知识点：

抽象类和接口的区别：

抽象类用extends被子类继承，接口用implements

抽象类可以有构造器，接口不可以有，

抽象类可以有默认的方法实现，接口不可以有方法实现

抽象类可以有public，protected，default这些修饰符，接口只有public

只能继承一个抽象类，但可以继承多个接口

抽象类和普通类的唯一区别就是不能创建实例对象和允许有abstract方法

堆栈：先进后出

队列：先进先出

②堆是在程序运行时，而不是在程序编译时，申请某个大小的内存空间。即动态分配内存，对其访问和对一般内存的访问没有区别。

③堆是应用程序在运行的时候请求操作系统分配给自己内存，一般是申请/给予的过程。

④堆是指程序运行时申请的动态内存，而栈只是指一种使用堆的方法(即先进后出)。

栈又叫堆栈

**Mysql的四种隔离级别**：

Read Uncommitted（读取未提交内容）

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

Read Committed（读取提交内容）

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。

epeatable Read（可重读）

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。

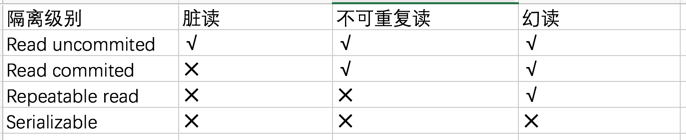
Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

脏读(Drity Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。

不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。

幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。



唯一索引可以为空值，但是主键索引不可以；

对于BTREE这种Mysql默认的索引类型，具有普遍的适用性

正是因为hash表在处理较小数据量时具有无可比拟的素的优势，所以hash索引很适合做缓存（内存数据库），NoSql数据库redis等，都使用了hash索引这种形式;

# java垃圾回收机制简单介绍

可达性分析法：该方法的基本思想是通过一系列的“GC Roots”对象作为起点进行搜索，如果在“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的，不过要注意的是被判定为不可达的对象不一定就会成为可回收对象。被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性，则基本上就真的成为可回收对象了。

目前Jvm使用的垃圾回收算法：它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。分为老年代（Tenured Generation）和新生代（Young Generation）。老年代的内存区域的对象一般回收频率比较低，采用了Mark-Compact算法，而新生代的内存区域由于每次需要回收大量对象，回收频率较高，所以将该区域又划分成了一个较大的Eden空间和两个较小的Suivivor空间，每次使用Eden空间和一个Survivor空间的，当需要回收垃圾时，将Eden空间和该Survivor空间的存活对象复制到另外一块Survivor空间上，然后清理到Eden和刚才使用的Survivor空间，实现垃圾回收机制.

**Mark-Compact（标记-整理）算法**   
Mark-Compact算法是在Mark-Sweep算法的基础上进行了改进，算法标记跟Mark-Sweep一样，只是在标记完之后，将标记的对象向一端移动，然后清理掉边界以外的内存区域，这样就解决了内存碎片化的问题

构造器不能被继承，因此不能被重写Override，但可以被重载Overload; 在子类构造对象时发现，访问子类构造函数时，父类构造函数也运行了； 原因是：在子类的构造函数第一行有一个默认的隐式语句：super();子类实例化过程：子类中所有的构造函数默认都会访问父类中的空参数的构造函数。

**HashTable** 是同步的(线程安全)，而**HashMap** 线程不安全，**效率上HashMap 更快**

**HashMap**允许空键值，而**Hashtable** 不允许

HashMap默认的初始大小是16，装载因子是0.75，并且扩容的大小一定是2的指数；而HashTable的默认初始大小是11，扩容的方式是 2\*old+1；

HashMap 的iterator 迭代器执行快速失败机制，也就是说在迭代过程中修改集合结构，除非调用迭代器自身的remove 方法，否则以其他任何方式的修改都将抛出并发修改异常。而Hashtable 提供了对键的列举(Enumeration)，Hashtable 返回的Enumeration 不是快速失败的。

Hashtable 和ConcurrentHashMap 都可以用于多线程的环境，但是当Hashtable 的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间；因为ConcurrentHashMap 引入了分割(segmentation)，不论它变得多么大，仅仅需要锁定map 的某个部分，而其它的线程不需要等到迭代完成才能访问map。简而言之，在迭代的过程中，ConcurrentHashMap 仅仅锁定map 的某个部分，而Hashtable 则会锁定整个map。

CurrentHashMap，是将HashMap分成了很多个片（一般默认是16片），引入了分段锁的概念，然后对每一片加锁，具体可以理解成一把大的Map分解成N个小的HashTable，根据key.hashCode（）来决定放到哪一个片上

String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，

# 五种进程调度算法的总结：

1、时间片轮转调度算法（RR）：给每个进程固定的执行时间，根据进程到达的先后顺序让进程在单位时间片内执行，执行完成后便调度下一个进程执行，时间片轮转调度不考虑进程等待时间和执行时间，属于抢占式调度。优点是兼顾长短作业；缺点是平均等待时间较长，上下文切换较费时。适用于分时系统。

2、先来先服务调度算法（FCFS）：根据进程到达的先后顺序执行进程，不考虑等待时间和执行时间，会产生饥饿现象。属于非抢占式调度，优点是公平，实现简单；缺点是不利于短作业。

3、优先级调度算法（HPF）：在进程等待队列中选择优先级最高的来执行。

4、多级反馈队列调度算法：将时间片轮转与优先级调度相结合，把进程按优先级分成不同的队列，先按优先级调度，优先级相同的，按时间片轮转。优点是兼顾长短作业，有较好的响应时间，可行性强，适用于各种作业环境。

5、高响应比优先调度算法：根据“响应比=（进程执行时间+进程等待时间）/ 进程执行时间”这个公式得到的响应比来进行调度。高响应比优先算法在等待时间相同的情况下，作业执行的时间越短，响应比越高，满足段任务优先，同时响应比会随着等待时间增加而变大，优先级会提高，能够避免饥饿现象。优点是兼顾长短作业，缺点是计算响应比开销大，适用于批处理系统。

Linux设置进程优先级：

Linux系统进程的优先级取值：-20 到 19，数越大优先级越低。  
 可以通过top命令来查看，NI那一列。

1，top命令。输入r，然后根据提示输入进程ID，再输入优先级数值。

2，renice命令。renice -n 2 -p 3432。-n，后面是优先级的值；-p，是进程号。

# Linux之uptime主机运行时间及平均负载含义

性能监控命令；输出的信息依此为：系统现在的时间，系统从上次开机到现在运行了多长时间，系统目前有多少登陆用户，系统在1分钟内、5分钟内、15分钟内的平均负载。

注意：load average的3个值大小一般不能大于系统CPU的个数。

         本系统有2个CPU，如果load average的3个值长期大于2时，说明CPU很繁忙，负载很高，可能会影响系统性能；

         如果偶尔大于2，一般不会影响系统性能；

 如果load average的输出值小于CPU的个数，则表示CPU还有空闲的时间。

**为什么要使用多线程?**

1.防止阻塞主线程,提高吞吐量

2,提高资源的利用率

3:需要异步操作的时候使用

进程：是执行中一段程序，即一旦程序被载入到内存中并准备执行，它就是一个进程。进程是表示资源分配的的基本概念，又是调度运行的基本单位，是系统中的并发执行的单位。

线程：单个进程中执行中每个任务就是一个线程。线程是进程中执行运算的最小单位。

一个线程只能属于一个进程，但是一个进程可以拥有多个线程。多线程处理就是允许一个进程中在同一时刻执行多个任务。

线程是一种轻量级的进程，与进程相比，线程给操作系统带来侧创建、维护、和管理的负担要轻，意味着线程的代价或开销比较小。

 我们都知道Java源文件，通过编译器，能够生产相应的.Class文件，也就是字节码文件，而字节码文件又通过Java虚拟机中的解释器，编译成特定机器上的机器码 。

JVM调优主要就是优化 Heap堆 和 Method Area 方法区。

JVM内存空间包含：方法区、java堆、java栈、本地方法栈

方法区是各个线程共享的区域，存放类信息、常量、静态变量。

java堆也是线程共享的区域，我们的类的实例就放在这个区域，可以想象你的一个系统会产生很多实例，因此java堆的空间也是最大的。如果java堆空间不足了，程序会抛出OutOfMemoryError异常。

java栈是每个线程私有的区域，它的生命周期与线程相同，一个线程对应一个java栈，每执行一个方法就会往栈中压入一个元素，这个元素叫“栈帧”，而栈帧中包括了方法中的局部变量、用于存放中间状态值的操作栈，如果java栈空间不足了，程序会抛出StackOverflowError异常，想一想什么情况下会容易产生这个错误，对，递归，递归如果深度很深，就会执行大量的方法，方法越多java栈的占用空间越大。

本地方法栈角色和java栈类似，只不过它是用来表示执行本地方法的，本地方法栈存放的方法调用本地方法接口，最终调用本地方法库，实现与操作系统、硬件交互的目的。

PC寄存器，说到这里我们的类已经加载了，实例对象、方法、静态变量都去了自己改去的地方，那么问题来了，程序该怎么执行，哪个方法先执行，哪个方法后执行，这些指令执行的顺序就是PC寄存器在管，它的作用就是控制程序指令的执行顺序。

对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

类加载过程:

整个生命周期包括：加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7个阶段。其中准备、验证、解析3个部分统称为连接（Linking）



加载、验证、准备、初始化、卸载这5个阶段的顺序是一定的，类的加载过程必须按照这种顺序按部就班地开始，而解析过程则不一定：它在某个情况下可以在初始化阶段之后再开始，这是为了支持Java语言语言的运行时绑定（也叫动态绑定和晚期绑定）;

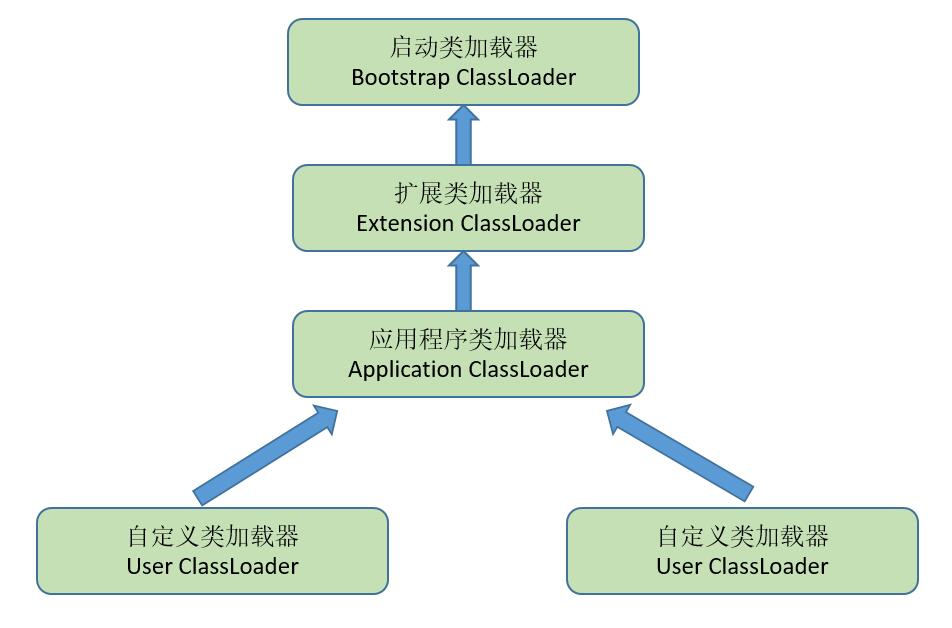
类加载阶段都是互相交叉地混合式进行的，通常是在一个阶段执行的过程中调用、激活另一阶段。

接口和类初始化的区别：当一个类在初始化时，其父类都基本上初始化过了，然而接口在初始化的时候，只有真正用到父接口的时候（如引用接口中定义的常量）才会进行初始化。

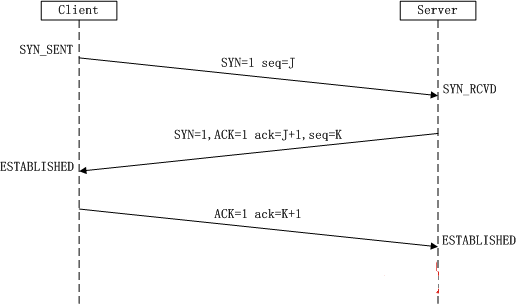
即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要类加载器不同，那么这两个类必定是不相等的。

**类加载的双亲委派模型过程**

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。



所谓三次握手（Three-Way Handshake）即建立TCP连接，是指建立一个TCP连接时，需要客户端和服务端总共发送3个包以确认连接的建立。在socket编程中，这一过程由客户端执行connect来触发;



（1）第一次握手：Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。

  （2）第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack (number )=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。

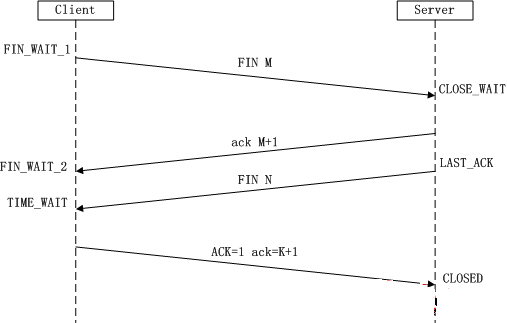
  （3）第三次握手：Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

SYN攻击：

  在三次握手过程中，Server发送SYN-ACK之后，收到Client的ACK之前的TCP连接称为半连接（half-open connect），此时Server处于SYN\_RCVD状态，当收到ACK后，Server转入ESTABLISHED状态。SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server回复确认包，并等待Client的确认，由于源地址是不存在的，因此，Server需要不断重发直至超时，这些伪造的SYN包将长时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络堵塞甚至系统瘫痪。SYN攻击时一种典型的DDOS攻击，检测SYN攻击的方式非常简单，即当Server上有大量半连接状态且源IP地址是随机的，则可以断定遭到SYN攻击了，使用如下命令可以让之现行：

  #netstat -nap | grep SYN\_RECV

所谓四次挥手（Four-Way Wavehand）即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close来触发，



由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭，上图描述的即是如此。

 （1）第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

  （2）第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

 （3）第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

  （4）第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

为什么建立连接是三次握手，而关闭连接却是四次挥手呢:

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

Hadoop无非就是：HDFS(文件系统)，yarn(任务调配)，mapReduce(编程模型，大数据并行运算)；

算法：

有左右之分，并且次序不能任意颠倒。

即使树中某结点只有一棵子树，也要区分它是左子树还是右子树。

二叉排序树又叫二叉查找树或者二叉搜索树，它首先是一个二叉树，而且必须满足下面的条件：

1）若左子树不空，则左子树上所有结点的值均小于它的根节点的值；

2）若右子树不空，则右子树上所有结点的值均大于它的根结点的值

3）左、右子树也分别为二叉排序树

4）没有键值相等的节点（？可能是因为不好处理键值相等的节点到底是左节点还是右节点吧）

二叉树链表：

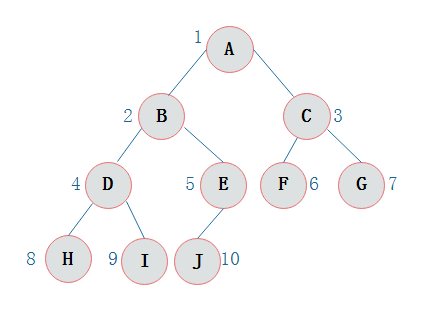
二叉树遍历：是指从二叉树的根结点出发，按照某种次序依次访问二叉树中的所有结点，使得每个结点被访问一次，且仅被访问一次。

访问次序分为四种：前序遍历，中序遍历，后序遍历，层序遍历

前序遍历：通俗的说就是从二叉树的根结点出发，当第一次到达结点时就输出结点数据，按照先向左在向右的方向访问。

从根结点出发，则第一次到达结点A，故输出A;  
继续向左访问，第一次访问结点B，故输出B；  
按照同样规则，输出D，输出H；  
当到达叶子结点H，返回到D，此时已经是第二次到达D，故不在输出D，进而向D右子树访问，D右子树不为空，则访问至I，第一次到达I，则输出I；  
I为叶子结点，则返回到D，D左右子树已经访问完毕，则返回到B，进而到B右子树，第一次到达E，故输出E；  
向E左子树，故输出J；  
按照同样的访问规则，继续输出C、F、G；

输出顺序：**ABDHIEJCFG**



**中序遍历：**就是从二叉树的根结点出发，当第二次到达结点时就输出结点数据，按照先向左在向右的方向访问。

从根结点出发，则第一次到达结点A，不输出A，继续向左访问，第一次访问结点B，不输出B；继续到达D，H；  
到达H，H左子树为空，则返回到H，此时第二次访问H，故输出H；  
H右子树为空，则返回至D，此时第二次到达D，故输出D；  
由D返回至B，第二次到达B，故输出B；  
按照同样规则继续访问，输出J、E、A、F、C、G；

**输出顺序：HDIBJEAFCG**

**后序遍历：**就是从二叉树的根结点出发，当第三次到达结点时就输出结点数据，按照先向左在向右的方向访问。

从根结点出发，则第一次到达结点A，不输出A，继续向左访问，第一次访问结点B，不输出B；继续到达D，H；  
到达H，H左子树为空，则返回到H，此时第二次访问H，不输出H；  
H右子树为空，则返回至H，此时第三次到达H，故输出H；  
由H返回至D，第二次到达D，不输出D；  
继续访问至I，I左右子树均为空，故第三次访问I时，输出I；  
返回至D，此时第三次到达D，故输出D；  
按照同样规则继续访问，输出J、E、B、F、G、C，A；

输出顺序：**HIDJEBFGCA**

层次遍历: 层次遍历就是按照树的层次自上而下的遍历二叉树

输出顺序：**ABCDEFGHIJ**

/\*二叉树的前序遍历递归算法\*/

void PreOrderTraverse(BiTree T)

{

if(T==NULL)

return;

printf("%c", T->data); /\*显示结点数据，可以更改为其他对结点操作\*/

PreOrderTraverse(T->lchild); /\*再先序遍历左子树\*/

PreOrderTraverse(T->rchild); /\*最后先序遍历右子树\*/

}

/\*二叉树的中序遍历递归算法\*/

void InOrderTraverse(BiTree T)

{

if(T==NULL)

return;

InOrderTraverse(T->lchild); /\*中序遍历左子树\*/

printf("%c", T->data); /\*显示结点数据，可以更改为其他对结点操作\*/

InOrderTraverse(T->rchild); /\*最后中序遍历右子树\*/

}

/\*二叉树的后序遍历递归算法\*/

void PostOrderTraverse(BiTree T)

{

if(T==NULL)

return;

PostOrderTraverse(T->lchild); /\*先后序遍历左子树\*/

PostOrderTraverse(T->rchild); /\*再后续遍历右子树\*/

printf("%c", T->data); /\*显示结点数据，可以更改为其他对结点操作\*/

}

**Hash去重：**

遇到两个对象数组A,B,找到A中B不存在的对象，并保存在一个数组中：

function unique2(arr){

//i遍历arr,同时创建两个空数组result和hash

for(var i=0,result=[],hash=[];

i<arr.length;

i++){

//如果hash中以当前元素为key的元素是undefined

if(hash[arr[i]]===undefined){

//将当前元素追加到result结尾

result[result.length]=arr[i];

//在hash中添加一个新元素: key为当前元素值,值为true

hash[arr[i]]=true;

}

}//(遍历结束)

return result;//返回result

}

HashSet去重原理差不多：它的add()方法实际上调用的是HashMap中的put()方法，把要添加进HashSet中的元素当做key存入，而value则是一个固定值：一个Object类对象。

先用hashCode()方法获得传入元素的哈希值，在集合中查找是否包含哈希值相同的元素，如果相同，则继续进行比较它们地址值，一般地址值都是不相同的，所以最后会用equals()方法比较对象内的属性值。 比较结果全为false就存入，如果比较结果有true则不存.

# [两个栈实现一个队列，两个队列实现一个栈](https://www.cnblogs.com/tracyhan/p/5490775.html)

1. 两个栈实现一个队列

将stack1作为存储空间，将stack2作为临时缓冲区，入队时，直接压入stack1中

 出队时，判断stack2是否为空，如果stack2为空，则将stack1中的元素倒入stack2中，否则直接弹出stack2中的元素

//入队操作

void EnQueue(stack<int> &s1,stack<int> &s2,int m)

{

s1.push(m);

}

//出队操作

void DeQueue(stack<int> &s1,stack<int> &s2,int &m)

{

if (s2.empty())

{

int p = s1.size();

for (int i=0;i<p;i++)

{

s2.push(s1.top());

s1.pop();

}

}

m = s2.top();

s2.pop();

}

1. 两个队列实现一个栈

将queue1用作进栈出栈，queue2作为一个中转站

入栈时，直接压入queue1中；

出栈时，先将queue1中的元素除最后一个元素外依次出队列，并压入队列queue2中，将留在queue1中的最后一个元素出队列即为出栈元素，最后还要把queue2中的元素再次压入queue1中

//进栈操作

void stackpush(queue<int> &q1,queue<int> &q2,int m)

{

q1.push(m);

}

//出栈操作

void stackpop(queue<int> &q1,queue<int> &q2,int &m)

{

int p = q1.size();

for (int i=0;i<p-1;i++)

{

q2.push(q1.front());

q1.pop();

}

m = q1.front();

q1.pop();

int l = q2.size();

for (int j = 0;j<l;j++)

{

q1.push(q2.front());

q2.pop();

}

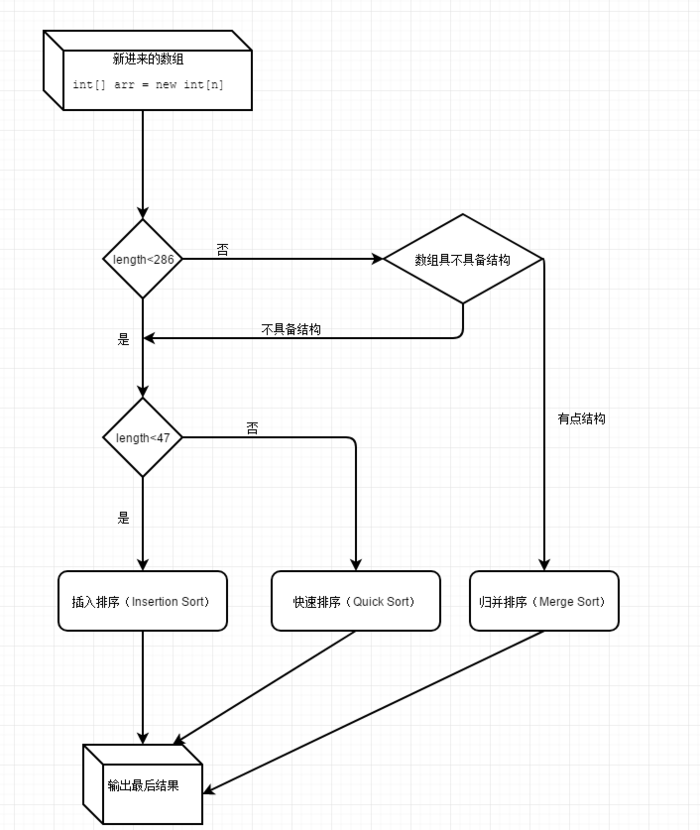
}

https://blog.csdn.net/weixin\_34342578/article/details/85821727

排序：

双轴快排的算法：双轴快排是基于两个轴来进行比较，跟普通的选择一个点来作为轴点的快排是有很大的区别的，双轴排序利用了区间相邻的特性，对原本的快排进行了效率上的提高；

jdk8中的数组排序算法Arrays.sort()：



数组具不具备结构：

SQL优化：

**in和exists区别：**

select \* from student s where s.stuid in(select stuid from score ss where ss.stuid = s.stuid)

select \* from student s where EXISTS(select stuid from score ss where ss.stuid = s.stuid)

exists()适合内表比外表数据大的情况；使用时一定要注意索引的影响，利用索引会使数据库查询有惊人的性能提升；

当内表数据与外表数据一样大时,in与exists效率差不多,可任选一个使用；

In（A为外表，B为内表）:

A表有10000条记录,B表有1000000条记录,那么最多有可能遍历10000\*1000000次,效率很差.  
A表有10000条记录,B表有100条记录,那么最多有可能遍历10000\*100次,遍历次数大大减少,效率大大提升.

Exists A为外表，B为内表）：

A表有10000条记录,B表有1000000条记录,那么exists()会执行10000次去判断A表中的id是否与B表中的id相等.

A表有10000条记录,B表有100000000条记录,那么exists()还是执行10000次,因为它只执行A.length次,可见B表数据越多,越适合exists()发挥效果.

not exists的执行顺序是：在表中查询，是根据索引查询的，如果存在就返回true，如果不存在就返回false，不会每条记录都去查询。  
之所以要多用not exists，而不用not in，也就是not exists查询的效率远远高与not in查询的效率。

Mysql的In查询会外表使用全表扫描，内表使用索引查询

JVM（over）

二叉树（over）

平衡树

红黑树

TCP/IP

Linux

Spring

Mybatis

Sql

Mysql

Hadoop

Java知识

Java8新特性

Java源码

Docker

Netty

Thrift

反射

排序算法重写