目录

[架构 3](#_Toc4931169)

[RPC (Dubbo) 3](#_Toc4931170)

[负载均衡 3](#_Toc4931171)

[消息队列 3](#_Toc4931172)

[Netty 3](#_Toc4931173)

[Redis 3](#_Toc4931174)

[JAVA 3](#_Toc4931175)

[JVM 3](#_Toc4931176)

[GC 4](#_Toc4931177)

[内存 5](#_Toc4931178)

[常用类 5](#_Toc4931179)

[并发 5](#_Toc4931180)

[锁机制 5](#_Toc4931181)

[线程 6](#_Toc4931182)

[框架 6](#_Toc4931183)

[Spring 6](#_Toc4931184)

[Spring MVC 6](#_Toc4931185)

[Hibernate 6](#_Toc4931186)

[数据结构 10](#_Toc4931187)

[Heap 10](#_Toc4931188)

[最大堆建堆，插入和删除 11](#_Toc4931189)

[海量数据topk 11](#_Toc4931190)

[树 12](#_Toc4931191)

[平衡二叉树（AVL） 12](#_Toc4931192)

[红黑树 （旋转调整） 12](#_Toc4931193)

[B树 13](#_Toc4931194)

[B+树 14](#_Toc4931195)

[B树B+树应用场景比较（典型：数据库索引） 14](#_Toc4931196)

[栈 14](#_Toc4931197)

[队列 15](#_Toc4931198)

[链表 15](#_Toc4931199)

[Map 16](#_Toc4931200)

[Hashtable: 16](#_Toc4931201)

[Hashmap: 16](#_Toc4931202)

[Hashmap与Hashtable的区别 16](#_Toc4931203)

[怎么保证map有序？-（LinkedHashmap） 17](#_Toc4931204)

[TreeMap 18](#_Toc4931205)

[设计模式 19](#_Toc4931206)

[单例模式 19](#_Toc4931207)

[数据库 20](#_Toc4931208)

[解析一条SQL语句 20](#_Toc4931209)

[常用引擎 20](#_Toc4931210)

[分类 20](#_Toc4931211)

[区别 20](#_Toc4931212)

[事务隔离 20](#_Toc4931213)

[锁机制 20](#_Toc4931214)

[索引 20](#_Toc4931215)

[计算机网络 20](#_Toc4931216)

[解析一条URL 20](#_Toc4931217)

[TCP/UDP 异同 20](#_Toc4931218)

[TCP/UDP 包头 20](#_Toc4931219)

[TCP连接建立(标识符，发包算法) 20](#_Toc4931220)

[TCP流量控制，拥塞避免 20](#_Toc4931221)

[七层协议/五层协议 20](#_Toc4931222)

[操作系统 23](#_Toc4931223)

[进程线程 23](#_Toc4931224)

[线程间通信 23](#_Toc4931225)

[进程间通信 23](#_Toc4931226)

[地址与内存 23](#_Toc4931227)

[中断 23](#_Toc4931228)

[上下文切换 23](#_Toc4931229)

[进程状态 23](#_Toc4931230)

[进程调度（算法） 23](#_Toc4931231)

# 架构

## RPC (Dubbo)

## 负载均衡

## 消息队列

## Netty

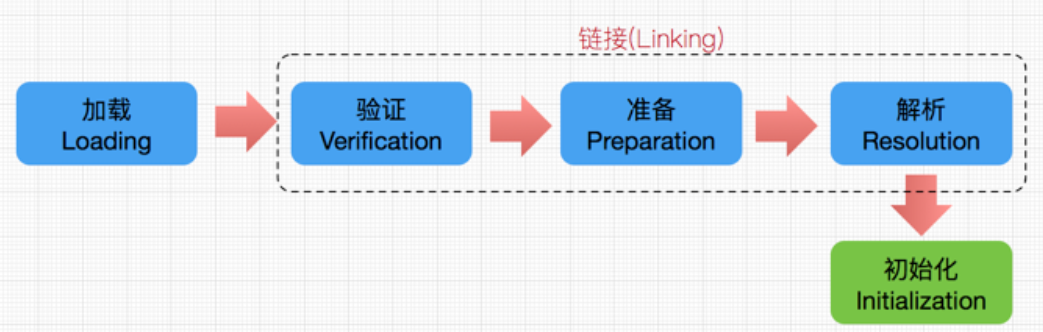
## Redis

# JAVA

## JVM

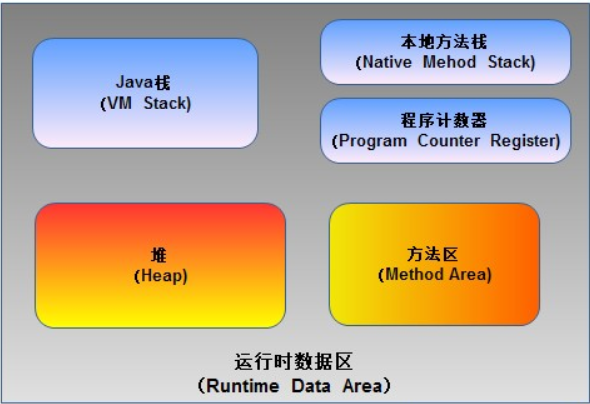
* 1. 类加载原理/加载机制

加载阶段 java =>.class文件， 然后经过字节码校验器，解释器或者是JIT.



**双亲委派模式**是在Java 1.2后引入的，其工作原理的是，如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式

* 1. JVM组成



蓝色区域为线程私有，

运行时每个线程都会分配一个栈

私有区域：

程序计数器，保存线程在运行代码的时候，具体运行导那个代码的指针。

本地方法栈： 实现由其他语言实现：（一些native的方法由C语言编写的库提供 ,object类中hashash（）就是native方法）执行的时候调用本地方法接口（xx.dll）c语言的库文件。

栈，包括方法出口，方法，栈帧。

公有区域：

堆：new 对象 （新生代占了1/3的空间，eden占其中8/10， survive占2/10； 老年代占2/3）

方法区“存放静态变量，常量，类信息，运行时常量池

* 1. JVM优化思想（参数优化）

## GC

* 1. 新生代主要有哪些收集算法（说了Serial、ParNew、Parallel Scvenge）

Serial收集器串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。新生代、老年代使用串行回收；新生代复制算法、老年代标记-压缩；垃圾收集的过程中会Stop The World（服务暂停）

ParNew收集器ParNew收集器其实就是Serial收集器的多线程版本。新生代并行，老年代串行；新生代复制算法、老年代标记-压缩参数控制：-XX:+UseParNewGC  ParNew收集器-XX:ParallelGCThreads 限制线程数量参数控制：-XX:+UseSerialGC  串行收集器

Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。可以通过参数来打开自适应调节策略，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制GC的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-压缩

* 1. 老年代主要的收集算法(说了Serial Old、Parallel Old、CMS)

5.1 Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法。这个收集器是在JDK 1.6中才开始提供参数控制： -XX:+UseParallelOldGC 使用Parallel收集器+ 老年代并行

5.2 CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。 其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。优点:并发收集、低停顿  缺点：产生大量空间碎片、并发阶段会降低吞吐量

* 1. 内存不够用时会回收软引用，那什么时候会发生这种事
  2. CMS垃圾回收过程：

初始标记（CMS initial mark）并发标记（CMS concurrent mark）重新标记（CMS remark）并发清除（CMS concurrent sweep）

* 1. FullGC次数太多了如何优化

根本原因是CMS请求消耗的内存量过大

有2种情况会触发full gc，在full gc时，整个应用会暂停

a）concurrent-mode-failure：当cms gc正进行时，此时有新的对象要进行old代，但是old代空间不足造成的

b）promotion-failed：当进行young gc时，有部分young代对象仍然可用，但是S1或S2放不下，因此需要放到old代，但此时old代空间无法容纳此

（1）针对cms gc的触发阶段，调整-XX:CMSInitiatingOccupancyFraction=50，提早触发cms gc，就可以 缓解当old代达到80%，cms gc处理不完，从而造成concurrent mode failure引发full gc

（2）修改-XX:CMSMaxAbortablePrecleanTime=500，缩小CMS-concurrent-abortable-preclean阶段 的时间

（3）考虑到cms gc时不会进行compact，因此加入-XX:+UseCMSCompactAtFullCollection （cms gc后会进行内存的compact）和-XX:CMSFullGCsBeforeCompaction=4 （在full gc4次后会进行compact）参数

## 内存

## 常用类

## 并发

Synchronized, Lock(看lock包，concurrent包， signal, await, condition)

## 锁机制

## 线程

## 框架

### Spring

### Spring MVC

### Hibernate

* 1. JVM OOM 如何定位 现在有个系统发现内存使用不断上升，OOM 异常，定位问题。回答 查日志。面试官：日志没有报错。查看dump 文件，怎么查看，具体命令记得吗，答jstack  具体怎么用的。。。 尴尬了。。如何查看dump文件，具体怎么查看定位。

JVM 发生 OOM 时，会自动在 /var/log/abc 目录下产生堆 dump 文件 java\_pidPID.hprof

这些文件记录了JVM运行期间的内存占用、线程执行等情况，这就是我们常说的dump文件。常用的有heap dump和thread dump（也叫javacore，或java dump）。我们可以这么理解：heap dump记录内存信息的，thread dump是记录CPU信息的。

Jmap => heap dump文件 Jstack => thread dump 文件

* 1. Java几种引用对象

强

软

弱

虚

* 1. JDK版本, 迭代历史中有什么变化
  2. JDK 代码大致了解

Throwable框架

一个 Throwable 对象包含线程创建到执行至异常处的所有栈。同时也包括一个额外的 message 信息，来描述异常。最重要的成员：private StackTraceElement[] stackTrace = UNASSIGNED\_STACK;其中一个StackTraceElement代表一个栈帧，最顶端的StackTraceElement代表异常抛出地点，其余每个StackTraceElement都表示一个方法调用。

* 1. 并发编程遵守几个性质

原子性：即一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

可见性：是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性即程序执行的顺序按照代码的先后顺序执行。

* 1. Volatile 和 Synchronized（i++是复杂操作 不是原子的 Volatile没有原子性）
  2. ReentrantLock（可重入锁）和Synchronized区别

两者都是同一个线程每进入一次，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。Synchronized是依赖于JVM实现的，而ReenTrantLock是JDK实现的，有什么区别，说白了就类似于操作系统来控制实现和用户自己敲代码实现的区别。很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。

ReenTrantLock特征

1.      ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。2.      ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。3.      ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。

* 1. Synchronized 和 lock的区别

类别 synchronized Lock**存在层次** Java的关键字，在jvm层面上 || 是一个类**锁的释放** 1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm会让线程**释放锁**  || 在finally中必须释放锁，不然容易造成线程死锁**锁的获取** 假设A线程获得锁，B线程等待。如果A线程阻塞，B线程会一直等待 分情况而定，Lock有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待。

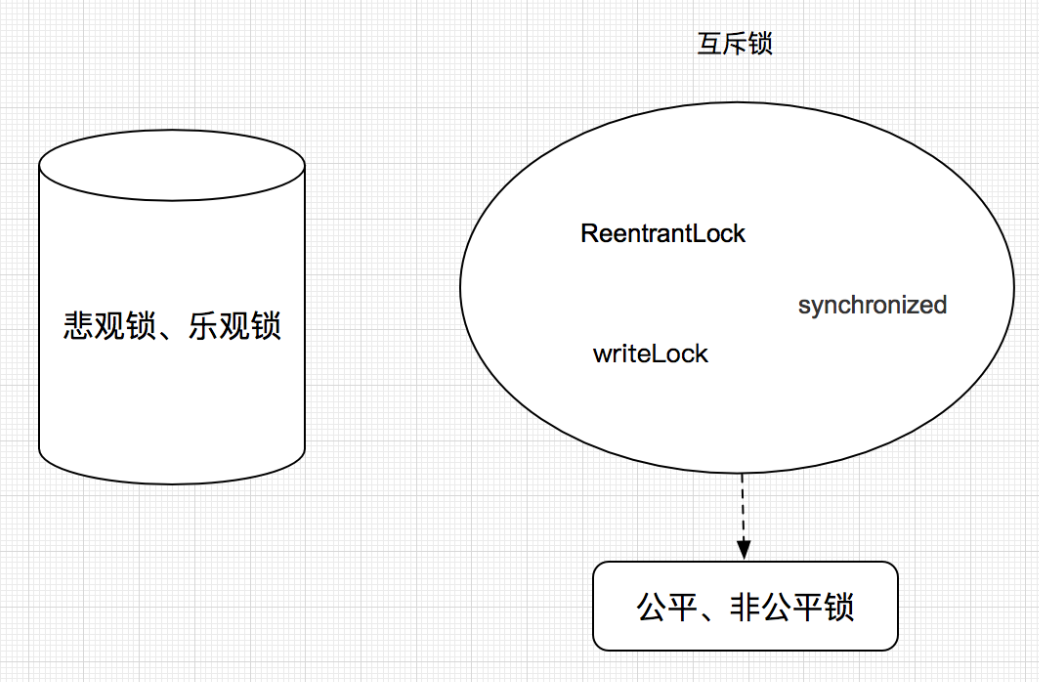
**锁状态** 无法判断 可以判断**锁类型** 可重入 不可中断 非公平 可重入 可判断 可公平（两者皆可）**性能**  少量同步 大量同步

* 1. 线程创建的几种方法
     1. 继承Thread类
     2. 实现Runnable接口
     3. 通过Callable和Future创建线程
  2. 并发 知道Future 吗， 实现线程的方式，哪种有返回值，使用Future 来接收，Future怎么用的，简单介绍下。

经常使用的Thread的Runnable()虽然被经常使用，但其有一个弊端，就是因为无法直接获取该线程的返回值，因为Runnable内的run方法，被定义为void类型.

从Java 5开始，Java在语言层级增加了支持线程返回结果的Future、Callable，用以支持和解决上述问题，完事线程编程模型。 用一个一个Future<string> 声明的值去或者一个线程submit的结果，用arraylist 来维护。

* 1. 乐观锁悲观锁



悲观锁、乐观锁使用场景是针对数据库操作来说的，是一种锁机制。**悲观锁(Pessimistic Lock)**：顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

**乐观锁(Optimistic Lock**)：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

* 1. 线程池的原理，corepoolsize，maxpoolsize和queue

Executors是java线程池的工厂类，通过它可以快速初始化一个符合业务需求的线程池

corePoolSize线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。maximumPoolSize线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize； 如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

* 1. 线程池有哪些类型，single cached和scheduled的各应用场景

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，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。newFixedThreadPool创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。newScheduledThreadPool创建一个定长线程池，支持定时及周期性任务执行。

* 1. 线程池的核心线程数如何设置，选择标准是什么
  2. CAS操作，CAS操作数据的有什么要求吗(这块没答好)

CAS是单词compare and set的缩写，意思是指在set之前先比较该值有没有变化，只有在没变的情况下才对其赋值。在没有锁的情况下实现了a++操作，这实际上是一种非阻塞算法。

CAS缺点

1、ABA问题CAS操作容易导致ABA问题,也就是在做a++之间，a可能被多个线程修改过了，只不过回到了最初的值，这时CAS会认为a的值没有变。a在外面逛了一圈回来，你能保证它没有做任何坏事，不能！！也许它讨闲，把b的值减了一下，把c的值加了一下等等，更有甚者如果a是一个对象，这个对象有可能是新创建出来的，a是一个引用呢情况又如何，所以这里面还是存在着很多问题的，解决ABA问题的方法有很多，可以考虑增加一个修改计数，只有修改计数不变的且a值不变的情况下才做a++，也可以考虑引入版本号，当版本号相同时才做a++操作等，这和事务原子性处理有点类似！2、比较花费CPU资源，即使没有任何争用也会做一些无用功。3、会增加程序测试的复杂度

* 1. 线程资源怎么回收

1. 设置线程池线程最大空闲时间，超出这个时间，对线程进行回allowCoreThreadTimeOut(true);
2. 关闭线程池（关闭之后，有新任务却不会执行）shutdown();
   1. Cookie原理 和 session

1、创建Cookie当用户第一次浏览某个使用Cookie的网站时，该网站的服务器就进行如下工作：①该用户生成一个唯一的识别码（Cookie id），创建一个Cookie对象；②默认情况下它是一个会话级别的cookie，存储在浏览器的内存中，用户退出浏览器之后被删除。如果网站希望浏览器将该Cookie存储在磁盘上，则需要设置最大时效（maxAge），并给出一个以秒为单位的时间（将最大时效设为0则是命令浏览器删除该Cookie）；③将Cookie放入到HTTP响应报头，将Cookie插入到一个 Set-Cookie HTTP请求报头中。④发送该HTTP响应报文。2、设置存储Cookie浏览器收到该响应报文之后，根据报文头里的Set-Cookied特殊的指示，生成相应的Cookie，保存在客户端。该Cookie里面记录着用户当前的信息。3、发送Cookie当用户再次访问该网站时，浏览器首先检查所有存储的Cookies，如果某个存在该网站的Cookie（即该Cookie所声明的作用范围大于等于将要请求的资源），则把该cookie附在请求资源的HTTP请求头上发送给服务器。4、读取Cookie 服务器接收到用户的HTTP请求报文之后，从报文头获取到该用户的Cookie，从里面找到所需要的东西。

* 1. Cookie和session 的不同

1、存放位置不同Cookie保存在客户端，Session保存在服务端。2 、存取方式的不同 Cookie中只能保管ASCII字符串，假如需求存取Unicode字符或者二进制数据，需求先进行编码。Cookie中也不能直接存取Java对象。若要存储略微复杂的信息，运用Cookie是比拟艰难的。 而Session中能够存取任何类型的数据，包括而不限于String、Integer、List、Map等。Session中也能够直接保管Java Bean乃至任何Java类，对象等，运用起来十分便当。能够把Session看做是一个Java容器类

* 1. 如果服务器是分布式的，如何应对Session的一致问题

什么是session一致性问题？只要用户不重启浏览器，每次http短连接请求，理论上服务端都能定位到session，保持会话。

1.session复制（同步）多个web-server之间相互同步session，这样每个web-server之间都包含全部的session2.客户端存储法 服务端存储所有用户的session，内存占用较大，可以将session存储到浏览器cookie中，每个端只要存储一个用户的数据了3.反向代理hash一致性 思路：web-server为了保证高可用，有多台冗余，反向代理层能不能做一些事情，让同一个用户的请求保证落在一台web-server上呢？

* 1. Object类有哪些方法？（说到Clone()方法被叫停）
  2. 那你认为有哪些深拷贝的方法？

  浅拷贝：使用一个已知实例对新创建实例的成员变量逐个赋值，这个方式被称为浅拷贝。  深拷贝：当一个类的拷贝构造方法，不仅要复制对象的所有非引用成员变量值，还要为引用类型的成员变量创建新的实例，并且初始化为形式参数实例值，这个方式称为深拷贝。

浅拷贝只复制一个对象，传递引用，不能复制实例。而深拷贝对对象内部的引用均复制，它是创建一个新的实例，并且复制实例。JAVA中Clone（）是浅拷贝

利用串行化来做**深复制：**把对象写到流里的过程是串行化（Serilization）过程，但是在Java程序师圈子里又非常形象地称为“冷冻”或者“腌咸菜（picking）”过程；而把对象从流中读出来的并行化（Deserialization）过程则叫做 “解冻”或者“回鲜(depicking)”过程。

* 1. 你认为两个对象的hashCode一样，是同一个对象吗？反过来呢？

1、相等（相同）的对象必须具有相等的哈希码（或者散列码）。2、如果两个对象的hashCode相同，它们并不一定相同。（哈希过程中产生冲突）

* 1. wait()、notify()、notifyAll()的使用场景有哪些？

（说了生产者、消费者模式） 等待生产 ， notify ,或者notifyAll.

Notify和NotifyAll的区别

调用了notify后只要一个线程会由等待池进入锁池，而notifyAll会将该对象等待池内的所有线程移动到锁池中，等待锁竞争

* 1. wait()、notify()、notifyAll()和JDK 1.5中的并发包Lock有什么区别？
  2. 线程的sleep方法和object类的wait方法有什么区别

# 数据结构

## Heap

**最大最小堆, 建堆时间(部分数据topk+海量数据topk)**

建堆时间： 建个n结点的堆，需要O（n），插入的average是O(1), worst case 是O（logn）

删除复杂度是O(logn)

### 最大堆建堆，插入和删除

class maxheap {

private int[] data;

private int size; //当前堆大小

private int capability;//堆容量

maxheap(int max){

this.data = new int[max+1];

this.capability = max ;

this.size=0;

}

//增

public boolean **insert**(int d) {

if(size == capability) {

return false;

}

data[size+1]=d;

size = size+1 ;

if(shiftup(d))

return true;

else

return false;

}

public boolean **shiftup**(int d) {

int i = size;

while(i>1&&data[i]>data[i/2]) {

int temp = data[i];

data[i] = data[i/2];

data[i/2]= temp;

i = i/2;

}

return true;

}

//删 （取）

public int **delete**() {

if(size==0) {

return -1;

}

int elements = data[1];

data[1] = data[size];

size --;

shiftdown(1);

return elements;

}

public void **shiftdown**(int i ) {

if(size ==i ) {

return;

}

while(i\*2 <= size) {

int j= i\*2;

if(j+1<=size&&data[j+1]>data[j]) {

j=j+1;

}

//如果子都小于结点 就结束 ， 不继续下面的交换循环

if(data[i]>data[j])

break;

int temp = data[i];

data[i]= data[j];

data[j]=temp;

i =j;

}

}

}

### 海量数据topk

        在大规模数据处理中，经常会遇到的一类问题：在海量数据中找出出现频率最好的前k个数，或者从海量数据中找出最大的前k个数，这类问题通常被称为top K问题。例如，在搜索引擎中，统计搜索最热门的10个查询词；在歌曲库中统计下载最高的前10首歌等。

        针对top K类问题，通常比较好的方案是分治+Trie树/hash+小顶堆（就是上面提到的最小堆），即先将数据集按照Hash方法分解成多个小数据集，然后使用[Trie树](#_TreeMap)或者Hash统计每个小数据集中的query词频，之后用小顶堆求出每个数据集中出现频率最高的前K个数，最后在所有top K中求出最终的top K。

## 树

### 平衡二叉树（AVL）

判断树的深度：

public int TreeDepth(TreeNode root) {

if(root==null) return 0;

return Math.max(1+TreeDepth(root.left),1+TreeDepth(root.right));

}

判断一个树是否是平衡二叉树（左右两边最大最小深度一样）：

public int nodedepth(TreeNode root) { //最大深度

if(root == null) return 0;

return Math.max(1+nodedepth(root.left),1+nodedepth(root.right));

}

public int nodemindepth(TreeNode root) { //最小深度

if(root == null) return 0;

return Math.min(1+nodedepth(root.left),1+nodedepth(root.right));

}

// 判断是否是平衡二叉树

public boolean IsBalanced\_Solution(TreeNode root) {

if(root==null) return true;

return nodedepth(root)- nodemindepth(root) >1 ? false : true;

}

**与红黑树性质比较**

[下一小节](#_红黑树_（旋转调整）)

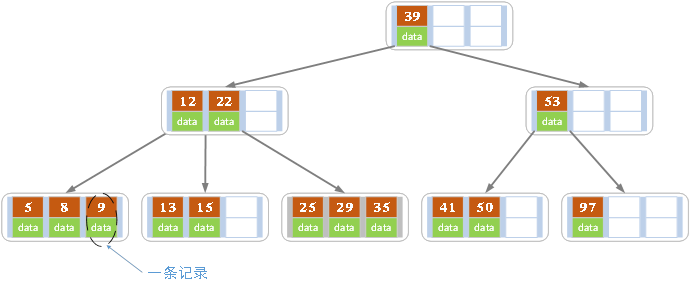
### 红黑树 （旋转调整）

**红黑树性质：**

一般的，红黑树，满足以下性质，即只有满足以下全部性质的树，我们才称之为红黑树：  
1）每个结点要么是红的，要么是黑的。  
2）根结点是黑的。  
3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。  
4）如果一个结点是红的，那么它的俩个儿子都是黑的。  
5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

**红黑树的各种操作的时间复杂度是多少？**能保证在最坏情况下，基本的动态几何操作的时间均为O（lgn）**红黑树相比于BST和AVL树有什么优点？**红黑树是牺牲了严格的高度平衡的优越条件为代价，它只要求部分地达到平衡要求，降低了对旋转的要求，从而提高了性能。红黑树能够以O(log2 n)的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较“便宜”的解决方案。相比于BST，因为红黑树可以能确保树的最长路径不大于两倍的最短路径的长度，所以可以看出它的查找效果是有最低保证的。在最坏的情况下也可以保证O(logN)的，这是要好于二叉查找树的。因为二叉查找树最坏情况可以让查找达到O(N)。**红黑树的算法时间复杂度和AVL相同**，但统计性能比AVL树更高，所以在插入和删除中所做的后期维护操作肯定会比红黑树要耗时好多，但是他们的查找效率都是O(logN)，所以红黑树应用还是高于AVL树的. 实际上插入 AVL 树和红黑树的速度取决于你所插入的数据.如果你的数据分布较好,则比较宜于采用 AVL树(例如随机产生系列数),但是如果你想处理比较杂乱的情况,则红黑树是比较快的

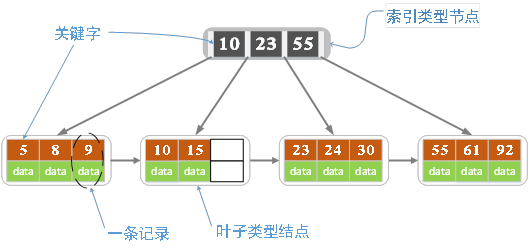
### B树



B树也称B-树,它是一颗多路平衡查找树。我们描述一颗B树时需要指定它的阶数，阶数表示了一个结点最多有多少个孩子结点，一般用字母m表示阶数。当m取2时，就是我们常见的二叉搜索树。一颗m阶的B树定义如下：1）每个结点最多有m-1个关键字。2）根结点最少可以只有1个关键字。3）非根结点至少有Math.ceil(m/2)-1个关键字。4）每个结点中的关键字都按照从小到大的顺序排列，每个关键字的左子树中的所有关键字都小于它，而右子树中的所有关键字都大于它。5）所有叶子结点都位于同一层，或者说根结点到每个叶子结点的长度都相同。

### B+树

B+树是应文件系统所需而产生的一种B树的变形树(文件的目录一级一级索引,只有最底层的叶子节点(文件)保存数据.),非叶子节点只保存索引,不保存实际的数据,数据都保存在叶子节点中.这不就是文件系统文件的查找吗?我们就举个文件查找的例子:有3个文件夹,a,b,c, a包含b,b包含c,一个文件yang.c, a,b,c就是索引(存储在非叶子节点), a,b,c只是要找到的yang.c的key,而实际的数据yang.c存储在叶子节点上. 所有的非叶子节点都可以看成索引部分



### B树B+树应用场景比较（典型：数据库索引）

B和**B+**树主要用在文件系统以及数据库做索引**.**比如**Mysql;(**为什么**)**

1、 B+树的磁盘读写代价更低：B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。

2、B+树的查询效率更加稳定：由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

3、由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。 [数据库索引](#_索引)

## 栈

定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的min函数（时间复杂度应为O（1））

import java.util.Stack;

public class Solution {

Stack<Integer> re = new Stack<Integer>();

Stack<Integer> min = new Stack<Integer>();

int mintmp= Integer.MAX\_VALUE;

**public void push(int node) {**

**re.push(node);**

**if(node<mintmp) {**

**mintmp=node;**

**min.push(node);**

**}else {**

**min.push(mintmp);**

**}**

}

public void pop() {

re.pop();

min.pop();

}

public int top() {

return re.peek();

}

public int min() {

return min.peek();

}

}

## 队列

优先队列(底层是堆)， 改写比较器，可用于topk题目。

## 链表

1. 在O(1)时间删除链表节点下一个结点数据覆盖需要删除的结点，但是不能是尾节点
2. 单链表的转置用三个临时指针 pre、head、next 在链表上循环一遍即可。

public ListNode ReverseList(ListNode head) {

ListNode next ,pre;

pre = null;

if (head == null)

return head;

while (head!=null) {

next = head .next;

head.next = pre;

pre = head;

head = next ;

}

return pre;

}

1. 找倒数k个结点

快慢指针， 距离为k

1. 判断是不是环

快慢指针 两倍速相遇

1. 判断环入口

两倍速， 相遇后， 快指针从头开始变一倍速

## Map

### **Hashtable:**

HashTable的主要方法的源码实现逻辑，与HashMap中非常相似，有一点重大区别就是所有的操作都是通过**synchronized**锁保护的。只有获得了对应的锁，才能进行后续的读写等操作。

### **Hashmap:**

底层实现：在JDK1.6，JDK1.7中，HashMap采用位桶+链表实现，即使用链表处理冲，而JDK1.8中，HashMap采用位桶+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树。通过hash的方法，通过put和get存储和获取对象。如果发生碰撞的时候，Hashmap通过链表将产生碰撞冲突的元素组织起来，在Java 8中，如果一个bucket中碰撞冲突的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。

**Hashmap可能存在的安全问题**：

如果多个线程同时使用 put 方法添加元素，而且假设正好存在两个 put 的 key 发生了碰撞（根据 hash 值计算的 bucket 一样），那么根据 HashMap 的实现，这两个 key 会添加到数组的同一个位置，这样最终就会发生其中一个线程 put 的数据被覆盖如果多个线程同时检测到元素个数超过数组大小 \* loadFactor，这样就会发生多个线程同时对 Node 数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给 table，也就是说其他线程的都会丢失，并且各自线程 put 的数据也丢失\*（put多了 rehash, get的时候死循环）Concurrenthashmap 以及使用场景ConcurrentHashMap是J.U.C(java.util.concurrent包)的重要成员，它是HashMap的一个线程安全的、支持高效并发的版本。ConcurrentHashMap 在默认并发级别下会创建16个Segment对象的数组，如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。每个segment实际上还是存储的哈希表，写入的时候，先找到对应的segment，然后锁这个segment，写完，解锁。锁segment的时候，其他segment还可以继续工作。分段锁。使用场景多线程并发MAP的时候。

Hashmap与Hashtable的区别：

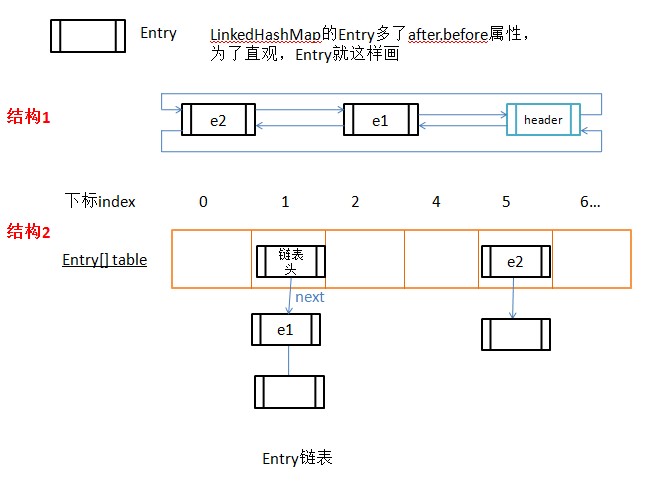
这两个类主要有以下几方面的不同：Hashtable和HashMap都实现了Map接口，但是Hashtable的实现是基于Dictionary抽象类。 在HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。 当get()方法返回null值时，即可以表示 HashMap中没有该键，也可以表示该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键，而应该用containsKey()方法来判断。而在Hashtable中，无论是key还是value都不能为null 。 这两个类最大的不同在于：1. Hashtable是线程安全的，它的方法是同步了的，可以直接用在多线程环境中。2. 而HashMap则不是线程安全的。在多线程环境中，需要手动实现同步机制。

### 怎么保证map有序？-（LinkedHashmap）

首先LinkedHashMap是非线程安全的。

LinkedHashMap 是 HashMap 的一个子类，它保留插入的顺序，如果需要输出的顺序和输入时的相同，那么就选用 LinkedHashMap。LinkedHashMap 是 Map 接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。LinkedHashMap 实现与 HashMap 的不同之处在于，LinkedHashMap 维护着一个运行于所有条目的**双重链接链表**。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。

下列是LinkedHashmap中Entry里面有的一些属性：K keyV valueEntry<K, V> nextint hash**Entry<K, V> beforeEntry<K, V> after**其中前面四个，也就是红色部分是从HashMap.Entry中继承过来的；后面两个，是LinkedHashMap独有的。不要搞错了next和before、After，**next**是用于维护HashMap指定table位置上连接的Entry的顺序的（从table中指向entry[]数组的），before、After是用于维护Entry插入的先后顺序的。关系图如下：



说明：

LinkedHashMap是继承HashMap，也就继承了HashMap的结构，也就是图中的结构2，在下文中我用"Entry数组+next链表"来描述。而LinkedHashMap有其自己的变量header，也就是图中的结构1，下文中我用"header链表"来描述。结构1中的Entry和结构2中的Entry本是同一个，结构1中应该就只有一个header，它指向的是结构2中的e1 e2，但这样会使结构图难画。为了说明问题的方便，我把结构2里的e1 e2在结构1中多画一个。

### TreeMap

对于TreeMap而言，由于它底层采用一颗“红黑树”来保存集合中的Entry，这意味着TreeMap添加元素、取出元素的性能都比HashMap低。

该映射根据其键的自然顺序进行排序，或者根据创建时提供的Comparator进行排序。

当TreeMap添加元素时，需要通过循环找到新增Entry的插入位置，因此比较耗性能；当从TreeMap中取出元素时，需要通过循环才能找到合适的Entry，也比较耗性能。但TreeMap、TreeSet相比HashMap、HashSet的优势在于：TreeMap中的所有Entry总是按key根据指定排序规则保存有序状态，TreeSet中的所有元素总是根据指定排序规则保存有序状态。[红黑树](#_红黑树_（旋转调整）)是一种自平衡二叉查找树，树中每个节点的值，都大于或等于在它的左子树中的所有节点的值，并且小于或等于在它的右子树中的所有节点的值，这确保红黑树运行时可以快速地在树中查找和定位的所需节点。

# 设计模式

## 单例模式

饿汉式

懒汉式

双重检验所

Enumerate

# 数据库

## 解析一条SQL语句

## 常用引擎

### 分类

### 区别

## 事务隔离

## 锁机制

## 索引

# 计算机网络

## 解析一条URL

## TCP/UDP 异同

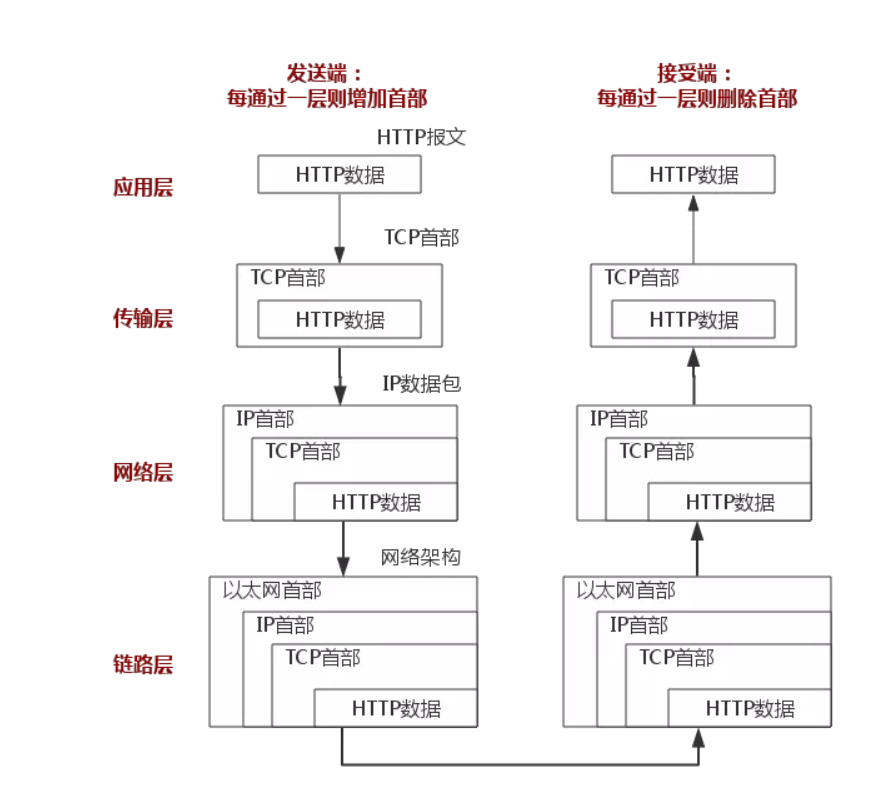
## TCP/UDP 包头

## TCP连接建立(标识符，发包算法)

## TCP流量控制，拥塞避免

## 七层协议/五层协议

* 1. 给你一个url 说一下如何浏览器如何解析的  DNS 解析几层 （递归解析 本地 根，顶级）
  2. 现在我有一台计算机，通过ISP接入了互联网，那么ISP就会给我分配一个DNS服务器，这个DNS服务器不是权威服务器，而是相当于一个代理的dns解析服务器，他会帮你迭代权威服务器返回的应答，然后把最终查到IP返回给你。2.现在的我计算机要向这台ISPDNS发起请求查询www.baidu.com这个域名了，(经网友提醒：这里其实准确来说不是ISPDNS，而应该是用户自己电脑网络设置里的DNS，并不一定是ISPDNS。比如也有可能你手工设置了8.8.8.8)3.ISPDNS拿到请求后，先检查一下自己的缓存中有没有这个地址，有的话就直接返回。这个时候拿到的ip地址，会被标记为非权威服务器的应答。4.如果缓存中没有的话，ISPDNS会从配置文件里面读取13个根域名服务器的地址（这些地址是不变的，直接在BIND的配置文件中），5.然后像其中一台发起请求。6.根服务器拿到这个请求后，知道他是com.这个顶级域名下的，所以就会返回com域中的NS记录，一般来说是13台主机名和IP。7.然后ISPDNS向其中一台再次发起请求，com域的服务器发现你这请求是baidu.com这个域的，我一查发现了这个域的NS，那我就返回给你，你再去查。（目前百度有4台baidu.com的顶级域名服务器）。8.ISPDNS不厌其烦的再次向baidu.com这个域的权威服务器发起请求，baidu.com收到之后，查了下有www的这台主机，就把这个IP返回给你了，9.然后ISPDNS拿到了之后，将其返回给了客户端，并且把这个保存在高速缓存中
  3. TCP 3握手4挥手
  4. HTTP传输数据 如何建立  七层网络模型从应用层说到物理层，说到 网络层



* 1. TCP头部信息，UDP头部消息





* 1. 为什么需要长连接 怎么实现长连接

在HTTP/1.0中，默认使用的是短连接。也就是说，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。如果客户端浏览器访问的某个HTML或其他类型的 Web页中包含有其他的Web资源，如JavaScript文件、图像文件、CSS文件等；当浏览器每遇到这样一个Web资源，就会建立一个HTTP会话。但从 HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头有加入这行代码

长连接优点缺点：长连接可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户来说，较适用长连接。不过这里存在一个问题，存活功能的探测周期太长，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，Client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候

短链接优点缺点：短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户请求频繁，将在TCP的建立和关闭操作上浪费时间和带宽。

* 1. Tomcat框架

# 操作系统

## 进程线程

## 线程间通信

## 进程间通信

## 地址与内存

## 中断

## 上下文切换

## 进程状态

## 进程调度（算法）