# 树的相关概念

在计算机科学中，**树**（英语：tree）是一种**抽象数据类型**（**A**bstract **D**ata **T**ype，**ADT**）的[数据结构](https://zh.wikipedia.org/wiki/%E8%B3%87%E6%96%99%E7%B5%90%E6%A7%8B)，用来模拟具[树状结构](https://zh.wikipedia.org/wiki/%E6%A8%B9%E7%8B%80%E7%B5%90%E6%A7%8B)性质的资料集合。

我们把老祖或者树根称为根（root）节点，老祖的儿子称为子节点，每个儿子作为根节点又可以形成一棵树，我们把这样的树称为根节点的子树。

树的标准定义：

树（tree）是包含n（n>0）个节点的有穷集合，其中：

　　（1）每个元素称为节点（node）；

　　（2）有一个特定的节点被称为根节点或树根（root）。

（3）除根节点之外的其余数据元素被分为m（m≥0）个互不相交的结合T1，T2，……Tm-1，其中每一个集合Ti（1<=i<=m）本身也是一棵树，被称作原树的子树（subtree）。

**树具有以下特点：**

（1）每个节点有零个或多个子节点。

（2）每个子节点只有一个父节点。

（3）没有父节点的节点称为根节点。

**关于树的一些术语**

1. 节点的度：一个节点含有的子树的个数称为该节点的度；
2. 树的度：一棵树中，最大的节点的度称为树的度；
3. 叶节点或终端节点：度为零的节点；
4. 非终端节点或分支节点：度不为零的节点；
5. 父亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点；
6. 孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点；
7. 兄弟节点：具有相同父节点的节点互称为兄弟节点；
8. 节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推；
9. 树的高度或深度：树中节点的最大层次；
10. 堂兄弟节点：父节点在同一层的节点互为堂兄弟；
11. 节点的祖先：从根到该节点所经分支上的所有节点；
12. 子孙：以某节点为根的子树中任一节点都称为该节点的子孙。
13. 森林：由m（m>=0）棵互不相交的树的集合称为森林；

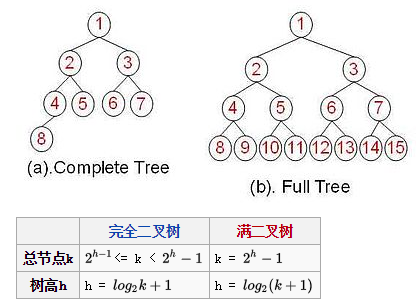
**二叉树**

二叉树的每个结点至多只有二棵子树（不存在度大于 2 的结点），二叉树的子树有左右之分，次序不能颠倒。

二叉树的第 i 层至多有 2i - 1 个结点；

深度为 k 的二叉树至多有 2k - 1 个结点；

对任何一棵二叉树 T，如果其终端结点数为 n0，度为 2 的结点数为 n2，则n0 = n2 + 1。

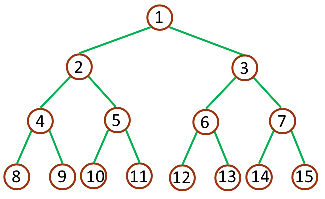


树和二叉树的三个主要差别：

* 树的结点个数至少为 1，而二叉树的结点个数可以为 0
* 树中结点的最大度数没有限制，而二叉树结点的最大度数为 2
* 树的结点无左、右之分，而二叉树的结点有左、右之分

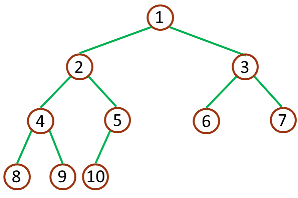
二叉树又分为完全二叉树（complete binary tree）和满二叉树（full binary tree）

满二叉树：一棵深度为 k，且有 2k - 1 个节点称之为满二叉树

[](http://bubkoo.qiniudn.com/full%C2%A0binary%C2%A0tree.png)

深度为 3 的满二叉树 full binary tree

完全二叉树：深度为 k，有 n 个节点的二叉树，当且仅当其每一个节点都与深度为 k 的满二叉树中序号为 1 至 n 的节点对应时，称之为完全二叉树

[](http://bubkoo.qiniudn.com/complete%C2%A0binary%C2%A0tree.png)

深度为 3 的完全二叉树 complete binary tree

## 树的遍历：

**深度优先遍历DFS**

在深度优先级中，我们希望从根结点访问最远的结点。和图的[深度优先搜索](https://zh.wikipedia.org/wiki/%E6%B7%B1%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2)不同的是，不需记住访问过的每一个结点，因为树中不会有环。前序，中序和后序遍历都是深度优先遍历的特例。

前序遍历：根节点->左子树->右子树

中序遍历：左子树->根节点->右子树

后序遍历：左子树->右子树->根节点

1. 

前序： G D A F E M H Z

中序： A D E F G H M Z

后序： A E F D H Z M G

**广度优先遍历BFS**

和深度优先遍历不同，广度优先遍历会先访问离根节点最近的节点。参见[广度优先搜索](https://zh.wikipedia.org/wiki/%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2)。 二叉树的广度优先遍历又称按层次遍历。算法借助队列实现。

层序遍历： G D M A F H Z E

1. **已知前序、中序遍历，求后序遍历**

例：

前序遍历: G D A F E M H Z

中序遍历: A D E F G H M Z

画树求法：第一步，根据前序遍历的特点，我们知道根结点为G

              第二步，观察中序遍历ADEFGHMZ。其中root节点G左侧的ADEF必然是root的左子树，G右侧的HMZ必然是root的右子树。

              第三步，递归，观察左子树中序遍历为ADEF，根据前序遍历序列DAFE，知道该左子树的根节点为D，继续划分得到结构：父节点D，左子树A，右子树EF。

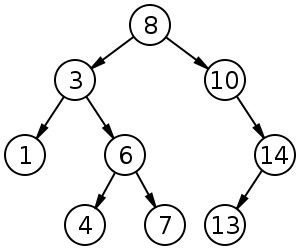
              第四步，同样的道理，root的右子树节点HMZ中的根节点也可以通过前序遍历求得。

      观察发现，上面的过程是递归的。先找到当前树的根节点，然后划分为左子树，右子树，然后进入左子树重复上面的过程，然后进入右子树重复上面的过程。最后就可以还原一棵树了。

# 二叉查找树

二叉查找树（英语：Binary Search Tree），也称二叉搜索树、有序二叉树（英语：ordered binary tree），排序二叉树（英语：sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

1. 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；
2. 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；
3. 任意节点的左、右子树也分别为二叉查找树；
4. 没有键值相等的节点。



二叉查找树的性质：对二叉查找树进行中序遍历，即可得到有序的数列。

　　二叉查找树的时间复杂度：它和二分查找一样，插入和查找的时间复杂度均为O(logn)，但是在最坏的情况下仍然会有O(n)的时间复杂度。原因在于插入和删除元素的时候，树没有保持平衡（比如，我们查找上图（b）中的“93”，我们需要进行n次查找操作）。我们追求的是在最坏的情况下仍然有较好的时间复杂度，这就是平衡查找树设计的初衷。

二叉查找树的高度决定了二叉查找树的查找效率。

二叉查找树的插入过程如下：

　　1) 若当前的二叉查找树为空，则插入的元素为根节点;

　　2) 若插入的元素值小于根节点值，则将元素插入到左子树中;

　　3) 若插入的元素值不小于根节点值，则将元素插入到右子树中。

二叉查找树的删除，分三种情况进行处理：

　　1) p为叶子节点，直接删除该节点，再修改其父节点的指针（注意分是根节点和不是根节点），如图a;

　　2) p为单支节点（即只有左子树或右子树）。让p的子树与p的父亲节点相连，删除p即可（注意分是根节点和不是根节点），如图b;

　　3) p的左子树和右子树均不空。找到p的后继y，因为y一定没有左子树，所以可以删除y，并让y的父亲节点成为y的右子树的父亲节点，并用y的值代替p的值；或者方法二是找到p的前驱x，x一定没有右子树，所以可以删除x，并让x的父亲节点成为y的左子树的父亲节点。

# 平衡二叉树

我们知道，对于一般的二叉搜索树（Binary Search Tree），其期望高度（即为一棵平衡树时）为log2n，其各操作的时间复杂度O(log2n)同时也由此而决定。但是，在某些极端的情况下（如在插入的序列是有序的时），**二叉搜索树将退化成近似链或链，此时，其操作的时间复杂度将退化成线性的，即O(n)**。我们可以通过随机化建立二叉搜索树来尽量的避免这种情况，但是在进行了多次的操作之后，由于在删除时，我们总是选择将待删除节点的后继代替它本身，这样就会造成总是右边的节点数目减少，以至于树向左偏沉。这同时也会造成树的平衡性受到破坏，提高它的操作的时间复杂度。于是就有了我们下边介绍的平衡二叉树。

　　平衡二叉树定义：平衡二叉树（Balanced Binary Tree）又被称为AVL树（有别于AVL算法），且具有以下性质：它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。平衡二叉树的常用算法有红黑树、AVL树等。在平衡二叉搜索树中，我们可以看到，其高度一般都良好地维持在O(log2n)，大大降低了操作的时间复杂度。

　　最小二叉平衡树的节点的公式如下：

　　F(n)=F(n-1)+F(n-2)+1

　　这个类似于一个递归的数列，可以参考Fibonacci数列，1是根节点，F(n-1)是左子树的节点数量，F(n-2)是右子树的节点数量。

## 平衡查找树之AVL树

<http://www.cnblogs.com/maybe2030/p/4732377.html>

有关AVL树的具体实现，可以参考[C小加的博客《一步一步写平衡二叉树（AVL）》](http://www.cppblog.com/cxiaojia/archive/2012/08/20/187776.html)。

**AVL树定义：**AVL树是最先发明的自平衡二叉查找树。AVL树得名于它的发明者 G.M. Adelson-Velsky 和 E.M. Landis，他们在 1962 年的论文 "An algorithm for the organization of information" 中发表了它。在AVL中任何节点的两个儿子子树的高度最大差别为1，所以它也被称为高度平衡树，n个结点的AVL树最大深度约1.44log2n。查找、插入和删除在平均和最坏情况下都是O(logn)。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。**这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。**

**AVL树的自平衡操作——旋转：**

　　AVL树最关键的也是最难的一步操作就是**旋转**。旋转主要是为了实现AVL树在实施了插入和删除操作以后，树重新回到平衡的方法。下面我们重点研究一下AVL树的旋转。

　　对于一个平衡的节点，由于任意节点最多有两个儿子，因此高度不平衡时，此节点的两颗子树的高度差2.容易看出，这种不平衡出现在下面四种情况：



1) 6节点的左子树3节点高度比右子树7节点大2，左子树3节点的左子树1节点高度大于右子树4节点，这种情况成为左左。

2) 6节点的左子树2节点高度比右子树7节点大2，左子树2节点的左子树1节点高度小于右子树4节点，这种情况成为左右。

3) 2节点的左子树1节点高度比右子树5节点小2，右子树5节点的左子树3节点高度大于右子树6节点，这种情况成为右左。

4) 2节点的左子树1节点高度比右子树4节点小2，右子树4节点的左子树3节点高度小于右子树6节点，这种情况成为右右。

从图2中可以可以看出，1和4两种情况是对称的，这两种情况的旋转算法是一致的，只需要经过一次旋转就可以达到目标，我们称之为单旋转。2和3两种情况也是对称的，这两种情况的旋转算法也是一致的，需要进行两次旋转，我们称之为双旋转。

**单旋转**

单旋转是针对于左左和右右这两种情况的解决方案，这两种情况是对称的，只要解决了左左这种情况，右右就很好办了。图3是左左情况的解决方案，节点k2不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且k1子树中，更深的一层的是k1的左子树X子树，所以属于左左情况。



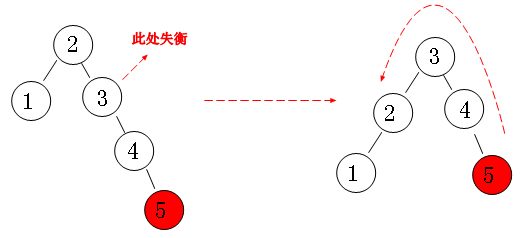
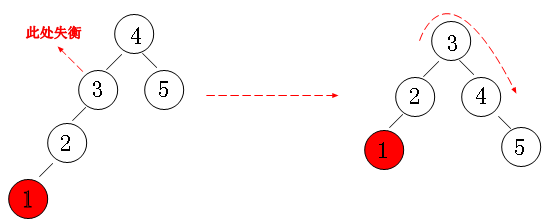
为使树恢复平衡，我们把k1变成这棵树的根节点，因为k2大于k1，把k2置于k1的右子树上，而原本在k1右子树的Y大于k1，小于k2，就把Y置于k2的左子树上，这样既满足了二叉查找树的性质，又满足了平衡二叉树的性质。

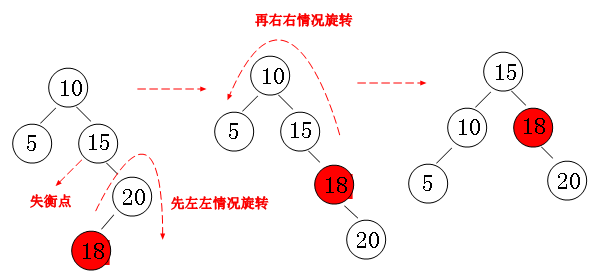
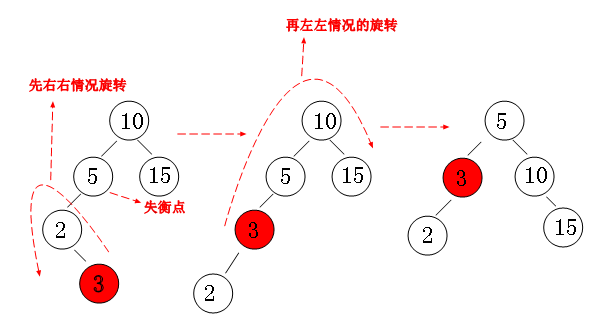
**双旋转**

对于左右和右左这两种情况，单旋转不能使它达到一个平衡状态，要经过两次旋转。双旋转是针对于这两种情况的解决方案，同样的，这样两种情况也是对称的，只要解决了左右这种情况，右左就很好办了。图4是左右情况的解决方案，节点k3不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且k1子树中，更深的一层的是k1的右子树k2子树，所以属于左右情况。



为使树恢复平衡，我们需要进行两步，第一步，把k1作为根，进行一次右右旋转，旋转之后就变成了左左情况，所以第二步再进行一次左左旋转，最后得到了一棵以k2为根的平衡二叉树。





## 平衡二叉树之红黑树

**红黑树的定义：**红黑树是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它是在1972年由鲁道夫·贝尔发明的，称之为"对称二叉B树"，它现代的名字是在 Leo J. Guibas 和 Robert Sedgewick 于1978年写的一篇论文中获得的。**它是复杂的，但它的操作有着良好的最坏情况运行时间，并且在实践中是高效的: 它可以在O(logn)时间内做查找，插入和删除，这里的n是树中元素的数目。**

红黑树和AVL树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。这不只是使它们在时间敏感的应用如实时应用（real time application）中有价值，而且使它们有在提供最坏情况担保的其他数据结构中作为建造板块的价值；例如，在计算几何中使用的很多数据结构都可以基于红黑树。此外，红黑树还是2-3-4树的一种等同，它们的思想是一样的，只不过红黑树是2-3-4树用二叉树的形式表示的。

**红黑树的性质：**

　　红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制的一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:

　　性质1. 节点是红色或黑色。

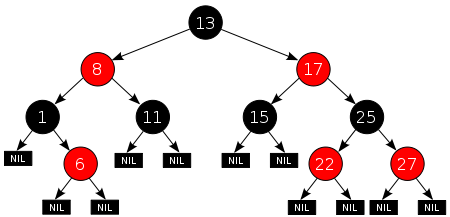
　　性质2. 根是黑色。

　　性质3. 所有叶子都是黑色（叶子是NIL节点）。

　　性质4. 每个红色节点必须有两个黑色的子节点。(从每个叶子到根的所有路径上不能有两个连续的红色节点。)

　　性质5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

　　下面是一个具体的红黑树的图例：



　　这些约束确保了红黑树的关键特性: 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

要知道为什么这些性质确保了这个结果，注意到性质4导致了路径不能有两个毗连的红色节点就足够了。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据性质5所有的路径都有相同数目的黑色节点(黑色节点大于红色节点数目)，这就表明了没有路径能多于任何其他路径的两倍长。

**红黑树的自平衡操作：**

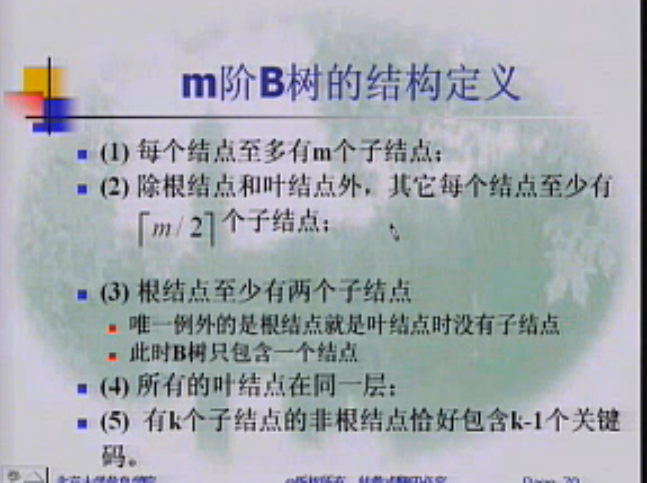
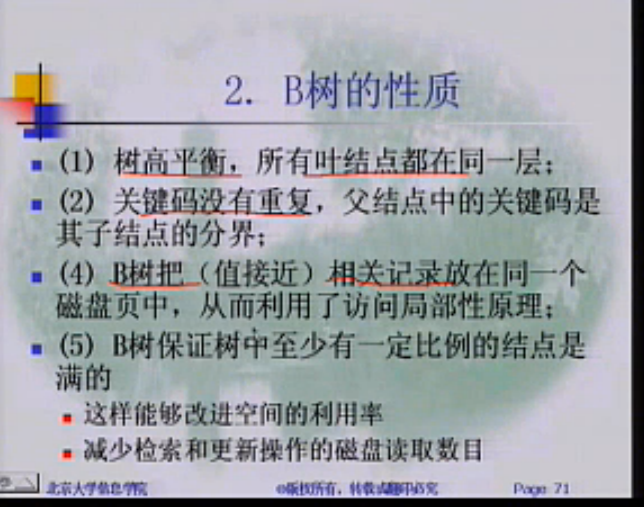
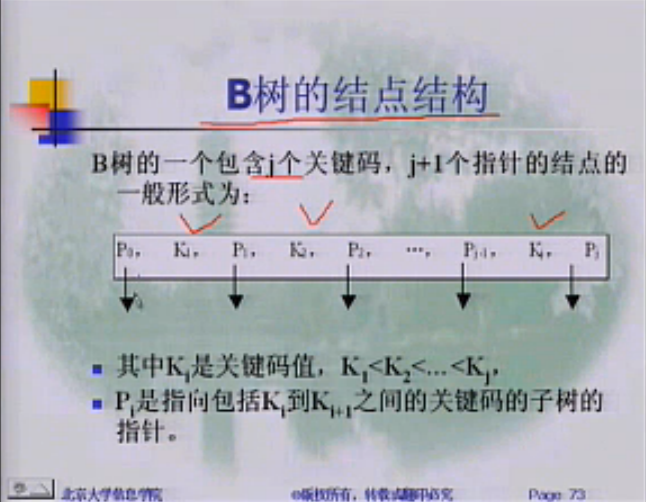
因为每一个红黑树也是一个特化的二叉查找树，因此红黑树上的只读操作与普通二叉查找树上的只读操作相同。然而，在红黑树上进行插入操作和删除操作会导致不再符合红黑树的性质。恢复红黑树的性质需要少量(O(logn))的颜色变更(实际是非常快速的)和不超过三次树旋转(对于插入操作是两次)。虽然插入和删除很复杂，但操作时间仍可以保持为O(logn) 次。

## B树

B树也是一种用于查找的平衡树，但是它不是二叉树。

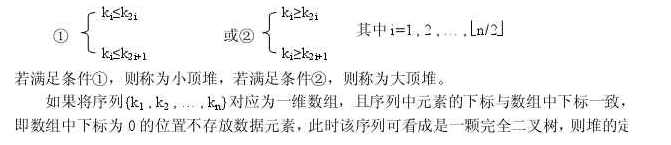
**B树的定义：**B树（B-tree）是一种树状数据结构，能够用来存储排序后的数据。这种数据结构能够让查找数据、循序存取、插入数据及删除的动作，都在对数时间内完成。B树，概括来说是一个一般化的二叉查找树，可以拥有多于2个子节点。与自平衡二叉查找树不同，B-树为系统最优化大块数据的读和写操作。B-tree算法减少定位记录时所经历的中间过程，从而加快存取速度。这种数据结构常被应用在数据库和文件系统的实作上。

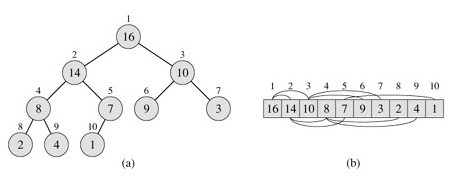
在B树中查找给定关键字的方法是，首先把根结点取来，在根结点所包含的关键字K1,…,Kn查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查找的关键字在Ki与Ki+1之间，Pi为指向子树根节点的指针，此时取指针Pi所指的结点继续查找，直至找到，或指针Pi为空时查找失败。

# 最大-最小堆

堆是完全二叉树。

堆有最大堆和最小堆之分，最大堆就是每个节点的值都>=其左右孩子（如果有的话）值的完全二叉树。最小堆便是每个节点的值都<=其左右孩子值的完全二叉树。   
  
  设有n个元素的序列{k1,k2,...,kn},当且仅当满足下列关系时,称之为堆。



用数组来表示堆，按照二叉树的BFS遍历。

数组是以下标0开始，所以父节点 i， 左孩子2i+1 右孩子2（i+1）

堆有两种基本操作（下面以最小堆为例）：

插入元素k：直接将k添加到数组最后，然后向上冒泡（bubble-up）调整堆。向上冒泡操作：将要调整的元素与其父节点比较，如果小于其父节点则交换，（否则结束）直到恢复堆的性质。

提取最值：最值即根元素。然后将其删除，令根元素=最后的叶子结点元素，然后从根元素开始向下冒泡（bubble-down）调整堆。

向下冒泡操作：每次应该从要调整节点，其左右孩子一共三个节点中选择最小的子节点来交换（如果最小就是其本身就不用交换，结束），直到恢复堆的性质，该操作的前提是，待调整节点以下的堆符合最大堆或者最小堆的特征。

**堆排序算法：**

static void heapSort**(**int**[]** arr**)** **{**

int len **=** arr**.**length**;**

//此处将数组调整为最大堆，叶节点不需要调整

**for** **(**int i**=**len**/**2**;** i**>=**0**;** i**--)** **{**

heapAdjust**(**arr**,** i**,** len**);**

**}**

//将堆根结点放到数组最后，重新调整堆，注意堆的调整范围也要改变

**for** **(**int i**=**len**-**1**;** i**>**0**;** i**--)** **{**

swap**(**arr**,** i**,** 0**);**

heapAdjust**(**arr**,** 0**,** i**);**

**}**

**}**

//向下调整最大堆，i为待调整节点的下标，len表示待调整的数组长度

private static void heapAdjust**(**int**[]** arr**,** int i**,** int len**)** **{**

int child**;**

**while** **((**child **=** i**\***2 **+** 1**)** **<** len**)** **{**

**if** **(**child **+** 1 **<** len **&&** arr**[**child**+**1**]** **>** arr**[**child**])** **{**

child**++;**

**}**

**if** **(**arr**[**child**]** **>** arr**[**i**])** **{**

swap**(**arr**,** i**,** child**);**

i **=** child**;**

**}** **else** **{**

**break;**

**}**

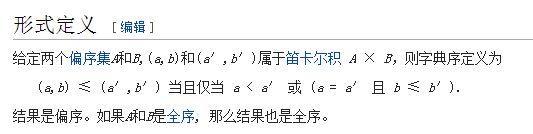
**}**

**}**

# 字典序

设想一本英语字典里的单词，何者在前何者在后？

显然的做法是先按照第一个字母、以 a、b、c……z 的顺序排列；如果第一个字母一样，那么比较第二个、第三个乃至后面的字母。如果比到最后两个单词不一样长（比如，sigh 和 sight），那么把短者排在前。



# ArrayList和LinkedList的区别

**LinkedList和ArrayList的区别**

LinkedList和ArrayList的差别主要来自于Array和LinkedList数据结构的不同。如果你很熟悉Array和LinkedList，你很容易得出下面的结论：

1) 因为Array是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。Array获取数据的时间复杂度是O(1),但是要删除数据却是开销很大的，因为这需要重排数组中的所有数据。

2) 相对于ArrayList，LinkedList插入是更快的。因为LinkedList不像ArrayList一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据重新装入一个新的数组，这是ArrayList最坏的一种情况，时间复杂度是O(n)，而LinkedList中插入或删除的时间复杂度仅为O(1)。ArrayList在插入数据时还需要更新索引（除了插入数组的尾部）。

3) 类似于插入数据，删除数据时，LinkedList也优于ArrayList。

4) LinkedList需要更多的内存，因为ArrayList的每个索引的位置是实际的数据，而LinkedList中的每个节点中存储的是实际的数据和前后节点的位置。

**什么场景下更适宜使用LinkedList，而不用ArrayList**

我前面已经提到，很多场景下ArrayList更受欢迎，但是还有些情况下LinkedList更为合适。譬如：

1) 你的应用不会随机访问数据。因为如果你需要LinkedList中的第n个元素的时候，你需要从第一个元素顺序数到第n个数据，然后读取数据。

2) 你的应用更多的插入和删除元素，更少的读取数据。因为插入和删除元素不涉及重排数据，所以它要比ArrayList要快。

以上就是关于ArrayList和LinkedList的差别。你需要一个不同步的基于索引的数据访问时，请尽量使用ArrayList。ArrayList很快，也很容易使用。但是要记得要给定一个合适的初始大小，尽可能的减少更改数组的大小。

**总结**

ArrayList和LinkedList在性能上各有优缺点，都有各自所适用的地方，总的说来可以描述如下：   
1．对ArrayList和LinkedList而言，在列表末尾增加一个元素所花的开销都是固定的。对ArrayList而言，主要是在内部数组中增加一项，指向所添加的元素，偶尔可能会导致对数组重新进行分配；而对LinkedList而言，这个开销是统一的，分配一个内部Entry对象。

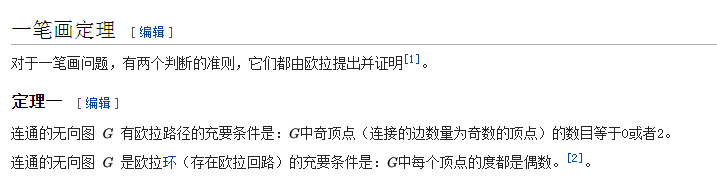
2．在ArrayList的中间插入或删除一个元素意味着这个列表中剩余的元素都会被移动；而在LinkedList的中间插入或删除一个元素的开销是固定的。

3．LinkedList不支持高效的随机元素访问。

4．ArrayList的空间浪费主要体现在在list列表的结尾预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗相当的空间

可以这样说：当操作是在一列数据的后面添加数据而不是在前面或中间,并且需要随机地访问其中的元素时,使用ArrayList会提供比较好的性能；当你的操作是在一列数据的前面或中间添加或删除数据,并且按照顺序访问其中的元素时,就应该使用LinkedList了。

# 欧拉问题



# IP地址

IP地址，IP地址是由32位二进制位组成，由网络地址和主机地址两部分组成，其编址的方法与街道地址的概念相似，比如XX街YY号，这里的XX街相当于网络地址，YY号相当于主机地址。

所谓分类的IP地址，就是将IP地址划分为若干固定类，每一类地址都由两个固定长度的字段组成，其中一个字段是**网络号 net-id**，它标志主机（或路由器）所连接到的网络，一个网络号在整个因特网范围内必须是唯一的。而另一个字段则是**主机号 host-id**，它标志该主机（或路由器），一个主机号在它前面的网络号所指明的网络范围内必须是唯一的。由此可见，一个IP地址在整个因特网范围内是唯一的。

由于一个路由器至少应当连接到两个网络（这样它才能将 IP 数据报从一个网络转发到另一个网络），因此一个路由器至少应当有两个不同的 IP 地址。

**子网掩码**不是一个地址,它是用来确定一个IP地址中哪一部分是网络ID,哪一部分是主机ID的,连续为1的部分代表网络ID,连续为0的部分代表主机ID.如C类网络的前24位为网络号,后8位为主机号,那么C类网的子网掩码就是255.255.255.0

IP地址分为5类：数量非常有限的，非常大的网络（A类）；数量较多规模中等的网络（B类）；为数众多的小网络（C类），以及用于组播（D类）和研究和实验之用（E类）。区分的办法就是根据第一个8位组来判断，常用的是ABC三类。



1. 对于A类地址，分配IP地址的前8位作为网络地址，其余24位作为主机地址。取值的范围应在0--127之间。再加上0和127已经被留做它用，所以A类网只有126个。网络号为127（01111111）保留作为本地软件环回测试本主机的进程间的通信。
2. 对于B类地址，分配IP地址的前16位作为网络地址，其余16位作为主机地址。并且，这前16位二进制位中的前两位必须是“10”。转换成十进制后，其取值的范围应在128--191之间。由于最高两位被限定为“10”，所以，实际上只有其后的14位用来区分网络地址，所以B；类地址有16384个。
3. 对于C类地址，分配IP地址的前24位作为网络地址，其余8位作为主机地址。并且，这前24位二进制位中的前三位必须是“110”。转换成十进制后，其取值的范围应在192--223之间。所以C类地址可以有2^21次方，共2097152个。同时由于只有8位数字用于表示主机机址，所以每个C类网只能有254台主机（主机地址不能全为0或全为1：全0为子网地址，全1为广播地址）。

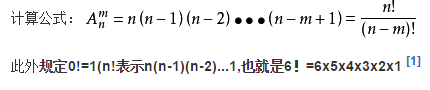
网关

那么网关到底是什么呢？网关实质上是一个网络通向其他网络的[IP地址](http://baike.baidu.com/view/3930.htm)。比如有网络A和网络B，网络A的[IP](http://baike.baidu.com/view/8370.htm)地址范围为“192.168.1.1~192. 168.1.254”，[子网掩码](http://baike.baidu.com/view/878.htm)为255.255.255.0；网络B的IP地址范围为“192.168.2.1~192.168.2.254”，子网掩码为255.255.255.0。在没有[路由器](http://baike.baidu.com/view/1360.htm)的情况下，两个网络之间是不能进行TCP/IP通信的，即使是两个网络连接在同一台[交换机](http://baike.baidu.com/view/1077.htm)（或[集线器](http://baike.baidu.com/view/7770.htm)）上，TCP/IP协议也会根据[子网掩码](http://baike.baidu.com/view/878.htm)（255.255.255.0）判定两个网络中的主机处在不同的网络里。而要实现这两个网络之间的通信，则必须通过网关。如果网络A中的主机发现[数据包](http://baike.baidu.com/view/25880.htm)的目的主机不在本地网络中，就把数据包转发给它自己的网关，再由网关转发给网络B的网关，网络B的网关再转发给网络B的某个主机（如附图所示）。网络A向网络B转发数据包的过程。

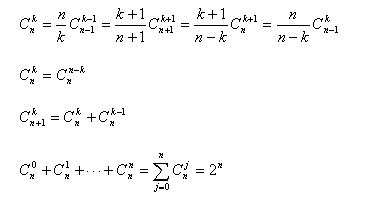
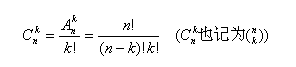
如果搞清了什么是网关，[默认网关](http://baike.baidu.com/view/119302.htm)也就好理解了。就好像一个房间可以有多扇门一样，一台主机可以有多个网关。[默认网关](http://baike.baidu.com/view/119302.htm)的意思是一台主机如果找不到可用的网关，就把数据包发给默认指定的网关，由这个网关来处理数据包。[默认网关](http://baike.baidu.com/view/119302.htm)。[默认网关](http://baike.baidu.com/view/119302.htm)一般填写192.168.x.1

# 排列组合

**排列：** 排列的定义：从n个不同元素中，任取m(m≤n,m与n均为自然数,下同）个元素按照一定的顺序排成一列，叫做从n个不同元素中取出m个元素的一个排列；从n个不同元素中取出m(m≤n）个元素的所有排列的个数，叫做从n个不同元素中取出m个元素的排列数，用符号 A(n,m）表示。

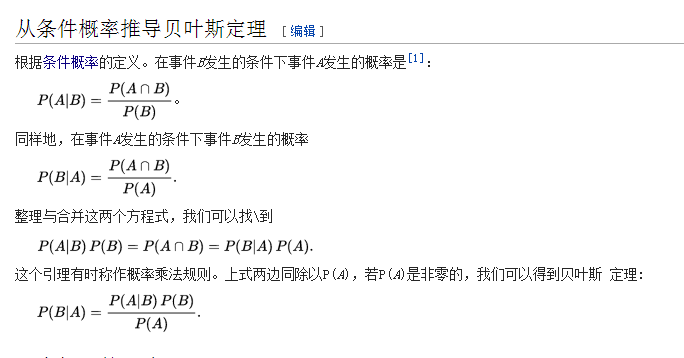


组合：从*n*个不同的元素中每次取出*k*个(*k*≤*n*)不同的元素，不管其顺序合并成一组，称为组合．其组合种数为：



**贝叶斯定理**

通常，事件A在事件B（发生）的条件下的概率，与事件B在事件A的条件下的概率是不一样的；然而，这两者是有确定的关系，贝叶斯定理就是这种关系的陈述。贝叶斯公式的用途在于通过己知三个概率函数推出第四个。它的内容是:在B出现的前提下,A出现的概率等于A和B都出现的概率除以B出现的概率。通过联系A与B,计算从一个事件产生另一事件的概率,即从结果上溯原。



# 生产者消费者实现

生产者消费者问题是线程模型中的经典问题：生产者和消费者在同一时间段内共用同一存储空间，生产者向空间里生产数据，而消费者取走数据。

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

这个阻塞队列就是用来给生产者和消费者解耦的。纵观大多数设计模式，都会找一个第三者出来进行解耦，如工厂模式的第三者是工厂类，模板模式的第三者是模板类。在学习一些设计模式的过程中，如果先找到这个模式的第三者，能帮助我们快速熟悉一个设计模式。

**import** java**.**util**.**concurrent**.**locks**.**Condition**;**

**import** java**.**util**.**concurrent**.**locks**.**Lock**;**

**import** java**.**util**.**concurrent**.**locks**.**ReentrantLock**;**

public class ProductConsumer **{**

public static void main**(**String**[]** args**)** **{**

Clerk clerk **=** **new** Clerk**();**

**new** Thread**(new** Product**(**"p1"**,** clerk**)).**start**();**

**new** Thread**(new** Consumer**(**"c1"**,** clerk**)).**start**();**

**new** Thread**(new** Consumer**(**"c2"**,** clerk**)).**start**();**

**new** Thread**(new** Consumer**(**"c3"**,** clerk**)).**start**();**

// new Thread(new Product("p2", clerk)).start();//当使用多个生产者时候就不正常了

**}**

**}**

class Clerk **{**

private int product **=** **-**1**;**

Lock lock **=** **new** ReentrantLock**();**

Condition consumerCondition **=** lock**.**newCondition**();**

Condition productCondition **=** lock**.**newCondition**();**

public void setProduct**(**int p**)** **throws** Exception **{**//被product调用

lock**.**lock**();**

**try** **{**

waitIfFull**();**

product **=** p**;**

consumerCondition**.**signal**();**//通知消费者等待集合，可以消费了

**}** **finally** **{**

lock**.**unlock**();**

**}**

**}**

private void waitIfFull**()** **throws** Exception **{**

**while** **(**product **!=** **-**1**)** **{**

productCondition**.**await**();**//加入生产者等待集合

**}**

**}**

public int getProduct**()** **throws** Exception **{**//被consumer调用

lock**.**lock**();**

int p **=** 0**;**

**try** **{**

waitIfEmpty**();**

p **=** product**;**

product **=** **-**1**;**

productCondition**.**signal**();**//通知生产者等待集合，开始生产了

**}** **finally** **{**

lock**.**unlock**();**

**}**

**return** p**;**

**}**

private void waitIfEmpty**()** **throws** Exception **{**

**while** **(**product **==** **-**1**)** **{**

consumerCondition**.**await**();** //加入消费者等待集合

**}**

**}**

**}**

class Product **implements** Runnable **{**

Clerk clerk**;**

String name**;**

public Product**(**String n**,** Clerk c**)** **{**

name **=** n**;**

clerk **=** c**;**

**}**

@Override

public void run**()** **{**

int p **=** 1**;**

**while** **(**p **<** 10**)** **{**

**try** **{**

clerk**.**setProduct**(**p**);**

System**.**out**.**println**(**name**+**" product "**+**p**);**

p**++;**

Thread**.**sleep**(**100**);**

**}** **catch** **(**Exception e**)** **{**

e**.**printStackTrace**();**

**}**

**}**

**}**

**}**

class Consumer **implements** Runnable **{**

Clerk clerk**;**

String name**;**

public Consumer**(**String n**,** Clerk c**)** **{**

name **=** n**;**

clerk **=** c**;**

**}**

@Override

public void run**()** **{**

int p **=** 0**;**

**while** **(true)** **{**

**try** **{**

p **=** clerk**.**getProduct**();**

System**.**out**.**println**(**name**+**" consum "**+**p**);**

Thread**.**sleep**(**500**);**

**}** **catch** **(**Exception e**)** **{**

e**.**printStackTrace**();**

**}**

**}**

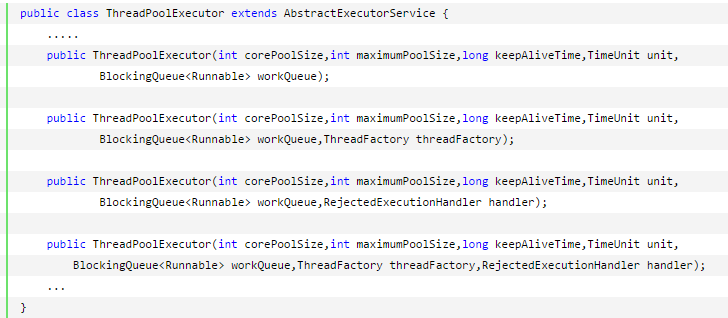
**}**

**}**

# 线程池

线程池类为 java.util.concurrent.ThreadPoolExecutor，在ThreadPoolExecutor类中提供了四个构造方法：：

<http://www.cnblogs.com/dolphin0520/p/3932921.html>



corePoolSize（线程池的基本大小）：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，如果当前线程数小于corePoolSize，哪怕此时有空闲线程，依旧会创建一个新的线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程，当提交的任务数大于maximumPoolSize + workQueue.size的时候，新提交的任务将被拒绝策略处理；

keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；

unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

TimeUnit.DAYS; //天

TimeUnit.HOURS; //小时

TimeUnit.MINUTES; //分钟

TimeUnit.SECONDS; //秒

TimeUnit.MILLISECONDS; //毫秒

TimeUnit.MICROSECONDS; //微妙

TimeUnit.NANOSECONDS; //纳秒

workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。

threadFactory：线程工厂，主要用来创建线程；

handler：表示当拒绝处理任务时的策略，有以下四种取值：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

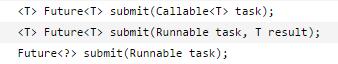
ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

在ThreadPoolExecutor类中有几个非常重要的方法：

execute()方法实际上是Executor中声明的方法，在ThreadPoolExecutor进行了具体的实现，这个方法是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。

void execute(Runnable command);

submit()方法是在ExecutorService中声明的方法，在AbstractExecutorService就已经有了具体的实现，在ThreadPoolExecutor中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和execute()方法不同，它能够返回任务执行的结果，去看submit()方法的实现，会发现它实际上还是调用的execute()方法，只不过它利用了Future来获取任务执行结果（Future相关内容将在下一篇讲述）。



shutdown()和shutdownNow()是用来关闭线程池的。

runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性；

下面的几个static final变量表示runState可能的几个取值。

当创建线程池后，初始时，线程池处于RUNNING状态；

如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；如果出现异常则抛出，如sleep的话抛出java.lang.InterruptedException:；

当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

**任务提交给线程池之后的处理策略，这里总结一下主要有4点：**

**关键是理解：任务先交给**corePoolSize内的线程处理，然后大于corePoolSize就交给缓存队列，如果缓存队列满了，就继续新建线程，如果线程数量大于maximumPoolSize，则采取拒绝策略。

如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；

如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

使用实例：

**import** java**.**util**.**concurrent**.**ArrayBlockingQueue**;**

**import** java**.**util**.**concurrent**.**ExecutorService**;**

**import** java**.**util**.**concurrent**.**Executors**;**

**import** java**.**util**.**concurrent**.**RejectedExecutionHandler**;**

**import** java**.**util**.**concurrent**.**ThreadPoolExecutor**;**

**import** java**.**util**.**concurrent**.**TimeUnit**;**

public class ProductConsumer **{**

public static void main**(**String**[]** args**)** **throws** Exception **{**

ThreadPoolExecutor threadPoolExecutor **=** **new**

ThreadPoolExecutor**(**2**,** 4**,** 10**,** TimeUnit**.**SECONDS**,**

**new** ArrayBlockingQueue**<>(**2**),** **new** RejectedExecutionHandlerImpl**());**

//submit work to the thread pool

**for(**int i**=**1**;** i**<=**10**;** i**++){**

threadPoolExecutor**.**execute**(new** WorkerThread**(**i**));**

System**.**out**.**println**(**

String**.**format**(**"[monitor] [%d/%d] Active: %d, Completed: %d, Task: %d, isShutdown: %s, isTerminated: %s"**,**

threadPoolExecutor**.**getPoolSize**(),**

threadPoolExecutor**.**getCorePoolSize**(),**

threadPoolExecutor**.**getActiveCount**(),**

threadPoolExecutor**.**getCompletedTaskCount**(),**

threadPoolExecutor**.**getTaskCount**(),**

threadPoolExecutor**.**isShutdown**(),**

threadPoolExecutor**.**isTerminated**()));**

**}**

threadPoolExecutor**.**shutdown**();**

**while** **(!**threadPoolExecutor**.**isTerminated**())** **{**

**}**

**}**

**}**

class RejectedExecutionHandlerImpl **implements** RejectedExecutionHandler **{**

@Override

public void rejectedExecution**(**Runnable r**,** ThreadPoolExecutor executor**)** **{**

System**.**out**.**println**(**r**.**toString**()** **+** " is rejected"**);**

**}**

**}**

class WorkerThread **implements** Runnable **{**

private int taskNum**;**

public WorkerThread**(**int num**)** **{**

**this.**taskNum **=** num**;**

**}**

@Override

public void run**()** **{**

System**.**out**.**println**(**"开始执行 task "**+**taskNum**);**

**try** **{**

Thread**.**currentThread**().**sleep**(**2000**);**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

System**.**out**.**println**(**"task "**+**taskNum**+**" 执行完毕"**);**

**}**

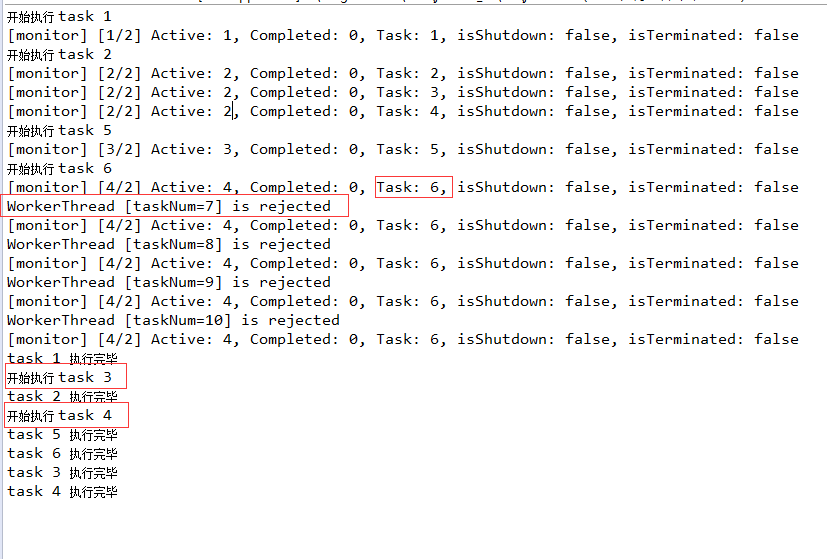
@Override

public String toString**()** **{**

**return** "WorkerThread [taskNum=" **+** taskNum **+** "]"**;**

**}**

**}**



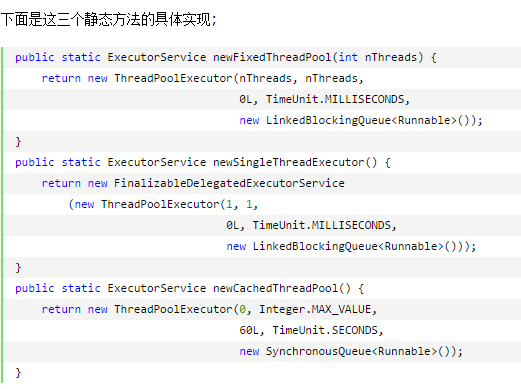
可见，缓存队列为2，corePoolSize=2，maximumPoolSize=4，当提交的任务数量大于缓存队列容量和maximumPoolSize之和的时候，新提交的任务被执行拒绝策略了。并且，最后执行的是放在缓存队列的2个任务。

**不过在java doc中，并不提倡我们直接使用ThreadPoolExecutor，而是使用Executors类中提供的几个静态方法来创建线程池：**

Executors.newCachedThreadPool(); //创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE

Executors.newSingleThreadExecutor(); //创建容量为1的缓冲池

Executors.newFixedThreadPool(int); //创建固定容量大小的缓冲池



从它们的具体实现来看，它们实际上也是调用了ThreadPoolExecutor，只不过参数都已配置好了。

　　newFixedThreadPool创建的线程池corePoolSize和maximumPoolSize值是相等的，它使用的LinkedBlockingQueue；

　　newSingleThreadExecutor将corePoolSize和maximumPoolSize都设置为1，也使用的LinkedBlockingQueue；

　　newCachedThreadPool将corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX\_VALUE，使用的SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程。

　　实际中，如果Executors提供的三个静态方法能满足要求，就尽量使用它提供的三个方法，因为自己去手动配置ThreadPoolExecutor的参数有点麻烦，要根据实际任务的类型和数量来进行配置。

　　另外，如果ThreadPoolExecutor达不到要求，可以自己继承ThreadPoolExecutor类进行重写。

ExecutorService executor **=** Executors**.**newCachedThreadPool**();**

ExecutorService fixExe **=** Executors**.**newFixedThreadPool**(**5**);**

ExecutorService singleExe **=** Executors**.**newSingleThreadExecutor**();**

注意：ArrayBlockingQueue的大小是设定好的，而LinkedBlockingQueue的大小是无限制的。

**import** java**.**util**.\*;**

**import** java**.**util**.**concurrent**.**ThreadPoolExecutor**;**

public class Main2 **{**

public static void main**(**String**[]** args**)** **{**

// TODO Auto-generated method stub

// 创建3个线程的线程池

ThreadPool t **=** ThreadPool**.**getThreadPool**(**3**);**

t**.**execute**(new** Runnable**[]** **{** **new** Task**(),** **new** Task**(),** **new** Task**()** **});**

t**.**execute**(new** Runnable**[]** **{** **new** Task**(),** **new** Task**()});**

System**.**out**.**println**(**t**);**

t**.**destroy**();**// 所有线程都执行完成才destory

System**.**out**.**println**(**t**);**

**}**

static int fun**(**int m**)** **{**

**if** **(**m **==** 0**)** **return** 0**;**

**if** **((**m **&** 1**)** **==** 1**)** **{**

System**.**out**.**println**(**"m\*fun(m-1)"**+**m**);**

**return** m**\***fun**(**m**-**1**);**

**}** **else** **{**

System**.**out**.**println**(**"m+fun(m-1)"**+**m**);**

**return** m**+**fun**(**m**-**1**);**

**}**

**}**

// 任务类

static class Task **implements** Runnable **{**

private static volatile int i **=** 1**;**

@Override

public void run**()** **{**// 执行任务

System**.**out**.**println**(**"任务 " **+** **(**i**++)** **+** " 完成"**);**

**}**

**}**

**}**

/\*\*

\* 线程池类，线程管理器：创建线程，执行任务，销毁线程，获取线程基本信息

\*/

final class ThreadPool **{**

// 线程池中默认线程的个数为5

private static int worker\_num **=** 5**;**

// 工作线程

private WorkThread**[]** workThrads**;**

// 处理的任务

private static volatile int finished\_task **=** 0**;**

// 任务队列，作为一个缓冲,List线程不安全

private List**<**Runnable**>** taskQueue **=** **new** LinkedList**<**Runnable**>();**

private static ThreadPool threadPool**;**

// 创建具有默认线程个数的线程池

private ThreadPool**()** **{**

**this(**5**);**

**}**

// 创建线程池,worker\_num为线程池中工作线程的个数

private ThreadPool**(**int worker\_num**)** **{**

ThreadPool**.**worker\_num **=** worker\_num**;**

workThrads **=** **new** WorkThread**[**worker\_num**];**

**for** **(**int i **=** 0**;** i **<** worker\_num**;** i**++)** **{**

workThrads**[**i**]** **=** **new** WorkThread**();**

workThrads**[**i**].**start**();**// 开启线程池中的线程

**}**

**}**

// 单态模式，获得一个默认线程个数的线程池

public static ThreadPool getThreadPool**()** **{**

**return** getThreadPool**(**ThreadPool**.**worker\_num**);**

**}**

// 单态模式，获得一个指定线程个数的线程池,worker\_num(>0)为线程池中工作线程的个数

// worker\_num<=0创建默认的工作线程个数

public static ThreadPool getThreadPool**(**int worker\_num1**)** **{**

**if** **(**worker\_num1 **<=** 0**)**

worker\_num1 **=** ThreadPool**.**worker\_num**;**

**if** **(**threadPool **==** **null)**

threadPool **=** **new** ThreadPool**(**worker\_num1**);**

**return** threadPool**;**

**}**

// 执行任务,其实只是把任务加入任务队列，什么时候执行有线程池管理器觉定

public void execute**(**Runnable task**)** **{**

synchronized **(**taskQueue**)** **{**

taskQueue**.**add**(**task**);**

taskQueue**.**notify**();**

**}**

**}**

// 批量执行任务,其实只是把任务加入任务队列，什么时候执行有线程池管理器觉定

public void execute**(**Runnable**[]** task**)** **{**

synchronized **(**taskQueue**)** **{**

**for** **(**Runnable t **:** task**)**

taskQueue**.**add**(**t**);**

taskQueue**.**notify**();**

**}**

**}**

// 批量执行任务,其实只是把任务加入任务队列，什么时候执行有线程池管理器觉定

public void execute**(**List**<**Runnable**>** task**)** **{**

synchronized **(**taskQueue**)** **{**

**for** **(**Runnable t **:** task**)**

taskQueue**.**add**(**t**);**

taskQueue**.**notify**();**

**}**

**}**

// 销毁线程池,该方法保证在所有任务都完成的情况下才销毁所有线程，否则等待任务完成才销毁

public void destroy**()** **{**

**while** **(!**taskQueue**.**isEmpty**())** **{**// 如果还有任务没执行完成，就先睡会吧

**try** **{**

Thread**.**sleep**(**10**);**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

**}**

// 工作线程停止工作，且置为null

**for** **(**int i **=** 0**;** i **<** worker\_num**;** i**++)** **{**

workThrads**[**i**].**stopWorker**();**

workThrads**[**i**]** **=** **null;**

**}**

threadPool**=null;**

taskQueue**.**clear**();**// 清空任务队列

**}**

// 返回工作线程的个数

public int getWorkThreadNumber**()** **{**

**return** worker\_num**;**

**}**

// 返回已完成任务的个数,这里的已完成是只出了任务队列的任务个数，可能该任务并没有实际执行完成

public int getFinishedTasknumber**()** **{**

**return** finished\_task**;**

**}**

// 返回任务队列的长度，即还没处理的任务个数

public int getWaitTasknumber**()** **{**

**return** taskQueue**.**size**();**

**}**

// 覆盖toString方法，返回线程池信息：工作线程个数和已完成任务个数

@Override

public String toString**()** **{**

**return** "WorkThread number:" **+** worker\_num **+** " finished task number:"

**+** finished\_task **+** " wait task number:" **+** getWaitTasknumber**();**

**}**

/\*\*

\* 内部类，工作线程

\*/

private class WorkThread **extends** Thread **{**

// 该工作线程是否有效，用于结束该工作线程

private boolean isRunning **=** **true;**

/\*

\* 关键所在啊，如果任务队列不空，则取出任务执行，若任务队列空，则等待

\*/

@Override

public void run**()** **{**

Runnable r **=** **null;**

**while** **(**isRunning**)** **{**// 注意，若线程无效则自然结束run方法，该线程就没用了

synchronized **(**taskQueue**)** **{**

**while** **(**isRunning **&&** taskQueue**.**isEmpty**())** **{**// 队列为空

**try** **{**

taskQueue**.**wait**(**20**);**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

**}**

**if** **(!**taskQueue**.**isEmpty**())**

r **=** taskQueue**.**remove**(**0**);**// 取出任务

**}**

**if** **(**r **!=** **null)** **{**

r**.**run**();**// 执行任务

**}**

finished\_task**++;**

r **=** **null;**

**}**

**}**

// 停止工作，让该线程自然执行完run方法，自然结束

public void stopWorker**()** **{**

isRunning **=** **false;**

**}**

**}**

**}**

# java反射

在java中， \*\*.class文件记录了很多信息，java在需要使用到某个类时才会将\*\*.class文档载入，并在JVM中产生一个java.lang.Class的实例代表该文档，从Class的实例开始，就可以获得类的很多类型。.class文档反映了类的基本信息，因而从Class等API获取类信息的方式就成为反射。

主要是通过Class以及java.lang.reflect包下的类获取特定类的Class对象，构建实例，获取类的成员方法和成员变量。

java反射机制是指在运行状态中，对应任意的一个类，都能通过反射知道这个类的所有属性和方法，并调用它的任意方法和属性，这种动态获取类的信息和动态调用类的方法的功能成为java的反射机制。

Class类中getFields、getMethods、getConstructors方法将返回该类中的public（域、方法、构造方法）的数组，以及其父类中的公有成员。

getDeclaredFields、getDeclaredMethods、getDeclaredConstructors将返回该类中声明的所有 域、方法、构造器的数组，包括公有，私有，受保护的成员，但不包括父类成员。

**import** java**.**lang**.**reflect**.**Field**;**

**import** java**.**lang**.**reflect**.**Method**;**

**import** java**.**util**.\*;**

public class MainDemo **{**

public static void main**(**String**[]** args**)** **throws** Exception **{**

//Class的3种获取方式

//Class<String> clz = Main.class;//编译错误

//Type mismatch: cannot convert from Class<Main> to Class<String>

Class clz **=** Student**.**class**;**

Class**<?>** clz2 **=** Class**.**forName**(**"test.Student"**);**

Student stu **=** **new** Student**();**

Class clz3 **=** stu**.**getClass**();**

//使用Class 构造类的，建立对象

Object object **=** clz**.**newInstance**();**

**((**Student**)**object**).**f**();**

Object object2 **=** clz**.**getConstructor**(**String**.**class**).**newInstance**(**"lucy"**);**

**((**Student**)**object2**).**f**();**

Object object3 **=**

clz**.**getConstructor**(**String**.**class**,** Integer**.**TYPE**)**//String类型和int类型

**.**newInstance**(**"hanmei"**,**18**);**

**((**Student**)**object3**).**f**();**

//建立数组

Object**[]** objects **=** **(**Object**[])** java**.**lang**.**reflect**.**Array**.**newInstance**(**clz**,** 10**);**

objects**[**0**]** **=** **new** Student**();**

**((**Student**)**objects**[**0**]).**f**();**

//操作对象方法

Class clzM **=** Class**.**forName**(**"test.Student"**);**

Object obj **=** clzM**.**newInstance**();** //1.生成获取对象的实例

Method methodF **=** clzM**.**getMethod**(**"f"**);**//2.不带参数的方法

methodF**.**invoke**(**obj**);**

//3.带参数的方法

Method methodP **=** clzM**.**getMethod**(**"p"**,** String**.**class**);**

methodP**.**invoke**(**obj**,** "method test"**);**

//4.带返回值的方法

Method methodSum **=** clzM**.**getMethod**(**"sum"**,** Integer**.**TYPE**,** Integer**.**TYPE**);**

int sum **=** **(**int**)** methodSum**.**invoke**(**obj**,** 3**,** 4**);**

System**.**out**.**println**(**sum**);**

//5.获取private或者protected方法

Method methodPrivete **=** clzM**.**getDeclaredMethod**(**"pp"**);**

methodPrivete**.**setAccessible**(true);**//设置可以获取，不可缺少

methodPrivete**.**invoke**(**obj**);**

//获取成员变量 1.获取公有成员变量

Field fieldName **=** clzM**.**getField**(**"name"**);**

fieldName**.**set**(**obj**,** "lilei"**);**

System**.**out**.**println**(**obj**.**toString**());**//显示 已经更改

//2.获取私有成员变量

Field fieldAge **=** clzM**.**getDeclaredField**(**"age"**);**

fieldAge**.**setAccessible**(true);**//设置可以获取

fieldAge**.**set**(**obj**,** 19**);**

System**.**out**.**println**(**obj**);**

**}**

**}**

class Student **{**

public String name**;**

private int age**;**

public void f**()** **{**

System**.**out**.**println**(**"f() "**+**name**+**" "**+**age**);**

**}**

public void p**(**String out**)** **{**

System**.**out**.**println**(**"p() "**+**out**);**

**}**

public int sum**(**int a**,** int b**)** **{**

**return** a**+**b**;**

**}**

private void pp**()** **{**

System**.**out**.**println**(**"private pp()"**);**

**}**

public Student**()** **{**

**}**

public Student**(**String n**)** **{**

name **=** n**;**

**}**

public Student**(**String n**,** int a**)** **{**

name **=** n**;**

age **=** a**;**

**}**

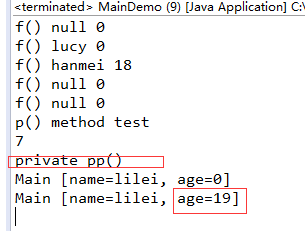
@Override

public String toString**()** **{**

**return** "Main [name=" **+** name **+** ", age=" **+** age **+** "]"**;**

**}**

**}**



代理

在反射API中有个Proxy类，可以动态建立接口的操作对象。

先看一个例子，实现执行某些方法时需要日志记录的功能，代码如下：

package review**;**

**import** java**.**util**.**logging**.**Level**;**

**import** java**.**util**.**logging**.**Logger**;**

public class HelloSpeaker **{**

public static void main**(**String**[]** args**)** **{**

**new** HelloSpeaker**().**hello**(**"jack"**);**

**}**

public void hello**(**String name**)** **{**

Logger**.**getLogger**(**HelloSpeaker**.**class**.**getName**())**

**.**log**(**Level**.**INFO**,** "hello() start……"**);**

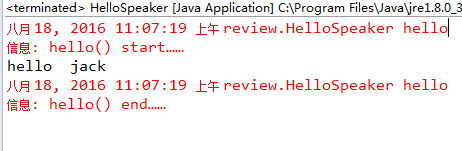
System**.**out**.**printf**(**"hello %s%n"**,** name**);**

Logger**.**getLogger**(**HelloSpeaker**.**class**.**getName**())**

**.**log**(**Level**.**INFO**,** "hello() end……"**);**

**}**

**}**



希望将打印日志和方法的本职工作解耦。

**静态代理**

在静态代理中，代理对象和被代理对象必须实现同一个接口，在代理对象中实现日志服务，必要时调用被代理对象，这样被代理对象中就可以仅仅实现自己的本职业务功能。

代码如下：

package review**;**

public interface Hello **{**

void hello**(**String name**);**//默认是 public abstract修饰

**}**

//

package review**;**

**import** java**.**util**.**logging**.**Level**;**

**import** java**.**util**.**logging**.**Logger**;**

public class HelloSpeaker **implements** Hello **{**

public static void main**(**String**[]** args**)** **{**

**new** HelloSpeaker**().**hello**(**"jack"**);**

**}**

@Override

public void hello**(**String name**)** **{**

System**.**out**.**printf**(**"hello %s%n"**,** name**);**

**}**

**}**

//

package review**;**

**import** java**.**util**.**logging**.**Level**;**

**import** java**.**util**.**logging**.**Logger**;**

public class HelloProxy **implements** Hello**{**

public static void main**(**String**[]** args**)** **{**

**new** HelloProxy**(new** HelloSpeaker**()).**hello**(**"lucy"**);**

**}**

private Hello helloObj**;**

public HelloProxy**(**Hello h**)** **{**

helloObj **=** h**;**

**}**

@Override

public void hello**(**String name**)** **{**

log**(**"hello() start……"**);**

helloObj**.**hello**(**name**);**

log**(**"hello() end……"**);**

**}**

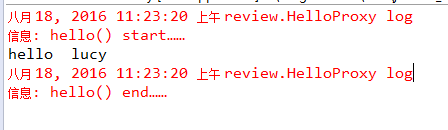
private void log**(**String text**)** **{**

Logger**.**getLogger**(**HelloProxy**.**class**.**getName**())**

**.**log**(**Level**.**INFO**,** text**);**

**}**

**}**



其实静态代理也没有使用Proxy类，大概是一种设计模式吧，并且，静态代理需要给每一个接口建立相应的代理类，如果接口很多，那么也就需要定义很多的代理对象，这样维护起来也很麻烦。

# 动态代理

反射API提供动态代理的相关类，可以不必为特定的接口操作特定的代理对象。使用动态代理机制，可以使用一个处理者（handler）代理多个接口的操作对象。

被代理的对象需要是实现某个接口。

package review**;**

public interface Hello **{**

void hello**(**String name**);**//默认是 public abstract修饰

**}**

//

package review**;**

**import** java**.**util**.**logging**.**Level**;**

**import** java**.**util**.**logging**.**Logger**;**

public class HelloSpeaker **implements** Hello **{**

public static void main**(**String**[]** args**)** **{**

**new** HelloSpeaker**().**hello**(**"jack"**);**

**}**

@Override

public void hello**(**String name**)** **{**

System**.**out**.**printf**(**"hello %s%n"**,** name**);**

**}**

**}**

//

package review**;**

public interface Bye **{**

int bye**(**int a**,** int b**);**

**}**

//

package review**;**

public class ByeSpeaker **implements** Bye**{**

@Override

public int bye**(**int a**,** int b**)** **{**

System**.**out**.**println**(**"bye sum="**+**a **+** " + "**+**b**);**

**return** a**+**b**;**

**}**

**}**

//

package review**;**

**import** java**.**lang**.**reflect**.**InvocationHandler**;**

**import** java**.**lang**.**reflect**.**Method**;**

**import** java**.**lang**.**reflect**.**Proxy**;**

**import** java**.**util**.**logging**.**Level**;**

**import** java**.**util**.**logging**.**Logger**;**

public class LoggingHandler **implements** InvocationHandler**{**

private Object target**;**

//使用bind来绑定被代理对象

public Object bind**(**Object t**)** **{**

target **=** t**;**

//建立代理对象，指定类加载器，要代理的接口，以及接口上定义方法被调用时的处理者

//生成代理对象，代理对象会操作指定要代理的接口

**return** Proxy**.**newProxyInstance**(**

target**.**getClass**().**getClassLoader**(),**

target**.**getClass**().**getInterfaces**(),** **this);**

**}**

//每次操作代理对象时，会调用处理器（InvocationHandler实例）的invoke()方法

//并传入代理对象，被调用方法的Method实例，参数

//可以在invoke方法中实现日志，利用被代理的对象，被调用的方法Method实例与参数值来实现被代理对象的职责。

@Override

public Object invoke**(**Object proxy**,** Method method**,** Object**[]** args**)** **throws** Throwable **{**

Object result **=** **null;**

**try** **{**

log**(**String**.**format**(**"%s() start……"**,** method**.**getName**()));**

result **=** method**.**invoke**(**target**,** args**);**

log**(**String**.**format**(**"%s() end……"**,** method**.**getName**()));**

**}** **catch** **(**Exception e**)** **{**

log**(**e**.**toString**());**

**}**

**return** result**;**

**}**

private void log**(**String text**)** **{**

Logger**.**getLogger**(**LoggingHandler**.**class**.**getName**())**

**.**log**(**Level**.**INFO**,** text**);**

**}**

**}**

//

package review**;**

**import** java**.**lang**.**reflect**.**Proxy**;**

public class ProxyDemo **{**

public static void main**(**String**[]** args**)** **{**

LoggingHandler loggingHandler **=** **new** LoggingHandler**();**

//生成代理对象 Proxy.newProxyInstance

Hello helloProxy **=** **(**Hello**)** loggingHandler**.**bind**(new** HelloSpeaker**());**

//每次操作代理对象时，会调用处理器（InvocationHandler实例）的invoke()方法

//并传入代理对象，被调用方法的Method实例，参数

//可以在invoke方法中实现日志，利用被代理的对象，被调用的方法Method实例与参数值来实现被代理对象的职责。

helloProxy**.**hello**(**"hanmei"**);**

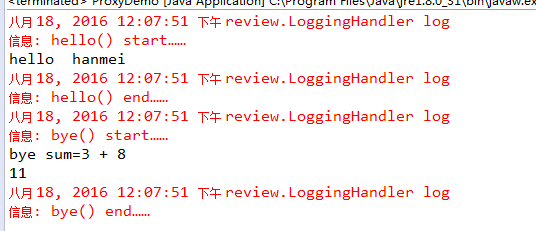
Bye byeProxy **=** **(**Bye**)** loggingHandler**.**bind**(new** ByeSpeaker**());**

int c **=** byeProxy**.**bye**(**3**,** 8**);**

System**.**out**.**println**(**c**);**

**}**

**}**



# 类加载器

类加载器的实际的职责是载入.class文档，JDK本身有自己默认的类加载器，用户也可以建立自己的类加载器加入现有的加载器层级。

一般执行 java XXX 命令后，java执行程序会寻找JRE的安装路径，然后寻找JVM，接着启动JVM并执行初始化动作，然后产生BootStrap Loader，BootStrap Loader会产生Extender Loader，并将Extended Loader的父加载器设定为BootStrap Loader，接着BootStrap Loader又产生System Loader，并将System Loader的父加载器设定为Extended Loader。

Bootstrap Loader是由C写成，在oracle/sun JDK中，实际显示null，Extended Loader由Java写成，在oracle/sun JDK中对应ExtClassLoader，System Loader由Java写成，对应AppClassLoader。

package test**;**

**import** java**.**lang**.**reflect**.**Array**;**

**import** java**.**lang**.**reflect**.**Field**;**

**import** java**.**lang**.**reflect**.**Method**;**

**import** java**.**net**.**URL**;**

**import** java**.**net**.**URLClassLoader**;**

**import** java**.**util**.\*;**

public class MainDemo **{**

public static void main**(**String**[]** args**)** **throws** Exception **{**

Class clz **=** MainDemo**.**class**;**

//会搜索系统参数java.class.path中指定位置的类，默认是当前工作路径下的.class文档

ClassLoader loader **=** clz**.**getClassLoader**();**

System**.**out**.**println**(**loader**);**//SystemLoader

//搜索系统参数 sun.boot.class.path中指定位置的类

loader **=** loader**.**getParent**();**//ExtendedLoader

System**.**out**.**println**(**loader**);**

//搜索系统参数 java.ext.dirs中指定位置的类

loader **=** loader**.**getParent**();**//BootstrapLoader

System**.**out**.**println**(**loader**);**

//不过在oracle的jre下分别显示null ExtClassLoader AppClassLoader

//在加载类时，每个类的加载器会先将加载类的任务交给父加载器

//如果父加载器找不到，才由自己加载。所以一个类的加载过程，会以BootstrapLoader->ExtendedLoader

// ->SystemLoader的顺序寻找类，如果都没有找到，则抛出NoClassDefFoundError

//如果程序在运行的过程中打算动态从其他路径加载类，就需要产生新的类加载器

URL url **=** **new** URL**(**"file:/c:/java/"**);**//指定类加载的搜索路径

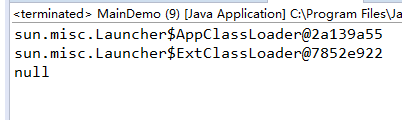
ClassLoader myLoader **=** **new** URLClassLoader**(new** URL**[]{**url**});**

//由同一个类加载器载入的.class文档，只会有一个Class实例，如果同一个.class文档由2个不同的类加载器

//载入，则会有2份不同的Class实例。

**}**

**}**



# 正则表达式

java正则表达式字符串中的 \ 都需要替换为 \\。

正则表达式中的元字符具有特殊的意义，如^ $ \* ( ) + = { } [ ] | \ : . ?等。如果要匹配这些元字符本身，则需要在其前面加转义符合\， 如 \^。

如果多个字符要用或的方式表示，使用[ ]，如[XYZ]，表示匹配X或Y或Z。若是XYZ，则表示匹配字符串“XYZ”。

如果要匹配 \ ,则其正则表达式为”\\\\”,匹配|| 为”\\|\\|”。

[abc]匹配a,b,c中的任一字符

[^abc]，匹配a,b,c以外的任一字符

[a-zA-z]匹配 a-z或者A-Z的任一字符，并集，等同于[a-z[A-Z]]

[a-z&&[bcd]] 匹配a-z并且是bcd中的任一字符，交集关系。 等同于 [bcd]，取2者的交集部分。

. 通配符，匹配任一字符，除换行 \n 以外。

\d 匹配任一数字 0-9

\D 任一非数字 [^0-9]

\s 任一空格符 [\t\n\x0B\r\f]

\S 任一非空格符

\w 任一ASCII字符，即[0-9a-zA-Z]

\W 任一非ASCII字符

X？ X项目出现一次或0次 <= 1, \* 表示出现 >=0 次，+表示 >= 1 次。

X{n} X 出现n次

X{n,m} X出现 n <= t <=m次，大于等于n次，不超过m次

X{n,} X出现大于n次。

边界比较常用：

^ 一行开头 $ 一行结尾 \b单词边界 \B非单词边界

public static void main**(**String**[]** args**)** **throws** Exception **{**

Pattern pattern **=** Pattern**.**compile**(**"([123]|[123]|[45])+"**);**

Matcher matcher **=** pattern**.**matcher**(**"12345"**);**

System**.**out**.**println**(**matcher**.**matches**());**

**}**

# try catch finally return顺序

结论一：

return语句并不是函数的最终出口，如果有finally语句，

这在return之后还会执行finally（return的值会暂存在栈里面，等待finally执行后再返回）

结论二：

finally里面不建议放return语句，根据需要，return语句可以放在try和catch里面和函数的最后。可行的做法

package test**;**

**import** java**.**lang**.**reflect**.**Array**;**

**import** java**.**lang**.**reflect**.**Field**;**

**import** java**.**lang**.**reflect**.**Method**;**

**import** java**.**net**.**URL**;**

**import** java**.**net**.**URLClassLoader**;**

**import** java**.**util**.\*;**

**import** java**.**util**.**regex**.**Matcher**;**

**import** java**.**util**.**regex**.**Pattern**;**

public class MainDemo **{**

/\*结论一：

return语句并不是函数的最终出口，如果有finally语句，

这在return之后还会执行finally（return的值会暂存在栈里面，等待finally执行后再返回）

结论二：

finally里面不建议放return语句，根据需要，return语句可以放在try和catch里面和函数的最后。可行的做法\*/

public static void main**(**String**[]** args**)** **throws** Exception **{**

System**.**out**.**println**(**NoException**());**

System**.**out**.**println**();**

System**.**out**.**println**(**NoException1**());**

System**.**out**.**println**();**

System**.**out**.**println**(**WithException**());**

System**.**out**.**println**();**

System**.**out**.**println**(**WithException1**());**

**}**

/\*

执行try块，执行到return语句时，先执行return的语句，--i，但是不返回到main方法，

执行finally块，遇到finally块中的return语句，执行--i,并将值返回到main方法，

这里就不会再回去返回try块中计算得到的值。

结论：try-catch-finally都有return语句时，没有异常时，返回值是finally中的return返回的。

\*/

public static int NoException**(){**

int i**=**10**;**

**try{**

System**.**out**.**println**(**"i in try block is："**+**i**);**

**return** **--**i**;**//执行--i,但不返回

**}**

**catch(**Exception e**){**

**--**i**;**

System**.**out**.**println**(**"i in catch - form try block is："**+**i**);**

**return** **--**i**;**

**}**

**finally{**

System**.**out**.**println**(**"i in finally - from try or catch block is："**+**i**);**

**return** **--**i**;**//从这里返回值

**}**

**}**

/\*

\* try中执行完return的语句后，不返回，执行finally块，

\* finally块执行结束后，返回到try块中，返回i在try块中最后的值。

\* 结论：try-catch都有return语句时，没有异常时，返回值是try中的return返回的。

\* \*/

public static int NoException1**(){**//最终返回9 而不是1或者0

int i**=**10**;**

**try{**

System**.**out**.**println**(**"i in try block is："**+**i**);**

**return** **--**i**;**

**}**

**catch(**Exception e**){**

**--**i**;**

System**.**out**.**println**(**"i in catch - form try block is："**+**i**);**

**return** **--**i**;**

**}**

**finally{**

System**.**out**.**println**(**"i in finally - from try or catch block is："**+**i**);**

i **=** 1**;**

System**.**out**.**println**(**"i in finally block is："**+**i**);**

//return --i;

**}**

**}**

/\*

\* 抛出异常后，执行catch块，在catch块的return的--i执行完后，

\* 并不直接返回而是执行finally，因finally中有return语句，所以，执行，返回结果6。

\*/

public static int WithException**(){**

int i**=**10**;**

**try{**

System**.**out**.**println**(**"i in try block is："**+**i**);**

i **=** i**/**0**;**

**return** **--**i**;**

**}**

**catch(**Exception e**){**

System**.**out**.**println**(**"i in catch - form try block is："**+**i**);**

**--**i**;**

System**.**out**.**println**(**"i in catch block is："**+**i**);**

**return** **--**i**;**

**}**

**finally{**

System**.**out**.**println**(**"i in finally - from try or catch block is--"**+**i**);**

**--**i**;**

System**.**out**.**println**(**"i in finally block is--"**+**i**);**

**return** **--**i**;**

**}**

**}**

/\*

\* 抛出异常后，执行catch块，执行完finally语句后，依旧返回catch中的执行return语句后的值，

\* 而不是finally中修改的值。

\*/

public static int WithException1**(){**//返回9

int i**=**10**;**

**try{**

System**.**out**.**println**(**"i in try block is："**+**i**);**

i**=**i**/**0**;**

**return** **--**i**;**

**}catch(**Exception e**){**

System**.**out**.**println**(**"i in catch - form try block is："**+**i**);**

**return** **--**i**;** //返回9

**}finally{**

System**.**out**.**println**(**"i in finally - from try or catch block is："**+**i**);**

**--**i**;**

System**.**out**.**println**(**"i in finally block is："**+**i**);**

//return i;

**}**

**}**

/\*

static int test() {

try {

} catch (Exception e) {

// TODO: handle exception

} finally {

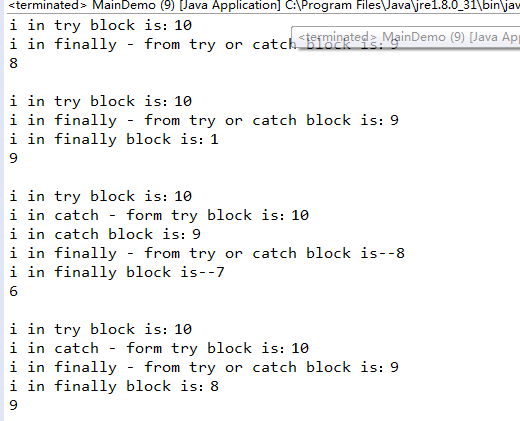
return 1;

}

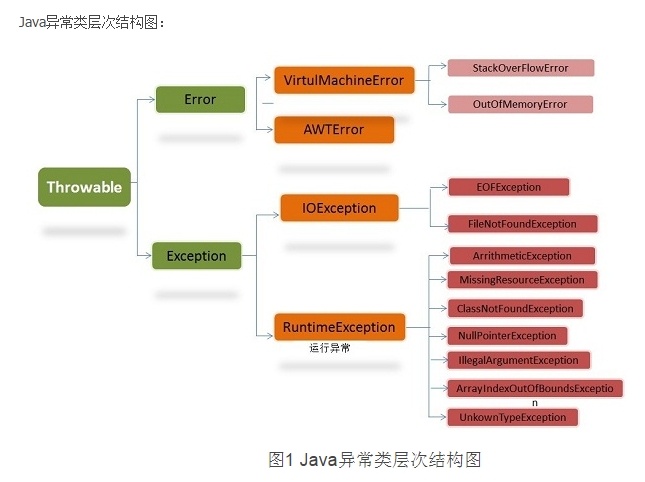
return 0;//finally有return时，编译器报错

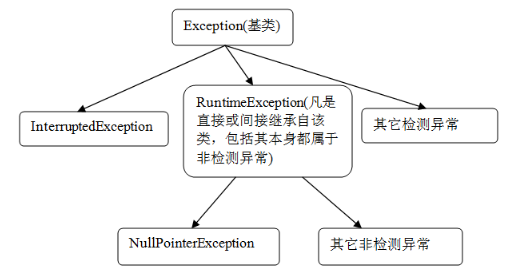
}\*/

**}**



异常的结构：

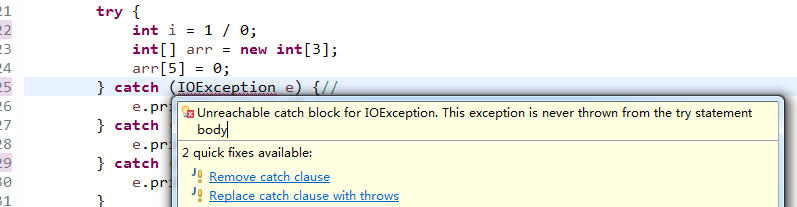




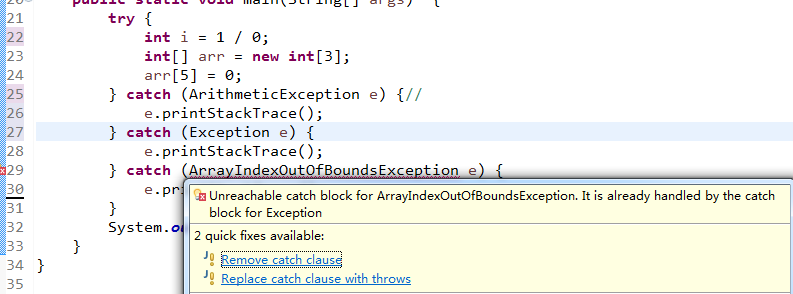
异常分为受检异常（编译器检查）和非受检异常（运行时异常），受检异常一定需要使用try catch或者throws异常声明，否则编译报错。

catch中的一些注意事项：

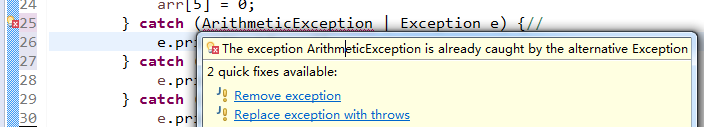
1. 在没有发生受检异常的情况下catch受检异常会编译报错



1. 使用多个catch捕获异常时，如果父类异常对象在其子类异常对象前被捕获，则子类永远无法被捕获，编译报错

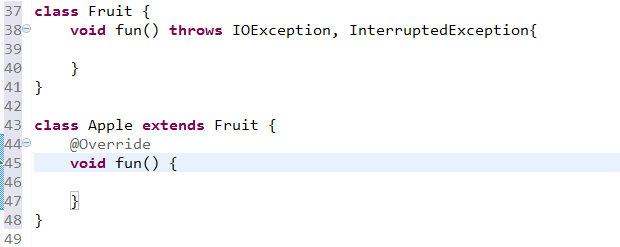


1. 使用多重捕捉的语法时候，catch括号中的异常不能有继承关系。

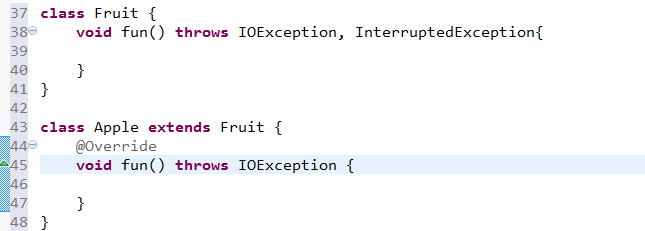


在子类重写父类方法时，如果父类方法throws了异常，则可以如下处理：

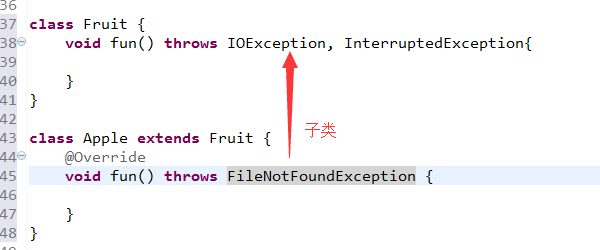
1. 子类方法不声明任何异常



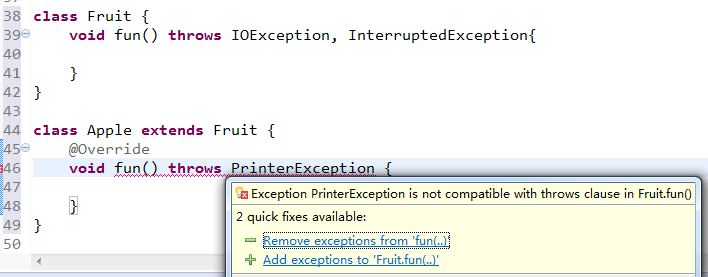
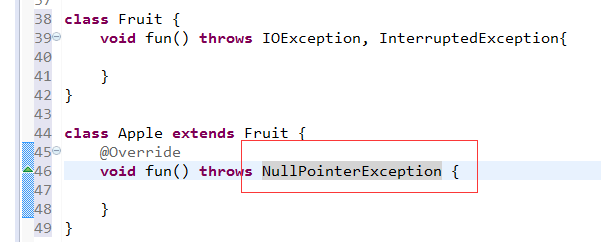
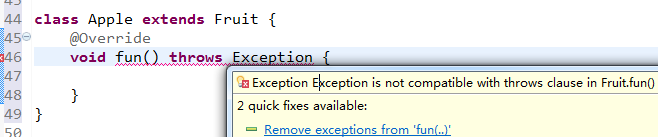
1. 子类方法声明父类方法的一些异常



1. 子类方法声明父类方法中异常的子类

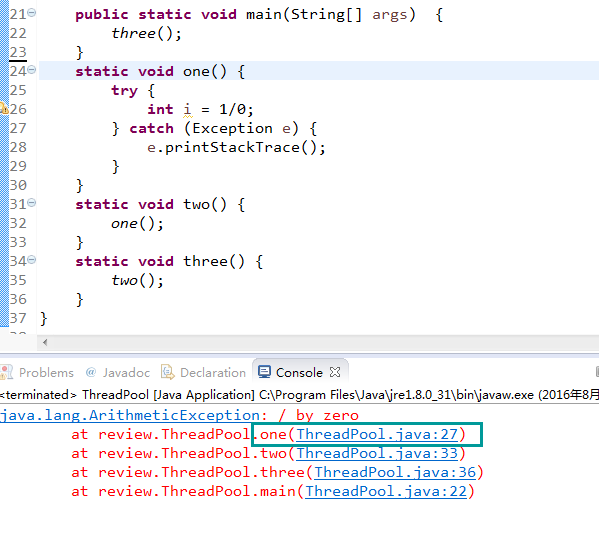


注意：不要声明父类方法中未声明的异常（前提该异常是受检异常，非受检异常无所谓）或者父类方法中异常的父类异常。

打印异常的堆栈信息：

e.printStackTrace()，最顶层是异常的根源，以下则是调用方法的顺序。

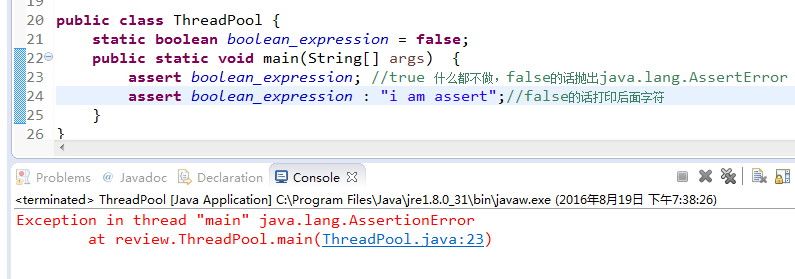


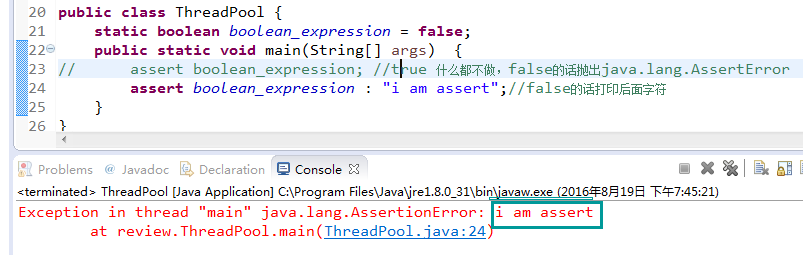
# assert

asset，断言检查：是用来检查程序执行的某个时间点或者某个情况下，是不是处于特定的状态，如果不是，则是严重的错误，需要特别关注。

默认情况下assert机制是关闭的，需要在执行java指令时指定 –ea 。

assert不能当做if子类的判断语句使用，它不是程序的一部分。





# 线程和进程

<http://www.cnblogs.com/obama/archive/2013/04/12/3016509.html>

简而言之,一个程序至少有一个进程,一个进程至少有一个线程.

进程：

是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。

进程和线程的区别？

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。**但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。**

**1) 简而言之,一个程序至少有一个进程,一个进程至少有一个线程.**

2) 线程的划分尺度小于进程，使得多线程程序的并发性高。

3) 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4) 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。**但是线程不能够独立执行，**必须依存在应用程序中，由应用程序提供多个线程执行控制。

5) 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。**这就是进程和线程的重要区别。**

1.线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位；

2.一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线；

3.进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段、数据集、堆等)及一些进程级的资源(如打开文件和信号)，某进程内的线程在其它进程不可见；

4.调度和切换：线程上下文切换比进程上下文切换要快得多。

a.进程是资源分配的基本单位，线程是cpu调度，或者说是程序执行的最小单位；一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线；。在Mac、Windows NT等采用微内核结构的操作系统中，进程的功能发生了变化：它只是资源分配的单位，而不再是调度运行的单位。在微内核系统中，真正调度运行的基本单位是线程。因此，实现并发功能的单位是线程。

　　b.进程有独立的地址空间，比如在linux下面启动一个新的进程，系统必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种非常昂贵的多任务工作方式。而运行一个进程中的线程，它们之间共享大部分数据，使用相同的地址空间，因此启动一个线程，切换一个线程远比进程操作要快，花费也要小得多。当然，线程是拥有自己的局部变量和堆栈（注意不是堆）的，比如在windows中用[\_beginthreadex](http://blog.csdn.net/morewindows/article/details/7421759)创建一个新进程就会在调用CreateThread的同时申请一个专属于线程的数据块（\_tiddata)。

　　c.线程之间的通信比较方便。统一进程下的线程共享数据（比如全局变量，静态变量），通过这些数据来通信不仅快捷而且方便，当然如何处理好这些访问的同步与互斥正是编写多线程程序的难点。而进程之间的通信只能通过[进程通信](http://baike.baidu.com/view/549640.htm)的方式进行。

　　d.由b，可以轻易地得到结论：多进程比多线程程序要健壮。一个线程死掉整个进程就死掉了，但是在[保护模式](http://blog.csdn.net/wukaiyu/article/details/1766804)下，一个进程死掉对另一个进程没有直接影响。

e.线程的执行与进程是有区别的。每个独立的线程有有自己的一个程序入口，顺序执行序列和程序的出口，但是线程不能独立执行，必须依附与程序之中，由应用程序提供多个线程的并发控制。

什么是线程安全？（2012年5月百度实习生面试）

　　如果多线程的程序运行结果是可预期的，而且与单线程的程序运行结果一样，那么说明是“线程安全”的。

多线程同步与互斥有几种实现方法？都是什么？（C++）

　　　临界区（CS:critical section）、事件（Event）、互斥量（Mutex）、信号量（semaphores），需要注意的是，临界区是效率最高的，因为基本不需要其他的开销，二内核对象涉及到用户态和内核态的切换，开销较大，另外，关键段、互斥量具有[线程所有权](http://blog.csdn.net/morewindows/article/details/7442639)的概念，因此只可以用于线程之间互斥，而不能用到同步中。只有互斥量能完美解决进程意外终止所造成的“遗弃问题”。

# 死锁

所谓死锁<DeadLock>: 是指两个或两个以上的进程在执行过程中,因争夺资源而造成的一种互相等待的现象,若无外力作用,它们都将无法推进下去.此时称系统处于死锁状态或系统产生了死锁,这些永远在互相等竺的进程称为死锁产生死锁的原因主要是：

　　（1） 因为系统资源不足。

　　（2） 进程运行推进的顺序不合适。

（3） 资源分配不当等。进程。

产生死锁的四个必要条件：

　　（1） 互斥条件：一个资源每次只能被一个进程使用。

　　（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

　　（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

　　（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

　　这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之

　　一不满足，就不会发生死锁。

### 银行家算法

一个银行家如何将一定数目的资金安全地借给若干个客户，使这些客户既能借到钱完成要干的事，同时银行家又能收回全部资金而不至于破产。银行家就像一个操作系统，客户就像运行的进程，银行家的资金就是系统的资源。

银行家算法需要确保以下四点：

1. 当一个顾客对资金的最大需求量不超过银行家现有的资金时就可接纳该顾客；
2. 顾客可以分期贷款, 但贷款的总数不能超过最大需求量；
3. 当银行家现有的资金不能满足顾客尚需的贷款数额时，对顾客的贷款可推迟支付，但总能使顾客在有限的时间里得到贷款；
4. 当顾客得到所需的全部资金后，一定能在有限的时间里归还所有的资金。

避免死锁的办法：

操作系统处理死锁问题的方式有三种：

（1）预防死锁。其原理是通过摒弃死锁的四个必要条件之一来保证进程不会发生死锁，死锁的四个必要条件是：互斥条件、请求与保持条件、进程已占用的资源不可剥夺、存在资源请求的环路。需要注意的是互斥条件是不能摒弃的，因为一旦摒弃互斥条件，有可能使访问临界资源的进程出现执行结果的错误。

（2）避免死锁。避免死锁的经典方式是银行家算法，利用银行家算法避免死锁的基本思想是：一个进程提出资源请求后，系统先进行资源的试分配，然后调用安全性检测算法检测本次的试分配是否使系统处于安全状态，若安全（能找到至少一个进程执行序列，进程不会进入死锁状态）则按试分配方案分配资源，否则不分配资源。

（3）检测并解除死锁。系统周期性检测系统中是否存在死锁的进程，若存在则通过剥夺资源或杀死进程的方式来解除死锁。

如果我们在死锁检查时发现了死锁情况，那么就要努力消除死锁，使系统从死锁状态中恢复过来。消除死锁的几种方式：

1. 最简单、最常用的方法就是进行系统的重新启动，不过这种方法代价很大，它意味着在这之前所有的进程已经完成的计算工作都将付之东流，包括参与死锁的那些进程，以及未参与死锁的进程；

2. 撤消进程，剥夺资源。终止参与死锁的进程，收回它们占有的资源，从而解除死锁。这时又分两种情况：一次性撤消参与死锁的全部进程，剥夺全部资源；或者逐步撤消参与死锁的进程，逐步收回死锁进程占有的资源。一般来说，选择逐步撤消的进程时要按照一定的原则进行，目的是撤消那些代价最小的进程，比如按进程的优先级确定进程的代价；考虑进程运行时的代价和与此进程相关的外部作业的代价等因素；

3. 进程回退策略，即让参与死锁的进程回退到没有发生死锁前某一点处，并由此点处继续执行，以求再次执行时不再发生死锁。虽然这是个较理想的办法，但是操作起来系统开销极大，要有堆栈这样的机构记录进程的每一步变化，以便今后的回退，有时这是无法做到的。

其实即便是商业产品，依然会有很多死锁情况的发生，例如 MySQL 数据库，它也经常容易出现死锁案例。

# [内核态和用户态的区别](http://blog.csdn.net/fatsandwich/article/details/2131707)

<http://my.oschina.net/liubin/blog/27795>

当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）。此时处理器处于特权级最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3级）用户代码中运行。当正在执行用户程序而突然被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。这与处于内核态的进程的状态有些类似。

内核态与用户态是操作系统的两种运行级别,跟intel cpu没有必然的联系, intel cpu提供Ring0-Ring3三种级别的运行模式，Ring0级别最高，Ring3最低。Linux使用了Ring3级别运行用户态，Ring0作为 内核态，没有使用Ring1和Ring2。Ring3状态不能访问Ring0的地址空间，包括代码和数据。Linux进程的4GB地址空间，3G-4G部 分大家是共享的，是内核态的地址空间，这里存放在整个内核的代码和所有的内核模块，以及内核所维护的数据。用户运行一个程序，该程序所创建的进程开始是运 行在用户态的，如果要执行文件操作，网络数据发送等操作，必须通过write，send等系统调用，这些系统调用会调用内核中的代码来完成操作，这时，必 须切换到Ring0，然后进入3GB-4GB中的内核地址空间去执行这些代码完成操作，完成后，切换回Ring3，回到用户态。这样，用户态的程序就不能 随意操作内核地址空间，具有一定的安全保护作用。

至于说保护模式，是说通过内存页表操作等机制，保证进程间的地址空间不会互相冲突，一个进程的操作不会修改另一个进程的地址空间中的数据。

# 进程通信的方式

1. **管道( pipe )**：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。  
   （2）**命名管道 (named pipe)** ： 命名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。  
   （3）**信号量( semophore )** ： 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的**同步手段**。  
   （4）**消息队列( message queue )** ： 消息队列是消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。  
   （5）信号 ( sinal ) ： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。  
   （6）**共享内存( shared memory )** ：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。  
   （7）**套接字( socket )** ： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信

# TCP三次握手、四次挥手

<http://www.seanyxie.com/wireshark%E6%8A%93%E5%8C%85%E5%9B%BE%E8%A7%A3-tcp%E4%B8%89%E6%AC%A1%E6%8F%A1%E6%89%8B%E5%9B%9B%E6%AC%A1%E6%8C%A5%E6%89%8B%E8%AF%A6%E8%A7%A3/>

<http://www.cnblogs.com/TankXiao/archive/2012/10/10/2711777.html>

# 

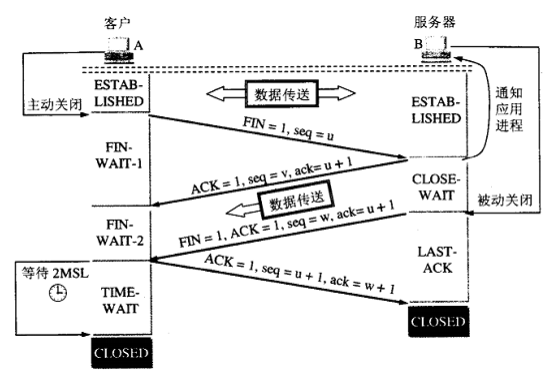
第一次握手：建立连接时发送SYN会选择一个初始序号（ISN），每个连接的ISN都是不同的。客户端发送数据包(SYN=1，seq=x)到服务器，并进入SYN\_SEND状态，等待服务器确认；

第二次握手：服务器收到数据包包，必须确认客户的SYN（ACK=1，ack=x+1），同时自己也发送一个SYN包（SYN=1，seq=y），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK=1(ack=y+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。

TCP四次挥手



第一次挥手：主动断开连接一方向被动断开连接发送FIN数据包，FIN=1,seq=x，告诉被动断开连接一方“我要跟你断开连接了，我不会再给你发送数据了”，这是主动断开连接方式可以接受数据的，如果一直没有收到被动连接方的确认包，则可以重新发送这个包。此时，主动断开连接方处于FIN\_WAIT\_1状态

第二次挥手：被动连接方接收到FIN包以后，发送确认包ACK=1,ack=x+1（FIN和SYN一样占用一个序列号），这个动作是告诉主动断开连接方我知道你要断开了，但是我还有数据没有发送完，等发送完了所有的数据就进行第三次挥手，此时被动断开连接方处于CLOSE\_WAIT状态，主动断开连接方处于FIN\_WAIT\_2状态

第三次挥手：被动断开连接方发送FIN=1，seq=y+1包，用来停止向主动断开连接方发送数据，也就是告诉主动断开连接方，我的数据也发完了，我也不给你发数据了，此时被动断开连接方处于LAST\_ACK状态，主动断开连接方处于TIME\_WAIT 状态

第四次挥手：等过了一定时间（2MSL（报文段最大生存时间）:为了保证最后ACK报文能够到达B，防止已失效连接请求报文段出现在此连接中）过后，主动断开连接方发送确认包ACK=1, ack=y+2，至此，四次挥手已经完成。

**【注意】** 在TIME\_WAIT状态中，如果TCP client端最后一次发送的ACK丢失了，它将重新发送。TIME\_WAIT状态中所需要的时间是依赖于实现方法的。典型的值为30秒、1分钟和2分钟。等待之后连接正式关闭，并且所有的资源(包括端口号)都被释放。

**【问题1】为什么连接的时候是三次握手，关闭的时候却是四次握手？**  
答：因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

**【问题2】为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？**

答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME\_WAIT状态就是用来重发可能丢失的ACK报文。



**批处理系统主要指多道批处理系统，由于多道程序能交替使用CPU，提高了CPU及其他系统资源的利用率，同时也提高了系统的效率。多道批处理系统的缺点是延长了作业的周转时间，用户不能进行直接干预，缺少交互性，不利于程序的开发与调试。**

# 中缀转后缀表达式

目的：将中缀表达式（即标准形式的表达式）转换为后缀式。

例子：a+b\*c+(d\*e+f)\*g转换成abc\*+de\*f+g\*+

转换原则：

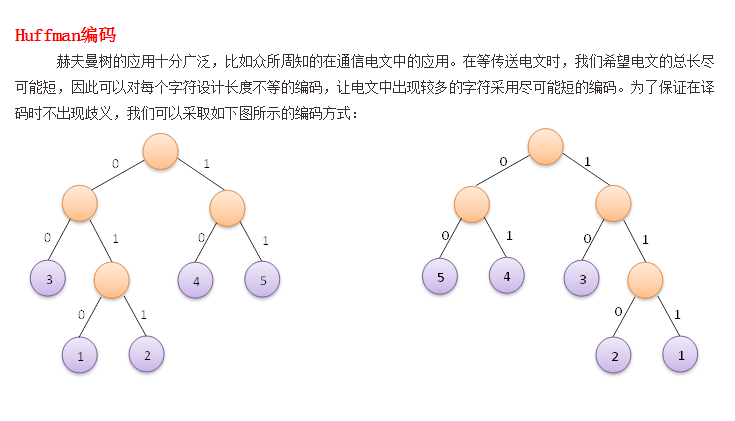
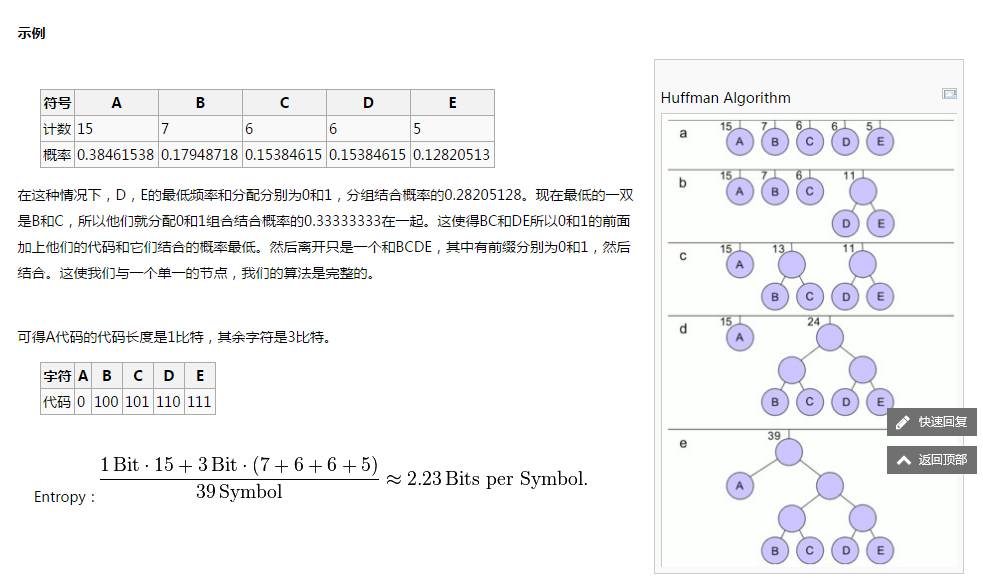
1.当读到一个操作数时，立即将它放到输出中。操作符则不立即输出，放入栈中。遇到左圆括号也推入栈中。

2.如果遇到一个右括号，那么就将栈元素弹出，将符号写出直到遇到一个对应的左括号。但是这个左括号只被弹出，并不输出。

3.在读到操作符时，如果此时栈顶操作符优先性大于或等于此操作符，弹出栈顶操作符(镇压不住呀)直到发现优先级更低的元素位置。除了处理）的时候，否则决不从栈中移走"（"。操作符中，+-优先级最低，（）优先级最高。

4.如果读到输入的末尾，将栈元素弹出直到该栈变成空栈，将符号写到输出中。

# 霍夫曼编码



# [原码, 反码, 补码 详解](http://www.cnblogs.com/zhangziqiu/archive/2011/03/30/ComputerCode.html)

机器数与真值

机器数，即数字在计算机中的二进制表示形式。  
真值，第一位用+-表示数字的正负，其余为二进制数。  
举个栗子：-3的机器数是10000011，真值是-0000011。

原码

原码，符号位加真值的绝对值。

|  |  |
| --- | --- |
| 1 2 | +3[原] = 00000011 -3[原] = 10000011 |

反码

正数的反码是其原码。  
负数的反码，符号位不变，其余各位取反，即1变成0，0变成1。

|  |  |
| --- | --- |
| 1 2 | +3[反] = 00000011[原] = 00000011[反] -3[反] = 10000011[原] = 11111100[反] |

补码

正数的的补码是其原码。  
负数的补码，是其反码+1。

|  |  |
| --- | --- |
| 1 2 | +3[补] = 00000011[原] = 00000011[反] = 00000011[补] -3[反] = 10000011[原] = 11111100[反] = 11111101[补] |

[1000 0000]补 就是-128，而使用补码表示的范围为[-128, 127].

# 大端小端 内存对齐

<http://www.cnblogs.com/luxiaoxun/archive/2012/09/05/2671697.html>

**C++内存对齐规则**

每个特定平台上的编译器都有自己的默认“对齐系数”(也叫对齐模数)。程序员可以通过预编译命令#pragma pack(n)，n=1,2,4,8,16来改变这一系数，其中的n就是你要指定的“对齐系数”。

1、第一个数据成员放在offset为0的地方，以后每个数据成员的对齐按照#pragma pack指定的数值和这个数据成员自身长度中，比较小的那个进行。

2、在数据成员完成各自对齐之后，类(结构或联合)本身也要进行对齐，对齐将按照#pragma pack指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。

　　很明显#pragma pack(n)作为一个预编译指令用来设置多少个字节对齐的。值得注意的是，n的缺省数值是按照编译器自身设置，一般为8，合法的数值分别是1、2、4、8、16。

　　即编译器只会按照1、2、4、8、16的方式分割内存。若n为其他值，是无效的。

<http://www.cnblogs.com/zrtqsk/p/4371773.html>

# C++笔试题

**2. 什么是“引用”？申明和使用“引用”要注意哪些问题？**

答：引用就是某个目标变量的“别名”(alias)，对应用的操作与对变量直接操作效果完全相同。申明一个引用的时候，切记要对其进行**初始化**。引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，**不能再把该引用名作为其他变量名的别名**。声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系**统也不给引用分配存储单元**。不能建立数组的引用。

**3. 将“引用”作为函数参数有哪些特点？**

（1）**传递引用给函数与传递指针的效果是一样的**。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

（2）**使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作**；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，**用引用比用一般变量传递参数的效率和所占空间都好。**

（3）使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中**同样要给形参分配存储单元**，且需要重复使用"\*指针变量名"的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，**在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。**

**结构与联合有和区别？**  
1. 结构和联合都是由多个不同的数据类型成员组成, 但在任何同一时刻, 联合中只存放了一个被选中的成员（所有成员共用一块地址空间）, 而结构的所有成员都存在（不同成员的存放地址不同）。   
2. 对于联合的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了, 而对于结构的不同成员赋值是互不影响的。

**include <file>和#include“file”有什么不同?**

1.用符号“<”和“>”将要包含的文件的文件名括起来。这种方法指示预处理程序到预定义的缺省路径下寻找文件。预定义的缺省路径通常是在INCLUDE环境变量中指定的，请看下例：     
    INCLUDE=C:\COMPILER\INCLUDE；S:\SOURCE\HEADERS；  
对于上述INCLUDE环境变量，如果用#include<file>语句包含文件，编译程序将首先到C:\COMPILER\INCLUDE目录下寻找文件；如果未找到，则到S:\SOURCE\HEADERS目录下继续寻找；如果还未找到，则到当前目录下继续寻找。     
(2)用双引号将要包含的文件的文件名括起来。这种方法指示预处理程序先到当前目录下寻找文件，再到预定义的缺省路径下寻找文件。

[**C/C++中extern关键字详解**](http://www.cnblogs.com/yc_sunniwell/archive/2010/07/14/1777431.html)

extern可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。此外extern也可用来进行链接指定。

（1）变量

尤其是对于变量来说。  
extern int a;//声明一个全局变量a  
int a; //定义一个全局变量a  
  
extern int a =0 ;//定义一个全局变量a 并给初值。  
int a =0;//定义一个全局变量a,并给初值，  
第四个 等于 第 三个，都是定义一个可以被外部使用的全局变量，并给初值。  
糊涂了吧，他们看上去可真像。但是定义只能出现在一处。也就是说，不管是int a;还是extern int a=0;还是int a=0;都只能出现一次，而那个extern int a可以出现很多次。

当你要引用一个全局变量的时候，你就要声明，extern int a;这时候extern不能省略，因为省略了，就变成int a;这是一个定义，不是声明。

（2）函数  
     函数，函数，对于函数也一样，也是定义和声明，定义的时候用extern，说明这个函数是可以被外部引用的，声明的时候用extern说明这是一个声明。 但由于函数的定义和声明是有区别的，定义函数要有函数体，声明函数没有函数体，所以函数定义和声明时都可以将extern省略掉，反正其他文件也是知道这个函数是在其他地方定义的，所以不加extern也行。两者如此不同，所以省略了extern也不会有问题。

总结下：

对变量而言，如果你想在本源文件中使用另一个源文件的变量，就需要在使用前用extern声明该变量，或者在头文件中用extern声明该变量；

对函数而言，如果你想在本源文件中使用另一个源文件的函数，就需要在使用前用声明该变量，声明函数加不加extern都没关系，所以在头文件中函数可以不用加extern。

<http://blog.sina.com.cn/s/blog_54a1bca70100hbf5.html>

**重载（overload)和重写(overried，有的书也叫做“覆盖”）的区别？**

从定义上来说：

**重载**：是指允许存在多个**同名函数**，而这些函数的**参数表不同**（或许参数个数不同，或许参数类型不同，或许两者都不同）。

**重写**：是指子类重新定义父类虚函数的方法。

从实现原理上来说：

**重载**：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数：function func(p:integer):integer;和function func(p:string):integer;。那么编译器做过修饰后的函数名称可能是这样的：int\_func、str\_func。对于这两个函数的调用，在编译器间就已经确定了，是**静态**的。也就是说，**它们的地址在编译期就绑定了（早绑定），**因此，**重载和多态无关**！

**重写**：和多态真正相关。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，**动态的调用**属于子类的该函数，这样的函**数调用在编译期间是无法确定的**（调用的子类的虚函数的地址无法给出）。因此，**这样的函数地址是在运行期绑定的（晚绑定）。**

**New delete 与mallocfree 的联系与区别?**答案：都是在堆(heap)上进行动态的内存操作。用malloc函数需要指定内存分配的字节数并且不能初始化对象，new 会自动调用对象的构造函数。delete 会调用对象的destructor，而free 不会调用对象的destructor.

**sizeof的概念**

sizeof是一个操作符（operator）。

其作用是返回一个对象或类型所占的内存字节数。

其返回值类型为size\_t。

sizeof有三种语法形式：

1） sizeof (object); //sizeof (对象)

2） sizeof object; //sizeof 对象

3） sizeof (type\_name); //sizeof (类型)

**1.      基本数据类型的sizeof**

这里的基本数据类型是指short、int、long、float、double这样的简单内置数据类型。

**2.      结构体的sizeof**

结构体的sizeof涉及到字节对齐问题。

1. **struct** S3
2. {
3. };
4. **sizeof**(S3); //值为1，空结构体也占内存。

**4.      数组的sizeof**

数组的sizeof值等于数组所占用的内存字节数。

**5.      指针的sizeof**

指针是用来记录另一个对象的地址，所以指针的内存大小当然就等于计算机内部地址总线的宽度。

在32位计算机中，一个指针变量的返回值必定是4。

指针变量的sizeof值与指针所指的对象没有任何关系。

**6.      函数的sizeof**

sizeof也可对一个函数调用求值，其结果是函数返回值类型的大小，函数并不会被调用。

对函数求值的形式：sizeof(函数名(实参表))

void Func ( char str[100] )  
{  
　sizeof( str ) =?  
}  
void\*p = malloc( 100 );  
sizeof ( p ) =?

**解答：**

sizeof( str ) = 4  
sizeof ( p ) = 4  
**剖析：**

　　Func ( char str[100] )函数中数组名作为函数形参时，在函数体内，数组名失去了本身的内涵，仅仅只是一个指针；在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。

**字符串函数**

头文件：#include <string.h>

定义函数：char \*strcpy(char \*dest, const char \*src);

函数说明：strcpy()会将参数src 字符串拷贝至参数dest 所指的地址。

返回值：返回参数dest 的字符串起始地址。

附加说明：如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出(buffer Overflow)的错误情况，在编写程序时请特别留意，或者用strncpy()来取代。

strlen()函数用来计算字符串的长度，其原型为：

unsigned int strlen (char \*s);

【参数说明】s为指定的字符串。

strlen()用来计算指定的字符串s 的长度，不包括结束字符"\0"。

【返回值】返回字符串s 的字符数。

strncat()用于将n个字符追加到字符串的结尾，其原型为：

char \* strncat(char \*dest, const char \*src, size\_t n);

【参数说明】dest为目标字符串，src为源字符串，n为要追加的字符的数目。

strncat()将会从字符串src的开头拷贝n 个字符到dest字符串尾部，dest要有足够的空间来容纳要拷贝的字符串。如果n大于字符串src的长度，那么仅将src全部追加到dest的尾部。

strncat()会将dest字符串最后的'\0'覆盖掉，字符追加完成后，再追加'\0'。

**const**

**1、定义常量**  
(1)const修饰变量，以下两种定义形式在本质上是一样的。它的含义是：const修饰的类型为TYPE的变量value是不可变的。

 TYPE const ValueName = value;   
   const TYPE ValueName = value;

**2、指针使用CONST**  
(1)指针本身是常量不可变  
     char\* const pContent;   
(2)指针所指向的内容是常量不可变  
     const char \*pContent;   
(3)两者都不可变  
      const char\* const pContent;

char \*const cp; //到char的const指针  
char const \*pc1; //到const char的指针  
const char \*pc2; //到const char的指针（后两个声明是等同的）

s  
(4)还有其中区别方法，沿着\*号划一条线：  
如果const位于\*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；  
如果const位于\*的右侧，const就是修饰指针本身，即指针本身是常量。

**const和static的作用**

太常见的问题了，下面给出一个较详细的参考答案：

**static关键字：**

1）函数体内static变量的作用范围为函数体。不同于auto变量。该变量的内存只被分配一次。因此其值在下次调用时仍维持上次的值。

2）在模块内的static全局变量可以被模块内的所有函数访问。但不能被模块外的其他函数访问。

3）在模块内的static函数只可被这一模块内的其它函数调用。这个函数的使用范围被限制在声明它的模块内。

4）在类中的static成员变量属于整个类所有，对类的所有对象只有一份复制。

5）在类中的static成员函数属于整个类所有，这个函数不接受this指针，因而只能访问类的static成员变量。

const关键字：

1）欲阻止一个变量被改变，可以使用const关键字。在定义该const变量时，通常需要对它进行初始化。因为以后就没有机会再改变它了。

2）对指针来说，可以指定指针的本身为const，也可以指定指针所指向的数为const。或二者同时为const。

3）在一个函数的声明中，const可以修饰形参，表明它是一个输入参数。在函数内不能改变其值。

4）对于类的成员函数，若指定其为const类型。则表明其是一个常量函数。不能修改类的成员变量。

5）对于类的成员函数，有时候必须指定其返回值为const类型。以使得其返回值不为“左值”。

1. 注意sizeof不是函数而是运算符，所以在计算变量所占用空间大小时，括号是可以省略的，但在计算类型大小时括号则不能省略，比如int i = 0; 则sizeof int是错误的。

6. 易误解：如果int a[5], 那么a与&a是等价的，因为两者地址相同。

解答：一定要注意a与&a是不一样的，虽然两者地址相同，但意义不一样，&a是整个数组对象的首地址，而a是数组首地址，也就是a[0]的地址，a的类型是int[5]，a[0]的类型是int，因此&a+1相当于a的地址值加上sizeof(int) \* 5，也就是a[5]，下一个对象的地址，已经越界了，而a+1相当于a的地址加上sizeof(int)，即a[1]的地址。

8. 可作为函数重载判断依据的有：参数个数、参数类型、const修饰符；

   不可以作为重载判断依据的有：返回类型。

9. 程序输出题：

int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int \*p = &(a + 1)[3];

printf("%d\n", \*p);

输出：5

说明：因为a+1指向a的第二个元素，[3]表示再向后移动3个元素。

char str1[] = "abc";

char str2[] = "abc";

const char str3[] = "abc";

const char str4[] = "abc";

const char \*str5 = "abc";

const char \*str6 = "abc";

char \*str7 = "abc";

char \*str8 = "abc";

cout << (str1 == str2) << endl;

cout << (str3 == str4) << endl;

cout << (str5 == str6) << endl;

cout << (str7 == str8) << endl;

[复制代码](javascript:void(0);)

输出：0 0 1 1

说明：输出str1~str8的地址为：

0x23aa80

0x23aa70

0x23aa60

0x23aa50

0x23aa48

0x23aa40

0x23aa38

0x23aa30

输出str1~str8内容“abc”的存储地址为：

0x23aa80

0x23aa70

0x23aa60

0x23aa50

0x100403030

0x100403030

0x100403030

0x100403030

可以发现str1~str4中的内容是存在栈上，地址各不相同，而str5~str8的内容都是存储在常量区，所以地址都相同。

**注意：**

char \*str = "abc";

printf("%p\n", str1);

cout << &str1 << endl;

上面打印的是字符串 “abc”的地址，下面打印的是 str1 变量的地址。

20. 虚函数原理考点，例如下面程序的输出是什么？

[复制代码](javascript:void(0);)

class A {

public:

virtual void funa();

virtual void funb();

void func();

static void fund();

static int si;

private:

int i;

char c;

};

[复制代码](javascript:void(0);)

问：sizeof(A) = ?

解答：

关于类占用的内存空间，有以下几点需要注意：

（1）如果类中含有虚函数，则编译器需要为类构建虚函数表，类中需要存储一个指针指向这个虚函数表的首地址，注意不管有几个虚函数，都只建立一张表，所有的虚函数地址都存在这张表里，类中只需要一个指针指向虚函数表首地址即可。

（2）类中的静态成员是被类所有实例所共享的，它不计入sizeof计算的空间

（3）类中的普通函数或静态普通函数都存储在栈中，不计入sizeof计算的空间

（4）类成员采用字节对齐的方式分配空间

答案：12（32位系统）或16（64位系统）

**new 、 delete 、 malloc 、 free 关系**

delete 会调用对象的析构函数 , 和 new 对应 free 只会释放内存， new 调用构造函数。 malloc 与 free 是 C++/C 语言的标准库函数， new/delete 是 C++ 的运算符。它们都可用于申请动态内存和释放内存。对于非内部数据类型的对象而言，光用 maloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free 。因此 C++ 语言需要一个能完成动态内存分配和初始化工作的运算符 new ，以及一个能完成清理与释放内存工作的运算符 delete 。注意 new/delete 不是库函数。

总结：new和delete会自动调用对象的构造与析构函数而malloc与free不会；

new和delete式C++运算符，而malloc和free是C/C++标准库函数。

**delete 与 delete [] 区别**

delete 只会调用一次析构函数，而 delete[] 会调用每一个成员的析构函数。

# Android笔试

## 四大组件

1. 应用程序中，一个Activity通常就是一个单独的屏幕，它上面可以显示一些控件也可以监听并处理用户的事件做出响应。Activity之间通过Intent进行通信。在Intent 的描述结构中，有两个最重要的部分：动作和动作对应的数据。android应用中每一个Activity都必须要在AndroidManifest.xml配置文件中声明，否则系统将不识别也不执行该Activity。
2. 应用可以使用它对外部事件进行过滤只对感兴趣的外部事件(如当电话呼入时，或者数据网络可用时)进行接收并做出响应。广播接收器没有用户界面。然而，它们可以启动一个activity或serice 来响应它们收到的信息，或者用NotificationManager 来通知用户。扮演一种打开其他组件的角色。通知可以用很多种方式来吸引用户的注意力──闪动背灯、震动、播放声音等。一般来说是在状态栏上放一个持久的图标，用户可以打开它并获取消息。广播接收者的注册有两种方法，分别是程序动态注册和AndroidManifest文件中进行静态注册。

生命周期只有十秒左右，如果在 onReceive() 内做超过十秒内的事情，就会报ANR，如果需要完成一项比较耗时的工作 , 应该通过发送 Intent 给 Service, 由Service 来完成 . 这里不能使用子线程来解决。

系统常见广播Intent,如开机启动、电池电量变化、时间改变等广播。广播被分为两种不同的类型：“普通广播（Normal broadcasts）”和“有序广播（Ordered broadcasts）”。普通广播是完全异步的，可以在同一时刻（逻辑上）被所有接收者接收到，消息传递的效率比较高，但缺点是：接收者不能将处理结果传递给下一个接收者，并且无法终止广播Intent的传播；然而有序广播是按照接收者声明的优先级别（声明在intent-filter元素的android:priority属性中，数越大优先级别越高,取值范围:-1000到1000。也可以调用IntentFilter对象的setPriority()进行设置），被接收者依次接收广播。如：A的级别高于B,B的级别高于C,那么，广播先传给A，再传给B，最后传给C。A得到广播后，可以往广播里存入数据，当广播传给B时,B可以从广播中得到A存入的数据。

1. 一个Service 是一段长生命周期的，通常位于后台运行，没有用户界面的程序，它一般不需要与用户交互，可以用来开发如监控类程序。就是一个没有界面的activity。比较好的一个例子就是一个正在从播放列表中播放歌曲的媒体播放器。服务不能自己运行,需要通过Contex.startService()或Contex.bindService()启动服务。started service(启动服务)是由其他组件调用startService()方法启动的，这导致服务的onStartCommand()方法被调用。当服务是started状态时，其生命周期与启动它的组件无关，并且可以在后台无限期运行，即使启动服务的组件已经被销毁。因此，服务需要在完成任务后调用stopSelf()方法停止，或者由其他组件调用stopService()方法停止。使用bindService()方法启用服务，调用者与服务绑定在了一起，调用者一旦退出，服务也就终止，大有“不求同时生，必须同时死”的特点。
2. ContentProvider实现不同应用间的数据共享，Content Provider提供数据对外访问接口，使一个应用程序的指定数据集提供给其他应用程序。其他应用可以通过ContentResolver类从该内容提供者中获取或存入数据。这是不同应用程序间共享数据的唯一方式，因为android没有提供所有应用共同访问的公共存储区。ContentProvider使用URI来唯一标识其数据集，这里的URI以content://作为前缀，表示该数据由ContentProvider来管理。只有需要在多个应用程序间共享数据是才需要内容提供者。例如，通讯录数据被多个应用程序使用，且必须存储在一个内容提供者中。它的好处是统一数据访问方式。就是对Sqlite进行了一层封装。

**4大组件的注册**

4大基本组件都需要注册才能使用，每个Activity、service、Content Provider都需要在AndroidManifest文件中进行配置。AndroidManifest文件中未进行声明的activity、服务以及内容提供者将不为系统所见，从而也就不可用。而broadcast receiver广播接收者的注册分静态注册(在AndroidManifest文件中进行配置)和通过代码动态创建并以调用Context.registerReceiver()的方式注册至系统。需要注意的是在AndroidManifest文件中进行配置的广播接收者会随系统的启动而一直处于活跃状态，只要接收到感兴趣的广播就会触发(即使程序未运行)。

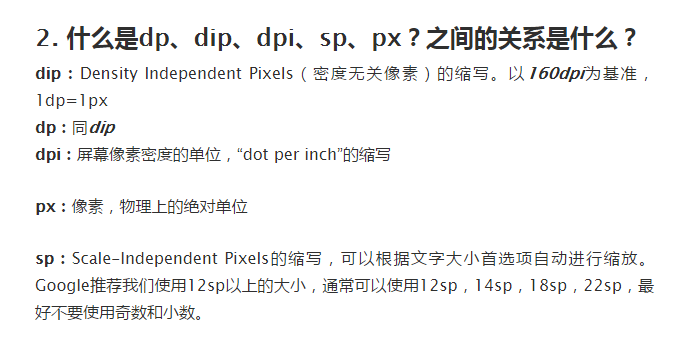
**4大组件的激活**

内容提供者的激活：当接收到ContentResolver发出的请求后，内容提供者被激活。而其它三种组件activity、服务和广播接收器被一种叫做intent的异步消息所激活。

**4大组件的关闭**

内容提供者仅在响应ContentResolver提出请求的时候激活。而一个广播接收器仅在响应广播信息的时候激活。所以，没有必要去显式的关闭这些组件。Activity关闭：可以通过调用它的finish()方法来关闭一个activity。服务关闭：对于通过startService()方法启动的服务要调用Context.stopService()方法关闭服务，使用bindService()方法启动的服务要调用Contex.unbindService()方法关闭服务。

## 屏幕适配



1. 使用dp,sp。

2. 应使用 “wrap\_content” 和 “match\_parent” 控制某些视图组件的宽度和高度，确保布局的灵活性并适应各种尺寸的屏幕，使用 “wrap\_content”，系统就会将视图的宽度或高度设置成所需的最小尺寸以适应视图中的内容，而 “match\_parent”（在低于 API 级别 8 的级别中称为 “fill\_parent”）则会展开组件以匹配其父视图的尺寸。weight是线性布局的一个独特的属性，我们可以使用这个属性来按照比例对界面进行分配，完成一些特殊的需求。

3. 使用相对布局，禁用绝对布局在开发中，我们大部分时候使用的都是线性布局、相对布局和帧布局，绝对布局由于适配性极差，所以极少使用。

4. 使用尺寸限定符。通过使用配置限定符，在运行时根据当前的设备配置自动选择合适的资源了，例如根据各种屏幕尺寸选择不同的布局。很多应用会在较大的屏幕上实施“双面板”模式，即在一个面板上显示项目列表，而在另一面板上显示对应内容。平板电脑和电视的屏幕已经大到可以同时容纳这两个面板了，但手机屏幕就需要分别显示。

res/layout/main.xml，单面板（默认）布局：

res/layout-large/main.xml，双面板布局：

 large 限定符。系统会在属于较大屏幕（例如 7 英寸或更大的平板电脑）的设备上选择此布局。系统会在较小的屏幕上选择其他布局（无限定符）。

使用最小宽度限定符：res/layout-sw600dp/main.xml，双面板布局，对于最小宽度大于等于 600 dp 的设备，系统会选择 layout-sw600dp/main.xml（双面板）布局，否则系统就会选择 layout/main.xml（单面板）布局。

1. 使用自动拉伸位图，自动拉伸位图，即android下特有的.9.png图片格式。支持各种屏幕尺寸通常意味着您的图片资源还必须能适应各种尺寸。例如，无论要应用到什么形状的按钮上，按钮背景都必须能适应。.9格式图片的制作。

## ANR

ANR定义：在Android上，如果你的应用程序有一段时间响应不够灵敏，系统会向用户显示一个对话框，这个对话框称作应用程序无响应（ANR：Application Not Responding）对话框。用户可以选择“等待”而让程序继续运行，也可以选择“强制关闭”。所以一个流畅的合理的应用程序中不能出现anr，而让用户每次都要处理这个对话框。因此，在程序里对响应性能的设计很重要，这样系统不会显示ANR给用户。

**原因**

* 应用在**5秒**内未响应用户的输入事件（如按键或者触摸）
* BroadcastReceiver未在**10秒**内完成相关的处理

主线程在Service的各个生命周期函数时20秒内没有处理完毕。

**出现场景**

* 主线程被IO操作（从4.0之后网络IO不允许在主线程中）阻塞。
* 主线程中存在耗时的计算
* 主线程中错误的操作，比如Thread.wait或者Thread.sleep等

Android系统会监控程序的响应状况，一旦出现下面两种情况，则弹出ANR对话框

**如何避免**

基本的思路就是将IO操作在工作线程来处理，减少其他耗时操作和错误操作

* 使用[AsyncTask](http://droidyue.com/blog/2014/11/08/bad-smell-of-asynctask-in-android/)或者子线程处理耗时IO操作。
* 使用[Handler](http://droidyue.com/blog/2014/12/28/in-android-handler-classes-should-be-static-or-leaks-might-occur/)处理工作线程结果，而不是使用Thread.wait()或者Thread.sleep()来阻塞主线程。
* Activity的onCreate和onResume回调中尽量避免耗时的代码
* BroadcastReceiver中onReceive代码也要尽量减少耗时，建议使用IntentService处理。
* 使用Thread或者HandlerThread时，调用Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND)设置优先级，否则仍然会降低程序响应，因为默认Thread的优先级和主线程相同。

## OOM

android上的app使用独立虚拟机，每开一个应用就会打开至少一个独立的虚拟机。

为了整个Android系统的内存控制需要，Android系统为每一个应用程序都设置了一个硬性的Dalvik Heap Size最大限制阈值，这个阈值在不同的设备上会因为RAM大小不同而各有差异。如果你的应用占用内存空间已经接近这个阈值，此时再尝试分配内存的话，很容易引起OutOfMemoryError的错误。

ActivityManager.getMemoryClass()可以用来查询当前应用的Heap Size阈值，这个方法会返回一个整数，表明你的应用的Heap Size阈值是多少Mb(megabates)。

因为android系统的app的每个进程或者每个虚拟机有个最大内存限制，如果申请的内存资源超过这个限制，系统就会抛出OOM错误。跟整个设备的剩余内存没太大关系。比如比较早的android系统的一个虚拟机最多16M内存，当一个app启动后，虚拟机不停的申请内存资源来装载图片，当超过内存上限时就出现OOM。

#### 2.1 Android的APP内存组成：

APP内存由 **dalvik内存** 和 **native内存** 2部分组成，dalvik也就是java堆，创建的对象就是就是在这里分配的，而native是通过c/c++方式申请的内存，Bitmap就是以这种方式分配的。（android3.0以后，系统都默认通过dalvik分配的，native作为堆来管理）。这2部分加起来不能超过android对单个进程，虚拟机的内存限制。

每个手机的内存限制大小是多少？

ActivityManager activityManager = (ActivityManager)context.getSystemService(Context.ACTIVITY\_SERVICE)

activityManager.getMemoryClass();

以上方法会返回以M为单位的数字，不同的系统平台或设备上的值都不太一样，比如HTC默认24M, Galaxy36M, emulator-2.3 24M，等等。我的moto xt681是42M。3

上面取到是虚拟机的最大内存资源。

而对于head堆的大小限制，可以查看/system/build.prop文件。

dalvik.vm.heapstartsize = 5m

dalvik.vm.heapgrowthlimit = 48m

dalvik.vm.heapsize = 256m

注： heapsize参数表示单个进程heap可用的最大内存，但如果存在以下参数"dalvik.vm.headgrowthlimit =48m"表示单个进程heap内存被限定在48m，即程序运行过程实际只能使用48m内存。

TIPS:编程要养成习惯，不用的对象设置为null。其实更好的是，不用的图片直接recycle。因为通过设置null让gc来回收，有时候还是会来不及。

### 常见避免OOM的几个注意点：

**3.1 适当调整图像大小** 。因为手机屏幕尺寸有限，分配给图像的显示区域有限，尤其对于超大图片，加载自网络或者sd卡，图片文件提及达到几M或者十几个M的：

加载到内存前，先算出该bitmap的大小，然后通过适当调节采样率使得加载的图片刚好，或稍大捷克在手机屏幕上显示就满意了：

BimtapFactory.Option opts = new BitampFactory.Option();

opts.inJustDecodeBounds = true ;

opts.inSampleSize=computeSample(opts, minSideLength, maxNumOfPixels); // Android 提供了一种动态计算的方法 computeSampleSize

opts.inJustDecodeBounds = false ;

try {

return BitmapFactory.decodeFile(imageFile, opts);

} catch (OutOfMemoryError err){

}

**3.2 图像缓存** 。在listview或Gallery等控件中一次性加载大量图片时，只加载屏幕显示的资源，尚未显示的不加载，移出屏幕的资源及时释放，采用强引用＋软引用2级缓存，提高加载性能。缓存图像到内存，采用软引用缓存到内存，而不是在每次使用的时候都从新加载到内存。

**3.3 采用低内存占用量的编码方式** 。比如Bitmap.Config.ARGB\_4444比Bitmap.Config.ARGB\_8888更省内存。

**3.4 及时回收图像** 。如果引用了大量的Bitmap对象，而应用又不需要同时显示所有图片。可以将暂时不用到的Bitmap对象及时回收掉。对于一些明确直到图片使用情况的场景可以主动recycle回收

App的启动splash画面上的图片资源，使用完就recycle。对于帧动画，可以加载一张，画一张，释放一张。

**3.5 不要在循环中创建过多的本地变量** 。慎用static，用static来修饰成员变量时，该变量就属于该类，而不是该类实例，它的生命周期是很长的。如果用它来引用一些内存占用太多的实例，这时候就要谨慎对待了。

**3.6 自定义堆内存分配大小** 。优化Dalvik虚拟机的堆内存分配。

public class ClassName{

private static Context mContext;

// 省略

}

## 如何避免OOM总结

**减小对象的内存占用**

避免OOM的第一步就是要尽量减少新分配出来的对象占用内存的大小，尽量使用更加轻量的对象。

使用更加轻量的数据结构

例如，我们可以考虑使用ArrayMap/SparseArray而不是HashMap等传统数据结构，下图演示了HashMap的简要工作原理，相比起Android系统专门为移动操作系统编写的ArrayMap容器，在大多数情况下，都显示效率低下，更占内存。通常的HashMap的实现方式更加消耗内存，因为它需要一个额外的实例对象来记录Mapping操作。另外，SparseArray更加高效在于他们避免了对key与value的autobox自动装箱，并且避免了装箱后的解箱。

避免在Android里面使用Enum

减小Bitmap对象的内存占用

Bitmap是一个极容易消耗内存的大胖子，减小创建出来的Bitmap的内存占用是很重要的，通常来说有下面2个措施：

inSampleSize：缩放比例，在把图片载入内存之前，我们需要先计算出一个合适的缩放比例，避免不必要的大图载入。

decode format：解码格式，选择ARGB\_8888/RBG\_565/ARGB\_4444/ALPHA\_8，存在很大差异。

使用更小的图片

在设计给到资源图片的时候，我们需要特别留意这张图片是否存在可以压缩的空间，是否可以使用一张更小的图片。尽量使用更小的图片不仅仅可以减少内存的使用，还可以避免出现大量的InflationException。假设有一张很大的图片被XML文件直接引用，很有可能在初始化视图的时候就会因为内存不足而发生InflationException，这个问题的根本原因其实是发生了OOM。

**内存对象的重复利用**

大多数对象的复用，最终实施的方案都是利用对象池技术，要么是在编写代码的时候显式的在程序里面去创建对象池，然后处理好复用的实现逻辑，要么就是利用系统框架既有的某些复用特性达到减少对象的重复创建，从而减少内存的分配与回收。

在Android上面最常用的一个缓存算法是LRU(Least Recently Use)

复用系统自带的资源

Android系统本身内置了很多的资源，例如字符串/颜色/图片/动画/样式以及简单布局等等，这些资源都可以在应用程序中直接引用。这样做不仅仅可以减少应用程序的自身负重，减小APK的大小，另外还可以一定程度上减少内存的开销，复用性更好。但是也有必要留意Android系统的版本差异性，对那些不同系统版本上表现存在很大差异，不符合需求的情况，还是需要应用程序自身内置进去。

注意在ListView/GridView等出现大量重复子组件的视图里面对ConvertView的复用

<http://blog.csdn.net/guolin_blog/article/details/9316683>

Bitmap对象的复用

在ListView与GridView等显示大量图片的控件里面需要使用LRU的机制来缓存处理好的Bitmap。

避免在onDraw方法里面执行对象的创建

类似onDraw等频繁调用的方法，一定需要注意避免在这里做创建对象的操作，因为他会迅速增加内存的使用，而且很容易引起频繁的gc，甚至是内存抖动。

StringBuilder

在有些时候，代码中会需要使用到大量的字符串拼接的操作，这种时候有必要考虑使用StringBuilder来替代频繁的“+”。

**避免对象的内存泄露**

内存对象的泄漏，会导致一些不再使用的对象无法及时释放，这样一方面占用了宝贵的内存空间，很容易导致后续需要分配内存的时候，空闲空间不足而出现OOM。显然，这还使得每级Generation的内存区域可用空间变小，gc就会更容易被触发，容易出现内存抖动，从而引起性能问题。

注意Activity的泄漏

通常来说，Activity的泄漏是内存泄漏里面最严重的问题，它占用的内存多，影响面广，我们需要特别注意以下两种情况导致的Activity泄漏：

1.内部类引用导致Activity的泄漏

最典型的场景是Handler导致的Activity泄漏，如果Handler中有延迟的任务或者是等待执行的任务队列过长，都有可能因为Handler继续执行而导致Activity发生泄漏。此时的引用关系链是Looper -> MessageQueue -> Message -> Handler -> Activity。为了解决这个问题，可以在UI退出之前，执行remove Handler消息队列中的消息与runnable对象。或者是使用Static + WeakReference的方式来达到断开Handler与Activity之间存在引用关系的目的。

2.Activity Context被传递到其他实例中，这可能导致自身被引用而发生泄漏。

内部类引起的泄漏不仅仅会发生在Activity上，其他任何内部类出现的地方，都需要特别留意！我们可以考虑尽量使用static类型的内部类，同时使用WeakReference的机制来避免因为互相引用而出现的泄露。

考虑使用Application Context而不是Activity Context

对于大部分非必须使用Activity Context的情况（Dialog的Context就必须是Activity Context），我们都可以考虑使用Application Context而不是Activity的Context，这样可以避免不经意的Activity泄露。

注意临时Bitmap对象的及时回收

虽然在大多数情况下，我们会对Bitmap增加缓存机制，但是在某些时候，部分Bitmap是需要及时回收的。例如临时创建的某个相对比较大的bitmap对象，在经过变换得到新的bitmap对象之后，应该尽快回收原始的bitmap，这样能够更快释放原始bitmap所占用的空间。

注意监听器的注销

在Android程序里面存在很多需要register与unregister的监听器，我们需要确保在合适的时候及时unregister那些监听器。自己手动add的listener，需要记得及时remove这个listener。

注意WebView的泄漏 注意缓存容器中的对象泄漏

注意Cursor对象是否及时关闭

在程序中我们经常会进行查询数据库的操作，但时常会存在不小心使用Cursor之后没有及时关闭的情况。这些Cursor的泄露，反复多次出现的话会对内存管理产生很大的负面影响，我们需要谨记对Cursor对象的及时关闭。

### 4. App使用图片时避免OOM的几种方式：

4.1  **直接null或recycle**

对于app里使用的大量图片，采用方式：使用时加载，不显示时直接置null或recycle。

这样处理是个好习惯，记本可以杜绝OOM，但是缺憾是代码多了，可能会忘记某些资源recycle。

而有些情况下会出现特定图片反复加载，释放，再加载等，低效率的事情。

**4.2 简单通过SoftReference引用方式管理图片资源**

建个SoftReference的hashmap

使用图片时先查询这个hashmap是否有softreference， softreference里的图片是否为空，

如果为空就加载图片到softreference并加入hashmap。

无需再代码里显式的处理图片的回收与释放，gc会自动处理资源的释放。

这种方式处理起来简单实用，能一定程度上避免前一种方法反复加载释放的低效率。但还不够优化。

**4.3 强引用＋软引用二级缓存**

Android示范程序ImageDownloader.java, 使用了一个二级缓存机制。就是有一个数据结构直接持有解码成功的Bitmap对象引用，同时使用一个二级缓存数据结构保持淘汰的Bitmap的softreference对象，由于softreference对象的特殊性，系统会再需要内存的时候首先将softreference持有的对象释放掉，也就是说当vm发现可用的内存较少需要出发gc的时候，二级缓存中的bitmap对象将被回收，而持有一级缓存的bitmap对象用于显示。

其实这个解决方案最为关键的一点是使用了一个比较合适的数据结构，那就是LinkedHashMap类型来进行一级缓存Bitmap的容器。由于LinkeHashMap的特殊性，我们可以控制其内存存储对象的个数并且将不在使用的对象从容器中移除，放到softreference二级缓存里，我们可以在一级缓存中一致保存最近被访问到的bitmap对象，而已经被访问过的图片在LinkedHashMap的容量超过我们预设值时将会把容器中存在的时间最长的对象移除，这个时候我么可以将被移除的LinkedHashMap中的放到二级缓存容器，而二级缓存中的对象管理就交给系统来做了，当系统需要gc时就会首先回收二级缓存容器的Bitmap对象了。

在获取图片对象时候先从一级缓存容器中查找，如果有对应对象并可用直接返回，如果没有的话从二级缓存中查找对应的SoftReference, 判断SoftReference对象持有的Bitmap是否可用，可用直接返回，否则返回空。如果二级缓存都找不到图片，那就直接加载图片资源。

   4， LruCache  + sd的缓存方式

### 5. 两种容易OOM的场景建议：

   5.1 网络下载大量图片

比如微博客户端： 多线程异步网络，小兔直接用LRUCache+SoftRef+Sd，大图按需下载：

    5.2 对于需要加载非常多条目信息的listview，gridview等的情况

在adapter的getView函数里有个convertView参数，告知你是否有可重用的view对象。 如果不使用convertView的话，每次调用getView时每次都会重新创建view，这样之前的view可能还没销毁，加之不断的新建view势必会造成内存剧增，从而导致OOM。另外在重用convertView时，里面原有的图片等资源就会变成无主的了。

这里Google官方推荐使用："convertview＋静态类viewholder"

官方给出解释是：

a 重用缓存convertView传递给getView()方法来避免填充不必要的视图。

  b 使用ViewHolder模式来避免没有必要的调用findViewById；因为太多的findViewById也会影响性能。

附ViewHolder类的作用：ViewHolder模式通过在getView方法返回的视图的标签（tag）中存储一个数据结构。这个数据结构包含了指向我们要绑定数据的视图的引用，从而避免每次调用getView()的时候调用findViewById();

### 6 申请超过内存限制的内存分配方式：

 6.1 从Native C分配内存。使用NDK(本地开发工具包)和JNI, 它可能从C级(如malloc/free或新建／删除)分配内存，这样的分配是不计入24MB的限制。这是真的，从本机代码分配内存是为了java方便，但它可以被用来存储在ram的数据（即使图片数据）的一些打击呢。

 6.2 使用OpenGL的纹理。纹理内存不计入限制，要查看你的应用程序确实分配了多少内存可以使用android.os.Debug.getNativeHeapAllocatedSize(), 可以使用上面介绍的两种技术的Nexus之一，我可以轻松地为一个单一的前台进程分配300MB-10倍以上的默认24MB 的限制，从上面看来使用native代码分配内存是不在24MB的限制内的（开放的GL的质地也是使用native代码分配内存）。

但是，这两个方法的风险就是，本地堆分配内存超过系统可用内存限制的话，通常都是直接崩溃。

# 综合面试：

自我介绍：

你的职业规划是什么？

开始工前的前两三年是提升技术最重要的成长期，这段时间里，尽量多做事多锻炼，不要怕苦怕累。在完成工作的同时还要花时间去学习了解其他技术，尽量让自己知识面广一点，可以不求甚解，但是要知道个大概。这样做的目的，一是可以在下次碰到要用时不会手忙脚乱，二是在别人谈及时不会云里雾里，同时也是为以后的职业规划做准备。

你的优势是什么？