继承

封装 对数据的访问只能通过已定义的接口 类中编写的方法就是对实现细节的封装 自动和普通洗衣机

多态性 允许不同子类型的对象对同一消息作出不同的响应

抽象类 一类对象的共同特征总结出来构造类的过程 关注于对象的属性和行为 接口更多的是方法

float f=3.4 不正确。3.4是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting）会造成精度损失，因此需要强制类型转换float f =(float)3.4; 或者写成float f =3.4F;

short s1 = 1; s1 = s1 + 1; 有错吗? short s1 = 1; s1 += 1; 有错吗？

答：对于short s1 = 1; s1 = s1 + 1;由于1是int类型，因此s1+1运算结果也是int 型，需要强制转换类型才能赋值给short型。而short s1 = 1; s1 += 1; 可以正确编译，因为s1+= 1; 相当于s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

&&之所以称为短路运算是因为，如果&&左边的表达式的值是false，右边的表达式会被直接短路掉，不会进行运算

Math.round(11.5)的返回值是12，Math.round(-11.5)的返回值是-11。四舍五入的原理是在参数上加0.5然后进行下取整。

数组没有length()方法，有length 的属性。String 有length()方法 数组属性 字符串方法

构造器不能被继承，因此不能被重写，但可以被重载。 有继承关系才能被override

String 类是final类，不可以被继承。

String和StringBuilder、StringBuffer的区别？

Java提供了两种类型的字符串：String和StringBuffer/StringBuilder，它们可以储存和操作字符串。其中String是只读字符串，也就意味着String引用的字符串内容是不能被改变的。而StringBuffer/StringBuilder类表示的字符串对象可以直接进行修改。StringBuilder是Java 5中引入的，它和StringBuffer的方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方面都没有被synchronized修饰，因此它的效率也比StringBuffer要高。 **StringBuilder单线程 StringBuffer多线程**

char类型可以存储一个中文汉字，因为**Java中使用的编码是Unicode**（不选择任何特定的编码，直接使用字符在字符集中的编号 **一个char两字节**

抽象类（abstract class）和接口（interface）有什么异同？抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全部都是抽象方法。抽象类中的成员可以是private、默认、protected、public的，而接口中的成员全都是public的。抽象类中可以定义成员变量，而**接口中定义的成员变量实际上都是常量**(final)。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法

抽象方法无法是static的 抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的

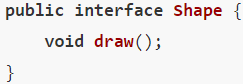
对象克隆 实现Serializable接口

接口可以继承接口，且支持多重继承。抽象类可以实现(implements)接口，抽象类可继承具体类也可以继承抽象类

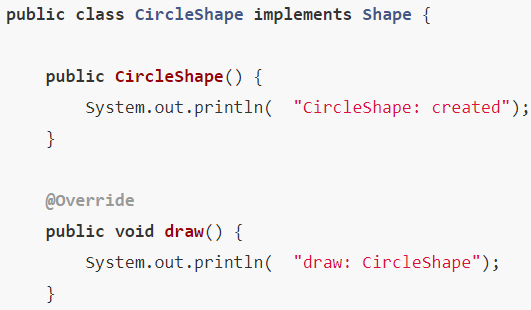
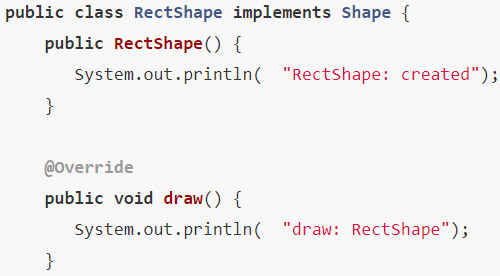
抽象类像一个模板 接口像扩展功能

工厂模式： https://juejin.im/entry/58f5e080b123db2fa2b3c4c6

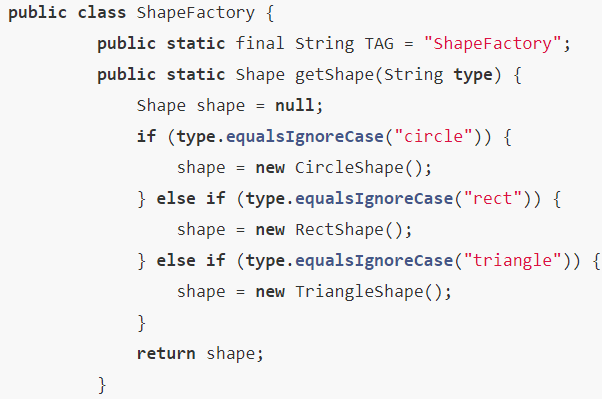
1.Define interface



2.individual class inherit interface

3.define factory class

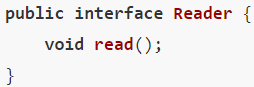


4.create instance

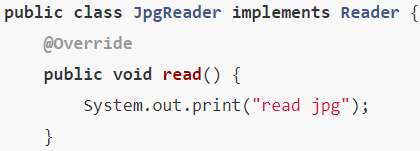
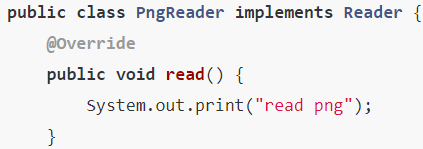
 

Factory Method Module

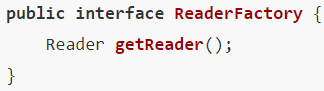
1.Define interface

 1:5.2 400 2000

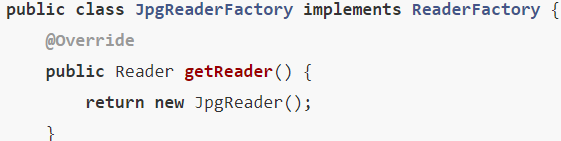
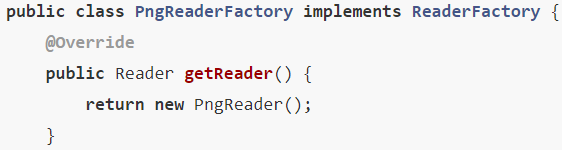
2.individual class inherit interface

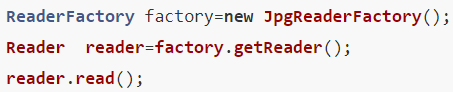
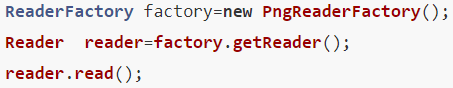
3.define abstract factory interface



4.define individual factory class implements factory interface

5.create instance

------------------------------

# 算法

算法稳定性 -- 假设在数列中存在a[i]=a[j]，若在排序之前，a[i]在a[j]前面；并且排序之后，a[i]仍然在a[j]前面。则这个排序算法是稳定的

## 二分查找

需要有序数组 每次折半查找

最多需要 log2n次

简单查找 线性时间 O(n) 需要n次

二分查找 对数时间 O(logn) 需要logn次

大O表示法 括号内的表示操作数---最糟情况下

算法的速度指的并非时间，而是操作数的增速。

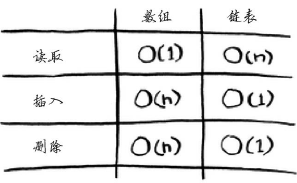
谈论算法的速度时，我们说的是随着输入的增加，其运行时间将以什么样的速度增加。

算法的运行时间用大O表示法表示。

数组和链表

数组---插值困难 读值简单-随机访问 数组中元素的位置称为索引 元素20位于索引1处

链表---随机读值困难 需要从头开始读-顺序访问 插值简单



## 选择排序

每次选择数组中最小的 与前面元素交换

第一次n 第二次n-1 ……. 平均每次检查的元素1/2\*n 所以总共1/2\*n2 省略常数 O(n2)

调用栈 *函数A包含函数B时 A先在内存中开辟区域 然后执行 栈顶是A 当执行到B时 在A上方插*

A ( you ) { *入一内存块执行B方法 执行完B后 把B的内存块弹走只剩A 继续A的执行* B( me ) {

}

}

递归指的是调用自己的函数。

每个递归函数都有两个条件：基线条件和递归条件。

栈有两种操作：压入和弹出。

所有函数调用都进入调用栈。

调用栈可能很长，这将占用大量的内存。

## 快速排序

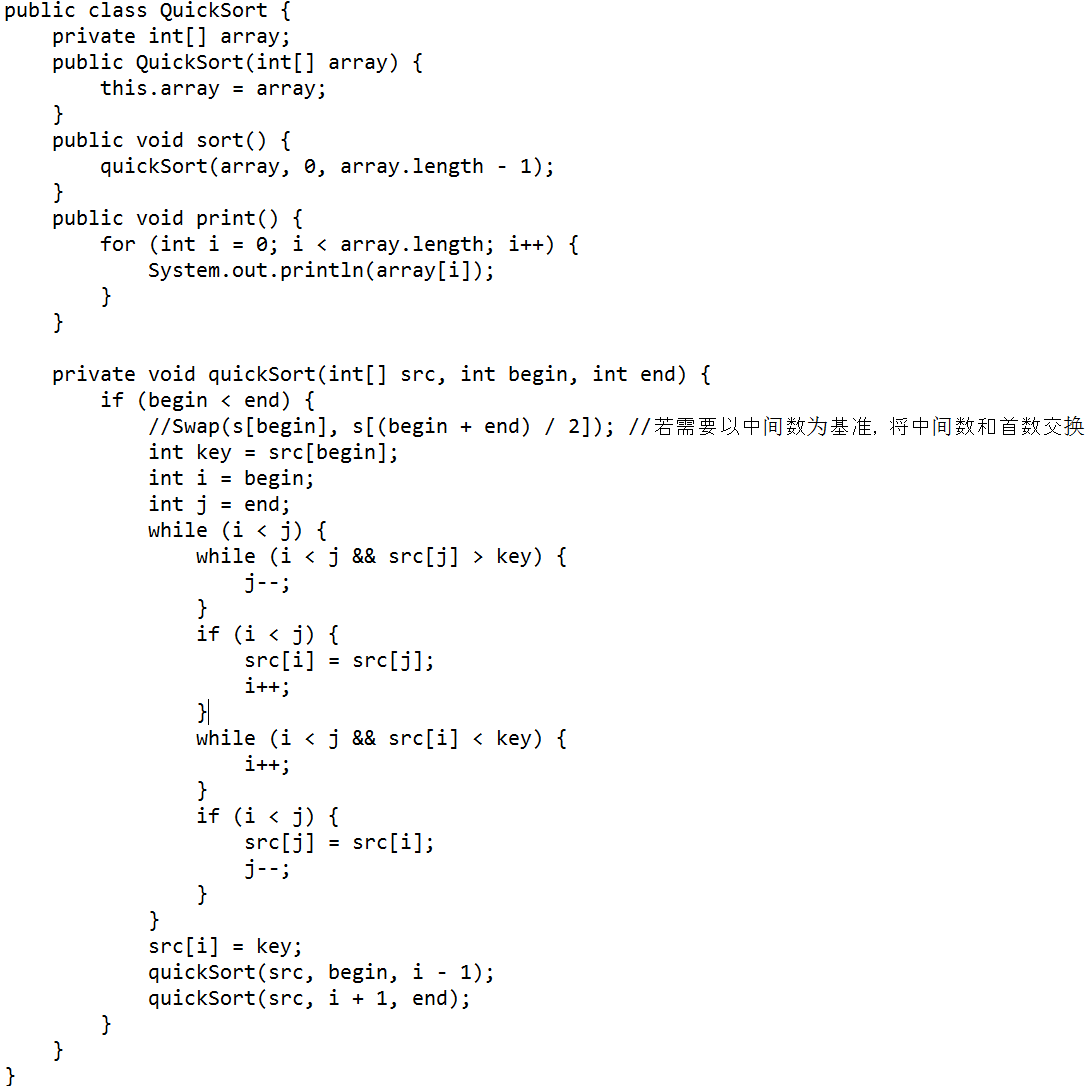
基线条件：停止条件

递归条件：循环体

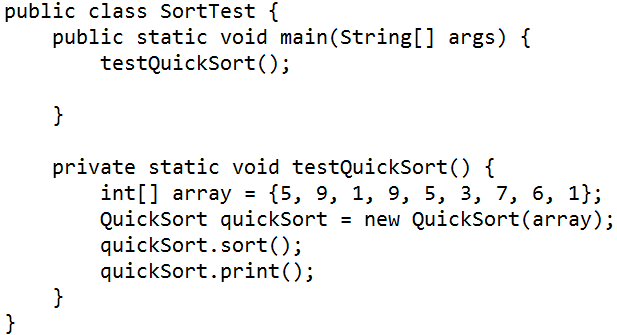
最优情况： O(logn)层 每层需要时间O(n) 总共O(nlogn)

最糟情况： O(n)层 每层需要时间O(n) 总共O(n2)

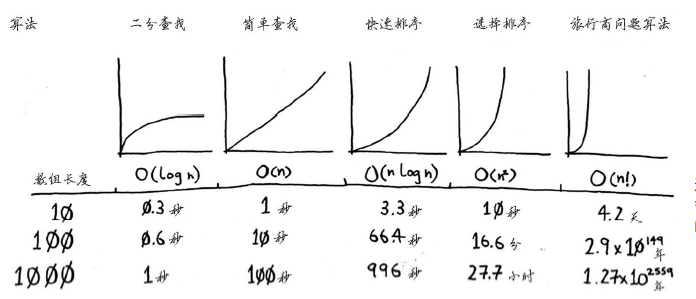
实现



测试类



各种算法的时间复杂度



## 散列表（映射）

输入---散列函数---数组索引---值 散列函数只会返回有效数组索引 相同输入返回相同数组索引

如果两个键映射到了同一个位置，就在这个位置存储一个链表

最优情况O(1) 最糟情况O(n)

想要避免冲突(长链表)，需要较低的填装因子(数组被占用/数组总格数)和良好的散列函数

一般填装因子大于0.7就调整散列表长度(翻倍)

缓存的数据储存在散列表中

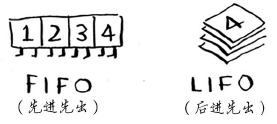
## 广度优先搜索BFS(图的算法)

段数最少的路径

图由节点和边组成

解决两类问题： 1.从A出发，有到B的路径么 2. 从A出发，到B的哪条路径最短

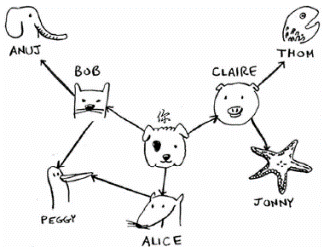
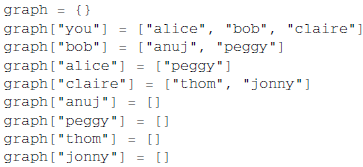
使用 队列 数据结构解决



队列 栈

这将先在一度关系中查找，再在二度关系中查找，因此找到的是距离最近的

先用用散列表来实现图 再用队列找最近路径

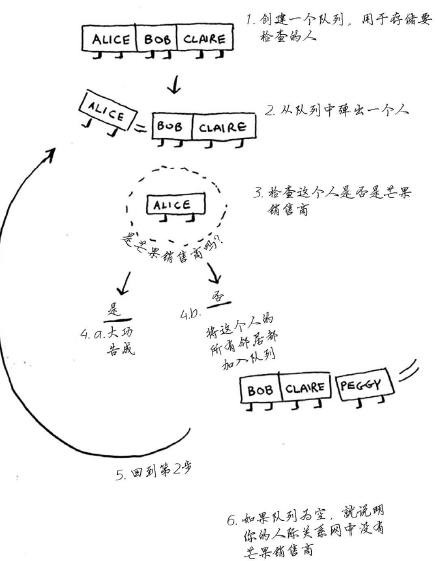
 

有向图 箭头表示单向关系 无向图 直线表示双向关系

 两图此时等价

算法实现

原理：

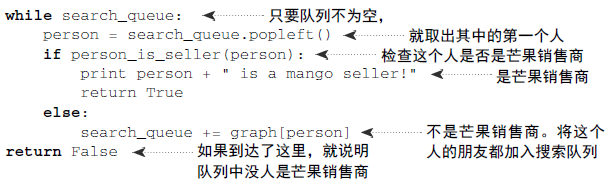


首先创建一个队列(使用deque创一个双端队列)

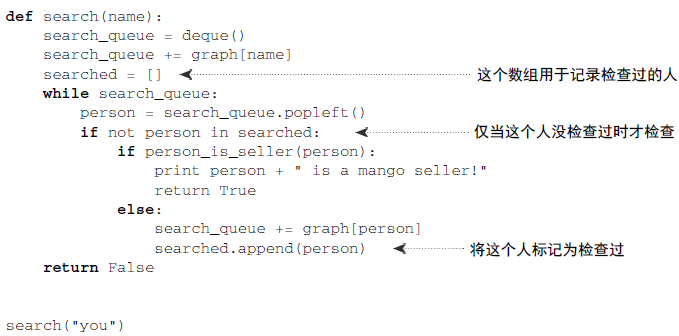
graph["you"]是一个数组，其中包含你的所有邻居，如["alice", "bob","claire"]



队列实现 此时会出现一个问题 还会重复检查 (如A既是B的邻居又是C的邻居) 所以需要创建一个数组记录

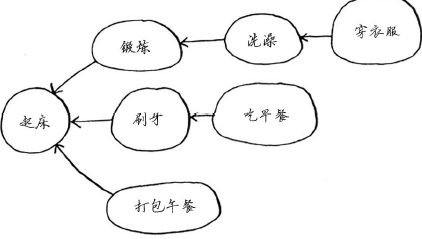


完整代码



广度优先搜索的运行时间为 O(人数(顶点数) + 边数)，这通常写作O(V + E)

## 拓扑排序



有序的 任务A依赖于任务B 在列表中A就必须排在B的后面 一条链上的顺序不可逆

## 树

结点的度：结点拥有的子树的数目。

叶子：度为零的结点。

分支结点：度不为零的结点。

树的度：树中结点的最大的度。

层次：根结点的层次为1，其余结点的层次等于该结点的双亲结点的层次加1。

树的高度：树中结点的最大层次。

无序树：如果树中结点的各子树之间的次序是不重要的，可以交换位置。

有序树：如果树中结点的各子树之间的次序是重要的, 不可以交换位置。

森林：0个或多个不相交的树组成。对森林加上一个根，森林即成为树；删去根，树即成为森林。

二叉树：

二叉树有以下几个性质：TODO(上标和下标)

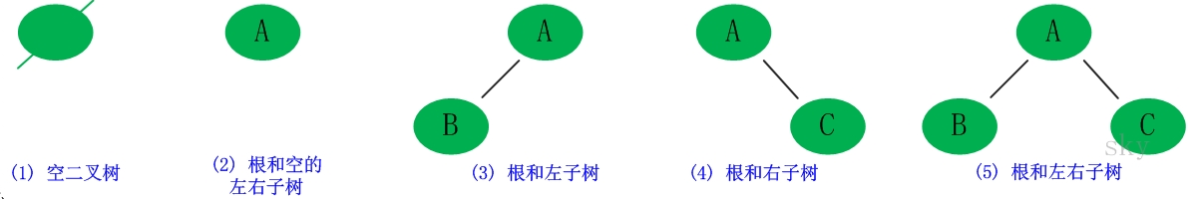
性质1：二叉树第i层上的结点数目最多为 2{i-1} (i≥1)。

性质2：深度为k的二叉树至多有2{k}-1个结点(k≥1)。

性质3：包含n个结点的二叉树的高度至少为log2 (n+1)。

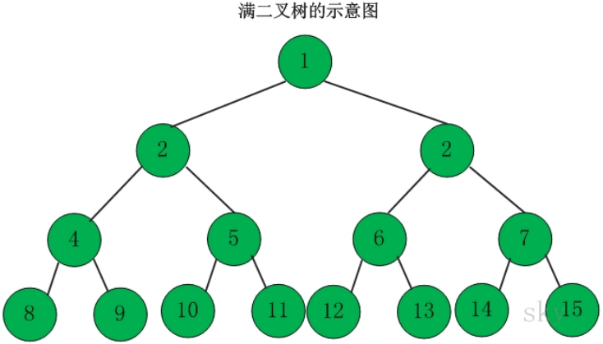
性质4：在任意一棵二叉树中，若终端结点的个数为n0，度为2的结点数为n2，则n0=n2+1。

每个节点最多有两个子树的树结构，二叉树可以是空集；根可以有空的左子树或右子树；或者左、右子树皆为空



满二叉树：

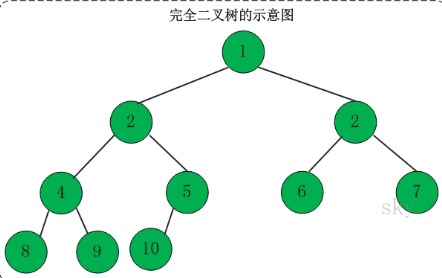
高度为h，并且由2{h} –1个结点的二叉树，被称为满二叉树



完全二叉树：

定义：一棵二叉树中，只有最下面两层结点的度可以小于2，并且最下一层的叶结点集中在靠左的若干位置上。这样的二叉树称为完全二叉树。

特点：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。显然，一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。



二叉查找树：

定义：二叉查找树(Binary Search Tree)，又被称为二叉搜索树。设x为二叉查找树中的一个结点，x节点包含关键字key，节点x的key值记为key[x]。如果y是x的左子树中的一个结点，则key[y] <= key[x]；如果y是x的右子树的一个结点，则key[y] >= key[x]

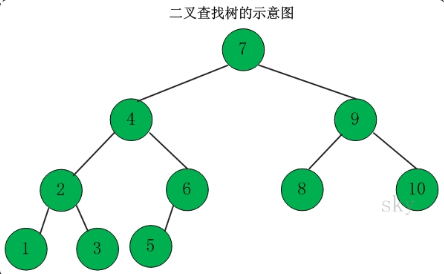
特点：

若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

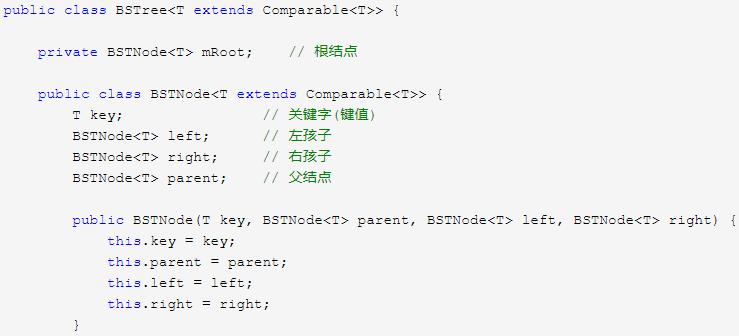
任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

任意节点的左、右子树也分别为二叉查找树。

没有键值相等的节点（no duplicate nodes）。



二叉查找树实现：

节点定义

BSTree是二叉树，它保护了二叉树的根节点mRoot；mRoot是BSTNode类型，而BSTNode是二叉查找树的节点，它是BSTree的内部类。

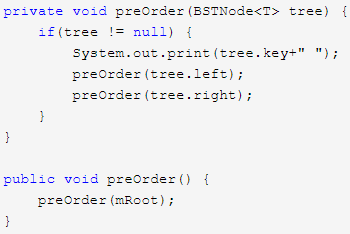
key -- 它是关键字，是用来对二叉查找树的节点进行排序的

前序遍历

若二叉树非空，则执行以下操作：

(01) 访问根结点；

(02) 先序遍历左子树；

(03) 先序遍历右子树。

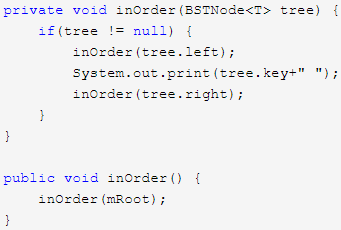
中序遍历

若二叉树非空，则执行以下操作：

(01) 中序遍历左子树；

(02) 访问根结点；

(03) 中序遍历右子树。



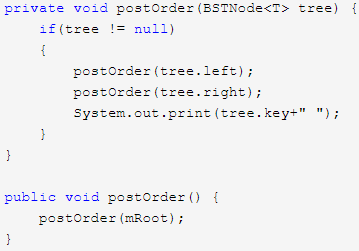
后序遍历

若二叉树非空，则执行以下操作：

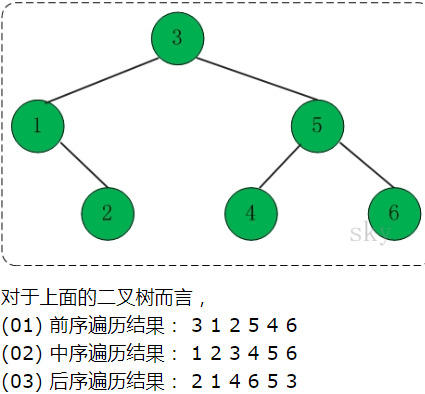
(01) 后序遍历左子树；

(02) 后序遍历右子树；

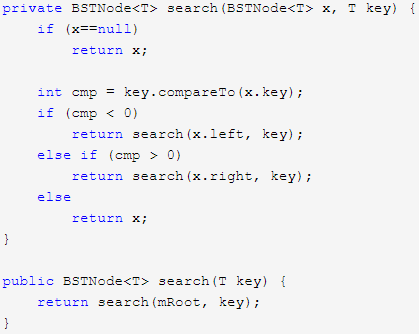
(03) 访问根结点。



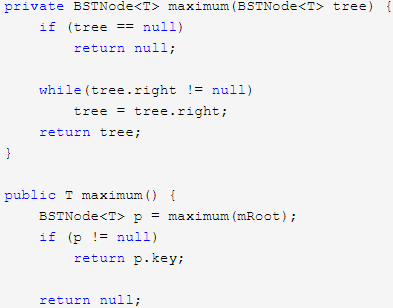
结果



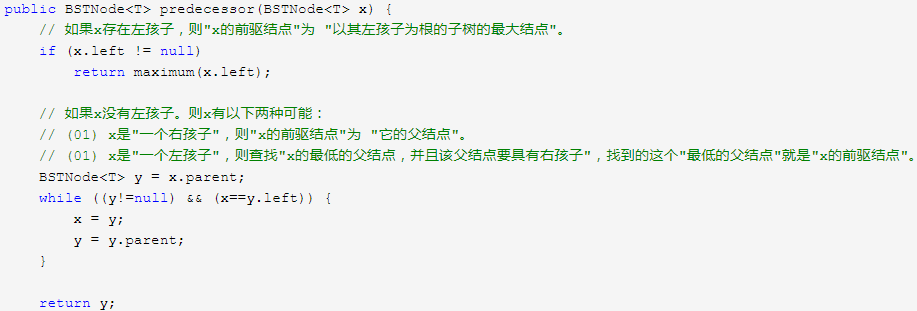
查找"二叉树x"中键值为key的节点



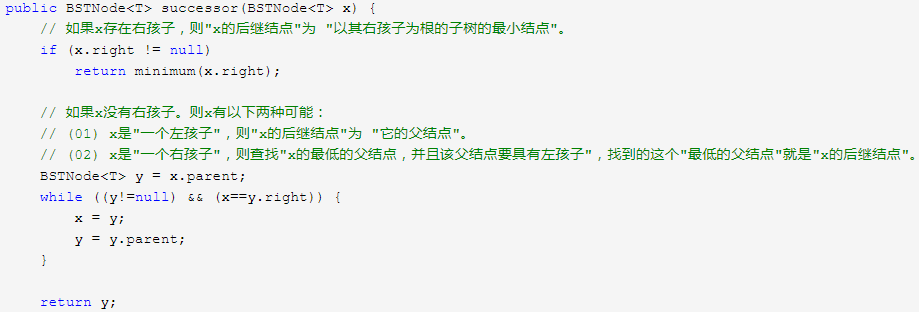
查找最大结点：返回tree为根结点的二叉树的最大结点(最小tree=tree.left;)



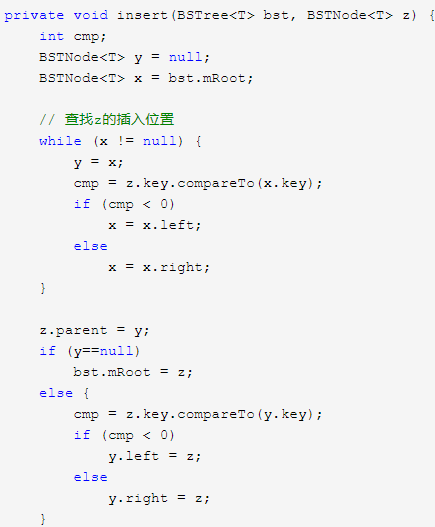
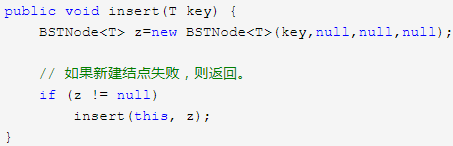
找结点(x)的前驱结点。即查找"二叉树中数据值小于该结点"的"最大结点"



找结点(x)的后继结点。即查找"二叉树中数据值大于该结点"的"最小结点"



将结点z插入到二叉树bst中

若z是个值就先创个z节点

删除

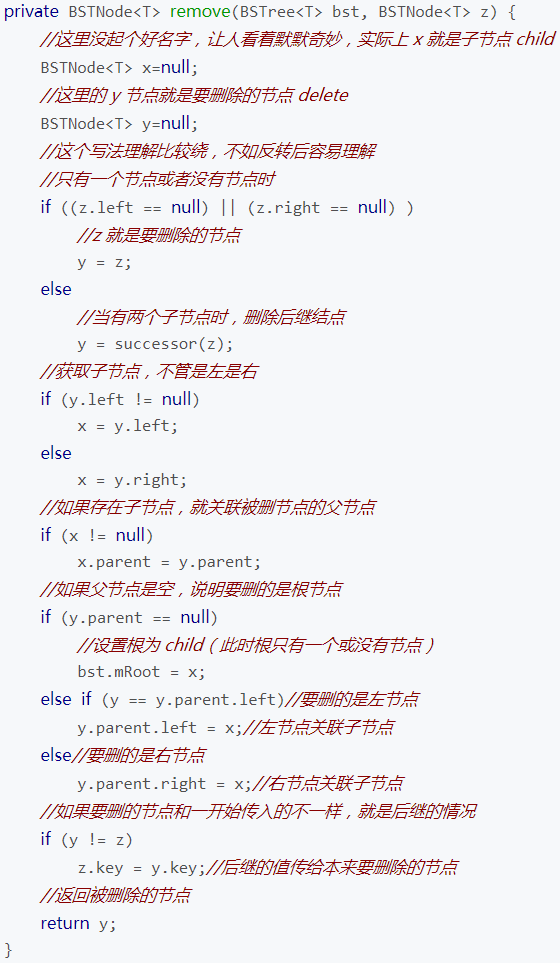
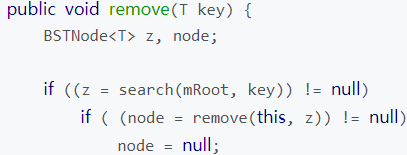
删除节点存在 3 种情况

没有左右子节点，可以直接删除

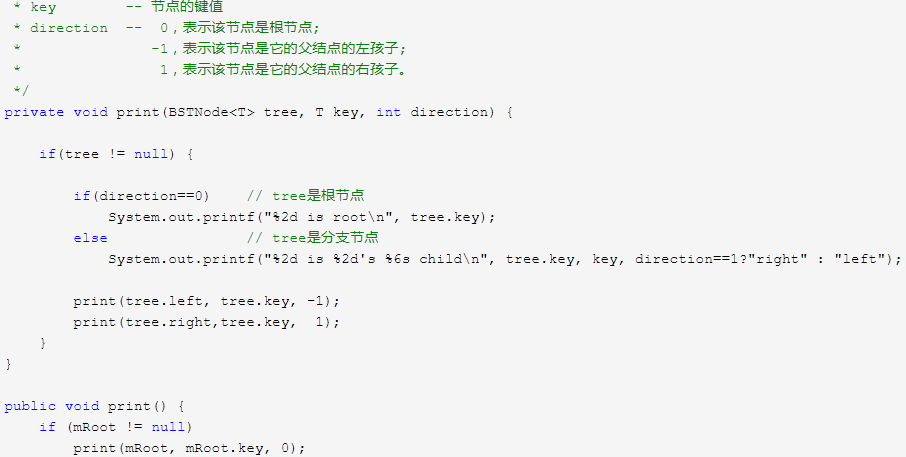
存在左节点或者右节点，删除后需要对子节点移动

同时存在左右子节点，不能简单的删除，但是可以通过和后继节点交换后转换为前两种情

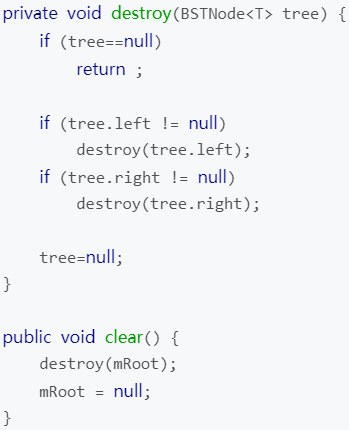
删除结点(z)，并返回被删除的结点

打印"二叉查找树”



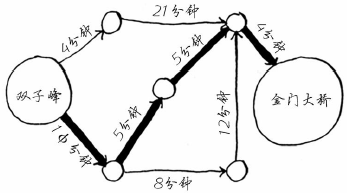
销毁



## 狄克斯特拉算法(Dijkstra’s algorithm)

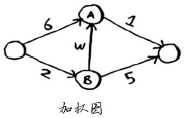
最短路径问题 只适用于有向无环图

关键理念：找出图中最便宜的节点，并确保没有到该节点的更便宜的路径



广度优先搜索找段数最少的路径 狄克斯特拉算法可以找最快路径

加权图 每条边上的数字叫权重 如果有负权边，就不能使用狄克斯特拉算法



原理;

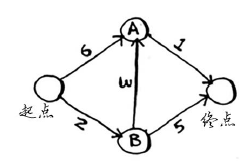
(1) 找出最便宜的节点，即可在最短时间内前往的节点。

(2) 对于该节点的邻居，检查是否有前往它们的更短路径，如果有，就更新其开销。

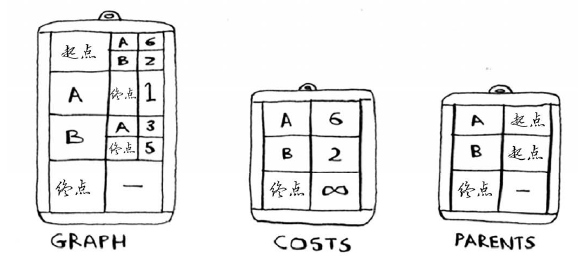
(3) 重复这个过程，直到对图中的每个节点都这样做了。

(4) 计算最终路径(再在前三步的过程中记录父节点)

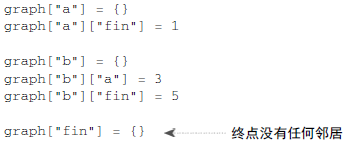
算法准备：

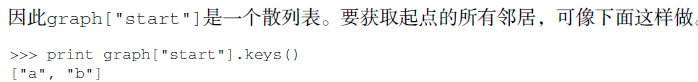
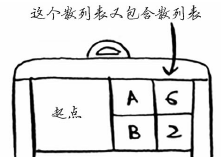


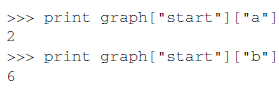
实现代码首先需要三个散列表

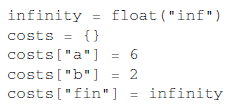


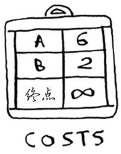
随着算法的进行，散列表costs和parents将不断更新

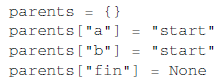
 此处意味着散列表的散列表 

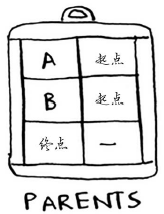


 路径表建立

 第一句代表无穷大

 开销表建立

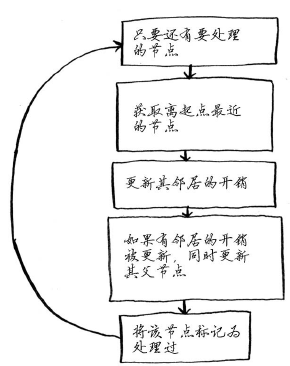
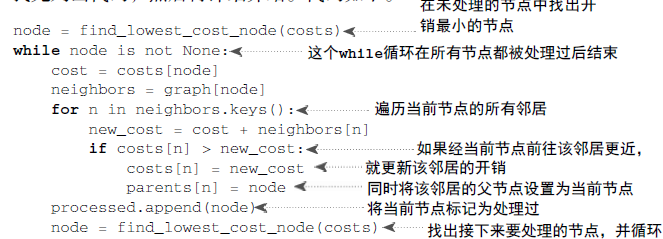


 存储父节点的散列表

最后需要一个数组，用于记录处理过的节点，因为对于同一个节点，你不用处理多次。

processed = [ ]

算法开始

1. 找出未处理节点中开销最小的点B
2. 获取B的开销2
3. 获取B的邻居A & 终点

neighbors是一个小散列表

4. 遍历邻居并算出其开销

每个节点都有开销。开销指的是从起点前往该节点需要多长时间。在这里，你计算从起点出

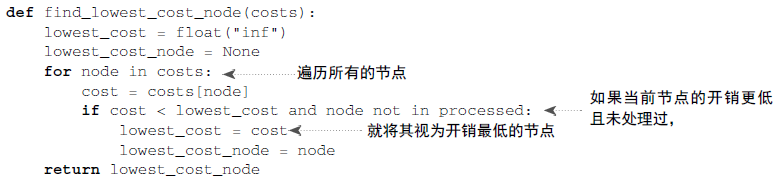
发，经节点B前往节点A（而不是直接前往节点A）需要多长时间。

5．新旧开销比较，若更小则更新开销表，父节点表

6．算出到A为5 到终点为7，结束遍历

7．标记当前节点(B) , 进行循环，开始寻找下一个未处理节点

函数find\_lowest\_cost\_node找出开销最低的节点 代码



 广度优先搜索用于在非加权图中查找最短路径。

 狄克斯特拉算法用于在加权图中查找最短路径。

 仅当权重为正时狄克斯特拉算法才管用。

 如果图中包含负权边，请使用贝尔曼-福德算法。

## 贪婪算法---单次最优

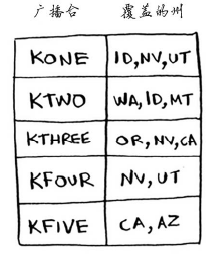
每步都选择局部最优解，最终得到的就是全局最优解

是一种近似算法（approximation algorithm）。在获得精确解需要的时间太长时，可使用近似算法。判断近似算法优劣的标准如下：

 速度有多快；

 得到的近似解与最优解的接近程度。

找出覆盖全美50个州的最小广播台集合

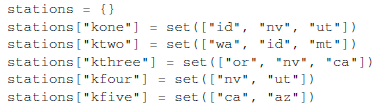


准备：

首先 创建数组包含要覆盖的州 再转化为set(排除重复元素)



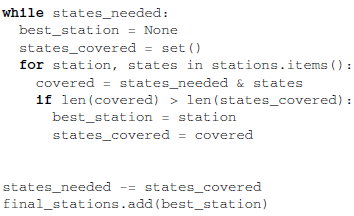
创建广播台清单散列表



创建集合来储存选择的广播台



算法开始：

遍历所有的广播台，从中选择覆盖了最多的未覆盖州的广播台。将这个广播台存储在best\_station中



对stations散列表将进行遍历，coved等于所有当前所有未覆盖的州和此台包含未覆盖的州取交集，找到cover最大的一次，加入best\_station中，将所有未覆盖的州删去已覆盖的

旅行商问题---找到最短路径

出发城市未定 两城市往返为不同路线

总次数为n!

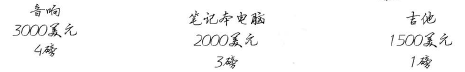
随便选择出发城市，然后每次选择要去的下一个城市时，都选择还没去的最近的城市

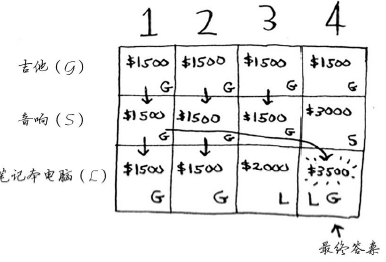
旅行商问题和集合覆盖问题都属于NP完全问题----计算所有解 选出最短/最小----此时应使用近似算法(贪心算法)

## 动态规划---网格方法

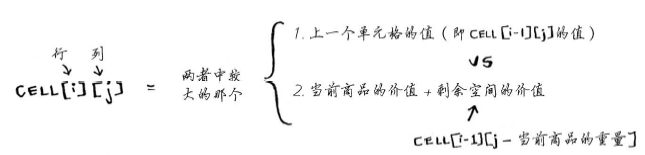
动态规划先解决子问题，再逐步解决大问题

如何使四磅的背包价值最高

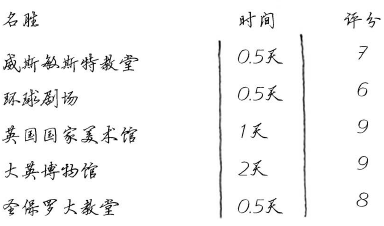




计算公式：



类似的问题：确定线路



动态规划可在给定约束条件下找到最优解。例如，你必须在背包容量给定的情况下，偷到价值最高的商品。

在问题可分解为彼此独立且离散的子问题时，就可使用动态规划来解决。

每种动态规划解决方案都涉及网格

单元格中的值通常就是你要优化的值。在前面的背包问题中，单元格的值为商品的价值。

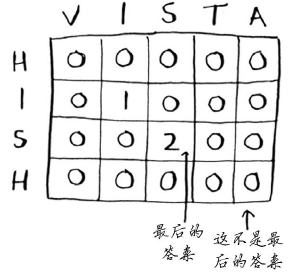
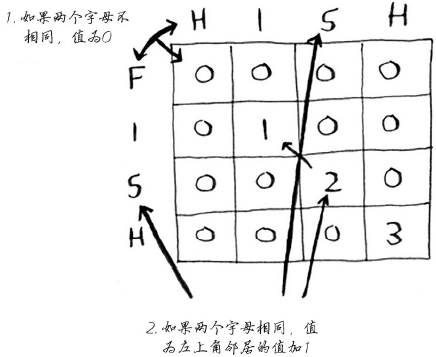
每个单元格都是一个子问题，因此你应考虑如何将问题分成子问题，这有助于你找出网格的坐标轴。

最长公共子串

问题 

伪代码：

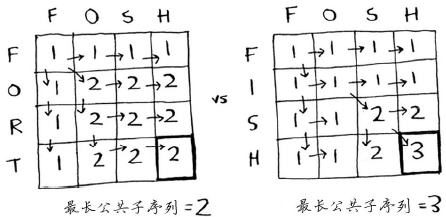




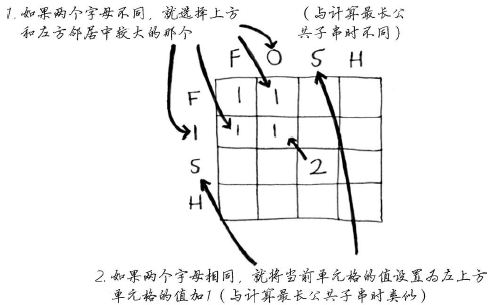
最长公共子序列

输入了fosh，他原本想输入的是fish还是fort

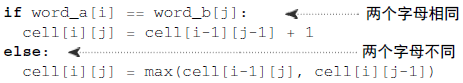
此时最长公共子串的长度相同，都包含两个字母 但fosh与fish更像



方法：



伪代码：



 需要在给定约束条件下优化某种指标时，动态规划很有用。

 问题可分解为离散子问题时，可使用动态规划来解决。

 每种动态规划解决方案都涉及网格。

 单元格中的值通常就是你要优化的值。

 每个单元格都是一个子问题，因此你需要考虑如何将问题分解为子问题。

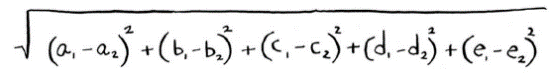
## K最近邻算法(KNN)---分类和回归

分类：

要对东西进行分类时,看其最近的K个邻居 如果有N位用户，应考虑sqrt(N)个邻居。

推荐系统：

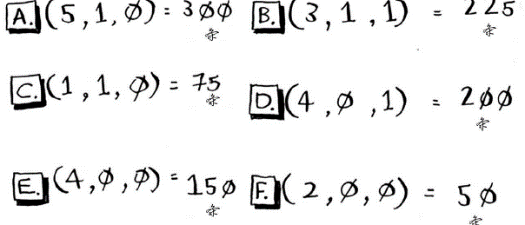
根据喜好确定图中位置 喜好相近用户相邻

如何确定相似程度：

将两用户对不同事物喜欢程度转化为坐标 确定两点间距离 越短越相近

回归：

根据A，B，C，D，E与X在表格中的距离找出最近的几个点 这几个点的平均值就是X的值





# 面试：

## 问题

说一说jvm的分区

说一说java文件编译成.class文件的过程

说一说jvm加载的过程

java的static修饰

反射机制

说一说线程池

线程池原理，怎么实现区分线程优先级的线程池

线程中保持同步有哪些方法(synchronized和Lock)

手写算法，找图中两点之间最短路径

spring的控制反转

知道哪些设计模式，说一说建造者模式

白板手写单例(懒加载。。双重检查。。静态内部类）

快速排序原理

看过java的源码吗？答: 看过一点集合类the whole things come here

那你说一说HashMap的实现(这里非常详细的展开说了实现原理）

那HashMap安全吗？(不安全，我们可以用...)

java里有哪些集合类，详细说一说HashSet

HashMap里的hashcode方法和equal方法什么时候需要重写？如果不重写会有什么后果？对此大家可以进一步了解HashMap的底层实现。

ArrayList和LinkedList底层实现有什么差别？它们各自适用哪些场合？对此大家也可以了解下相关底层代码。

volatile关键字有什么作用？由此展开，大家可以了解下线程内存和堆内存的差别。

SQL

SQL高级方面，比如group by, having，左连接，子查询（带in），行转列等高级用法。

建表方面，你可以考虑下，你项目是用三范式还是反范式，理由是什么？

数据库优化，准备下如何通过执行计划查看SQL语句改进点的方式，能改善SQL性能的方式（比如建索引等）

数据库索引，使用时要注意什么

详细介绍数据库的隔离机制

--------------------------------

简历<https://www.jianshu.com/p/7e495e22cff7>

---------------------------------

Async(异步)與Sync(同步)

Async指发出请求到收到结果间可以做其他事情 例如一个持续10s的任务 发出请求后10s内可以做其它操作

Sync指发出请求后需等待任务完成后再做其他 如一个10s任务 发出请求后10s内待机等待结果

## Equals方法

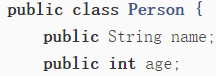
数据类型分为基本数据类型和引用数据类型 基本数据类型用==直接判断值是否相等 而引用数据类型一般用equals进行比较，如果你直接用的话判断的是对象在堆中的地址值是否相等(是否是同一个对象)，若想比较是否值相等，需要重写equals方法(string已经重写过了)

放入List的类需要覆写equals方法才能进行contains，indexOf等比较查找操作

Equals方法中对null的比较：即x.equals(null)永远返回false

String 已经覆写了equals方法

如一个person类

 相等逻辑：对于person类 name和age相同就相等



用instanceof判断传入的待比较的Object是不是当前类型，如果是，继续比较，否则，返回false；

如果不调用List的contains()、indexOf()这些查找方法，那么放入的元素就不需要实现equals()方法

## HashCode方法

想使用map，作为key的类必须正确覆写equals()和hashCode()方法

HashMap同样如此 作为key的对象必须覆写equals方法(这也是为什么经常用string作key的原因)

通过key计算索引的方式就是调用key对象的hashCode()方法，它返回一个int整数。HashMap正是通过这个方法直接定位key对应的value的索引，继而直接返回value

因此，正确使用Map必须保证：

作为key的对象必须正确覆写equals()方法，相等的两个key实例调用equals()必须返回true；

作为key的对象还必须正确覆写hashCode()方法，且hashCode()方法要严格遵循以下规范：

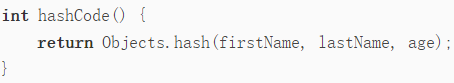
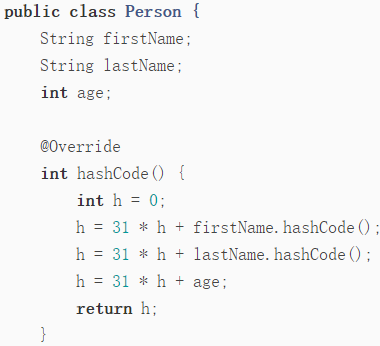
1.如果两个对象相等，则两个对象的hashCode()必须相等；

2.如果两个对象不相等，则两个对象的hashCode()尽量不要相等。

第一条必须保证实现，否则HashMap不能正常工作。

第二条如果尽量满足，则可以保证查询效率，因为不同的对象，如果返回相同的hashCode()，会造成Map内部存储冲突，使存取的效率下降。

Person类中hashCode覆写

 原始写法

原始写法中计算hashCode()时，用31\*h是为了尽量把不同的Person实例的hashCode()均匀分布整个int范围

equals () 用到的用于比较的每一个字段，都必须在hashCode () 中用于计算；equals () 中没有使用到的字段，绝不可放在hashCode () 中计算。

hashCode()返回的int范围高达±21亿，先不考虑负数，HashMap内部使用的数组得有多大？

实际上HashMap初始化时默认的数组大小只有16，任何key，无论它的hashCode()有多大，都可以简单地通过： int index = key.hashCode() & 0xf; // 0xf = 15 把索引确定在0～15，即永远不会超出数组范围，上述算法只是一种最简单的实现。

如果添加超过16个key-value到HashMap，数组不够用了怎么办？

自动扩容 每次乘2 扩容会导致重新分布已有的key-value，所以，频繁扩容对HashMap的性能影响很大 所以最好每次创建时就制定容量 Map<String, Integer> map = new HashMap<>(100); 虽然指定100 但实际上是2的n的整数次

## Iterator&Enumeration

Iterator(迭代器)和Enumeration(枚举类)都可以遍历集合

两个都是接口

Enumeration只能读取集合的数据，而不能对数据进行修改。Iterator除了能读取数据外，也能对数据进行删除

Iterator支持fail-fast机制：当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件 Enumeration不支持 所以Enumeration会稍微快点(进行操作少)

## final finally finalize

final可以修饰类、变量、方法

修饰类表示该类不能被继承、修饰方法表示该法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。

finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块 中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。

finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，该方法一般由垃圾回收器来调 用，当我们调用 System.gc() 方法的时候，由垃圾回收器调用finalize()，回收垃圾，一个对象是否可回收的最后判断。

# 线性表

## 数组

数据是连续的 随机访问速度快

包括：一维数组 多维数组 动态数组 动态数组Java Collection集合中提供了ArrayList和Vector

## 单向链表

链表与数组同级 是一种数据结构 ArrayList---数组实现 LinkedList---链表实现

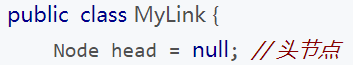
由节点node组成 每一个Node记录本Node的数据及下一个Node位置 向外暴露的只有一个头节点（Head）

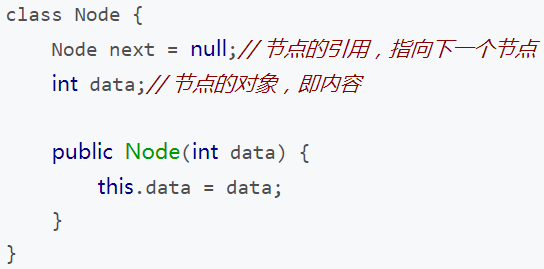


节点拥有两个成员：储存的对象、对下一个节点的引用

节点的链接方向是单向的 随机访问速度慢 但删除/添加数据的效率高

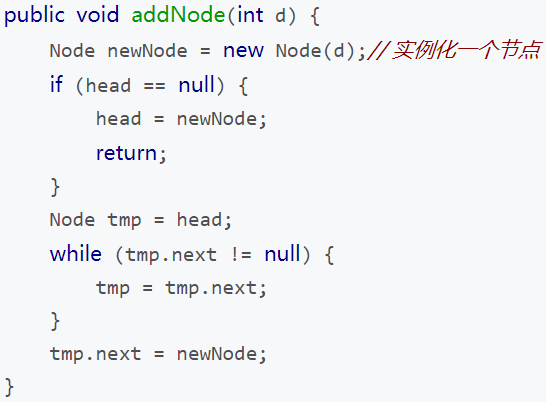
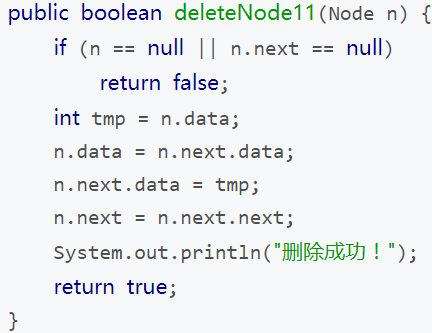
### 创建链表



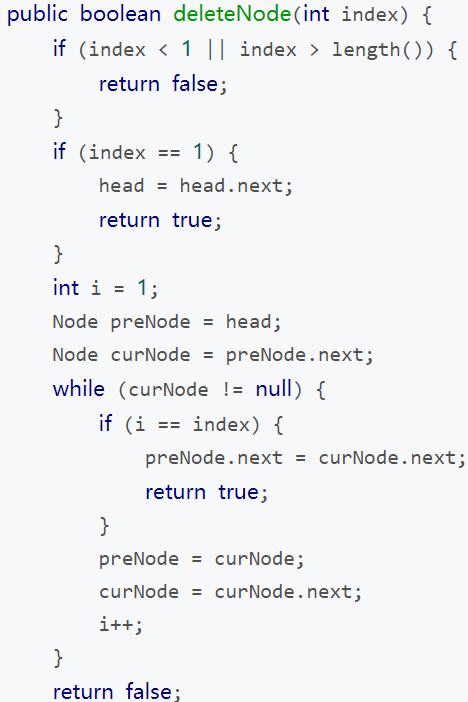


### 链表方法

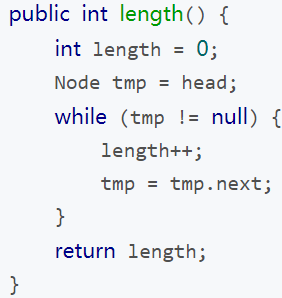
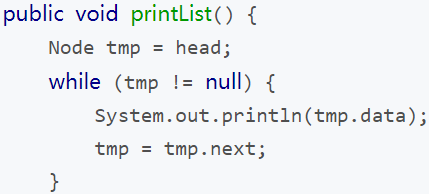
向链表中插入数据 在不知道头指针的情况下删除指定节点

删除第index节点

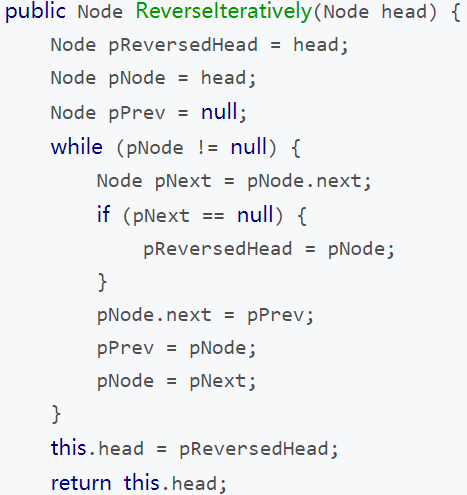


返回节点长度 输出链表

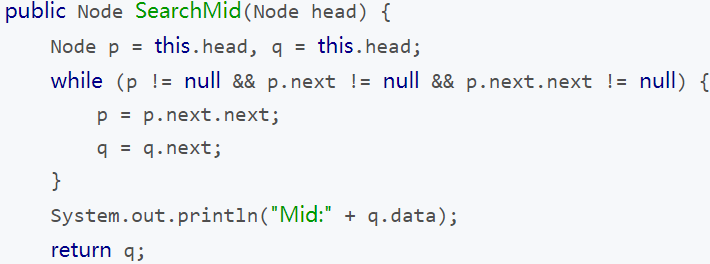
### 链表技巧

反转链表



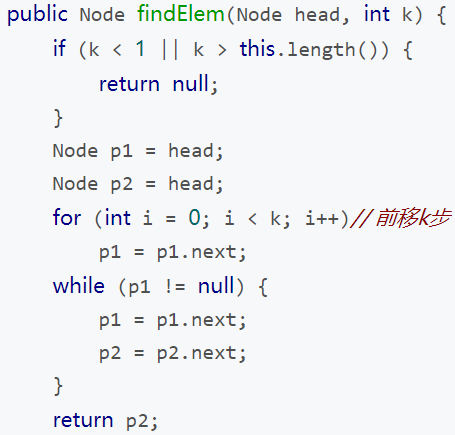
查找单链表中间结点

采用快慢指针的方式查找单链表的中间节点，快指针一次走两步，慢指针一次走一步，当快指针走完时，慢指针刚好到达中间节点

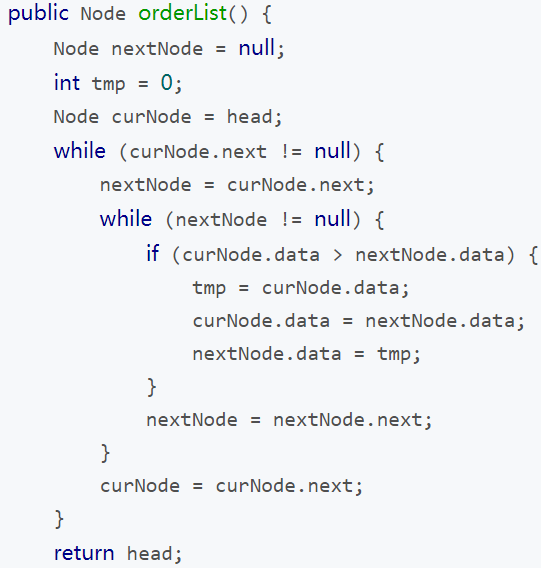


查找倒数第k个元素

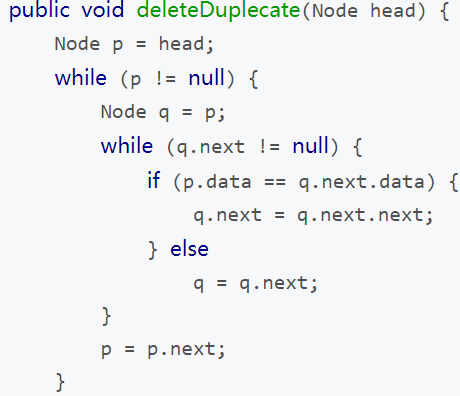
采用两个指针P1,P2，P1先前移K步，然后P1、P2同时移动，当p1移动到尾部时，P2所指位置的元素即倒数第k个元素



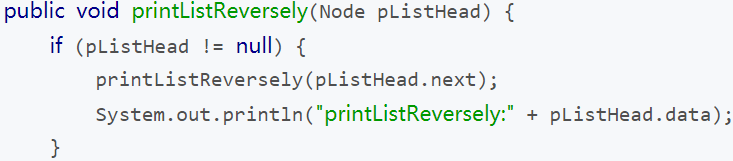
对链表进行排序



删除链表中重复节点

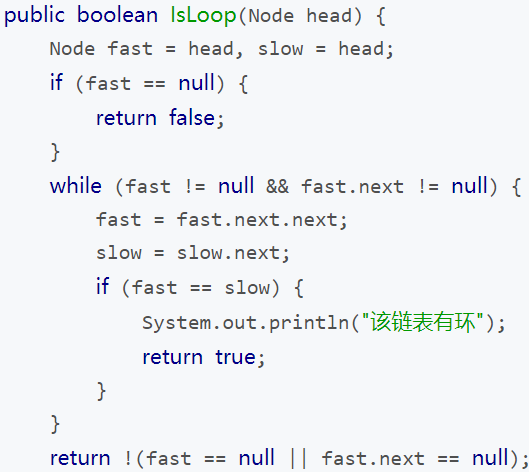


从尾到头输出单链表，采用递归方式实现

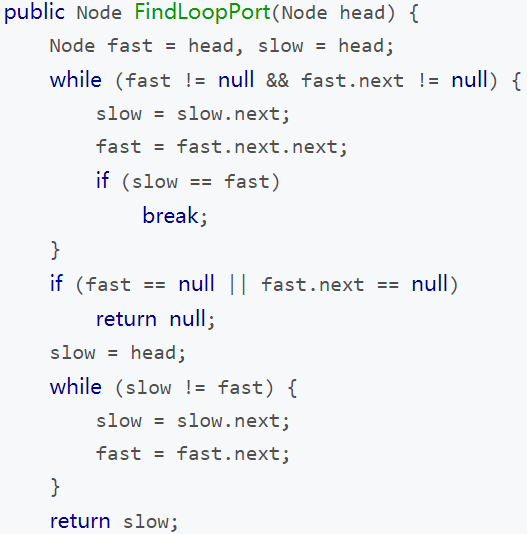


判断链表是否有环，有环情况下找出环的入口节点

判断链表是否有环，单向链表有环时，尾节点相同



找出链表环的入口



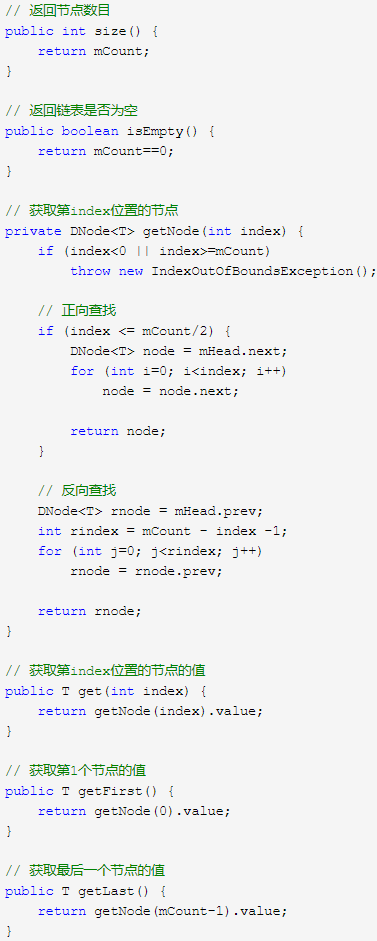
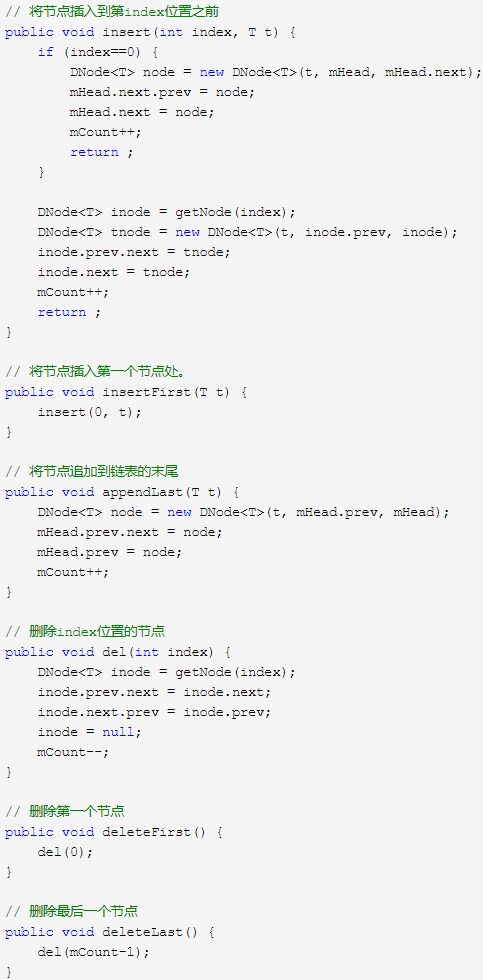
## 双向链表

节点的链接方向是双向的 每个数据结点中都有两个指针 分别指向直接后继和直接前驱

### 双向链表实现



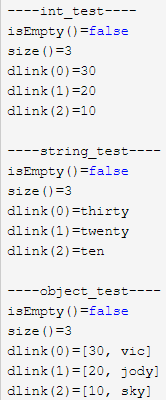
### 方法

### 测试



结果



# 集合类(Collection)：

都继承自collection接口

List 有序列表的集合 LinkedList, ArrayList, Vector, Stack 是List的4个实现类

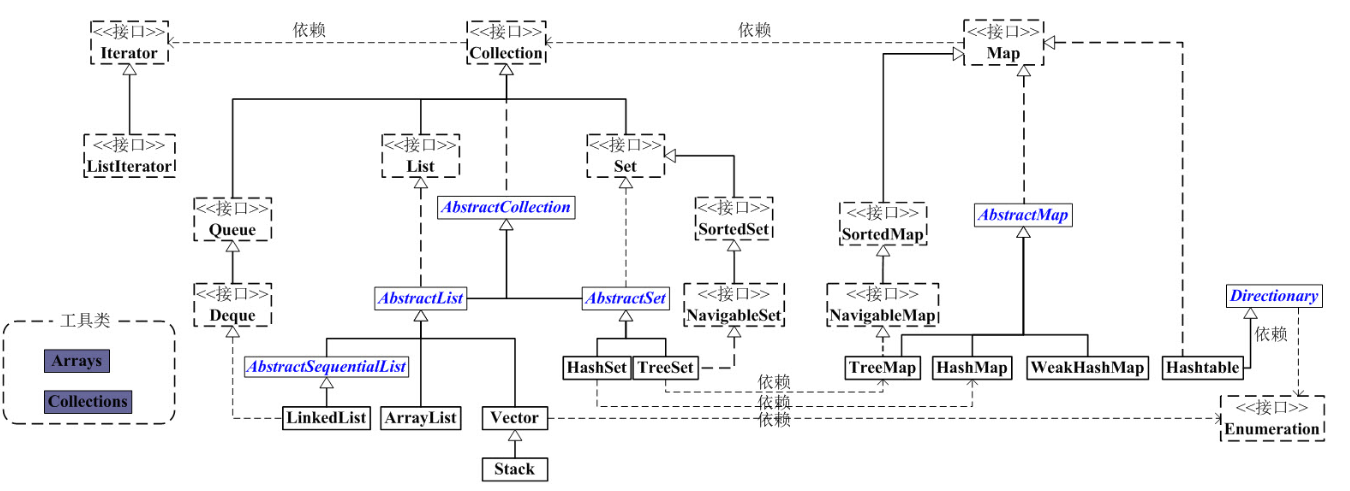
Set 无重复元素的集合(重复覆盖) HastSet和TreeSet。HashSet依赖于HashMap TreeSet依赖于TreeMap

Map 通过键值对查找的映射表集合

Java集合设计的特点：实现接口和实现类相分离 如有序表的接口是List 具体的实现类是ArrayList，LinkedList

支持泛型 可限制一个集合中只能放入同一种数据类型的元素List<String> list = new ArrayList<>();

Java访问集合通过迭代器（Iterator）实现 优点是无需知道集合内部元素按什么方式储存



Map接口和Collection接口是所有集合框架的父接口：

Collection接口的子接口包括：Set接口和List接口

Map接口的实现类主要有：HashMap、TreeMap、Hashtable、ConcurrentHashMap以及Properties等

Set接口的实现类主要有：HashSet、TreeSet、LinkedHashSet等

List接口的实现类主要有：ArrayList、LinkedList、Stack以及Vector等

list 支持for循环，也就是通过下标来遍历，也可以用迭代器，但是set只能用迭代，因为他无序，无法用下标来取得想要的值

List：和数组类似，可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变

Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。

Map：包含键值对的数组

# List

## ArrayList

是一个数组队列 相当于动态数组 与普通数组比它能动态增长

与Vector不同 它不是线程安全 多线程使用Vector

有三个构造函数：1. ArrayList () ；默认容量10 不够时新的容量=“(原始容量x3)/2 + 1”

2. ArrayList(int capacity) 设置初始化容量

3. ArrayList (Collection<? extends E> collection) 创建一个包含collection的ArrayList

继承关系：Object--- AbstractCollection--- AbstractList--- ArrayList

ArrayList的克隆函数（ArrayList.copy()），即是将全部元素克隆到一个数组中。

ArrayList实现java.io.Serializable的方式。当写入到输出流时，先写入“容量”，再依次写入“每一个元素”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”。

序列化：https://www.runoob.com/java/java-serialization.html

三种遍历方式：

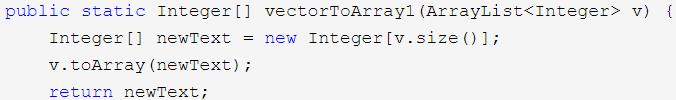
第一种效率最低 第二种效率最高

有时调用toArray()时会出现异常

ArrayList提供了2个toArray()函数： Object[] toArray() <T> T[] toArray(T[] contents)

toArray() 会抛出异常是因为 toArray() 返回的是 Object[] 数组，将 Object[] 转换为其它类型(如如，将Object[]转换为的Integer[])则会抛出“java.lang.ClassCastException”异常，因为Java不支持向下转型

解决该问题的办法是调用 <T> T[] toArray(T[] contents) ， 而不是 Object[] toArray()

使用方式：

常用的API测试程序：<https://www.cnblogs.com/xrq730/p/4989451.html> 具体运作原理



fail-fast 机制是java集合(Collection)中的一种错误机制。当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。

例如：当某一个线程A通过iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出ConcurrentModificationException异常，产生fail-fast事件。

## LinkedList

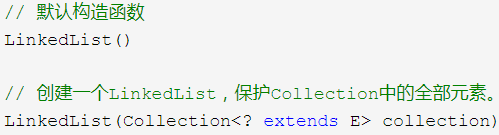
### LinkedList方法

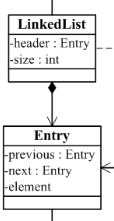
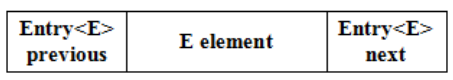
是一个继承于AbstractSequentialList()的双向链表。它也可以被当作堆栈、队列或双端队列进行操作。

实现 List 接口，能对它进行队列操作。实现 Deque 接口，即能将LinkedList当作双端队列使用。

是异步的，线程不安全

AbstractSequentialList 实现了链表中根据index索引值操作链表的全部函数



LinkedList包含两个重要的成员：header 和 size。

　　header是双向链表的表头，它是双向链表节点所对应的类Entry的实例。Entry中包含成员变量：previous, next, element。其中，previous是该节点的上一个节点，next是该节点的下一个节点，element是该节点所包含的值。

　　size是双向链表中节点的个数。

LinkedList为双向链表 顺序访问高效 随机访问效率较低 因为其实现list接口 实现了一些可根据索引值来获取和删除的函数(通过一个计数索引值，例如，当我们调用get(int location)时，首先会比较“location”和“双向链表长度的1/2”；若前者大，则从链表头开始往后查找，直到location位置；否则，从链表末尾开始先前查找，直到location位置。)

LinkedList内部包含一个重要的内部类Entry，Entry是双向链表节点所对应的数据结构

不存在LinkedList容量不足的问题



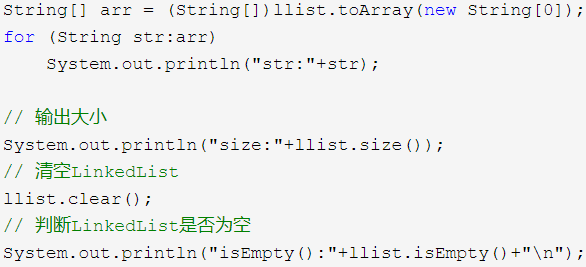
LinkedList遍历

使用removeFist()或removeLast()效率最高。但用它们遍历时，会删除原始数据

应使用for循环遍历for (Integer integ:list)

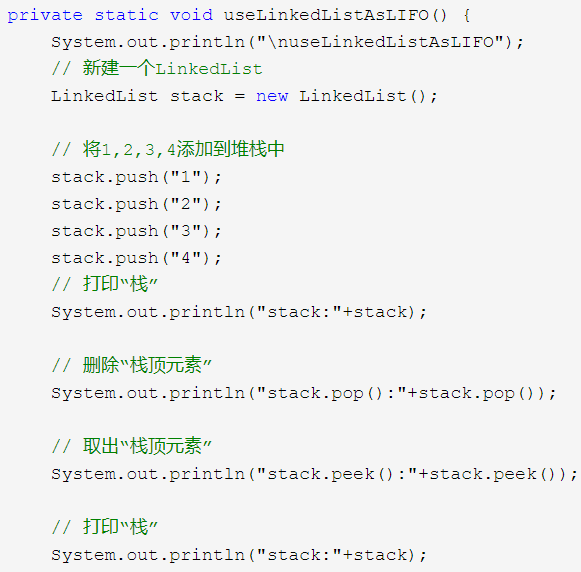
不要用随机访问(for i=0;i++) 效率太低

将LinkedList转行为数组



### LinkedList实现堆栈

将LinkedList当作 LIFO(后进先出)的堆栈



### LinkedList实现队列

将LinkedList当作 FIFO(先进先出)的队列



## 队列(Queue)

队列是一种特殊的线性表，它只允许在表的前端进行删除操作，而在表的后端进行插入操作。

LinkedList类实现了Queue和Deque接口，因此我们可以把LinkedList当成Queue/Deque来用



推荐使用offer poll peek 满元素/空集合/空队列 时会返回flase/null/null而不是异常

### 队列(Vector实现)

通过数组实现 先进后出

是矢量队列，实现了List；所以，它是一个队列，支持相关的添加、删除、修改、遍历等功能。

提供了随机访问功能，在Vector中，我们即可以通过元素的序号快速获取元素对象；这就是快速随机访问。

Vector的默认容量大小是10

当Vector容量不足以容纳全部元素时，Vector的容量会增加。若容量增加系数 >0，则将容量的值增加“容量增加系数”；否则，将容量大小增加一倍。

Vector的克隆函数，即是将全部元素克隆到一个数组中

ArrayList已经可以完全的替代Vector Vector线程安全，ArrayList线程不安全

### 队列(Deque实现)



## 堆栈(Stack)

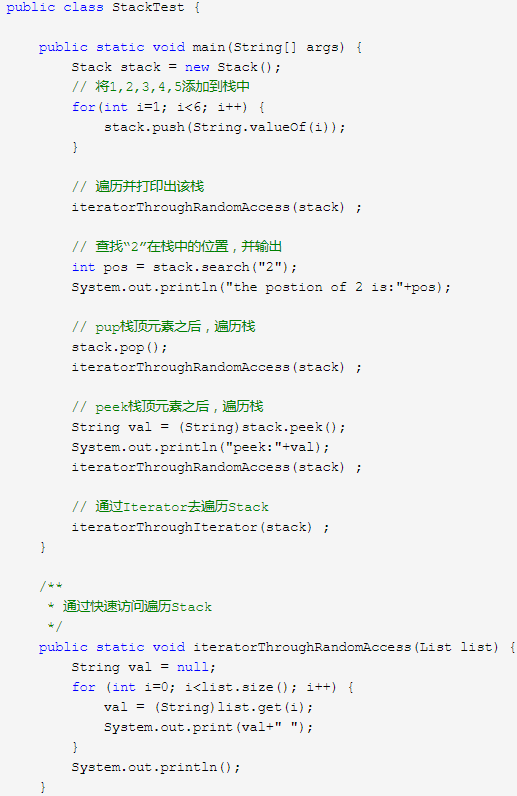
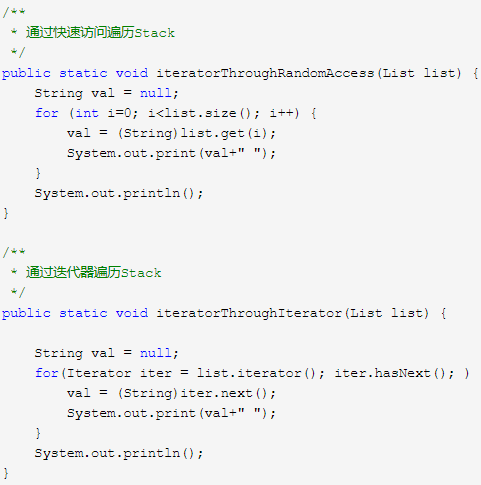
栈 先进后出 继承于Vector 通过数组实现

执行push时(即将元素推入栈中)将元素追加的数组的末尾中。

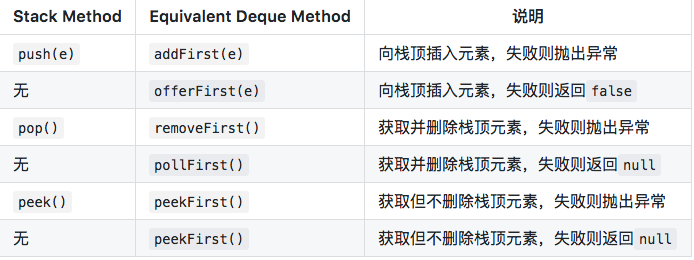
执行peek时(即取出栈顶元素，不执行删除)返回数组末尾的元素。

执行pop时(即取出栈顶元素，并将该元素从栈中删除)取出数组末尾的元素并将该元素从数组中删除。

### 栈(Stack实现)

### 栈(Deque实现)



要实现Stack，至少应该包括：

1. pop() 出栈操作，弹出栈顶元素。

2. push(E e) 入栈操作

3. peek() 查看栈顶元素(第一个元素)

4. isEmpty() 栈为空

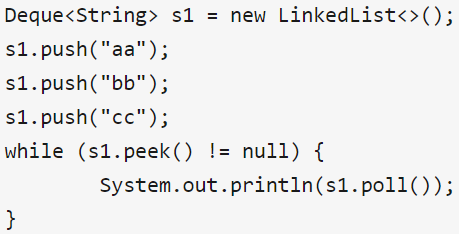
还应该考虑到几个问题：

1. 栈的初始大小以及栈满以后如何新增栈空间

2. 对栈进行更新时需要进行同步

栈的三种实现 <https://segmentfault.com/a/1190000002516799>

Java的新特性中的封装方法deque



Deque既可以用作后进先出的栈，也可以用作先进先出的队列

## 总结List

ArrayList 是一个数组队列，相当于动态数组。它由数组实现，随机访问效率高，随机插入、随机删除效率低。

LinkedList 是一个双向链表。它也可以被当作堆栈、队列或双端队列进行操作。LinkedList随机访问效率低，但随机插入、随机删除效率高。

Vector 是矢量队列，和ArrayList一样，它也是一个动态数组，由数组实现。但是ArrayList是非线程安全的，而Vector是线程安全的。

Stack 是栈，它继承于Vector。它的特性是：先进后出(FILO, First In Last Out)。

使用情况：

如果涉及到栈，队列，链表等操作，应该考虑用List，具体的选择哪个List，根据下面的标准来取舍。

对于需要快速插入，删除元素，应该使用LinkedList。

对于需要快速随机访问元素，应该使用ArrayList。

对于”*单线程环境*”或者”*多线程环境，但List仅仅只会被单个线程操作*”，此时应该使用异步的类如ArrayList，LinkedList

对于”多线程环境，且List可能同时被多个线程操作”，此时应该使用同步的类(如Vector，Stack)。

LinkedList中插入元素很快，而ArrayList中插入元素很慢原因：

LinkedList先是在双向链表中找到要插入节点的位置index；找到之后，再插入一个新节点。双向链表查找index位置的节点时，有一个加速动作：若index < 双向链表长度的1/2，则从前向后查找; 否则，从后向前查找。

ArrayList会移动index之后所有元素即可。这就意味着，ArrayList的add(int index, E element)函数，会引起index之后所有元素的改变

LinkedList中随机访问很慢，而ArrayList中随机访问很快

LinkedList获取LinkedList第index个元素时。先是在双向链表中找到要index位置的元素；找到之后再返回。

双向链表查找index位置的节点时，有一个加速动作：若index < 双向链表长度的1/2，则从前向后查找; 否则，从后向前查找

ArrayList通过get(int index)获取ArrayList第index个元素时。直接返回数组中index位置的元素，而不需要像LinkedList一样进行查找

Vector和ArrayList

同：都是List 都支持快速随机访问RandomAccess 都是通过数组实现的，本质上都是动态数组

默认数组容量是10

异：线程安全性不一样

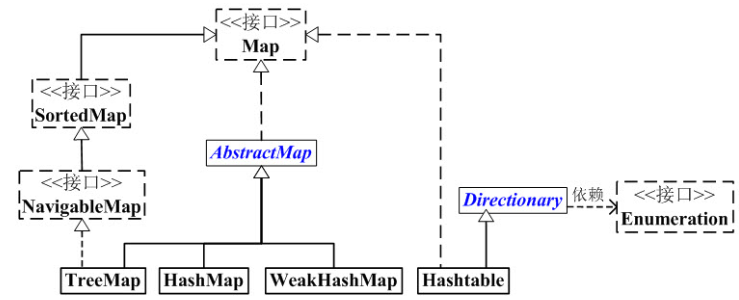
容量增加方式不同

ArrayList容量不足时，新的容量=(原始容量x3)/2 + 1

Vector的容量增长与增长系数有关，若指定了增长系数，且增长系数有效(大于0)；那么，每次容量不足时，新的容量=原始容量+增长系数。若增长系数无效(即小于/等于0)，则新的容量=原始容量 x 2

# Map

Map架构



Map是映射接口 里面储存的内容是键值对 Map映射中不能包含重复的键；每个键最多只能映射到一个值。

TreeMap实现了NavigableMap接口 因此TreeMap中的内容是**有序**的键值对

HashMap没有实现NavigableMap接口 里面的内容是**无序**的键值对

Hashtable和HashMap一样 不过Hashtable是线程安全的，而且它支持通过Enumeration去遍历

## HashMap(散列表)

1.HashMap里面存储的内容是键值对(key-value是一个整体) 映射 键值对储存在table数组中 table是一个Entry[]数组类型 table=new Entry[]

2.异步 线程不安全 Key，value值都可为null 映射无序

3.两个参数影响性能(初始容量(默认16)和加载因子(默认0.75)) 当大于容量和因子乘积时 进行rehash操作(double容量)

4.当我们想一个HashMap中添加一对key-value时，系统首先会计算key的hash值，然后根据hash值确认在table中存储的位置。若该位置没有元素，则直接插入。否则迭代该处元素链表并依此比较其key的hash值。如果两个hash值相等且key值相等(e.hash == hash && ((k = e.key) == key || key.equals(k))),则用新的Entry的value覆盖原来节点的value。如果两个hash值相等但key值不等 ，则将该节点插入该链表的链头

HashMap是通过"拉链法"实现的哈希表(解决哈希冲突)

成员变量包括：table size threshold loadFactor modCount

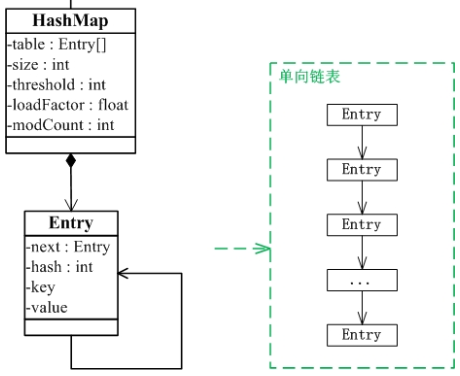
table是一个Entry[]数组类型，而Entry实际上是一个单向链表。哈希表的"key-value键值对"都是存储在Entry数组中的。其中Entry为HashMap的内部类，它包含了键key，值value，下一个节点next，hash值

size是HashMap的大小，它是HashMap保存的键值对的数量。默认16

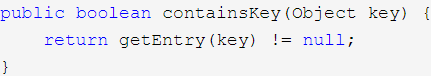
threshold是HashMap的阈值，用于判断是否需要调整HashMap的容量。threshold的值="容量\*加载因子"，当HashMap中存储数据的数量达到threshold时，就需要将HashMap的容量加倍。

loadFactor就是加载因子---0.75

modCount是用来实现fail-fast机制的



containsKey() 的作用是判断HashMap是否包含key。



containsKey() 首先通过getEntry(key)获取key对应的Entry，然后判断该Entry是否为null。



HashMap将key为null的元素都放在table的位置0处，即table[0]中；key不为null的放在table的其余位置

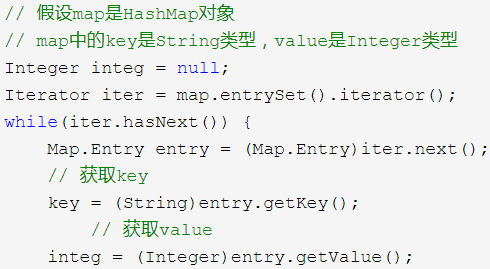
containsValue()和containsKey()的实现基本都分为两步

若key/value值为null查是否表中含null 不为null查是否值相等

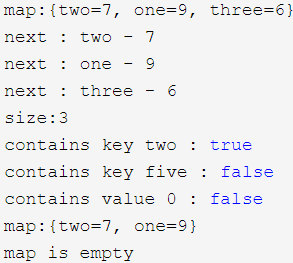
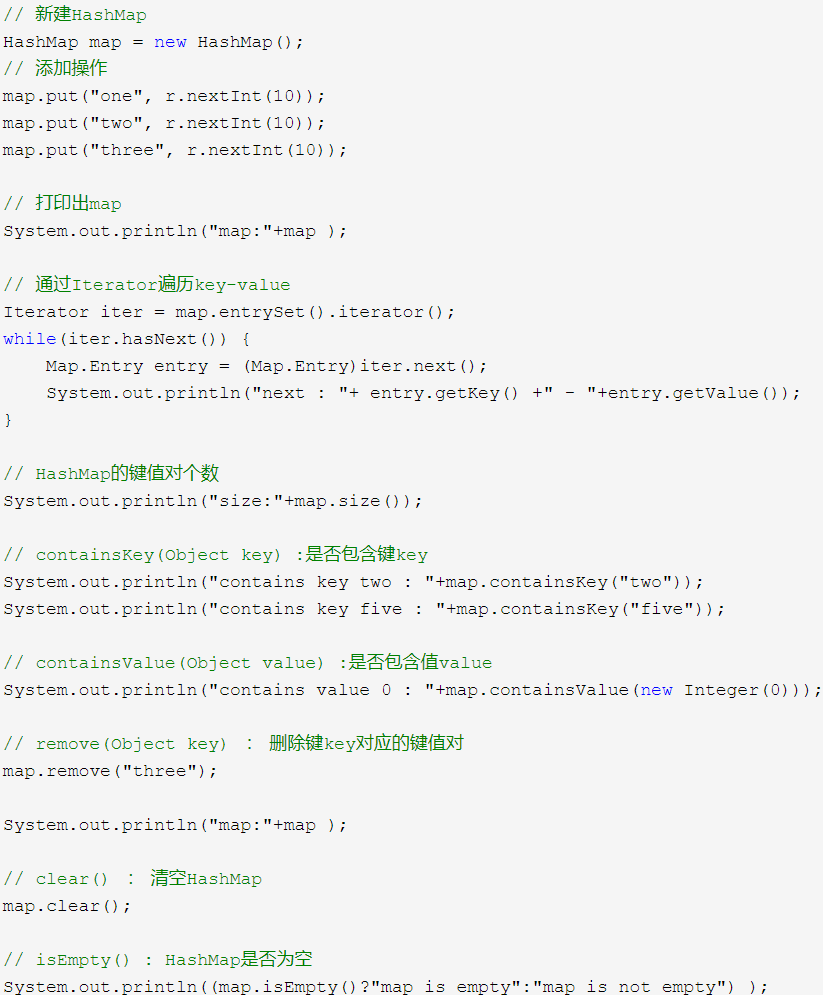
遍历HashMap的键值对

第一步：根据entrySet()获取HashMap的“键值对”的Set集合。

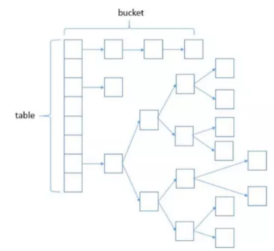
第二步：通过Iterator迭代器遍历“第一步”得到的集合。



HashMap示例



HashMap 1.7 与 1.8 的 区别，说明 1.8 做了哪些优化？



JDK1.7 及之前的版本中， HashMap的hash值冲突的时候， 就将对应节点以链表的形式存储。

JDK1.8 中，当同一个hash值（ Table 上元素）的链表节点数不小于8时，将不再以单链表的形式存储了，会被调整成一颗红黑树。这就是 JDK7 与 JDK8 中 HashMap 实现的最大区别。

其下基于 JDK1.7.0\_80 与 JDK1.8.0\_66 做的分析

JDK1.7中

使用一个 Entry 数组来存储数据，用key的HashCode 取模来决定key会被放到数组里的位置，如果 HashCode相同，或者 HashCode 取模后的结果相同（ hash collision ），那么这些 key 会被定位到 Entry 数组的同一个 格子里，这些 key 会形成一个链表。

在 HashCode 特别差的情况下，比方说所有key的 HashCode 都相同，这个链表可能会很长，那么 put/get 操作 都可能需要遍历这个链表，也就是说时间复杂度在最差情况下会退化到 O(n)

JDK1.8中

使用一个 Node 数组来存储数据，但这个 Node 可能是链表结构，也可能是红黑树结构

如果插入的 key 的 HashCode 相同，那么这些key也会被定位到 Node 数组的同一个格子里。

如果同一个格子里的key不超过8个，使用链表结构存储。

如果超过了8个，那么会调用 treeifyBin 函数，将链表转换为红黑树。

那么即使 HashCode 完全相同，由于红黑树的特点，查找某个特定元素，也只需要O(log n)的开销

也就是说put/get的操作的时间复杂度最差只有 O(log n) 听起来挺不错，但是真正想要利用 JDK1.8 的好处，有一个限制： key的对象，必须正确的实现了 Compare 接口 如果没有实现 Compare 接口，或者实现得不正确（比方说所有 Compare 方法都返回0） 那 JDK1.8 的 HashMap 其实还是慢于 JDK1.7 的

HashMap 是线程安全的吗，为什么不是线程安全的?

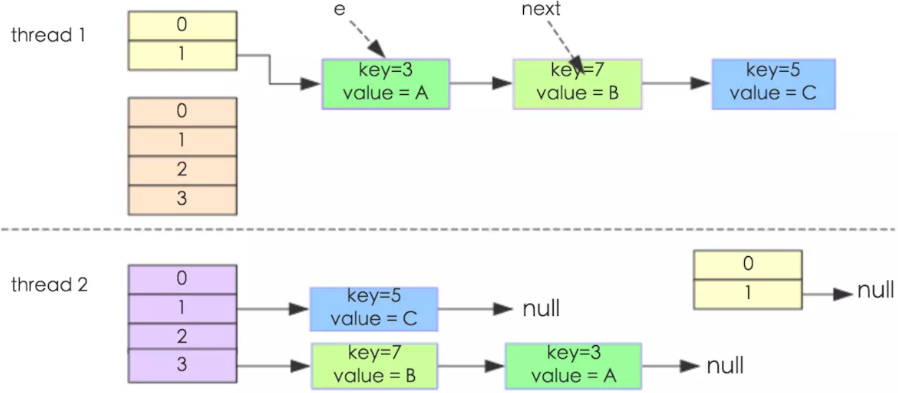
1、put的时候导致的多线程数据不一致。

这个问题比较好想象，比如有两个线程A和B，首先A希望插入一个key-value对到HashMap中，首先计算记录所要落到的桶的索引坐标，然后获取到该桶里面的链表头结点，此时线程A的时间片用完了，而此时线程B被调度得以执行，和线程A一样执行，只不过线程B成功将记录插到了桶里面，假设线程A插入的记录计算出来的桶索引和线程B要插入的记录计算出来的桶索引是一样的，那么当线程B成功插入之后，线程A再次被调度运行时，它依然持有过期的链表头但是它对此一无所知，以至于它认为它应该这样做，如此一来就覆盖了线程B插入的记录，这样线程B插入的记录就凭空消失了，造成了数据不一致的行为

2. HashMap的get操作可能因为resize而引起死循环

我们假设有两个线程同时需要执行resize操作，我们原来的桶数量为2，记录数为3，需要resize桶到4，原来的记录分别为：[3,A],[7,B],[5,C]，在原来的map里面，我们发现这三个entry都落到了第二个桶里面

假设线程thread1执行到了transfer方法的Entry next = e.next这一句，然后时间片用完了，此时的e = [3,A], next = [7,B]。线程thread2被调度执行并且顺利完成了resize操作，需要注意的是，此时的[7,B]的next为[3,A]。此时线程thread1重新被调度运行，此时的thread1持有的引用是已经被thread2 resize之后的结果。线程thread1首先将[3,A]迁移到新的数组上，然后再处理[7,B]，而[7,B]被链接到了[3,A]的后面，处理完[7,B]之后，就需要处理[7,B]的next了啊，而通过thread2的resize之后，[7,B]的next变为了[3,A]，此时，[3,A]和[7,B]形成了环形链表，在get的时候，如果get的key的桶索引和[3,A]和[7,B]一样，那么就会陷入死循环。



如果在取链表的时候从头开始取（现在是从尾部开始取）的话，则可以保证节点之间的顺序，那样就不存在这样的问题了

 为什么是2的幂次： 



## HashTable(散列表)

Hashtable 的函数都是同步的 线程安全 key、value都不可以为null 映射无序

HashMap当key = null时，也有hash值 = 0，所以HashMap的key 可为null

对比HashTable，由于HashTable对key直接hashCode() 若key为null时 会抛出异常 所以HashTable的key不可为null

HashMap的键key 可为null且只能为1个，但值value可为null且为多个

其他同HashMap

HashMap没有考虑同步，是线程不安全的；Hashtable使用了synchronized关键字，是线程安全的；

HashMap允许K/V都为null；后者K/V都不允许为null；

HashMap继承自AbstractMap类；而Hashtable继承自Dictionary类；

## TreeMap(基于红黑树实现)

TreeMap 是一个有序的key-value集合，它是通过红黑树实现的。

该映射根据其键的自然顺序进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。

TreeMap的基本操作 containsKey、get、put 和 remove 的时间复杂度是 log(n) 。

TreeMap是异步的。 它的iterator 方法返回的迭代器是fail-fast的

## WeakHashMap

# Set

Set 是继承于Collection的接口。它是一个不允许有重复元素的集合

Set的实现类都是基于Map来实现的

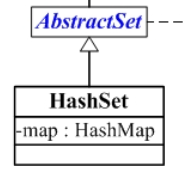
HashSet依赖于HashMap，它实际上是通过HashMap实现的。 HashSet中的元素是无序的。

TreeSet依赖于TreeMap，它实际上是通过TreeMap实现的。 TreeSet中的元素是有序的。

## Hashset

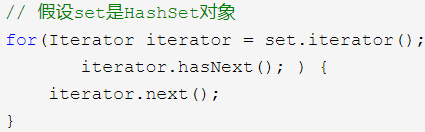
一个没有重复元素的集合 元素无序 允许null元素 异步

如果多个线程同时访问一个哈希 set，而其中至少一个线程修改了该 set，那么它必须保持外部同步---封装该set对象并执行同步操作，最好在创建时完成这一操作 Set s = Collections.synchronizedSet(new HashSet(...));



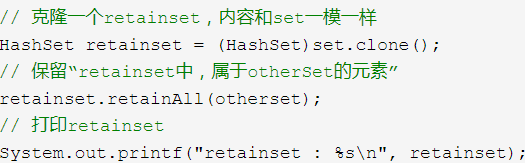
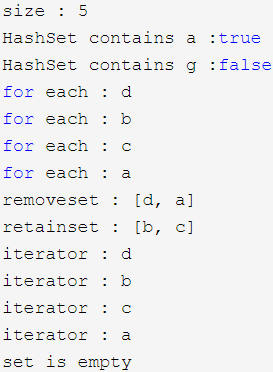
HashSet中含有一个"HashMap类型的成员变量"map，HashSet的操作函数，实际上都是通过map实现的。

遍历



实例



HashSet 是如何保证不重复的

向 HashSet 中 add ()元素时，判断元素是否存在的依据，不仅要比较hash值，同时还要结合 equals 方法比较。HashSet 中的 add () 方法会使用 HashMap 的 add () 方法。

HashMap 的 key 是唯一的， HashSet 添加进去的值就是作为 HashMap 的key。所以不会重复（ HashMap 比较key是否相等是先比较 HashCode 在比较 equals ）。

## Treeset

## 内部类

内部类的好处：

每个内部类都能独立地继承一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响。

在程序设计中有时候会存在一些使用接口很难解决的问题，这个时候我们可以利用内部类提供的、可以继承多个具体的或者抽象的类的能力来解决这些程序设计问题。可以这样说，接口只是解决了部分问题，而内部类使得**多重继承**的解决方案变得更加完整。

Father类 Mother类 Son类 son类中可以出现两个内部类一个继承father一个继承mother

内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立

在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类

创建内部类对象的时刻并不依赖于外围类对象的创建。

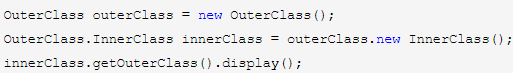
内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。

内部类提供了更好的封装，除了该外围类，其他类都不能访问。

内部类可以使用外部类的成员变量和方法(即使private也可以)

外围类要访问内部类需要先创建内部类实例再进行调用

成员内部类中不能存在任何static的变量和方法

或者

静态内部类

创建不需要依赖外部类

只能使用外部类的static成员变量和static方法



匿名内部类

必须继承一个父类或实现一个接口 同时也只能继承一个类或者实现一个接口

匿名内部类中是不能定义构造函数的。

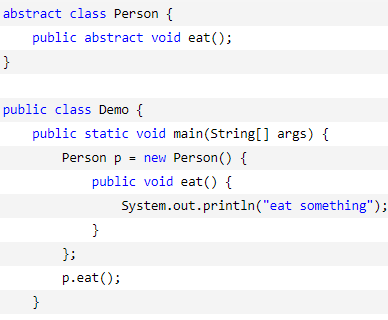
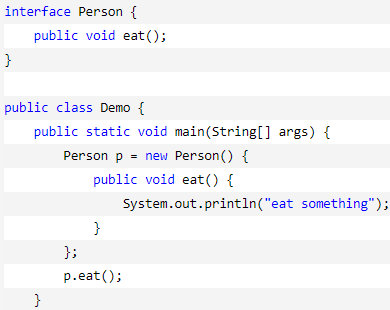
匿名内部类中不能存在任何的静态成员变量和静态方法。

匿名内部类为局部内部类，所以局部内部类的所有限制同样对匿名内部类生效。

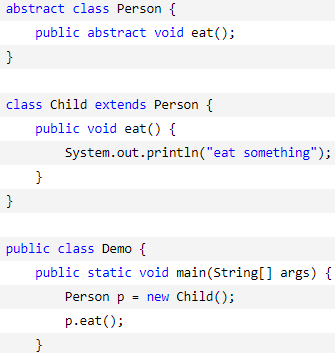
匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。

给匿名内部类传递参数的时候，若该形参在内部类中需要被使用，那么该形参必须要为final。也就是说：当所在的方法的形参需要被内部类里面使用时，该形参必须为final。

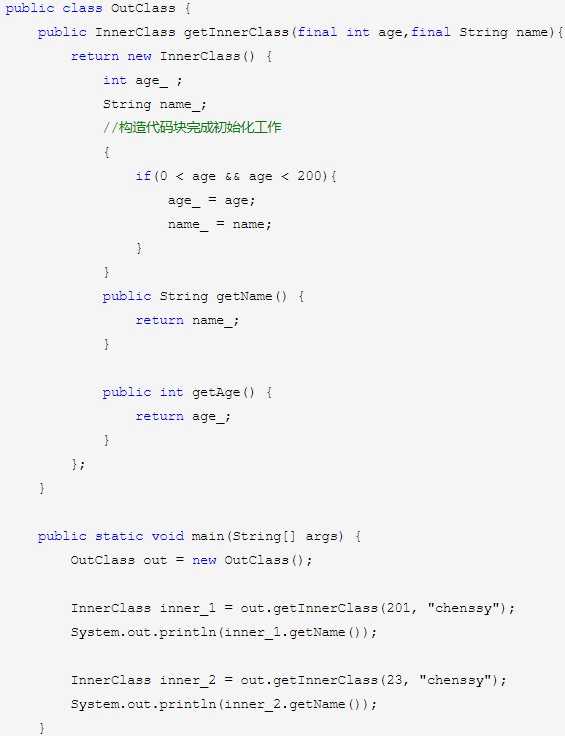
对于抽象类 对于接口

若不使用匿名内部类

用Child继承了Person类，然后实现了Child的一个实例，将其向上转型为Person类的引用 此处的Child类只使用一次，将其编写为独立的一个类很麻烦

利用构造代码块能够达到为匿名内部类创建一个构造器的效果



判断一个元素在亿级数据中是否存在？HashMap会内存溢出

## 布隆过滤器

它主要就是用于解决判断一个元素是否在一个集合中，但它的优势是只需要占用很小的内存空间以及有着高效的查询效率。

对写入的数据做 H 次 hash 运算定位到数组中的位置，同时将数据改为 1 。当有数据查询时也是同样的方式定位到数组中。 一旦其中的有一位为 0 则认为数据肯定不存在于集合，否则数据可能存在于集合中。

特点：

只要返回数据不存在，则肯定不存在。

返回数据存在，但只能是大概率存在。

同时不能清除其中的数据。

为什么返回存在的数据却是可能存在呢，这其实也和 HashMap 类似。

在有限的数组长度中存放大量的数据，即便是再完美的 Hash 算法也会有冲突，所以有可能两个完全不同的 A、B 两个数据最后定位到的位置是一模一样的。这时拿 B 进行查询时那自然就是误报了。

删除数据也是同理，当我把 B 的数据删除时，其实也相当于是把 A 的数据删掉了，这样也会造成后续的误报。

基于以上的 Hash 冲突的前提，所以 BloomFilter 有一定的误报率，这个误报率和 Hash 算法的次数 H，以及数组长度 L 都是有关的。

## 图

由点-顶点(vertex)和连线-边/弧(edge)组成 通常记为G=(V,E)

一条边上的两个顶点叫做邻接点

顶点的入边，是指以该顶点为终点的边。而顶点的出边，则是指以该顶点为起点的边

无向图中，某个顶点的度是邻接到该顶点的边(或弧)的数目

有向图中，某个顶点的入度，是指以该顶点为终点的边的数目。而顶点的出度，则是指以该顶点为起点的边的数目。顶点的度=入度+出度

路径：如果顶点(Vm)到顶点(Vn)之间存在一个顶点序列。则表示Vm到Vn是一条路径。

路径长度：路径中"边的数量"。

简单路径：若一条路径上顶点不重复出现，则是简单路径。

回路：若路径的第一个顶点和最后一个顶点相同，则是回路。

简单回路：第一个顶点和最后一个顶点相同，其它各顶点都不重复的回路则是简单回路。

连通图：对无向图而言，任意两个顶点之间都存在一条无向路径，则称该无向图为连通图。 对有向图而言，若图中任意两个顶点之间都存在一条有向路径，则称该有向图为强连通图。

连通分量：非连通图中的各个连通子图称为该图的连通分量。

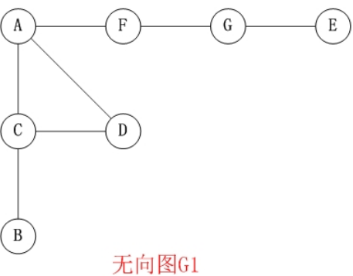
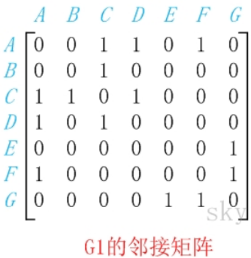
图的存储结构，常用的是"邻接矩阵"和"邻接表"。

邻接矩阵：一个一维数组用来保存顶点信息，一个二维数组来用保存边的信息。  
优点是可以快速判断两个顶点之间是否存在边，可以快速添加边或者删除边。而其缺点是如果顶点之间的边比较少，会比较浪费空间。因为是一个 n∗n 的矩阵。

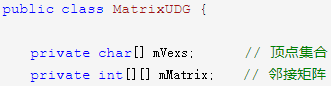
邻接表是图的一种链式存储表示方法。它是改进后的"邻接矩阵"

优点是节省空间，只存储实际存在的边。其缺点是关注顶点的度时，就可能需要遍历一个链表。还有一个缺点是，对于无向图，如果需要删除一条边，就需要在两个链表上查找并删除

### 邻接矩阵无向图

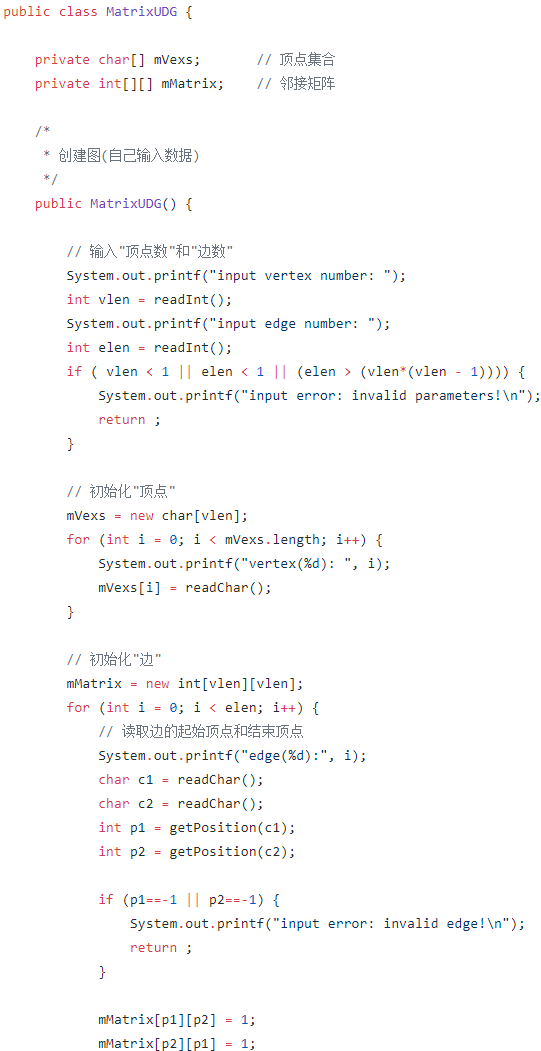
 

基本定义

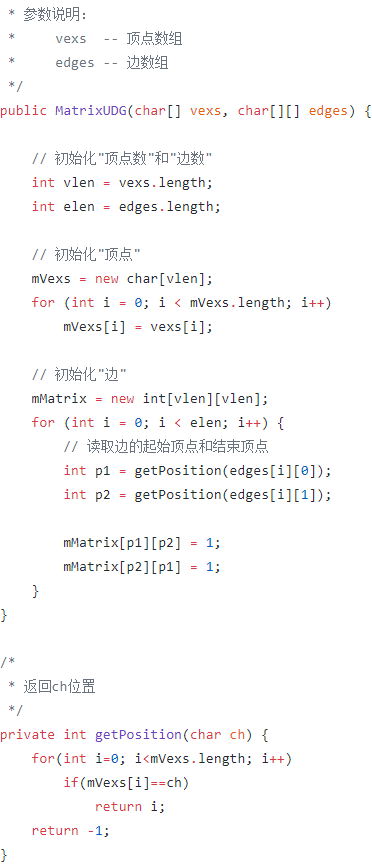
MatrixDG是邻接矩阵有向图对应的结构体

mVexs用于保存顶点，mMatrix则是用于保存矩阵信息的二维数组

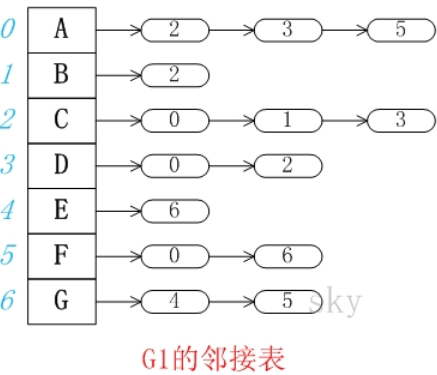
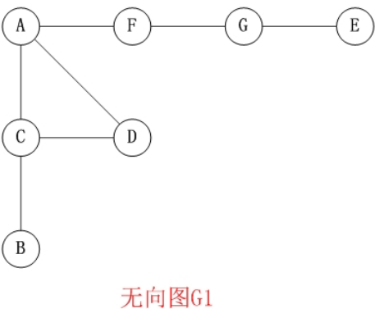
创建图(自己输入数据)

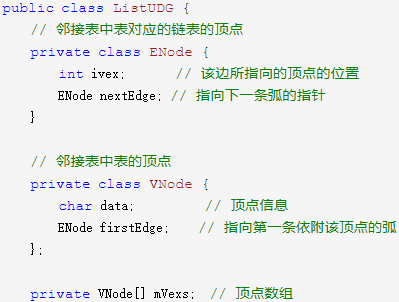


创建图(用已提供的矩阵)

### 邻接表无向图



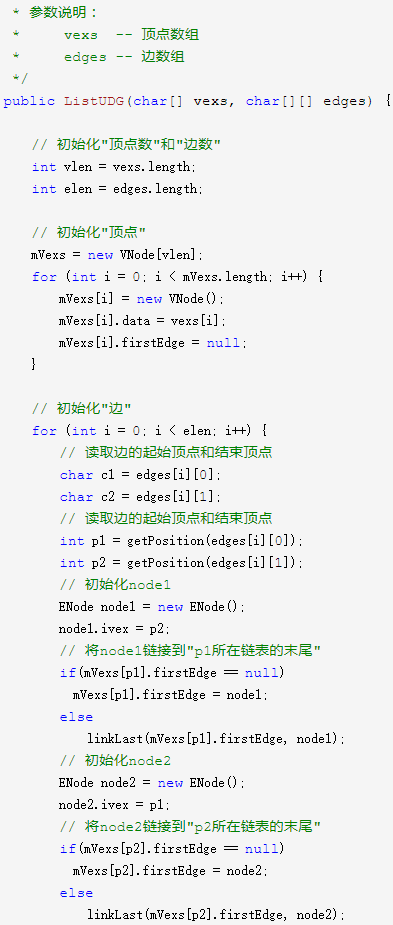
基本定义

ListUDG是邻接表对应的结构体。mVexs则是保存顶点信息的一维数组

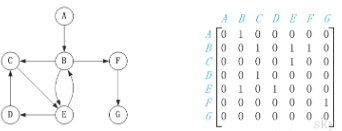
VNode是邻接表顶点对应的结构体。 data是顶点所包含的数据，而firstEdge是该顶点所包含链表的表头指针。

ENode是邻接表顶点所包含的链表的节点对应的结构体。 ivex是该节点所对应的顶点在vexs中的索引，而nextEdge是指向下一个节点的

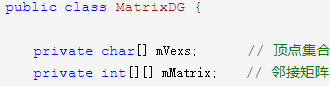
创建图(用已提供的矩阵) 创建图(自己输入数据)

### 邻接矩阵有向图



基本定义

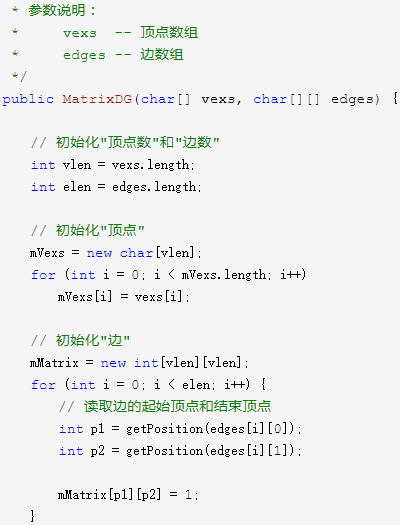
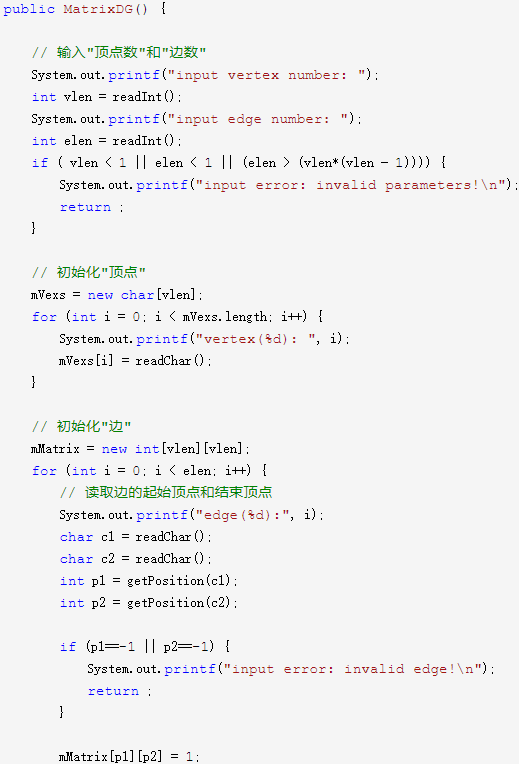


MatrixDG是邻接矩阵有向图对应的结构体

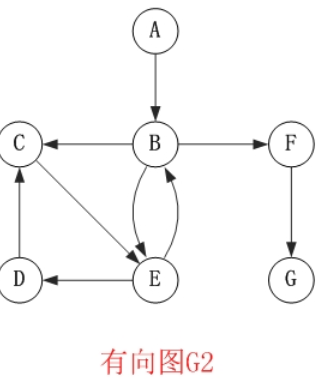
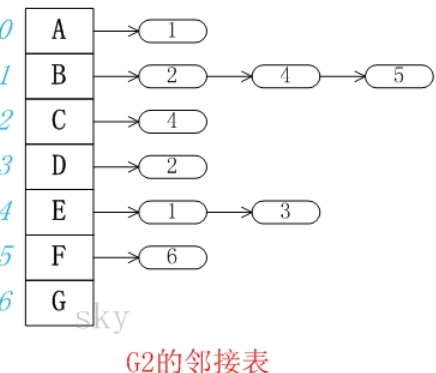
mVexs用于保存顶点，mMatrix则是用于保存矩阵信息的二维数组

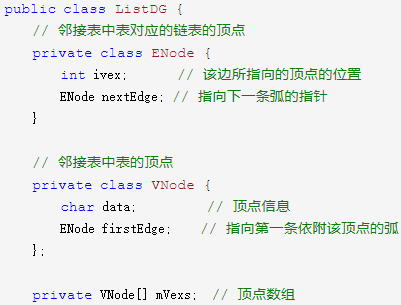
mMatrix[i][j]=1，则表示"顶点i(即mVexs[i])"和"顶点j(即mVexs[j])"是邻接点，且顶点i是起点，顶点j是终点

创建图(用已提供的矩阵) 创建图(自己输入数据)

### 邻接表有向图

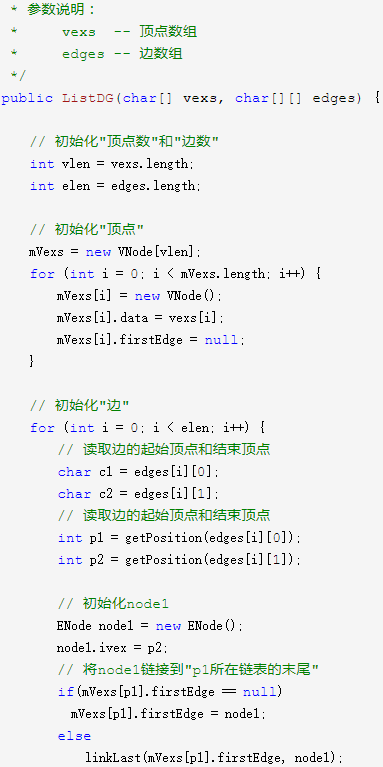
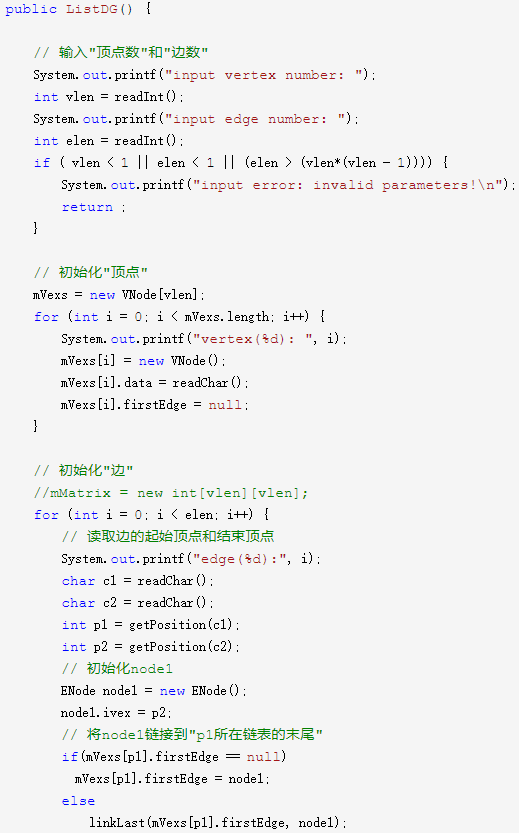
基本定义

ListDG是邻接表对应的结构体。 mVexs是保存顶点信息的一维数组

VNode是邻接表顶点对应的结构体。 data是顶点所包含的数据，而firstEdge是该顶点所包含链表的表头指针

ENode是邻接表顶点所包含的链表的节点对应的结构体。 ivex是该节点所对应的顶点在vexs中的索引，而nextEdge是指向下一个节点的

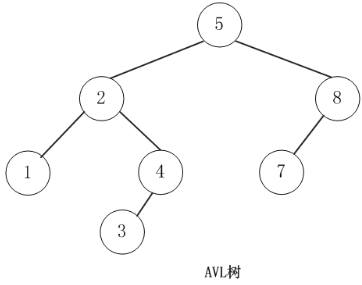
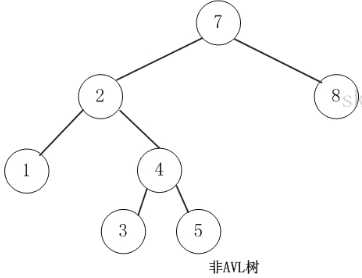
创建图(用已提供的矩阵) 创建图(自己输入数据)

# Round 2

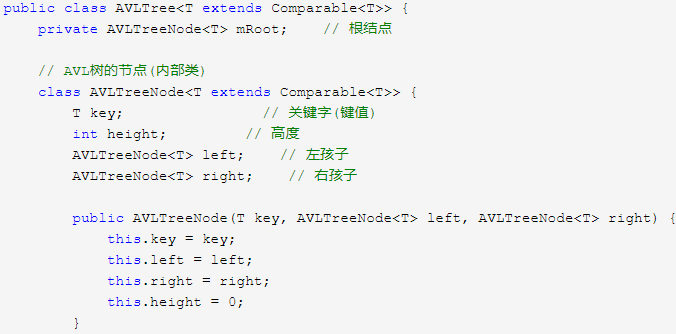
# AVL树

AVL树是高度平衡的二叉树。它的特点是：AVL树中任何节点的两个子树的高度最大差别为1

左边的是AVL树，它的任何节点的两个子树的高度差别都<=1；而右边的不是AVL树，因为7的两颗子树的高度相差为2(以2为根节点的树的高度是3，而以8为根节点的树的高度是1)

节点定义



AVLTree是AVL树对应的类，而AVLTreeNode是AVL树节点，它是AVLTree的内部类。AVLTree包含了AVL树的根节点，AVL树的基本操作也定义在AVL树中。AVLTreeNode包括的几个组成对象:

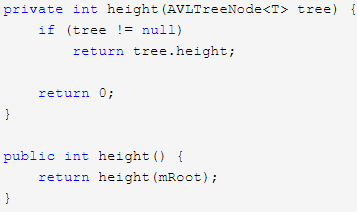
(01) key -- 是关键字，是用来对AVL树的节点进行排序的。

(02) left -- 是左孩子。

(03) right -- 是右孩子。

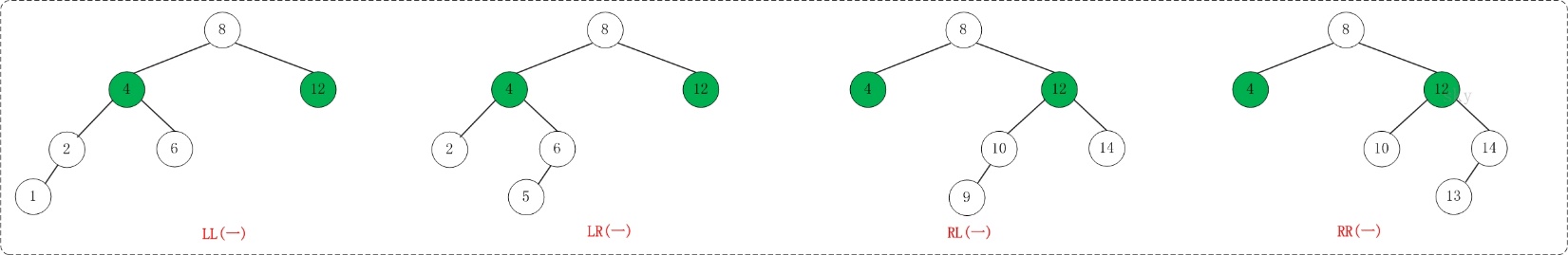
(04) height -- 是高度。

获取树高

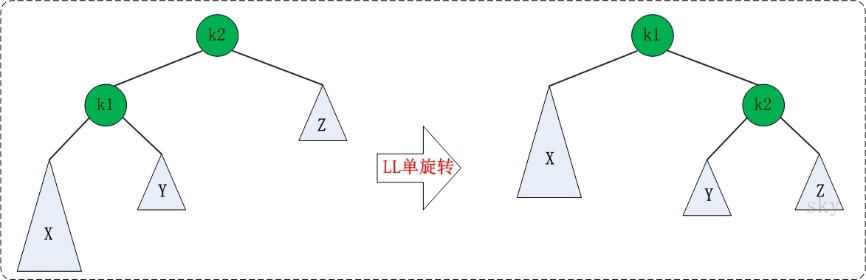


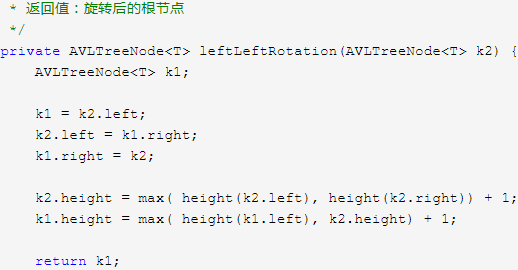
旋转

AVL树中进行插入或删除节点后，可能导致AVL树失去平衡。这种失去平衡的可以概括为4种姿态：LL(左左)，LR(左右)，RR(右右)和RL(右左) 可通过旋转使其恢复平衡

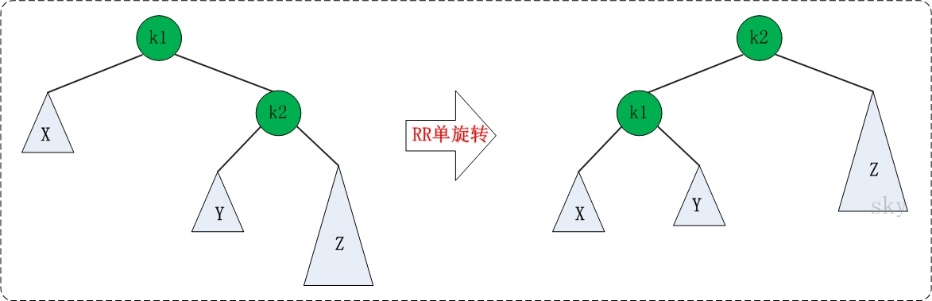


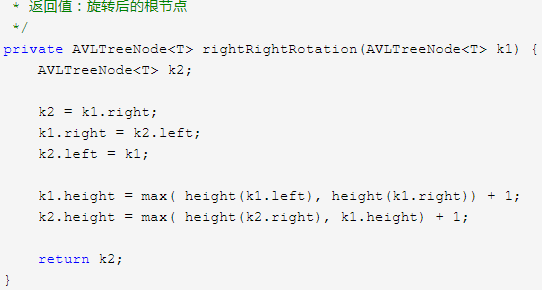
LL旋转





RR旋转

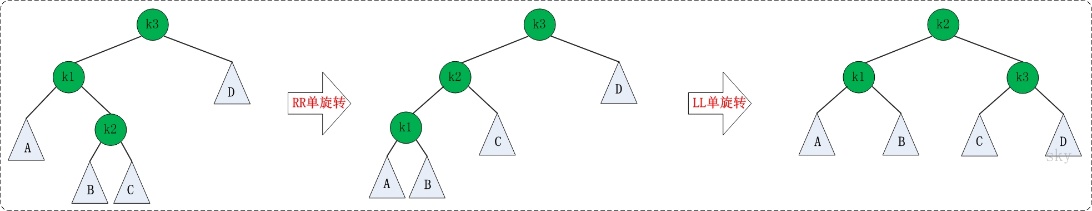


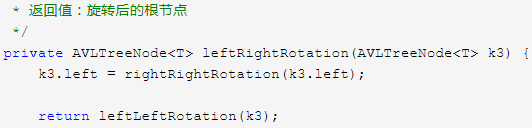


LR旋转

LR失去平衡的情况，需要经过两次旋转才能让AVL树恢复平衡

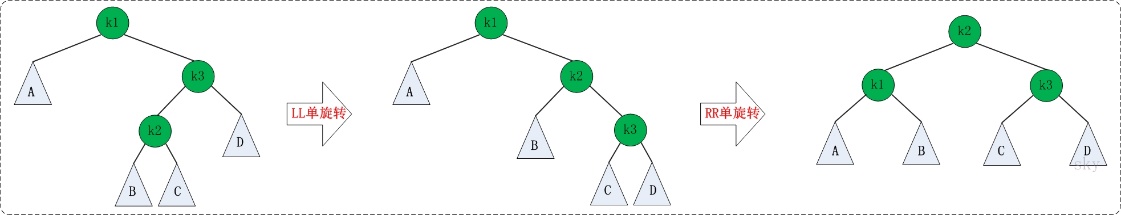
第一次旋转是围绕"k1"进行的"RR旋转"，第二次是围绕"k3"进行的"LL旋转"。

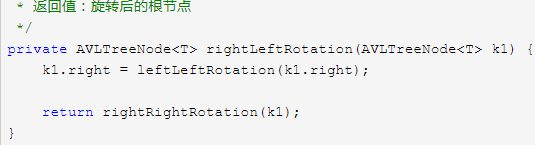




RL旋转

第一次旋转是围绕"k3"进行的"LL旋转"，第二次是围绕"k1"进行的"RR旋转"。



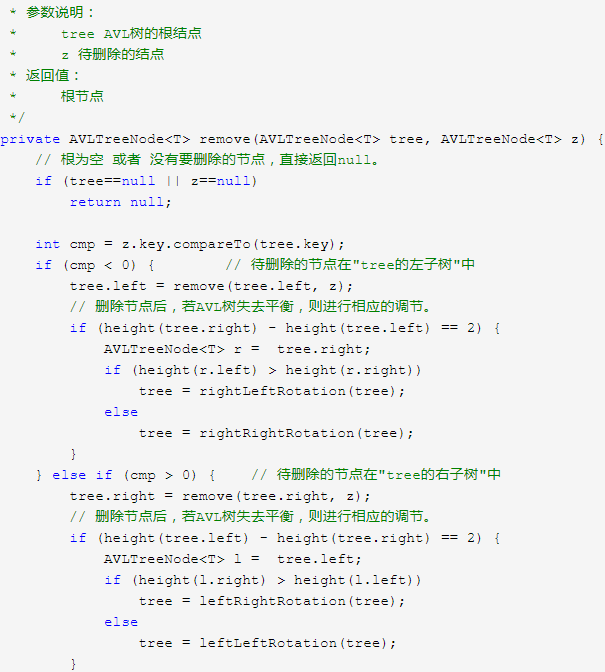


插入节点代码

将结点插入到AVL树中，并返回根节点



删除节点—返回根节点



# 红黑树(R-B Tree)

全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树的特性:

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树

应用

主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率非常之高。

例如，Java集合中的TreeSet和TreeMap 以及Linux虚拟内存的管理，都是通过红黑树去实现的。

红黑树的时间复杂度为: O(lgn) 一棵含有n个节点的红黑树的高度至多为2log(n+1).

常见的AVL的插入重建平衡

对于红黑树，你可以只了解：红黑树是一种常用的平衡二叉搜索树，单次的查找、插入、删除的复杂度都是lg(n)，这些操作的复杂度都不会恶化；红黑树的定义；红黑树结点数与树高的约束：lg(n+1) >= h/2，及其证明；STL中的map和set是用红黑树实现的，以及它们的用法,还有原理---是如何实现的 不需要具体代码

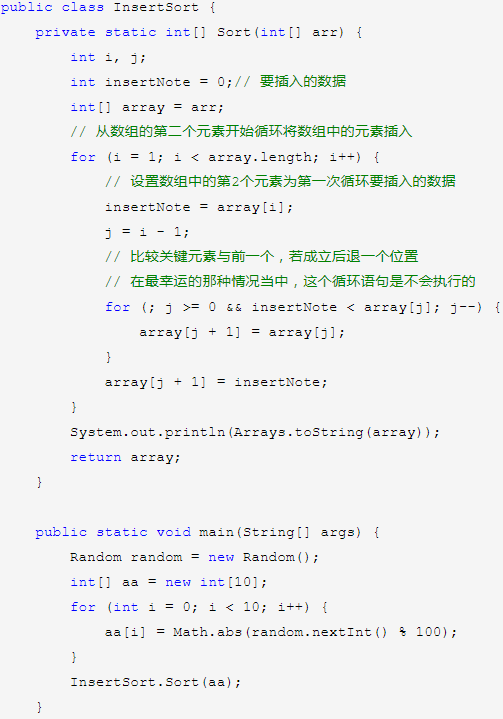
要记住这些算法到底解决了什么问题。干什么用的就行

在面试中常考的树的形状有：普通二叉树、平衡二叉树、完全二叉树、二叉搜索树、四叉树（Quadtree）、多叉树（N-ary Tree）。对于一些特殊的树，例如红黑树（Red-Black Tree）、自平衡二叉搜索树（AVL Tree），大家不必花费太多时间去准备，一般在面试中不会被问到，除非你所涉及的研究领域跟它们相关或者你十分感兴趣。树的结构十分直观，而树的很多概念定义都有一个相同的特点：递归，也就是说，一棵树要满足某种性质，往往要求每个节点都必须满足。例如，在定义一棵二叉搜索树时，每个节点也都必须是一棵二叉搜索树。正因为树有这样的性质，大部分关于树的面试题都与递归有关，换句话说，面试官希望通过一道关于树的问题来考察你对于递归算法掌握的熟练程度。关于树的考题，无非就是要考查树的遍历以及序列化（serialization)。树的基本遍历有三种：前序遍历（Preorder Traversal）中序遍历（Inorder Traversal）后序遍历（Postorder Traversal）掌握好这3种遍历的递归写法和非递归写法是非常重要的，同时，还需要懂得分析各种写法的时间复杂度和空间复杂度同样重要。无论你是前端工程师，还是后端工程师，在准备面试的时候，树这个数据结构可以说是最应该花时间学习的。掌握好树，能证明你对递归有很好的认识，能帮助你学习图论。另外，树的许多性质都是面试的热门考点，尤其是二叉搜索树（BST）。

# 直接插入排序

时间复杂度是O(N2) 稳定算法

把n个待排序的元素看成为一个有序表和一个无序表。开始时有序表中只包含1个元素，无序表中包含有n-1个元素，排序过程中每次从无序表中取出第一个元素，将它插入到有序表中的适当位置，使之成为新的有序表，重复n-1次可完成排序过程 ；

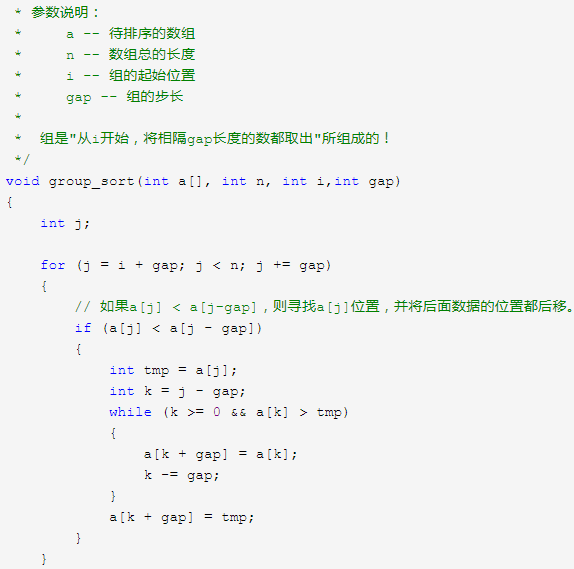


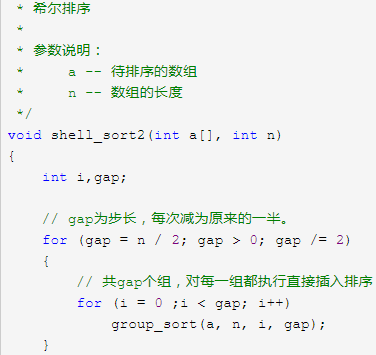
# 希尔排序

希尔排序的时间复杂度与增量(即，步长gap)的选取有关。例如，当增量为1时，希尔排序退化成了直接插入排序，此时的时间复杂度为O(N²)，而Hibbard增量的希尔排序的时间复杂度为O(N3/2)

它是不稳定算法

希尔排序实质上是一种分组插入方法。它的基本思想是：对于n个待排序的数列，取一个小于n的整数gap(gap被称为步长)将待排序元素分成若干个组子序列，所有距离为gap的倍数的记录放在同一个组中；然后，对各组内的元素进行直接插入排序。 这一趟排序完成之后，每一个组的元素都是有序的。然后减小gap的值，并重复执行上述的分组和排序。重复这样的操作，当gap=1时，整个数列就是有序的





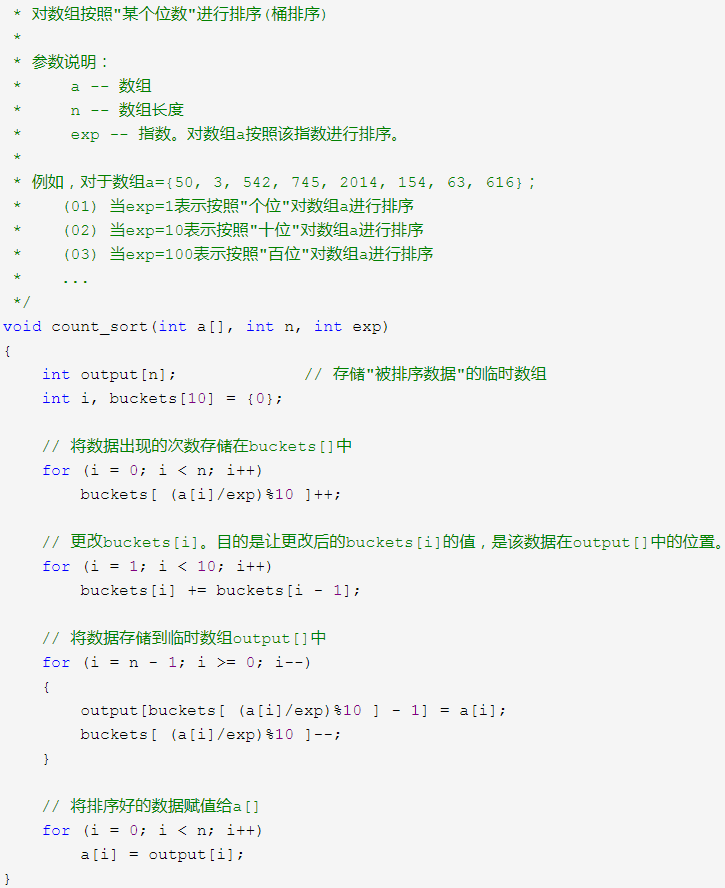
# 基数排序

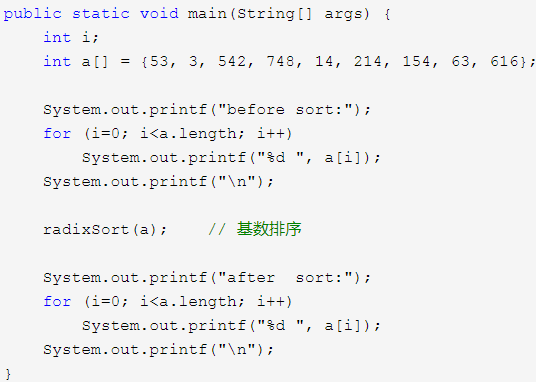
基本思想是：将所有待比较数值(注意,必须是正整数)统一为同样的数位长度,数位较短的数前面补零. 然后, 从最低位开始, 依次进行一次稳定排序.这样从最低位排序一直到最高位排序完成以后, 数列就变成一个有序序列.

如果要从高位排序, 那么次高位的排序会影响高位已经排好的大小关系. 在数学中, 数位越高,数位值对数的大小的影响就越大.从低位开始排序,就是对这种影响的排序. 数位按照影响力从低到高的顺序排序, 数位影响力相同则比较数位值

稳定排序的意思是指, 待排序相同元素之间的相对前后关系,在各次排序中不会改变.比如实例中具有十位数字5的两个数字58和356, 在十位排序之前356在58之前,在十位排序之后, 356依然在58之前.稳定排序能保证,上一次的排序成果被保留,十位数的排序过程能保留个位数的排序成果,百位数的排序过程能保留十位数的排序成果.



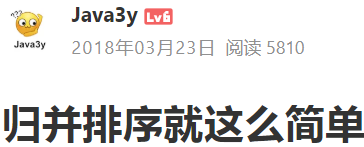


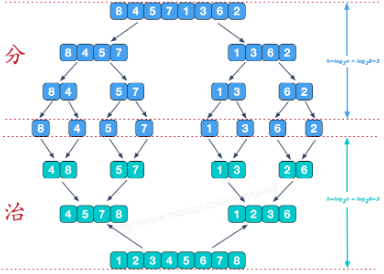


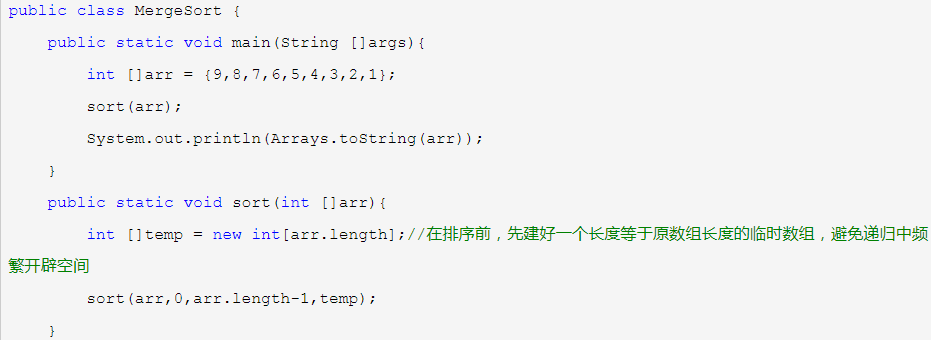
# 归并排序

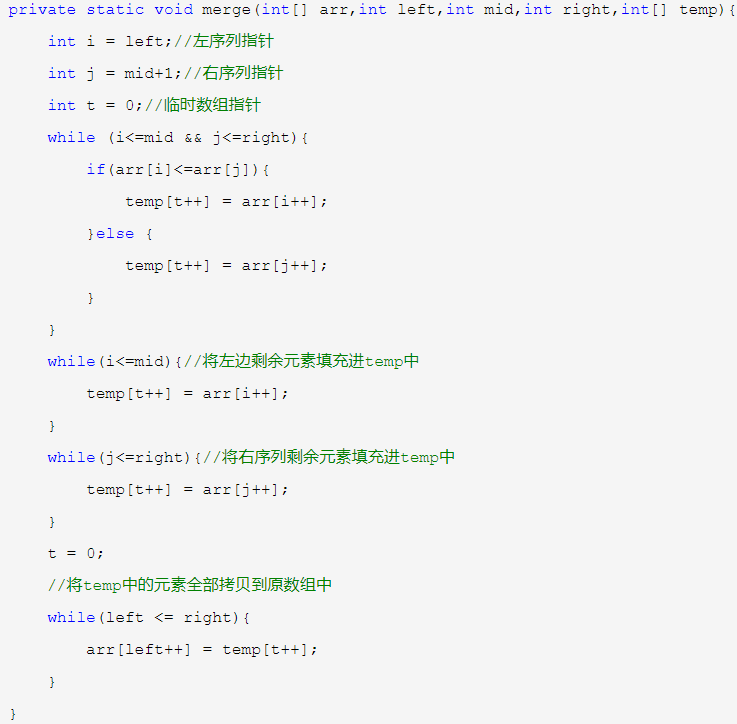
采用经典的分治（divide-and-conquer）策略

是一种稳定排序，每次合并操作的平均时间复杂度为O(n)，而完全二叉树的深度为|log2n|。总的平均时间复杂度为O(nlogn)。而且，归并排序的最好，最坏，平均时间复杂度均为O(nlogn)。









Java和python的Array.sort排序通过TimSort算法实现

Timsort本质是归并排序（merge sort）混合了插入排序（insertion sort）做优化

日常中需要排序的数据通常不是完全随机的，而是部分有序的，或者部分逆序的，所以TimSort充分利用已有序的部分进行归并排序。

首先规定一个最小归并长度。检查数组中原本有序的片段，如果已有序的长度小于规定的最小归并长度，则通过插入排序对已有序的片段进行进行扩充（这样做的原因避免归并长度较小的片段，因为这样的效率比较低）。将有序片段的起始索引位置和已有序的长度入栈。只有相邻片段可进行归并

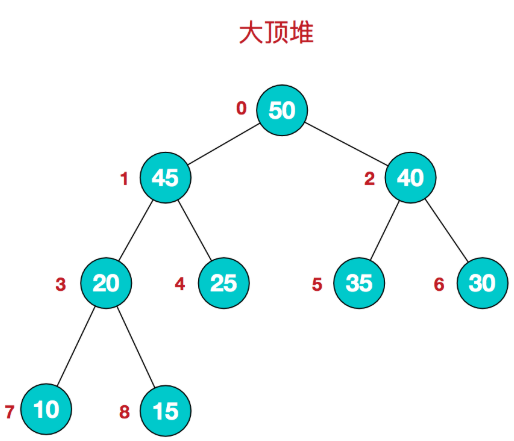
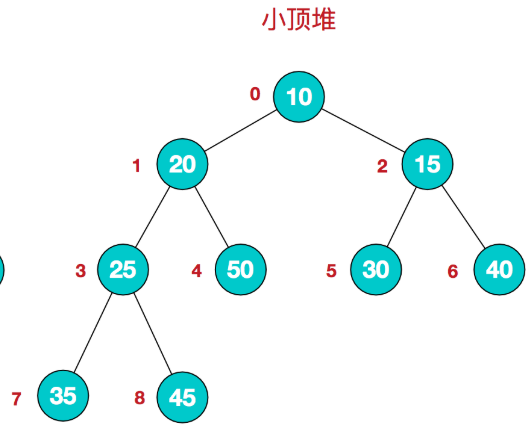
避免一个较长的有序片段和一个较小的有序片段进行归并，因为这样的效率比较低

# 堆排序

堆排序是利用堆这种数据结构而设计的一种排序算法，堆排序是一种选择排序，它的最坏，最好，平均时间复杂度均为O(nlogn)，它是不稳定排序

堆

堆是具有以下性质的完全二叉树：每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆；或者每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆

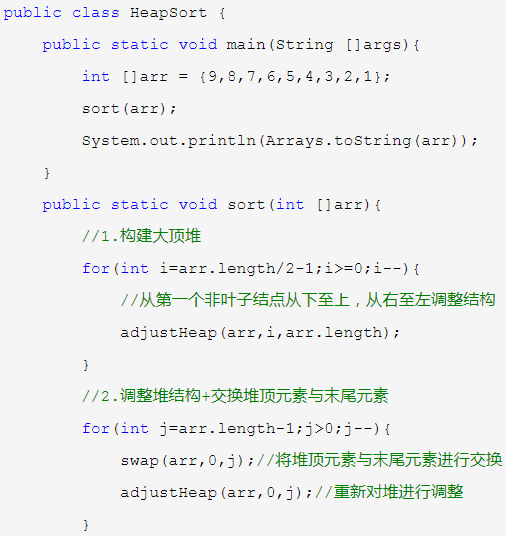
将堆中节点映射到数组中

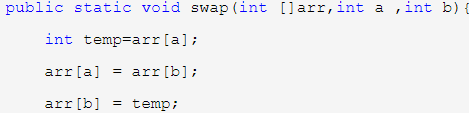
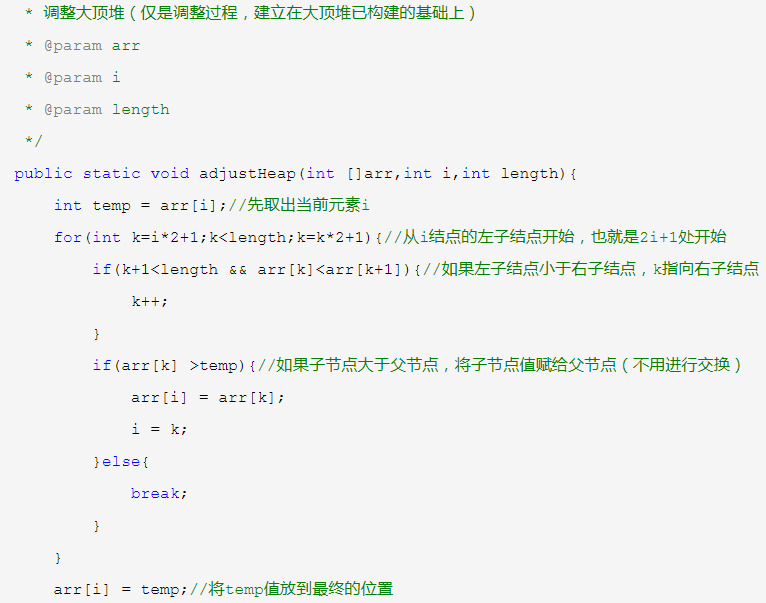
一般都用数组来表示堆，i结点的父结点下标就为(i – 1) / 2。它的左右子结点下标分别为2 \* i + 1和2 \* i + 2。

该数组从逻辑上讲就是一个堆结构，我们用简单的公式来描述一下堆的定义就是：

大顶堆：arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]

小顶堆：arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]

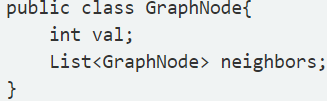




整体主要由构建初始堆+交换堆顶元素和末尾元素并重建堆两部分组成。其中构建初始堆经推导复杂度为O(n)，在交换并重建堆的过程中，需交换n-1次，而重建堆的过程中，根据完全二叉树的性质，[log2(n-1),log2(n-2)...1]逐步递减，近似为nlogn。所以堆排序时间复杂度一般认为就是O(nlogn)级。

# 深度优先搜索DFS(图的算法)

由于树可以看成是一个graph. Graph类的定义还是用每一个节点保存邻居信息:

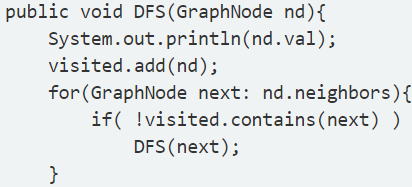


为了防止重复, 用一个HaseSet记录走过的节点:

HasheSet<GraphNode> visited = new HasheSet<GraphNode>();

递归写法

每到一个节点, 标记其已经访问过了, 然后对于邻居里面没有访问的节点继续递归进行DFS.

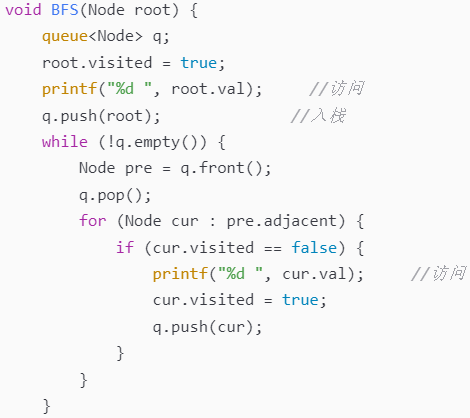


非递归写法

非递归的版本效率高一些, 另外这个算法和bfs非常相似, 只不过把队列queue换成了栈stack而已



非递归BFS



dfs另一个有用的性质是: 对于二叉树而言, dfs得到的节点顺序正是其前序遍历(preorder traversal)的顺序.

判断有向图中是否存在回路

在递归版本的dfs里, 我们对每一个点改为三种标记:

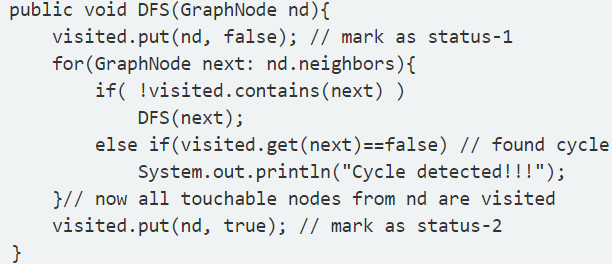
未访问过(0)， 正在访问其邻居节点(1)， 已经访问完该节点以及所有该节点可以到达的节点(2).

什么时候会出现回路呢? 就是当前节点v的一个邻居u的状态为1的时候. 因为该节点状态为1, 即还没有把它以后的节点全部遍历, 所以当前节点v肯定可以从u到达, 而现在又可以从v到达u, 所以构成一个回路.

为了表示一个节点的三种状态, 我们把visited的定义改一下, 定义为一个hashmap:

HasheMap<GraphNode, Boolean> visited = new HasheMap<GraphNode, Boolean>();

节点不在visited表示还未访问过, 节点对应为false表示正在访问, 节点对应为true表示已经访问该节点以及所有可以从它到达的节点.



# 拓扑排序

是dfs的应用，需要在有向无回路图中

拓扑排序的一个应用就是对于各种依赖性(比如学习课程A需要先学习过课程B)组成的图寻找一个节点遍历的顺序使其可行

拓扑排序的结果不唯一.

有回路的图不存在拓扑顺序.

如果一个节点没有出边, 那么它可以放在拓扑排序的最后面(没有节点以来它).

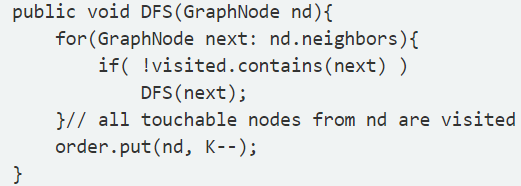
如果一个节点没有入边, 那么它可以放在拓扑排序的最后面.

维护一个计数器K(初始化为n=所有节点数), 每当一个点已经遍历完毕(所有通过这个点可以到达的点都已经被走过)以后, 就把这个点的顺序设为K, 同时减少K.

就用一个HashMap来为每个节点关联一个序号好了:

HasheMap<GraphNode, Integer> order = new HasheMap<GraphNode, Integer>();

此为对一个点的拓扑，为了给所有点，只要从一个没有出边的节点出发进行遍历, 一直运行到所有的节点都已经访问过为止



<https://www.coursera.org/specializations/algorithms>



作者：一只Tom猫  
链接：https://zhuanlan.zhihu.com/p/86536581

**1.1 java知识点**

* Hashmap 源码级掌握，扩容，红黑树，最小树化容量，hash冲突解决，有些面试官会提出发自灵魂的审问，比如为什么是红黑树，别的树不可以吗；为什么8的时候树化，4不可以吗，等等
* concureentHashMap，段锁，如何分段，和hashmap在hash上的区别，性能，等等
* HashTable ，同步锁，这块可能会问你synchronized关键字 1.6之后提升了什么，怎么提升的这些
* ArrayList 优势，扩容，什么时候用
* LinkedList 优势，什么时候用，和arraylist的区别 等等
* 基本类型和包装类型的区别，涉及自动装箱和拆箱，怎么做的，原理
* String ，StringBuffer，StringBuilder哪个是安全的
* 字符串编码的区别，被问到过，我觉得比较容易被忽视的一个点
* 什么是泛型，怎么用泛型
* static能不能修饰threadLocal，为什么，这道题我当时一听到其实挺懵逼的
* Comparable和Comparator接口是干什么的，其区别
* 多态的原理是什么，感觉这个很容易被问到
* 接口和抽象类，面试官问我是怎么理解的，我说接口对应功能，抽象类对应属性，然后面试官给我说了他的看法，说抽象类更偏向于一种模板~ 然后又交流了一下各自的想法
* 如何通过反射和设置对象私有字段的值
* 快速失败(fail-fast)和安全失败(fail-safe)的区别是什么
* synchronized 的实现原理以及锁优化？
* volatile 的实现原理？
* Java 的信号灯？
* synchronized 在静态方法和普通方法的区别？
* 怎么实现所有线程在等待某个事件的发生才会去执行？
* CAS？CAS 有什么缺陷，如何解决？
* synchronized 和 lock 有什么区别？
* Hashtable 是怎么加锁的 ？
* List，Map，Set接口在取元素师，各有什么特点
* 如何线程安全的实现一个计数器
* 生产者消费者模式，要求手写过代码，还是要知道的
* 单例模式，饿汉式，懒汉式，线程安全的做法，两次判断instance是否为空，每次判断的作用是什么。
* 线程池，这个还是很重要的，在生产中用的挺多，四个线程池类型，其参数，参数的理解很重要，corepoolSize怎么设置，maxpoolsize怎么设置，keep-alive各种的，和美团面试官探讨过阻塞队列在生产中的设置，他说他一般设置为0，防止用户阻塞
* cyclicbarrier 和countdownlatch的区别，个人理解 赛马和点火箭
* 线程回调，这块 被问过让我设计一个RPC，怎么实现，其实用到了回调这块的东西
* sleep 和yeild方法有什么区别
* volatile关键字，可见性。
* 乐观锁和悲观锁的使用场景
* 悲观锁的常见实现方式：lock synchronized retreentlock
* 乐观锁：CAS MVCC
* 读写锁的实现方式，16位int的前八位和后八位分别作为读锁和写锁的标志位
* 死锁的条件，怎么解除死锁，怎么观测死锁。
* 希望大家能够好好看一下反射的原理，怎么确定类，怎么调方法
* RPC框架，同步异步，响应时间，这些都被问到过，还让设计过
* 同步，异步，阻塞，非阻塞 在深信服的面试中遇到过，最好再找一些应用场景加以理解

**1.2 JVM**

* 内存模型以及分区，需要详细到每个区放什么。
* 堆里面的分区：Eden，survival （from+ to），老年代，各自的特点。
* 对象创建方法，对象的内存分配，对象的访问定位。
* GC 的两种判定方法
* GC 的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路？
* GC 收集器有哪些？CMS 收集器与 G1 收集器的特点
* Minor GC 与 Full GC 分别在什么时候发生？
* JVM 内存分哪几个区，每个区的作用是什么?
* 如和判断一个对象是否存活?(或者 GC 对象的判定方法)
* java 中垃圾收集的方法有哪些?
* 类加载器双亲委派模型机制？
* java 内存模型，java 类加载过程?
* 什么是类加载器，类加载器有哪些?
* 简述 java 内存分配与回收策率以及 Minor GC 和Major GC

**2.1 MySQL**

* 事务四大特性（ACID）原子性、一致性、隔离性、持久性？
* 事务的并发？事务隔离级别，每个级别会引发什么问题，MySQL默认是哪个级别？
* MySQL常见的三种存储引擎（InnoDB、MyISAM、MEMORY）的区别？
* MySQL的MyISAM与InnoDB两种存储引擎在，事务、锁级别，各自的适用场景？
* 查询语句不同元素（where、jion、limit、group by、having等等）执行先后顺序
* 索引为什么要用B+树，B+树和B-树的区别是什么
* mysql的默认事务级别，一共有哪些事务级别
* mysql的一些语句，这些肯定需要掌握的
* mysql锁，行锁，表锁 ，什么时候发生锁，怎么锁，原理
* 数据库优化，最左原则啊，水平分表，垂直分表
* 什么是临时表，临时表什么时候删除?
* MySQL B+Tree索引和Hash索引的区别？
* sql查询语句确定创建哪种类型的索引？如何优化查询？
* 聚集索引和非聚集索引区别？
* 有哪些锁（乐观锁悲观锁），select 时怎么加排它锁？
* 非关系型数据库和关系型数据库区别，优势比较？
* 数据库三范式，根据某个场景设计数据表？
* 数据库的读写分离、主从复制，主从复制分析的 7 个问题？
* 使用explain优化sql和索引？
* MySQL慢查询怎么解决？
* 什么是 内连接、外连接、交叉连接、笛卡尔积等？
* mysql都有什么锁，死锁判定原理和具体场景，死锁怎么解决？
* varchar和char的使用场景？
* mysql 高并发环境解决方案？
* 数据库崩溃时事务的恢复机制（REDO日志和UNDO日志）？

03 Spring相关

* spring的两大特性- ioc aop，实现原理
* 如果存在A依赖B，B依赖A，那么是怎么加到IOC中去的
* beanFactory的理解，怎么加载bean
* FactoryBean的理解
* 基于注解的形式，是怎么实现的， 你知道其原理吗，说一下
* 依赖冲突，有碰到过吗，你是怎么解决的~
* bean的生命周期
* spring中的自动装配方式
* BeanFactory 和 FactoryBean
* Spring IOC 的理解，其初始化过程？
* BeanFactory 和 ApplicationContext？
* Spring Bean 的生命周期，如何被管理的？Spring Bean 的加载过程是怎样的？
* 如果要你实现Spring AOP，请问怎么实现？
* 如果要你实现Spring IOC，你会注意哪些问题？
* Spring 是如何管理事务的，事务管理机制？
* Spring 的不同事务传播行为有哪些，干什么用的？
* Spring 中用到了那些设计模式？
* Spring MVC 的工作原理？
* Spring 循环注入的原理？
* Spring 如何保证 Controller 并发的安全？
* 你一般是怎么对mvc项目进行分层的
* dispatch-servlet的工作原理
* 为什么有了springmvc还要在项目中使用spring？
* springmvc的运行机制，dispatch -》 hanldermapping-—》handler -》handlerAdapter-》执行handler-》modelandview -》 返回mv -》 视图解析器-》返回view -》 渲染响应
* 怎么防止依赖注入
* 怎么让mapper 和xml对应
* 如何自动包装对象
* 和spring相比，做了什么改变
* starter你知道哪些
* 如何部署springmvc项目 以及如何部署springboot项目
* springboot的插件，你使用过哪些

04 中间件

4.1 redis

* Redis用过哪些数据数据，以及Redis底层怎么实现
* Redis缓存穿透，缓存雪崩
* 如何使用Redis来实现分布式锁
* Redis的并发竞争问题如何解决
* Redis持久化的几种方式，优缺点是什么，怎么实现的
* Redis的缓存失效策略
* Redis集群，高可用，原理
* Redis缓存分片，Redis的数据淘汰策略
* 为什么选择redis，有什么好处，基于内存，抗压
* redis集群怎么进行数据分配，hash槽
* redis的主从复制是怎么实现的
* redis的数据结构 最常问 hash是什么， sorted set怎么实现的
* 因为项目的原因，问我redis是怎么保证高可用的，主从和集群怎么加在一起
* redis 和memcache的区别
* redis 分布式锁的实现原理 setNX 啥的
* redis模拟session，除了redis你还考虑过别的吗
* redis的缓存击穿，怎么处理这个问题
* redis是基于内存的，那么它有持久化吗，aof rdb
* aof和rdb的优缺点，你在项目中使用的哪一个

06 Linux

* 常见的命令
* sed 和 awk 感觉linux必考。。
* linux的使用场景，你什么时候会用linux -- 》 布置服务器
* 怎么查看进程和杀死进程
* 打印一个文件夹中的所有文件
* float在计算机中是怎么存储的，当时被问到的时候，我也在问自己，怎么存的~~~ 佛了
* 线程和进程的区别
* 线程的通信方式，进程的通信方式
* 系统线程的数量上限是多少
* 页式存储的概念
* 内存碎片，你有了解过吗，有想过解决方案吗~

7.1 排序算法

* 八大排序算法真的是面试宠儿
* 最常考 快速排序 和归并排序
* 哪些排序算法是稳定的 哪些是不稳定的
* 堆排 也应该掌握

7.2 树

* 根据遍历结果恢复树，递归
* 二叉搜索树第k大
* 树的和为k的路径
* 层次遍历
* 根据层次遍历和后序遍历恢复树
* 镜像树
* 树的深度
* 是不是平衡二叉树

7.3 链表

* 反转链表
* 链表环的入口
* 交叉链表的交点
* 复杂链表的复制
* 二叉搜索树变成双向链表

7.4 回溯算法

* 走迷宫
* 游戏通关

7.5 递推算法

* 走台阶
* 断钢筋

7.6 背包问题

* 装最多的东西

7.7 贪心算法

* 覆盖问题
* 时间问题

08 设计模式

面试中设计模式其实也是挺重要的

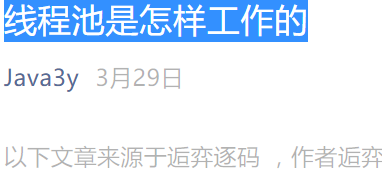
* Java 中什么叫单例设计模式？请用 Java 写出线程安全的单例模式
* 在 Java 中，什么叫观察者设计模式（observer design pattern）
* 使用工厂模式最主要的好处是什么？在哪里使用
* 举一个用 Java 实现的装饰模式(decorator design pattern) ？它是作用于对象层次还是类层次？
* 在 Java 中，什么时候用重载，什么时候用重写？
* 举例说明什么情况下会更倾向于使用抽象类而不是接口
* 观察者模式
* 适配模式
* 工厂模式
* ...

地哥的腾讯面试经历

快速排序，我本来想补充的，例如对中轴的选择问题，与其他排序算法的一些比较，应用场景等

说说网络建立连接之后是怎么关闭的（其实就是四次挥手，我详细说了，之前写过文章）。如果出现大量time\_wait 状态的连接是怎么回事？（我说了第四次挥手出现了丢包、超时等），假如不会出现丢包、超时等，网络非常流畅呢？（说了短连接导致的）那怎么解决？（我说了限制最大连接，用队列来缓冲），问我还有其他方法吗？（我不大知道，好像面试官不大满意我的回答。不过最好的做法应该是修改 time\_wait 等待的最大之间，把时间改小一点）

线程池是怎样工作的



<https://blog.csdn.net/qq_34337272/category_6481583.html>

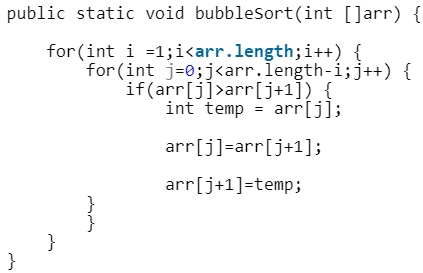
# 八大排序

冒泡，选择，插入，希尔，快速，归并，堆，基数

## 冒泡排序

a，b，c，d，e

比较12，13，14，15 // 23，24，25 // 34，35 // 45 若前数大于后数则交换



时间复杂度

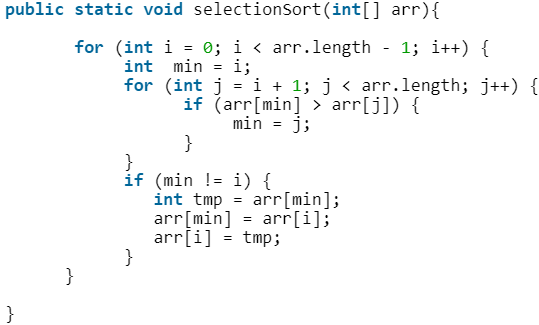
空间复杂度

稳定排序

## 选择排序

a，b，c，d，e

设1是最小位t，循环2345位，若发现最小改变t的位值，最后把1位与最小位交换，剩下元素找第二小



时间复杂度

空间复杂度

不稳定排序

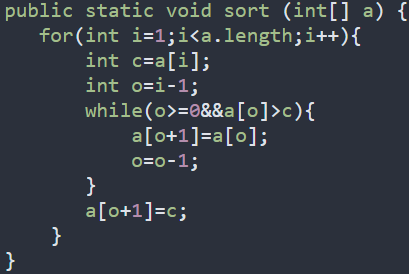
## 插入排序

在要排序的一组数中，假定前n-1个数已经排好序，现在将第n个数插到前面的有序数列中，使得这n个数也是排好顺序的。如此反复循环，直到全部排好顺序。 如果数据序列基本有序，使用插入排序会更加高效。

a，b，c，d，e

设第一个数有序，外数一与最后一个内数比，若大，不循环，外数等于外数，加入内数组中(外数指针后移一位)，若小，外数一的位置被最后一个内数赋值，开始循环(与倒数第二个内数比较)

第一个循环是从数组第二个元素开始往后，第二个循环是要比较的元素与前面有序数组里的元素挨个比较，数组倒序比，若大，有序数组里比较的值后移一位，若小，要比较的元素赋值为此时数组元素位置加一



时间复杂度

空间复杂度

稳定排序

## 希尔排序

是插入排序的高效版本

初始增量gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]

从第一个元素开始循环i，i与i+gap一组，此时调用插入排序，接着gap继续除2，再次插入排序，直到gap=1，最后一次排序



快速排序

# JVM

JVM是sun公司旗下产品，有三个版本，分别是javaSE(平台标准版)，javaEE(平台企业版)和javaME(平台移动设备版)。sun公司在jdk8之后的JVM技术实现是HotSpot，这里还有一层关系，JDK是java开发环境，JRE是java运行环境，JDK包含JRE，而JRE包含JVM。也就是说HotSpot是JVM的实现技术，是用C+汇编语言编写的

在 Java 中，JVM 可以理解的代码就叫做字节码（即扩展名为 .class 的文件），它不面向任何特定的处理器，只面向虚拟机 可以说.class文件是不同的语言在 Java 虚拟机之间的重要桥梁，同时也是支持 Java 跨平台很重要的一个原因。

## 类的加载

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口

类加载器并不需要等到某个类被“首次主动使用”时再加载它，JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）若这个类一直没有被程序主动使用，那么类加载器就不会报告错误

## Java类的加载过程

类的加载过程:加载—验证---准备---解析---初始化(initialization) 除了解析阶段都是是按顺序开始 而不是顺序完成，解析在某些情况下可以在初始化阶段之后开始，这是为了支持Java语言的动态绑定

静态绑定

在程序执行前方法已经被绑定，针对java简单的可以理解为程序编译期的绑定；

java当中的方法只有final，static，private和构造方法是前期绑定

动态绑定

在运行时根据具体对象的类型进行绑定。提供了一些机制，可在运行期间判断对象的类型，并分别调用适当的方法。也就是说，编译器此时依然不知道对象的类型，但方法调用机制能自己去调查，找到正确的方法主体

1.静态绑定是发生在编译阶段；而动态绑定是在运行阶段；

2.private, final and static方法和变量使用静态绑定，而虚函数(virtual methods)则会根据运行时的具体对象进行绑定（注：在Java语言中, 所有的方法默认都是”虚函数”。只有以关键字 final 标记的方法才是非虚函数。）

3.静态绑定使用的是类信息，而动态绑定使用的是对象信息

4.重载方法(overloaded methods)使用的是静态绑定，而重写方法(overridden methods)使用的是动态绑定

### 加载

加载主要是将.class文件（并不一定是.class。可以是ZIP包，网络中获取）中的二进制字节流读入到JVM中。

在加载阶段，JVM需要完成3件事：

1）通过类的全限定名获取该类的二进制字节流；---读取.class文件 类文件结构

2）将字节流所代表的静态存储结构转化为方法区的运行时数据结构； ---方法区类

3）在**内存**中生成一个该类的java.lang.Class对象，可通过该对象访问方法区的类数据。---堆中类对象

(可用系统提供的类加载器或自定义类加载器)

### 验证

验证是连接阶段的第一步，主要确保加载进来的字节流符合JVM规范。  
验证阶段会完成以下4个阶段的检验动作：  
1）文件格式验证 (验证字节流是否符合Class文件格式的规范 )  
2）元数据验证(是否符合Java语言规范)  
3）字节码验证（确定程序语义合法，符合逻辑）  
4）符号引用验证（确保下一步的解析能正常执行）

采用 -Xverifynone参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

### 准备

为静态变量在--**方法区--**分配内存，并将其初始化为默认值。

内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在**--堆--**中，这里所设置的初始值通常情况下是数据类型默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值

public static int value = 3；那么变量value在准备阶段过后的初始值为0，而不是3，因为这时候尚未开始执行任何Java方法，而把value赋值为3的指令是在程序编译后，存放于类构造器<clinit>（）方法之中的，所以把value赋值为3的动作将在**初始化**阶段才会执行。

1.对基本数据类型来说，对于类变量（static）和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于局部变量来说，在使用前必须显式地为其赋值，否则编译时不通过。

2.对于同时被static和final修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过，此时这个常量被放在运行时常量池中；

public static final int value = 3；

编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为3。我们可以理解为static final常量在编译期就将其结果放入了调用它的类的常量池中。

3.对于引用数据类型reference来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的零值，即null。

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有 “指针碰撞” 和 “空闲列表” 两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

虚拟机采用两种方式来保证线程安全：

CAS+失败重试： CAS 是乐观锁的一种实现方式。所谓乐观锁是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。保证更新操作的原子性

TLAB： 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

### 解析

是虚拟机将常量池内的**符号引用**替换为**直接引用**的过程。---类文件结构

解析阶段可能开始于初始化之前，也可能在初始化之后开始，到底是在类被加载器加载时就对常量池中的符号引用进行解析（初始化之前），还是等到一个符号引用将要被使用前才去解析它（初始化之后）

### 初始化

只有当对类的主动使用的时候才会导致类的初始化，类的主动使用包括以下六种：

1）new该类实例化对象的时候；

2）访问类/接口的静态变量（但被final修饰的字段，在编译器时就被放入常量池的静态字段除外static final）

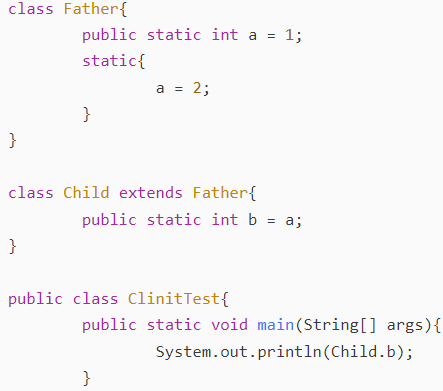
3）调用类静态方法的时候；

4）反射Class.forName(“xxxx”) 对类进行反射调用的时候，该类需要初始化；

5） 初始化一个类的时候，有父类，先初始化父类

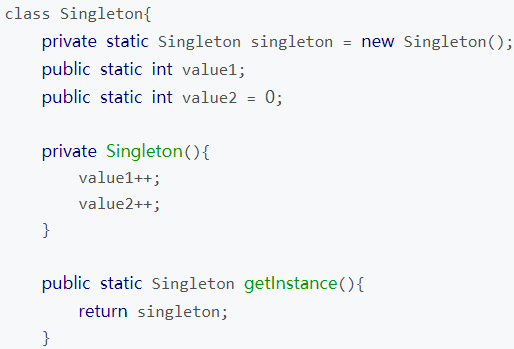
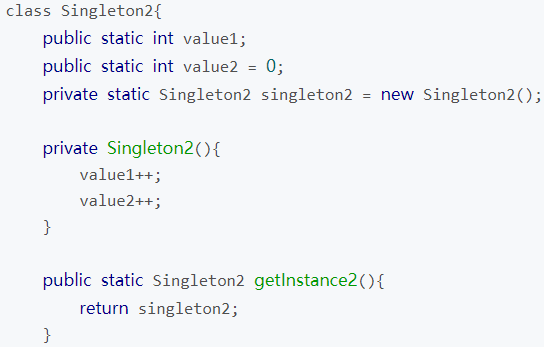
（注：1. 接口除外，父接口在调用的时候才会被初始化；2.子类引用父类静态字段，只会引发父类初始化）；

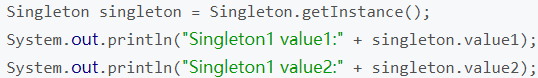
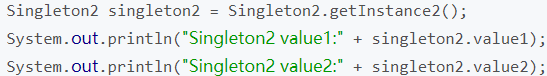
6） 被标明为启动类的类（即包含main()方法的类）要初始化；

结果2

Static和静态语句块顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句中可以赋值，但是不能访问（xxx=a --- wrong）

New的时候运行class的构造器，静态变量先跟随类的初始化而初始化，待完成加载类后再赋值

首先类加载，类中各成员全是初始值，接着new 类与类构造器是一起的，看先new类还是先给变量赋值





真正执行字节码的操作，在加载完成后才真正开始

但从 Java 程序的视角来看，对象创建才刚开始，<init> 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 <init> 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来

#### 类初始化的被动引用

到初始化阶段，才真正开始执行类中的Java程序代码。类的初始化，称为对一个类进行主动引用，除此之外所有引用类的方式都不会触发其初始化，称为被动引用。下面举一些例子来说明被动引用。



例2



Super的m是11

Father 的m是33

看通过谁调用m



常量编译时

直接存入

调用它的类的

常量池中

Const的Name是

Public，所有类可调

用其常量并存入调用

它的类的常量池

接口和类二者在初始化时最主要的区别是：当一个类在初始化时，要求其父类全部已经初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量），才会初始化该父接口。调用类中的static final常量时并不会 触发该类的初始化，但是调用接口中的static final常量时便会触发该接口的初始化

## 类加载器

类加载器实现的功能是即为加载阶段获取二进制字节流的时候

JVM提供了以下3种系统的类加载器：

1.启动类加载器（Bootstrap ClassLoader）：最顶层的类加载器，负责加载Library

2.扩展类加载器(Extension ClassLoader)： 负责加载Library Extra

3.应用程序类加载器(Application ClassLoader)：负责加载用户路径（classpath）上的类库。如果没有自定义类加载器，一般这个就是默认的类加载器。

4.自定义类加载器：有的时候，我们需要自定义类加载器。比如应用是通过网络来传输 Java类的字节码，为保证安全性，这些字节码经过了加密处理，这时系统类加载器就无法对其进行加载，这样需要自定义类加载器来实现。自定义类加载器一般都是继承自 ClassLoader类，重写 loadClass方法中的findclass方法

## JVM类加载机制

1.全盘负责，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入

2.父类委托，先让父类加载器加载该类，只在父类加载器无法加载该类时才尝试从自己的类路径中加载该类

3.缓存机制，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效

## 双亲委派模型



这里父类加载器并不是通过继承关系来实现的，而是采用组合实现的

解释：如果一个类接受到类加载请求，他自己不会去加载这个请求，而是将这个类加载请求委派给父类加载器，这样一层一层传送，直到到达启动类加载器（Bootstrap ClassLoader）。启动类加载器开始尝试加载, 无法加载这个请求时，交给子加载器去加载。一层一层传送, 直到达到应用程序加载器或自定义加载器

意义：保证了Object类在程序中的各种类加载器中都是同一个类加载器 不同加载器加载出来的类不算同一类 保证了Java程序的稳定运作

代码实现

双亲委派模型的代码实现集中在java.lang.ClassLoader的loadClass()方法当中。

1）首先检查类是否被加载，没有则调用父类加载器的loadClass()方法；

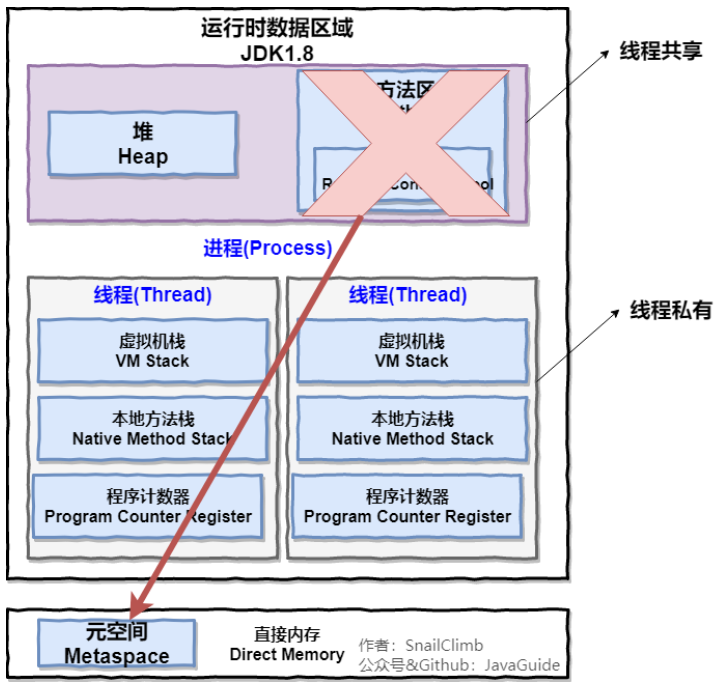
2）若父类加载器为空，则默认使用启动类加载器作为父加载器；

3）若父类加载失败，抛出ClassNotFoundException 异常后，再调用自己的findClass() 方法。

## Java内存区域

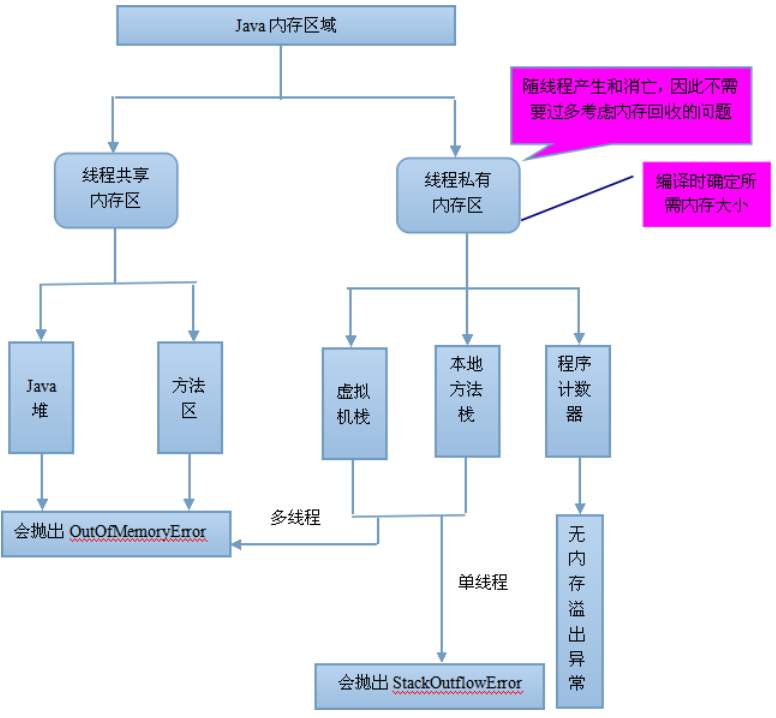
堆内存是JVM中最大的一块

1.8前后区别



线程私有：程序计数器 虚拟机栈 本地方法栈

线程共享的：堆 方法区 直接内存 (非运行时数据区的一部分)



### 程序计数器

一块较小的内存空间，它是当前线程所执行的字节码的行号指示器，字节码解释器工作时通过改变该计数器的值来选择下一条需要执行的字节码指令。每条线程都有一个独立的的程序计数器，各线程间的计数器互不影响，因此该区域内存是线程私有的。

当线程在执行一个Java方法时，该计数器记录的是正在执行的虚拟机字节码指令的地址，当线程在执行的是Native方法（调用本地操作系统方法）时，该计数器的值为空。

该内存区域是唯一一个在Java虚拟机规范中没有规定任何内存溢出（OutOfMemoryError）的区域。

### java虚拟机栈

它的生命周期也与线程相同。虚拟机栈描述的是Java方法执行的内存模型：**每个方法被执行的时候都会同时创建一个栈帧**，它是用于支持续虚拟机进行方法调用和方法执行的数据结构。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。对于执行引擎来讲，**活动线程中，只有栈顶的栈帧是有效的，称为当前栈帧，这个栈帧所关联的方法称为当前方法。**在编译程序代码时，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定了。因此，一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

在Java虚拟机规范中，对这个区域规定了两种异常情况：

1、如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常。

2、如果虚拟机在动态扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常。

    这两种情况存在着一些互相重叠的地方：当栈空间无法继续分配时，到底是内存太小，还是已使用的栈空间太大，其本质上只是对同一件事情的两种描述而已。在单线程的操作中，无论是由于栈帧太大，还是虚拟机栈空间太小，当栈空间无法分配时，虚拟机抛出的都是StackOverflowError异常，而不会得到OutOfMemoryError异常。而在多线程环境下，则会抛出OutOfMemoryError异常。

栈帧包含局部变量表，操作数栈，动态链接，方法返回地址

Java虚拟机的解释执行引擎称为“基于栈的执行引擎”，其中所指的“栈”就是操作数栈。因此我们也称Java虚拟机是基于栈的，这点不同于Android虚拟机，Android虚拟机是基于寄存器的。

Java 栈可用类比数据结构中栈，Java 栈中保存的主要内容是栈帧，每一次函数调用都会有一个对应的栈帧被压入 Java 栈，每一个函数调用结束后，都会有一个栈帧被弹出。

Java 方法有两种返回方式： return 语句 抛出异常 不管哪种返回方式都会导致栈帧被弹出。

### 本地方法栈

虚拟机栈为虚拟机执行Java方法服务，而本地方法栈则为使用到的本地操作系统（Native）方法服务

本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。

### Java堆(heap)

Java Heap是Java虚拟机所管理的内存中最大的一块。几乎所有的对象实例和数组都在这类分配内存。是垃圾收集器管理的主要区域，也被称为“GC堆”。

根据Java虚拟机规范的规定，Java堆可以处在物理上不连续的内存空间中，只要逻辑上是连续的即可。如果在堆中没有内存可分配时，并且堆也无法扩展时，将会抛出OutOfMemoryError异常。

Java 堆还可以细分为：新生代和老年代



上图所示的 eden 区、s0 区、s1 区都属于新生代，tentired 区属于老年代。大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置。

### 方法区

方法区存储类信息、常量、静态变量等数据

虚拟机规范允许该区域可以选择不实现垃圾回收。该区域的内存回收目标主要针是对废弃常量的和无用类的回收。

运行时常量池是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池（Class文件常量池），用于存放编译器生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。运行时常量池相对于Class文件常量池的另一个重要特征是具备动态性，Java语言并不要求常量一定只能在编译期产生

JDK1.7 已经将运行时常量池从方法区中移了出来，在 Java 堆中开辟了一块区域存放运行时常量池

JDK 1.8 的时候，方法区被彻底移除了，取而代之是元空间，元空间使用的是直接内存。常用参数：

-XX:MetaspaceSize=N //设置 Metaspace 的初始（和最小大小）

-XX:MaxMetaspaceSize=N //设置 Metaspace 的最大大小

根据Java虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

### 直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，它直接从操作系统中分配，因此不受Java堆大小的限制，但是会受到本机总内存的大小及处理器寻址空间的限制，因此它也可能导致OutOfMemoryError异常出现。在JDK1.4中新引入了NIO机制，它是一种新I/O方式，它可以直接使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样就能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆之间来回复制数据。

#### 内存溢出和内存泄漏

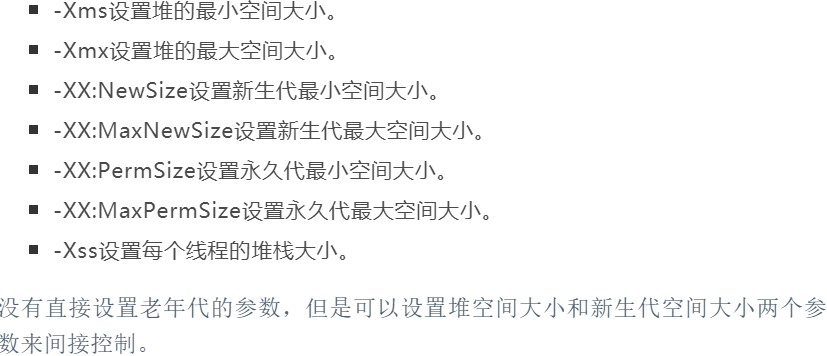
这里有一点要重点说明，在多线程情况下，给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。操作系统为每个进程分配的内存是有限制的，虚拟机提供了参数来控制Java堆和方法区这两部分内存的最大值，忽略掉程序计数器消耗的内存（很小），以及进程本身消耗的内存，剩下的内存便给了虚拟机栈和本地方法栈，每个线程分配到的栈容量越大，可以建立的线程数量自然就越少。因此，如果是建立过多的线程导致的内存溢出，在不能减少线程数的情况下，就只能通过减少最大堆和每个线程的栈容量来换取更多的线程。

另外，由于Java堆内也可能发生内存泄露（Memory Leak），这里说明一下内存泄露和内存溢出的区别：

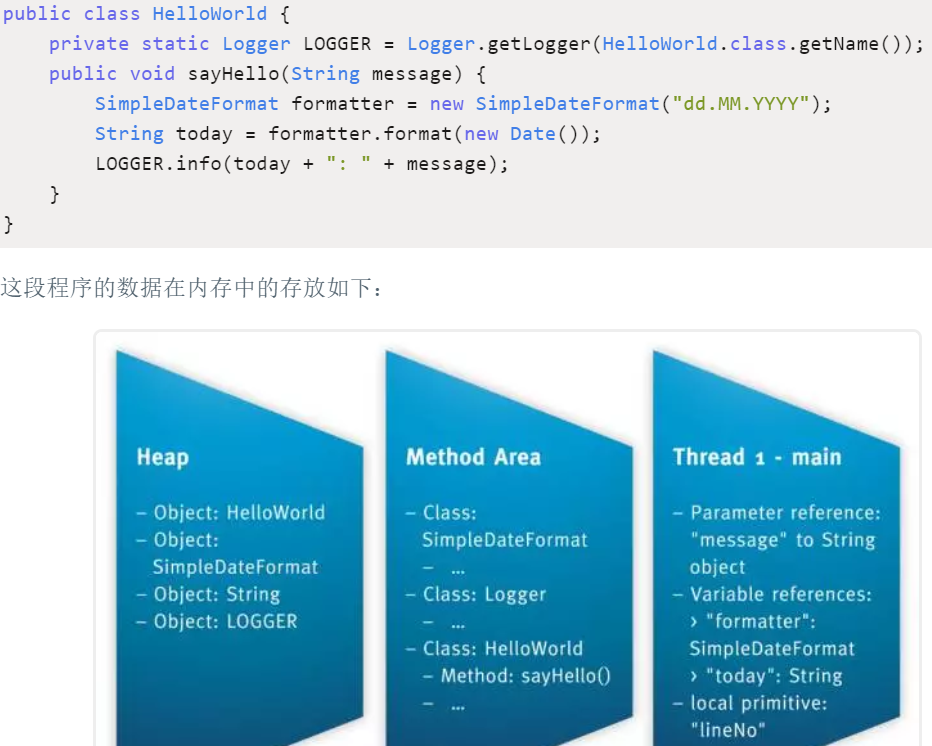
内存泄露是指分配出去的内存没有被回收回来，由于失去了对该内存区域的控制，因而造成了资源的浪费。Java中一般不会产生内存泄露，因为有垃圾回收器自动回收垃圾，但这也不绝对，当我们new了对象，并保存了其引用，但是后面一直没用它，而垃圾回收器又不会去回收它，这边会造成内存泄露，

内存溢出是指程序所需要的内存超出了系统所能分配的内存（包括动态扩展）的上限

控制参数



类加载实例



## JVM垃圾回收

如何判断对象是否死亡（两种方法）。

简单介绍一下强引用、软引用、弱引用、虚引用（虚引用与软引用和弱引用的区别、使用软引用能带来的好处）

如何判断一个常量是废弃常量

如何判断一个类是无用的类

垃圾收集有哪些算法，各自的特点？

HotSpot 为什么要分为新生代和老年代？

常见的垃圾回收器有那些？

介绍一下 CMS,G1 收集器。

Minor Gc 和 Full GC 有什么不同呢？

jvm 中，程序计数器、虚拟机栈、本地方法栈都是随线程而生随线程而灭，栈帧随着方法的进入和退出做入栈和出栈操作，实现了自动的内存清理，因此，我们的内存垃圾回收主要集中于 java 堆和方法区中

### 堆

新生代：eden & survival 老年代：tentired

大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s1("To")，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 ），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置

经过这次GC后，Eden区和"From"区已经被清空。这个时候，"From"和"To"会交换他们的角色，也就是新的"To"就是上次GC前的“From”，新的"From"就是上次GC前的"To"。不管怎样，都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程，直到“To”区被填满，"To"区被填满之后，会将所有对象移动到年老代中。

对象优先在eden分配

eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC.

新生代 GC（Minor GC）:指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。

老年代 GC（Major GC/Full GC）:指发生在老年代的 GC， Major GC 速度一般会比 Minor GC 的慢 10 倍以上

给 allocation2 分配内存的时候 eden 区内存几乎已经被分配完了，当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC.GC 期间虚拟机又发现 allocation1 无法存入 Survivor 空间，所以只好通过 分配担保机制 把新生代的对象提前转移到老年代中去，老年代上的空间足够存放 allocation1，所以不会出现 Full GC。执行 Minor GC 后，后面分配的对象如果能够存在 eden 区的话，还是会在 eden 区分配内存

大对象直接进入老年代---字符串，数组

为了更好的适应不同程序的内存情况，虚拟机不是永远要求对象年龄必须达到了某个值才能进入老年代，如果 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需达到要求的年龄。



### 对象存活判断

#### 引用计数法

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加 1；当引用失效，计数器就减 1；任何时候计数器为 0 的对象就是不可能再被使用的。无法解决对象相互循环引用

#### 可达性分析算法

这个算法的基本思想就是通过一系列的称为 “GC Roots” 的**对象**作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的

#### 对象的引用

无论是通过引用计数法判断对象引用数量，还是通过可达性分析法判断对象的引用链是否可达，判定对象的存活都与“引用”有关。

1．强引用（StrongReference）

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

2．软引用（SoftReference）

如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

3．弱引用（WeakReference）

只能生存到下一次垃圾回收之前。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4．虚引用（PhantomReference）

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。

虚引用与软引用和弱引用的一个区别在于： 虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。特别注意，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生。

在可达性分析法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑阶段”，要真正宣告一个对象死亡，至少要经历两次标记过程；可达性分析法中不可达的对象被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize 方法。当对象没有覆盖 finalize 方法，或 finalize 方法已经被虚拟机调用过时，虚拟机将这两种情况视为没有必要执行。

如果该对象被判定为有必要执行finalize（）方法，那么这个对象将会被放置在一个名为F-Queue队列中，并在稍后由一条由虚拟机自动建立的、低优先级的Finalizer线程去执行finalize（）方法。finalize（）方法是对象逃脱死亡命运的最后一次机会（因为一个对象的finalize（）方法最多只会被系统自动调用一次），稍后GC将对F-Queue中的对象进行第二次小规模的标记，如果要在finalize（）方法中成功拯救自己，只要在finalize（）方法中让该对象重引用链上的任何一个对象建立关联即可。而如果对象这时还没有关联到任何链上的引用，那它就会被回收掉。

无用的类 ：

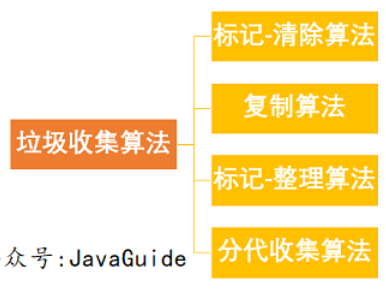
该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。

加载该类的 ClassLoader 已经被回收。

该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收

### 垃圾收集算法



#### 标记-清除算法

该算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象 缺点：效率低 空间产生大量不连续碎片

#### 复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一半，持续复制长生存期的对象则导致效率降低。

#### 标记-整理算法

根据老年代的特点提出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

#### 分代收集算法

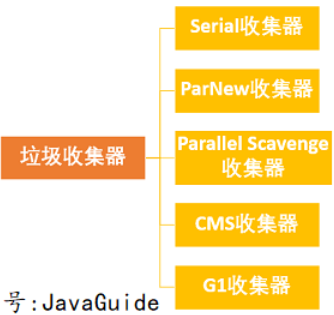
当前虚拟机的垃圾收集都采用分代收集算法，根据对象存活周期的不同将内存分为几块。一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

延伸面试问题： HotSpot 为什么要分为新生代和老年代？ 垃圾收集容易

### 垃圾收集器

收集算法是内存回收的方法论，垃圾收集器是内存回收的具体实现。根据不同场景选择适合自己的垃圾收集器



#### Serial 收集器

单线程收集器，简单高效，它的 “单线程” 的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（ "Stop The World" ），直到它收集结束。适用于client模式下

新生代采用复制算法，老年代采用标记-整理算法。

参数控制： -XX:+UseSerialGC 串行收集器

#### ParNew 收集器

Serial的多线程版本，可与CMS收集器配合使用，同样暂停应用程序

并行（Parallel） ：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。

并发（Concurrent）：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。

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：

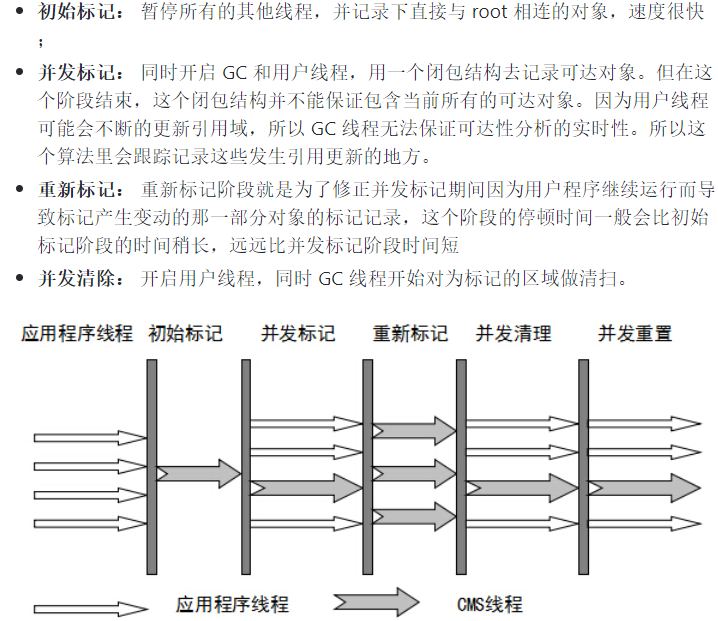
-XX:+UseParNewGC ParNew收集器

-XX:ParallelGCThreads 限制线程数量

#### CMS收集器

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。它非常符合在注重用户体验的应用上使用。是 HotSpot 虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。

基于“标记-清除”算法实现，分为四个阶段



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

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行，老年代收集器（新生代使用ParNew）

优点：并发收集、低停顿

缺点： “标记-清除”会导致收集结束时会有大量空间碎片，并发阶段会降低吞吐量

#### G1收集器

G1 (Garbage-First) 是一款面向服务器的垃圾收集器，低GC 停顿时间，高吞吐量

1.并行与并发：缩短 Stop-The-World 停顿时间。部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 java 程序继续执行。

2.空间整合：G1 从整体来看是基于“标记整理”算法实现的收集器

3.可预测的停顿： G1 除了追求低停顿外，还能建立可预测的停顿时间模型

G1 收集器的运作大致分为以下几个步骤：初始标记，并发标记，最终标记，筛选回收

上面的垃圾收集器收集范围都是整个新生代或者老年代，而G1将整个Java堆划分为多个大小相等的独立区域(Region)，虽然还保留有新生代和老年代的概念，优先选择回收价值最大的 Region

## JVM调优

### 调优工具

Jconsole（Java Monitoring and Management Console）

用于对JVM中内存，线程(可点击检测死锁)，类(已加载的类)和VM的监控，获取并展示虚拟机中运行的应用程序的性能和资源消耗等信息。

可选择监控本机的java应用，也可以选择远程的java服务来监控，若监控远程服务需要在tomcat启动脚本中添加代码

VisualVM (Java Virtual Machine, JVM)

调优很厉害，主要功能：监控、线程、Visual GC

# 复杂度



如果一个算法的复杂度为1 、 logn 、n 、 n\*logn ,那么这个算法时间效率比较高 ，如果是2n ,3n ,n!，那么稍微大一些的n就会令这个算法不能动了，居于中间的几个则差强人意

## 时间复杂度

一行代码一次运算

int aFunc(int n) {

for (int i = 0; i<n; i++) { // 需要执行 (n + 1) 次

printf("Hello, World!\n"); // 需要执行 n 次

}

return 0; // 需要执行 1 次

} 结果复杂度(n + 1 + n + 1) = 2n + 2 = O(n)

void aFunc(int n) {

for(int i = 0; i < n; i++) { // 循环次数为 n

for(int j = 0; j < n; j++) { // 循环次数为 n

printf("Hello, World!\n"); // 循环体时间复杂度为 O(1)

}

}

} 结果复杂度O(n × n × 1 ) = O(n^2)。

void aFunc(int n) {

// 第一部分时间复杂度为 O(n^2)

for(int i = 0; i < n; i++) {

for(int j = 0; j < n; j++) {

printf("Hello, World!\n");

}

}

// 第二部分时间复杂度为 O(n)

for(int j = 0; j < n; j++) {

printf("Hello, World!\n");

}

} 结果复杂度max(O(n^2), O(n)) = O(n^2)

void aFunc(int n) {

if (n >= 0) {

// 第一条路径时间复杂度为 O(n^2)

for(int i = 0; i < n; i++) {

for(int j = 0; j < n; j++) {

printf("输入数据大于等于零\n");

}

}

} else {

// 第二条路径时间复杂度为 O(n)

for(int j = 0; j < n; j++) {

printf("输入数据小于零\n");

}

}

} 结果复杂度max(O(n^2), O(n)) = O(n^2)

void aFunc(int n) {

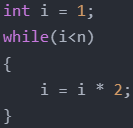
for (int i = 2; i < n; i++) {

i \*= 2;

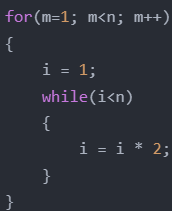
printf("%i\n", i);

}

} 结果复杂度 = O(log n)



从上面代码可以看到，在while循环里面，每次都将 i 乘以 2，乘完之后，i 距离 n 就越来越近了。假设循环x次之后，i 就大于 2 了，此时这个循环就退出了，也就是说 2 的 x 次方等于 n，那么 x = log2n，也就是说当循环 log2n次以后，代码结束。因此这个代码的时间复杂度为：O(logn)



将时间复杂度为O(logn)的代码循环N遍的话，那么它的时间复杂度就是 n \* O(logN)，也就是了O(nlogN)。

long aFunc(int n) {

if (n <= 1) {

return 1;

} else {

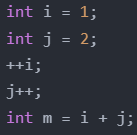
return aFunc(n - 1) + aFunc(n - 2);

}

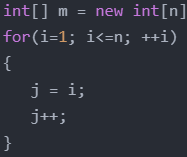
} 复杂度可表示为 O((5/3)^n)，简化后为 O(2^n)

## 空间复杂度

如果算法执行所需要的临时空间不随着某个变量n的大小而变化，即此算法空间复杂度为一个常量，可表示为 O(1)



第一行new了一个数组出来，这个数据占用的大小为n，这段代码的2-6行，虽然有循环，但没有再分配新的空间，因此，这段代码的空间复杂度主要看第一行即可，即 S(n) = O(n)



# To be continue

Io

算法题

多线程

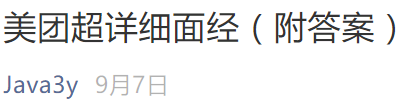
数据库 <https://blog.csdn.net/justloveyou_/article/details/78308460>

网络 <https://blog.csdn.net/justloveyou_/article/details/78303617>

计算机系统 <https://blog.csdn.net/justloveyou_/article/details/78304294>

一个有项目的面试<https://blog.csdn.net/qq_34337272/article/details/89315360>

面试美团问题



<https://mp.weixin.qq.com/s?__biz=MzI4Njg5MDA5NA==&mid=2247485636&idx=1&sn=3ab554f1c7e9745bf033d89577dfaa2b&chksm=ebd749c5dca0c0d3874ead44f3a03d7bd88ae1dbba7ab2828fc966598a85d7a4aaeeee5ca1be&token=1948873548&lang=zh_CN#rd>

比如互联网金融类公司的岗位，如果有过支付和银行相关的系统开发经验肯定会有加分，这点和每个人的长期规划有关。

Crm 面试问题

<https://crmtipsbyprm.wordpress.com/category/interview-questions/>

* 应付算法，除了常见的二分查找，快速排序，链表判断有环，链表逆序，二叉树的三种遍历
* 了解下java虚拟机的结构，重点是堆的结构，以及各部分的功能要知道，必须掌握现代Java垃圾回收算法的种类，简单调优，和分代算法的思想，了解四种Java引用类型是什么。看周志明的那本《深入理解Java虚拟机》就够了，应届生。
* 熟悉一下LRU的缓存实现原理，重点是如何实现并发，顺便梳理下操作系统的内存存储方式，各种算法，分页，分段，碎片啥的理解下。
* Java面试常考，可以互相引申的就是Java集合的知识点：hashmap和concurrentHashMap原理、区别，hashTable，hashset，各种list等，看《编程思想》结合网上博客文章总结即可。
* 熟悉下多线程设计模式，比如生产者和消费者，不变模式等。有一本日本人写的比较老的书可以看看《Java多线程设计模式》，深入的JUC并发知识最好看下《Java并发编程实践》，老外的书非常棒。并且结合操作系统课程可以梳理下进程，线程的各种知识点。
* 计算机网络，起码的几层协议要会，记住，理解tcp三次握手（具体报文格式），概念：流量控制，滑动窗口机制，选择重传等，一般应届生也就够用了，可以看下老外那本经典的《tcp/ip详解》卷一，深入可以把卷二看了，虽然我没看过。但是确实好。
* 设计模式起码掌握单例，工厂，观察者等，系统的可以看下《Head First设计模式》，应付日常工作和面试足以。设计模式这也是各个框架的考察中常考的点，重点一般就是单例的各种版本的优缺点，线程安全实现，枚举的使用，反射，序列化的攻击应对方法。工厂的类型等。
* 反射，序列化的概念
* clone的一些知识点
* 最好熟悉下类加载机制
* 框架暂时没有遇到考察的。熟悉原理最好，了解过，读过源码更好了。如果是Java大数据方向的求职，相应的就是hadoop，spark等，对应的大数据的算法也要掌握一些。

public static void Main(string[] args)

{

DateTime date = DateTime.Parse(args[0]);

DateTime dateEnd= DateTime.Parse(args[0]);

//check input is period time or not

if (args.Length == 2)

{

dateEnd = DateTime.Parse(args[1]);

}

通过命令行运行时 可以直接通过命令行输入传参数

* Please explain how to reduce packet loss on each layer of TCP/IP stack?
* What is memory reordering? Please give an example and explanation.
* How can CPU cache affect your algorithm design? Please give an example and explanation.
* 自己编程实现栈，并用双栈实现队列
* introduce yourself
* why not tencent
* introduce a project you did and what problem you solved
* what is recursion？
* what is hashtable？
* how to implement a hashtable, what data structure should be used？
* stack and heap
* in what scenario，we can only use heap.
* memory leak
* how to detect memory leak
* what is virtual function
* how to implement virtual function and polymorphism
* what is copy constructor
* for a given string, how to output all the different combination.