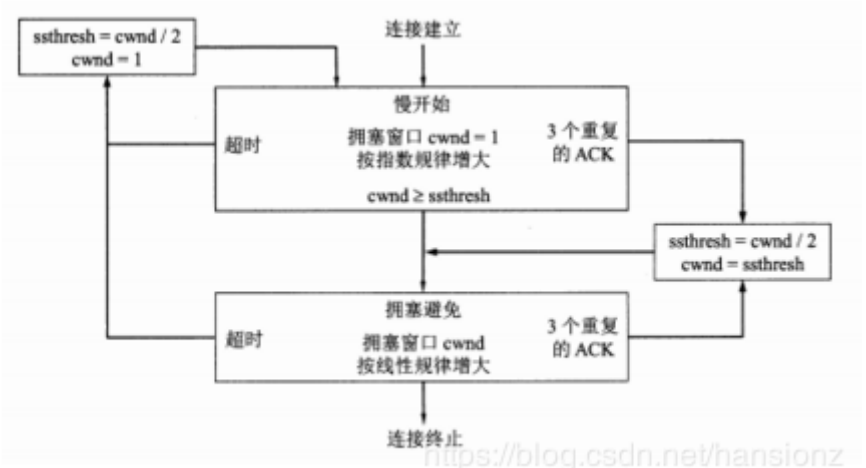
当连续收到三个ACK后，就会开始快重传。

- sshthresh = cwnd /2
- cwnd 重置为 1
- 进入慢启动过程

快恢复

如果我们连续收到三个确认报文后，我们就开始新的慢启动，则有点反应强烈了。
所以在快重传时，不采用慢启动，而是采用下面这个算法：

- cwnd = cwnd /2
- sshthresh = cwnd



TCP和UDP的区别

1. TCP是面向连接的，而UDP是不面向连接的

2. TCP保证数据的可靠性，UDP不保证
3. TCP对系统资源要求多，因为需要三次握手，期间有拥塞控制，重传，确认应答机制等
4. TCP是面向字节流的，将数据拆分成字节流后发送，UDP是面向报文的
5. TCP是一对一的，而UDP支持一对多，一对一，多对一，多对多等

HTTP和HTTPS的区别

1. HTTP是不加密传输，HTTPS是加密传输
2. HTTPS经由HTTPS传输，但是传输的内容经过对称加密

URL各部分详解

http://127.0.0.1:8080/demo/demo.jsp?id=5&p=3

> http 所使用的协议名称
> 127.0.0.1 域名 ip地址也可以作为域名 8080 端口号，URL中没有端口号的话会使用默认的端口
> demo 端口或者域名后面的/到最后一个/中间夹着的就是虚拟目录
> demo.jsp 最后一个/后面知道？或者#为文件名 即向服务器请求的文件
> ?id=5&p=3 参数，可以后台通过解析来获得参数
> #point 根据name或者id界面将定位到div

token

token的验证过程

1. 用户第一次进行登录
2. 验证登录
3. 服务端返回一个签名的token给客户端
4. 客户端再请求api时，带上token
5. 服务端拿到token

死锁

什么是死锁

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者彼此通信造成阻塞的一种现象，若无外力作用，他们都将无法推动下去。进程永远在等待要请求的资源。

例如：进程A占有着打印机，之后要访问输入设备，进程B占有者输入设备，之后要访问打印机，A等待B释放输入设备，B等待A释放打印机，一直等待下去，就是死锁。

MySQL

数据库范式

第一范式 1NF

属性不可分割，比如id，name字段，就是不可分割的，而如果是姓名和性别，那么就不满足第一范式

第二范式 2NF

第二范式要求表中要有主键，其他的字段都依赖于主键。主键要可以区分一条记录

第三范式 3NF

第三范式要求一个表中不能含有另一个表中非主键的字段，即我们如果需要存储其他表中的数据，就需要以外键形式关联另一张表的主键。属性不依赖其他非主属性

MySQL索引采用什么数据类型

采用B+树

为什么采用B+树而不是B-树或者红黑树

红黑树也是属于二叉树的一种，则其相较于B-树以及B+树而言，它的深度是非常深的，会进行大量的IO，性能较差

B-树是多叉树，其分支节点上存储的是子节点的索引以及数据，所以其相较于二叉树来说更加矮胖，节点中K个元素中包含着子节点的范围，所以查询时可以查询分支节点确定叶子结点的位置。

B-的查询特点是不确定，有可能查询的元素在根节点中，则触发一次磁盘IO即可查到对应数据，有可能需要进入到叶子节点中才能查询到。但是其缺点并不是如此，而是其应对范围查询的能力较差。当我们需要进行范围查询时，B-树首先查询到最小值，之后需要中序遍历得到范围中的其他值。

B+树的特点是所有的分支节点中不存数据，只有叶子节点中存储数据；并且分支节点中存储叶子节点的索引(叶子节点的最大值或最小值)；每个叶子结点中包含了指向下一个叶子结点的指针，由于叶子结点中的值本身就是有序的，则叶子节点之间形成了一个有序链表。

综上所述，B+树较B-树的优势是：

1. 单一节点存储更多的数据，所以比B-树更加矮胖，减少了IO次数
2. 所有的查询都会找到叶子结点，查询性能稳定
3. 范围查找时只需要找到对应的叶子结点之后进行链表内向后查找即可。

B+树的特征：

1. 有k个子树的中间节点包含有k个元素（B树中是k-1个元素），每个元素不保存数据，只用来索引，所有数据都保存在叶子节点。
2. 所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针(聚簇索引为数据)，且叶子结点本身依关键字的大小自小而大顺序链接。
3. 所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素。

二级索引

什么是二级索引

叶子结点中存储的不是数据，而是主键值

为什么要采用二级索引呢

因为如果所有的索引都是聚簇索引的话，那么会占用大量的内存空间。所以在InnoDB中，主键索引采用聚簇索引，而唯一索引，普通索引，前缀索引等都是用的二级索引。

为什么要在叶子结点中存储主键值

InnoDB在移动行时，无需维护二级索引，因为如果存储的是指针的话，移动行后位置发生变化，需要维护二级索引的指针值，而存储主键值则没有这个麻烦。

为什么主键最好采用增加的值

如果主键是增加的，那么当我们需要插入新的记录时，只需要在叶子节点最后添加元素即可，而不需要插入到中间，之后出发B+树的重新排序。

最左匹配原则

适用于联合索引，即(a,b)索引，当进行

```
# 如下是不会使用索引的
select something from tableName where b = 1
# 当使用a作为查询条件会走索引
select something from tableName where a = 1 and b > 1
```

如下图就是联合索引的图解

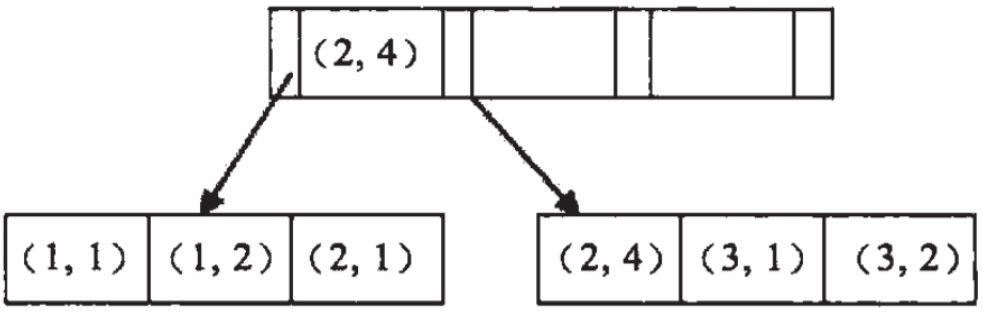


图 9-22　多个键值的 B+ 树

事务

> 事务表示一系列操作要么全部成功，要么全部失败

事务的四大特性

原子性

一个事务中的操作要不全部执行，要么全部不执行，不会结束在中间某个环节，事务在执行过程中发生错误，会被回滚到开始事务之前的状态，就像这个事务从来没有执行过一样

一致性

在事务的开始之前和结束之后，数据库的一致性没有被破坏。这表示写入的规则必须完全符合预设规则，包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。

隔离性

数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。

持久性

事务结束后数据会写入到数据库中，即使重启机器仍然有效

事务隔离级别

MySQL采用客户端/服务端的方式，即可能有多个客户端连接同一个服务端，那么就可能产生一些问题。
脏读：一个事务读取到另一个事务未提交的数据：
当A事务修改了一条记录时，B事务读取，而此时A事务可能回滚或者更改该记录为另一个值，那么B读到的数据就是错的，称为脏读。
不可重复读：一个事务查询两次同一条记录但是数据不一样。
A事务查询一条记录，B事务在A查询后更改了记录并且提交，此时A事务又去查询该记录，而此时数据已经改变，所以读到了不同的数据。
幻读：A事务在读取某个范围内的记录时，B事务在该范围内插入了新的记录，当A事务再次读取范围内的记录时，会产生幻行。

所以我们需要引出事务的隔离级别：
读未提交：其他事务可以读到当前事务的未提交的记录，可能发生脏读、不可重复读以及幻读。
读已提交：其他事务只能读到当前事务已经提交的记录，解决了脏读，但是会发生不可重复读，幻读。

时间	事务一	事务二
T1	开启事务	
T2		开启事务
T3		查询余额 1000
T4	查询余额 1000	
T5		取出1000 余额 0

时间	事务一	事务二
T6		提交
T7	查询余额 0	
T8	提交	

如上表，即读到了其他事务提交后的数据，发生了不可重复读。
可重复读：在一些业务场景中，一个事务只能读到另一个已经提交的事务修改过的数据，但是第一次读过某条记录后，即使其他事务修改了该记录的值并且提交，该事务之后再读该条记录时，读到的仍是第一次读到的值，而不是每次都读到不同的数据。那么这种隔离级别就称之为可重复读

时间	事务一	事务二
T1	开启事务	
T2		开启事务
T3		查询余额 1000
T4	查询余额 1000	
T5		取出1000 余额 0
T6		提交
T7	查询余额 1000	
T8	提交	

串行化：以上3种隔离级别都允许对同一条记录进行读-读、读-写、写-读的并发操作，如果我们不允许读-写、写-读的并发操作，可以使用SERIALIZABLE隔离级别

时间	事务一	事务二
T1	开启事务	
T2		开启事务
T3		取出1000
T4	查询余额 等待	
T5		提交
T6	查询余额 0	
T7	提交	

读已提交和可重复读的区别

会有一个ReadView，记录着当前活动的事务，读已提交每次都会去获取ReadView，而可重复读在第一读的时候生成一个ReadView，之后一直使用该ReadView。

关于事务的隔离级别：https://blog.csdn.net/qq_38538733/article/details/88902979

mysql中事务的写法

BEGIN...COMMIT

MySQL的锁机制

锁的划分：

1. 根据粒度划分：表锁，行锁，页锁
2. 根据使用方式划分：共享锁，排它锁
3. 根据概念划分：乐观锁，悲观锁
4. 几种行级锁：Record Lock、Gap Lock、Next-key Lock
5. Record Lock：在索引记录上加锁
6. Gap Lock：间隙锁
7. Next-key Lock：Record Lock+Gap Lock

行锁

是MySQL粒度最低的锁，只针对当前行加锁。**行锁能大大减少操作数据库的冲突，但是锁的开销也最大，并且可以造成死锁。**行锁按照使用方式分为共享锁和排它锁两种。

共享锁（S锁，读锁）

如果事务A对行添加S锁，则其他事务不可以对该行进行写操作，可以进行读操作。

```
select ... lock in share mode;
```

共享锁就是可以允许多个线程持有同一个锁，同一个锁可以被多个线程持有

排它锁（X锁，写锁）

如果事务A对行添加X锁，则其他事务无法向该行添加任何锁，直到事务A释放该锁。

```
select ... for update
```

排它锁，也称作独占锁，一个锁在某一时刻只能被一个线程占有，其它线程必须等待锁被释放之后才可能获取到锁。

表锁

表级锁是MySQL中粒度最大的锁，表示对整张表加锁，锁的开销比较小，**不会发生死锁，但是性能较差，发生锁冲突的概率很大**

页锁

页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。BDB支持页级锁

关于MySQL锁：https://blog.csdn.net/qq_38238296/article/details/88362999

操作系统

进程和线程的区别

进程是资源分配的最小单位，线程是CPU调度的最小单位

1. 进程在线程中运行
2. 一个进程可以包含多个线程
3. 进程间信息较难共享，进程间比较方便
4. 进程间互不影响，一个挂了其他的可以继续正常运行
5. 进程可以在多机中使用，而线程最多是单机多核运行

进程间通信的方式

1. 管道：只能用于父子进程，半双工，有固定的收方和发方，速度慢
2. FIFO：任何进程都能通信，速度慢
3. 消息队列：任何线程之间都可以通信，通过存放消息以及读取消息来进行通信
4. 信号量：不能用来传递消息，常和共享内存同用
5. 共享内存：利用内存缓冲区直接交换消息，多个进程都可以访问，通过信号量来实现同步和通信。

线程间通信的方式

1. 使用volatile关键字，其他线程监听该值的变化
2. 通过wait()以及notify(),notityAll()方法来进行通信

Redis

Redis是什么

Redis是用C语言开发的一个开源的高性能的键值对的内存数据库，可以用作数据库，缓存，消息队列等，

Redis数据类型

1. String 是一个key-value组合，value可以是string，也可以是数字

2. Hash，是一个key-value的集合。hash特别适合存储对象，常用命令有hget，hset，hgetall
3. List，列表，Redis的列表是简单的字符串列表，按照插入顺序排序，可以添加到列表的头部或者尾部，常用命令：lpush、rpush、lpop、rpop、lrange（获取列表片段）等。
4. Set，字符串的无序集合，不允许重复
5. Zset，带score的字符串集合，根据score排序

Zset底层数据结构

Redis如何保持数据同步

当我们查询数据，先去缓存中查，缓存中有的话直接拿缓存中的数据就好了。如果缓存中没有，那么就去数据库中查询，再讲数据插入缓存中。
在读数据时，是没有什么问题的，在写入数据时，并发情况下就会出现数据库与缓存一致性问题：

1. 如果先删除缓存再写库：线程A删除了缓存，去数据库写库，这时B来查询，发现缓存中没有，就去查询并且写入数据库，此时缓存中为脏数据。
2. 如果先写库，再更新缓存，在写库完成后，宕机了，数据库与缓存中的数据也不一致了。

1. 延时双删策略

1. 先删除缓存
2. 再写库
3. 休眠一会儿
4. 再删除缓存

2. 订阅binlog

我们设置所有的读操作都在Redis中进行
所有的写操作都在数据库中进行
程序读取mysql的binlog，得到增删改查操作，之后将操作添加进消息队列等，消费队列中的操作消息来进行缓存同步。

3. 设置过期时间

设置过期时间，对数据同步是比较有利的，当key过期后，我们查询不到就会去查询数据库并更新到缓存中了。

Redis雪崩

当某一时间出现大量的Redis键失效的时候，请求一瞬间就会到数据库，那么数据库就会承受不了这么多的请求而挂掉。此时怎么重启数据库都没用了，重启的瞬间就会被打宕机，直到流量下降到数据库可以承受的程度。

如何避免Redis雪崩

在批量添加缓存的时候，给缓存的过期时间设置一个随机值，可以避免一时间大量的缓存过期。

缓存穿透

当用户不停的去请求一个数据库与缓存中都没有的数据的时候，因为缓存中没有，所以会去数据库中查询，数据库中也没有，所以不会更新缓存，就会每次发都会去请求数据库。

如何解决缓存穿透

在接口层设置校验，校验token，校验逻辑，如不符合的数据不能去缓存，还有一种布隆过滤器，它会判断数据库中是否有该数据。

缓存击穿

缓存击穿是有一个热点数据，会被经常访问，在这个数据失效的瞬间，很多请求会到数据库请求

如何解决缓存击穿

设置热点数据永不过期，或者在Redis中查不到数据后，给去数据库查询的代码块或者方法上加互斥锁，避免过多请求到数据库。

Redis为什么这么快？为什么是单线程？

因为Redis是基于内存的，绝大部分请求完全是内存操作，数据存在内存中，查询修改特别快，使用单线程，避免了线程上下文切换，不存在CPU的调度，不用考虑锁的问题，使用多路复用IO。

淘汰策略

策略	详细描述
volatile-lru	从已设置过期时间的kv集中优先淘汰最少使用的
volatile-ttl	从已设置过期时间的kv集中优先淘汰剩余时间最短的
volatile-random	从已设置过期时间的kv集中随机挑选淘汰的
allkeys-lru	从所有kv集中优先淘汰最少使用的
allkeys-ttl	从所有kv集中淘汰剩余时间最短的
noeviction	不淘汰，如果超出内存限制则报错
volatile-lfu	对设置过期时间的访问频率最低的进行淘汰
allkeys-lfu	对所有kv集中访问频率最低的进行淘汰

持久化

持久化即将缓存中的数据写入硬盘中，即使重启或者断电，数据仍然存在。Redis会周期性的进行持久化。

RDB

快照形式，直接将Redis中的数据存储到一个RDB文件中，定时保存
当需要进行备份时，Redis fork一个子进程来写入RDB文件，优点是数据恢复很快，缺点是可能会丢失一段时间的数据。

AOF

将Redis除读以外的操作命令记录到aof文件中，当进行恢复时，执行这些命令来进行恢复数据。每隔一秒或者短时间内进行写入，比RDB的数据更完整，但是AOF文件会大于RDB文件并且数据恢复慢，高频率的写入文件会造成CPU效率低。

主从复制

Redis单节点存在单点故障问题，解决单点问题一般都需要部署Redis配置从节点，主从复制的过程和原理：
主从配置结合哨兵模式能解决单点故障问题，提高Redis可用性。
从节点执行`slaveof [masterIP] [masterPort]`,保存主节点信息。之后主从节点建立socket连接，建立连接后，主节点将所有数据发送给从节点（主从同步），把当前的数据同步给从节点后，就完成了主从复制。之后就持续的将写命令发送给从节点，保持主从同步。

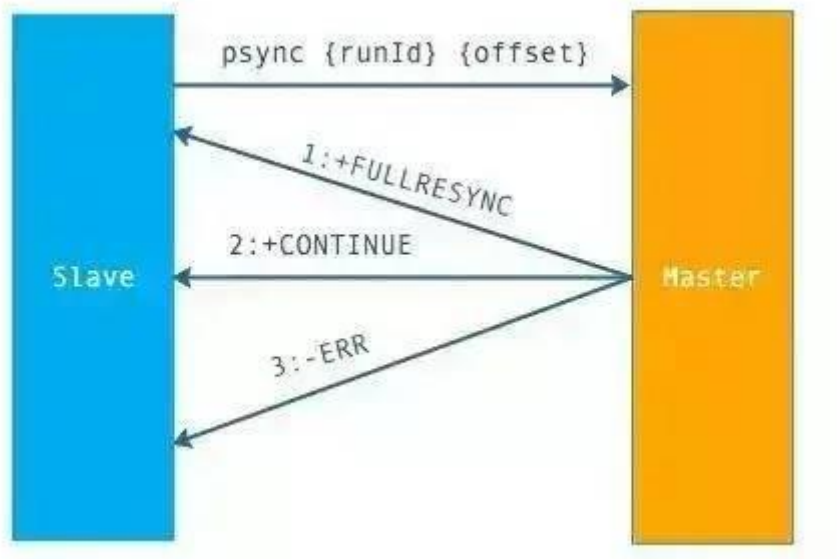
主从同步的过程

runId：每个 Redis 节点启动都会生成唯一的 uuid，每次 Redis 重启后，runId 都会发生变化。
offset：主节点和从节点都各自维护自己的主从复制偏移量 offset，当主节点有写入命令时，offset=offset+命令的字节长度。

在主节点发送数据给从节点的过程中，主节点还会进行一些写的操作，这时候数据存储在数据缓存区中。

当全量数据同步完成后，会将缓存区的数据同步给从节点。

主节点响应写命令时，不但会把命令发送给从节点，还会把数据复制在缓存区，用于复制命令失败后的数据补救。



如图，从节点发送psync {runId} {offset}命令到主节点，主节点有三种响应方式

1. FULLRESYNC：第一次连接，进行全量复制
2. CONTINUE：进行增量复制
3. ERR：可能因为偏移量大于缓冲区内存储的数据，无法进行增量复制，开始进行全量复制

具体增量复制和全量复制的过程

全量复制

一开始从节点发送 psync ? -1 （第一次不知道主节点的runId），主节点发现从节点是第一次进行主从同步，于是进行FULLRESYNC响应，FULLRESYNC {runId} {offset} 返回自己的runId和目前的offset，从节点接收到主节点信息后保存。主节点响应过后，会生成RDB文件用来全量同步从节点数据。之后向从节点发送RDB文件。

数据结构

栈

设计模式

单例模式

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

懒汉式

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

    public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

线程不安全，线程一进入到 `instance == null` 后，线程一被挂起，线程二此时进入判断，创建了实例，然后线程一继续运行，此时线程一和线程二分别持有了一个单独的实例

懒汉式线程安全

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}
```

```
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

登记式/静态内部类

```
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

饿汉式

```
public class Singleton {
        private static Singleton instance = new Singleton();
        private Singleton (){}
        public static Singleton getInstance() {
        return instance;
    }
}
```

java基础

JVM垃圾回收

JVM垃圾回收发生在堆内存中，因为堆是线程共享的，并且new出来的对象存放在堆上。

垃圾回收算法

复制算法

在垃圾回收时，将不需要回收的对象复制到另一块区域内。速度较快，但是会造成内存分区，也就是需要分出两个区域来存放复制后的对象。

标记-清除算法

标记出需要回收的对象，之后将回收的对象清除，优点是速度快，缺点是容易造成内存碎片。

标记-整理算法

标记-清除算法会造成内存碎片，于是我们在标记-清除后，将内存中的对象重排列，可以避免标记-清除算法造成的内存碎片。

Object类常用的方法

1. toString():返回一个String对象，用来标识自己
2. equals():在Object类中是比较地址值，在String中对equals()重写了
3. hashCode():Object类中返回的是地址值，可以重写该方法
4. wait():必须卸载synchronize代码块中，因为wait()会释放锁，如果没有锁，就没办法释放了
5. notify():唤醒等待队列的第一个线程
6. notifyAll():唤醒等待队列的所有线程
7. finalize():在对象被回收时调用

Java实现线程同步的方式

1. synchronize同步方法

使用synchronize修饰方法，那么当一个线程获得该锁后，其余线程要等待该线程释放锁才能运行

```
public class Bank {
    private int count = 0;// 账户余额

    // 存钱
    public synchronized void addMoney(int money) {
        count += money;
        System.out.println(System.currentTimeMillis() + "存进：" + money);
    }

    // 取钱
    public synchronized void subMoney(int money) {
        if (count - money < 0) {
            System.out.println("余额不足");
            return;
        }
        count -= money;
        System.out.println(+System.currentTimeMillis() + "取出：" + money);
    }

    // 查询
    public void lookMoney() {
        System.out.println("账户余额：" + count);
    }
}
```

2. 同步代码块

使用同步方法对性能影响过大，粒度太大，我们可以选择同步代码块。

```
public class Bank {
    private int count = 0;// 账户余额

    // 存钱
    public  void addMoney(int money) {
        synchronized(this){
            count += money;
        }

        System.out.println(System.currentTimeMillis() + "存进：" + money);
    }

    // 取钱
    public  void subMoney(int money) {
        synchronized(this){
            if (count - money < 0) {
                System.out.println("余额不足");
                return;
            }
            count -= money;
        }

        System.out.println(+System.currentTimeMillis() + "取出：" + money);
    }

    // 查询
    public void lookMoney() {
        System.out.println("账户余额：" + count);
    }
}
```

Volatile关键字

并不能保证原子性，在多线程情况下对库存或者价格进行修改可能会出问题，它通知虚拟机该变量可能会被修改，强制当前线程修改后的volatile的变量的值写入内存，并且使其他线程的该变量的值失效。
为什么会失效呢?

```
    private volatile int a = 1;
    a ++;
    // 同时两个线程进行，可能会得到2，
    temp = 1 + 1;
    a = temp;
    // 两个线程同时获得a，相加后给a赋值，那么就得到了2
```

思考，可以使用boolean值，只涉及赋值操作来进行线程间同步

Lock接口

Lock接口提供了获得锁和释放锁的方法，ReentrantLock类为Lock的实现类。使用`lock()`或者`tryLock()`方法获取锁，一定要注意需要关闭锁`unLock()`

ThreadLocal

ThreadLocal是什么

线程变量，是一个类，它有set()和get()方法，set()的值存在当前线程内。其内填充的变量属于当前线程，对于其他线程是不可见的。

threadLocal的set()方法：

```
public void set(T value) {
        // 获取当前线程
        Thread t = Thread.currentThread();
        // 获取ThreadLocalMap，map在Thread内
        // 这个map的键是ThreadLocal，值就是我们set的值
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
    }
```

ThreadLocal的get()方法：

```
public T get() {
        // 获取当前线程
        Thread t = Thread.currentThread();
        // 根据当前线程获取线程内的map
        ThreadLocalMap map = getMap(t);
        // 然后下面就是获取值啦
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }
```

ThreadLocal总结

1. ThreadLocal类并不存储数据，他只是作为Thread的ThreadLocalMap中的键
2. ThreadLocalMap是ThreadLocal的一个内部类，用Entity存储数据
3. 每个Thread中存在一份ThreadLocalMap的引用

ThreadLocal内存泄漏问题

如上我们知道了，ThreadLocalMap存储在Thread中，ThreadLocal只是一个键，那么当ThreadLocal为null，需要被垃圾回收器回收，但是此时ThreadLocalMap的键没了，值还在，因为

map存储在Thread中，只要Thread没挂掉，就仍然存在，所以此时的现象是：ThreadLocalMap的key没了，但是value还在，这就造成了内存泄漏。
解决办法是每次使用完之后记得remove掉。

线程的状态

1. 初始：新创建了线程对象，但是没有调用其start()方法
2. 运行：分为就绪和运行两个状态，就绪状态下，等待CPU调度；运行中是已经获取到了CPU时间片，正在运行中
3. 阻塞：阻塞在等待其他线程释放synchronized锁的过程
4. 等待：调用Object.wait()或者Thread.join()方法，等待其他线程的特定动作：notify()等。
5. 超时等待：wait(timeout)在经过设置时间后，会自动唤醒
6. 终止：run()方法调用完后，线程结束。

HashMap详解

数据结构

底层数据结构是一个哈希表

hash冲突

采用拉链的方式解决冲突，JDK1.7以前使用头插法，JDK1.8及以后采用尾插法。在链表数据大于8时转换成红黑树，链表中元素为6则退化成链表

HashMap的并发问题

1. 多线程扩容可能会造成死循环
2. 多线程put可能会丢值
3. 多线程情况下put后get为null

1.8前后对比

1.8以前上述三个情况都可能发生
1.8及以后不会造成死循环，但是其他两种情况可能会出现

HashMap的键值可以为null吗

可以，如果为null则将元素放到数组的第一个位置

用可变类作为HashMap的键会有什么问题

当put进元素后，修改了键对象，那么导致get时重算hashcode，导致无法取到元素。

为什么扩容时可能会死循环

```
void transfer(Entry[] newTable)
{
    Entry[] src = table;
    int newCapacity = newTable.length;
    //下面这段代码的意思是：
    //  从OldTable里摘一个元素出来，然后放到NewTable中
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            } while (e != null);
        }
    }
}
```

以上是hashMap扩容时复制元素到新table的过程
如果当线程一运行到*Entry<K,V> next = e.next;*，此时线程二开始扩容，并且完成了扩容，那么造成的问题就是，可能这时e.next.next = e；当线程一继续扩容时就会现将e放到newTable，然后放e.next，而此时e.next.next = e e.next = e.next，造成了死循环。

为什么put进去可能为null

还是上面的代码块，*src[j] = null;*，线程一将table的j位置职位null，然后线程二正好开始get()，就get到了null

为什么put进去的元素可能被覆盖

这个比较好理解，1.8以后采用尾插法，那么如果线程一和线程二得到的数组同一位置的链表尾部元素相等，那么就会造成同一个尾部e.next = 线程一.entity，线程二此时 e.next = 线程二.next;

HashMap相关：https://blog.csdn.net/qq_37217713/article/details/102659570

JAVA的锁相关

乐观锁和悲观锁

乐观锁和悲观锁是一种锁的实现思想。

悲观锁

悲观锁认为同一个数据的并发操作一定是会被其他线程修改的，所以对于并发的数据，采用加锁的方式。悲观的认为数据一定会被修改

乐观锁

乐观锁认为数据不一定会被其他线程修改，所以现将数据读取到，修改后比较数据是否与读数据时的状态一致，如果一致则提交修改，不一致则不进行更新。

公平锁和非公平锁

公平锁

公平锁指的是多个线程按请求的顺序来获取锁

非公平锁

非公平锁指的是多个线程并不按照请求的顺序来获取锁，可能后请求的线程先获取锁。

独占锁和共享锁

独占锁

类似mysql的写锁，只允许一个线程占有锁

共享锁

可以有多个线程持有锁并读取数据，但是只能有一个做出修改 CAS：Compare and Swap，基于乐观锁的思想，得到内存中的数据后，记录下来，当对该数据进行修改时，需要比较内存中的数据是否和记录下来的预期值相同，如果不同的话，再次获取该变量，然后再次修改，再次比较，直到比较相同，则提交。

wait()&sleep()的区别

1. sleep()是线程类的方法，wait()是object的方法
2. sleep()不会释放锁，wait()会释放锁，如果不设置时间，wait()需要被notify()
3. wait()需要写在synchronized里，而sleep()不需要
4. wait()需要被唤醒，sellp()在经过设置时间后会再运行。
5. wait()如果设置了时间，到了时间也会自动唤醒

算法

二叉树层序遍历

```
public void levelOrder(TreeNode root) {
        List<TreeNode> list = new LinkedList<>();
        list.add(root);
        while (!list.isEmpty()) {
            TreeNode node = list.remove(0);
            if (node.left != null) {
                list.add(node.left);
                next += 1;
            }
            if (node.right != null) {
```

```
                list.add(node.right);
                next += 1;
            }
            System.out.println(node.val);
        }
    }
```

从右侧看一颗二叉树

层序遍历取每层的最后一个

```
public void levelOrder(TreeNode root) {
        List<TreeNode> list = new LinkedList<>();
        list.add(root);
        int cur, next;
        cur = 1;
        next = 0;
        while (!list.isEmpty()) {
            TreeNode node = list.remove(0);
            cur--;
            if (node.left != null) {
                list.add(node.left);
                next += 1;
            }
            if (node.right != null) {
                list.add(node.right);
                next += 1;
            }
            if (cur == 0) {
                cur = next;
                next = 0;
                System.out.println(node.val);
            }
        }
    }
```

链表反转

采用头结点，指向首元结点，之后新增一个头结点，头插法

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode n = new ListNode(0);
        while (head != null) {
```

```
            ListNode next = head.next;
            head.next = n.next;
            n.next = head;
            head = next;
        }
        return n.next;
    }
}
```

求平方根

```
package binarySearch;

public class Sqrt {
    public static void main(String[] args) {
        System.out.println(Sqrt.sqrt(25));
    }
    public static double sqrt(double num) {
        double left = 0;
        double right = num;
        double mid = (left + right) / 2;
        while (left < right) {
            if (mid * mid == num) {
                break;
            }
            if (mid * mid < num) {
                left = mid;
                mid = (left + right) / 2;
            } else {
                right = mid;
                mid = (left + right) / 2;
            }
        }
        return mid;
    }
}
```

字符串的全排列

```
package orther;

public class QuanPaiLie {
    public static void main(String[] args) {
        char[] arr = {'a', 'b', 'c', 'd'};
        perm(arr, 0, arr.length);
    }

    public static void swap(char[] arr, int i, int j) {
        if (i == j) {
            return;
        }
        char temp = arr[i];
```

```
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void perm(char[] arr, int start, int end){
        if (start == end){
            System.out.println(arr);
        } else {
            for (int i = start; i < end; i++) {
                swap(arr, i, start);
                perm(arr, start + 1, end);
                swap(arr, i, start);
            }
        }
    }
}
```

二叉树的最近公共祖先

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || root == p || root == q) {
            return root;
        }
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if (left == null && right == null) {
            return null;
        }
        if (left == null) {
            return right;
        }
        if (right == null) {
            return left;
        }
        return root;
    }
```

旋转数组找最小值

```
public int minValue(int[] numbers) {
        int left = 0;
        int right = numbers.length - 1;
        int mid = (left + right) / 2;
        while (left < right) {
            if (numbers[mid] < numbers[right]) {
                right = mid;
            } else if (numbers[mid] > numbers[right]) {
                left = mid + 1;
            } else {
                right -= 1;
            }
            mid = (left + right) / 2;
```

```
        }
        return numbers[left];
}
```

二叉树的镜像

```
// 递归进行，当越过叶子结点返回null
// 递归更换root的左右子节点
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if (root == null) {
            return null;
        }
        TreeNode left = root.left;
        root.left = mirrorTree(root.right);
        root.right = mirrorTree(left);
        return root;
    }
}
```