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

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。

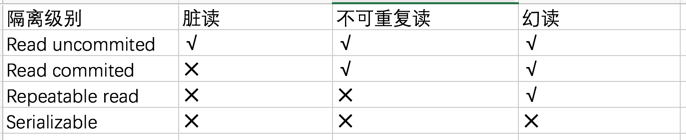
Serializable（可串行化）

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

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：

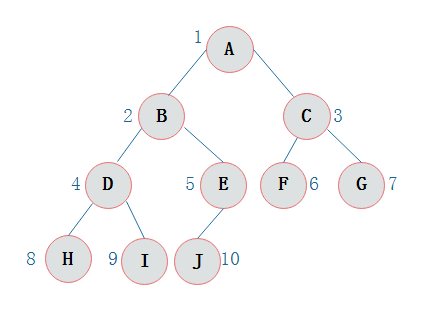
二叉树遍历：是指从二叉树的根结点出发，按照某种次序依次访问二叉树中的所有结点，使得每个结点被访问一次，且仅被访问一次。

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.