突破程序员基本功的16课

一、实例变量的初始化时机

语法规定，程序可在3个地方对实例变量执行初始化：

定义实例变量时指定初始值。

非静态初始化块中对实例变量指定初始值。

构造器中对实例变量指定初始值。

其中，前两种方式比第三种方式更早执行，但第1、2种方式的执行顺序与它们在源程序中的排列顺序相同。

```
public class InitTest {

	public static void main(String[] args) {
		Cat cat = new Cat("Lily", 1);
		System.out.println(cat);
		Cat cat2 = new Cat("Lucy", 2);
		System.out.println(cat2);
	}
}

class Cat {
	String name;
	int age;

	public Cat(String name, int age) {
		System.out.println("执行构造器");
		this.name = name;
		this.age = age;
	}

	{
		System.out.println("执行非静态初始化块");//总是在构造器执行之前执行
		weight = 2.0;
	}

	double weight = 2.3;

	public String toString() {
		return "name:" + name + ",age:" + age+ ",weight:" + weight;
	}
}
```

以上程序段输出：

执行非静态初始化块

执行构造器

name:Lily,age:1,weight:2.3

执行非静态初始化块

执行构造器

name:Lucy,age:2,weight:2.3

注意：定义实例变量时指定的初始值、初始化块中为实例变量指定初始值的语句的地位是平等的，当经过编译器处理后，它们都将被提取到构造器中。也就是说，对于类定义中的语句：

```
double weight = 2.3;
```

实际上会被分成如下2次执行：

（1）double weight;：创建Java对象时系统根据该语句为该对象分配内存。

（2）weight = 2.3;：这条语句将会被提取到Java类的构造器中执行。

二、父类构造器与静态、非静态代码块的执行顺序

当创建任何Java对象时，程序总会先依次调用每个父类非静态初始化块，父类构造器（总是从Object开始）进行初始化，最后才调用本类的非静态初始化块、构造器执行初始化。

当调用某类的构造器来创建Java对象时，系统总会先调用父类的非静态初始化块进行初始化。这个调用是隐式的，而且父类的静态初始化块总是会被执行。接着会调用父类的一个或多个构造器执行初始化，这个调用可以是通过super显式调用，也可以是隐式调用。当所有父类的非静态初始化块、构造器依次调用完后，系统调用本类的非静态初始化块、构造器执行初始化，最后返回本类的实例。

如：在一中代码段增加以下内容，并让Cat继承自Animal

```
class Animal extends Obj{
        String name;
        int age;

        static{
        System.out.println("执行父类静态初始化块");
     }

    {
        System.out.println("执行父类非静态初始化块");
    }
}

class Obj{
```

```
    String name;

  static{
    System.out.println("执行Obj类静态初始化块");
  }

  {
    System.out.println("执行Obj类非静态初始化块");
  }
}
```

则程序输出：

执行Obj类静态初始化块
执行父类静态初始化块
执行Obj类非静态初始化块
执行父类非静态初始化块
执行非静态初始化块
执行构造器
name:Lily,age:1,weight:2.3
执行Obj类非静态初始化块
执行父类非静态初始化块
执行非静态初始化块
执行构造器
name:Lucy,age:2,weight:2.3

同样，若有以下继承关系：
Object 《 Parent 《 Mid 《 Sub （右侧继承自左侧）
则程序会按以下步骤进行初始化：
（0）依次执行Object 、 Parent 、Mid、 Sub 的静态初始化块（如果有的话）
（1）执行Object类非静态初始化块（如果有的话）
（2）隐式或显示调用Object类的一个或多个构造器执行初始化
（3）执行Parent类非静态初始化块（如果有的话）
（4）隐式或显示调用Parent类的一个或多个构造器执行初始化
（5）执行Mid类非静态初始化块（如果有的话）
（6）隐式或显示调用Mid类的一个或多个构造器执行初始化
（7）执行Sub类非静态初始化块（如果有的话）
（8）隐式或显示调用Sub类的一个或多个构造器执行初始化

综上，当涉及到继承时，按照如下顺序执行

（1）执行父类的静态成员变量定义与静态初始化块，执行子类的静态成员变量定义与静态初始化块

（2）执行父类的非静态成员变量定义与动态初始化块，执行父类构造方法

（3）执行子类的非静态成员变量定义与动态初始化块，执行子类构造方法

注意：

（1）父类静态代码块优先于子类静态代码块执行，因为Java虚拟机加载类时，就会执行类的静态代码块。

（2）父类构造函数优先于子类构造函数执行，因为“先有父亲，后有孩子” 。

（3）类的静态代码块执行且仅执行一次，非静态代码块根据实例化的次数会执行多次。

（4）super调用和this调用都只能在构造器中使用，而且super和this的调用必须作为构造器的第一行代码。

（5）父类构造方法中用到的方法如果已被子类重写，那么在构造子类对象过程中，调用父类构造方法时使用子类重写的方法 。

三、父子实例的内存控制

1、继承成员变量和继承成员方法的区别

且看下边程序实例：

```
public class FiledAndMethodTest {
	public static void main(String[] args) {
		Base bd = new Derived();
		System.out.println(bd.count);
		bd.dispaly();
	}
}

class Base {
	int count = 2;

	public void dispaly() {
		System.out.println(this.count);
	}
}

class Derived extends Base {
	int count = 20;

	@Override
	public void dispaly() {
		System.out.println(this.count);
```

```
    }
}
```
读者能很快给出程序输出吗？答案应该是：

2
20
解释，直接通过db访问count实例变量，输出的将是Base（声明时类型）对象的count实例变量的值；如果通过db来调用display()方法，该方法将表现出Derive（运行时类型）对象的行为方式。

原因：如果子类重写了父类方法，就意味着子类里定义的方法彻底覆盖了父类里的同名方法，系统将不可能把父类的方法转移到子类中。对于实例变量则不存在这样的现象，即使子类中定义的与父类完全同名的实例变量，这个实例变量依然不可能覆盖父类中定义的实例变量。因为继承成员变量和继承方法之间存在这样的差异，所以对于一个引用类型的变量而言，当通过该变量访问它所引用的对象的实例变量时，该变量的值取决于声明该变量时类型；当通过该变量来调用它所引用的对象的方法时，该方法取决于它所实际引用的对象的类型。

2、内存中子类实例
首先应该知道，当程序创建一个子类对象时，系统不仅会为该类中定义的实例变量分配内存，也会为其父类中定义的所有实例变量分配内存，即使子类定义与父类中同名实例变量。也就是说，当系统创建一个Java对象的时候，如果该Java类有两个父类（一个直接父类A，一个间接父类B），假设A类中定义了2个实例变量，B类中定义了3个实例变量，当前类中定义了2个实例变量，那这个Java对象将会保存2+3+2个实例变量。

如果在子类里定义了与父类中已有变量同名的变量，那么子类中定义的变量会隐藏父类中定义的变量。注意不是完全覆盖，因此系统为创建子类对象时，依然会为父类中定义的、被隐藏的变量分配内存空间。为了在子类方法中访问父类中定义的、被隐藏的实例变量，或者为了在子类方法中调用父类中定义的、被覆盖（Override）的方法，可以通过super.作为限定来修饰这些实例变量和实例方法。

注意：super关键字本身并没有引用任何对象，它甚至不能被当做一个真正的引用变量使用。如：不能直接使用return supper;，但可以使用return this;返回调用该方法的对象。

3、内部类访问的局部变量都要使用final修饰

原因：对于普通局部变量，其作用域就是停留在该方法内，当方法执行结束，该局部变量也随之消失，但内部类则可能产生隐式的“闭包”，闭包将使得局部变量脱离它所在的方法继续存在。

如下程序：

```
public class ClosureTest {
	public static void main(String[] args) {
		final String str = "Java";

		new Thread(new Runnable() {
			public void run() {
				for (int i = 0; i < 1000; i++) {
					System.out.println(str + i);
				}
			}
		}).start();//①
	}
}
```

正常情况下，程序执行完①行代码后，main的生命周期就结束了，局部变量str的作用域也会随之结束。但实际上这个程序中只要新线程里的run方法没有执行完，匿名内部类的实例的生命周期就没有结束，将一直可以访问str局部变量的值，这就是内部类会扩大局部变量作用域的实例。由于内部类可能扩大局部变量的作用域，如果再加上这个被内部类访问的局部变量没有使用final修饰，也就是说该变量的值可以随意改变，那将引起极大的混乱，因此，Java编译器要求所有被内部类访问的局部变量必须要使用final修饰。

四、常见Java集合实现细节

1、Set、Map

Set：无序、不可重复。

Map：多个key-value对组成的集合，本质是关联数组（key数组和value数组）。

联系：Map的所有key集中起来，可组成一个Set集合。

2、HashMap、HashSet

对于HashSet而言，系统采用Hash算法决定集合元素的存储位置，这样可以保证快速存、取集合元素；对于HashMap而言，系统将value当成key的附属，系统根据Hash算法来决定key的存储位置，而完全没有考虑Entry中的value。

这里得提到一个重要接口Map.Entry，每个Map.Entry其实就是一个key-value对。当程序试图将一个key-value放入HashMap中时，首先根据该key的hashCode()返回值决定Entry的存储位置：如果两个Entry的key的hashCode()返回值相同，那它们的存储位置相同；如果这两个Entry的key通过equals比较返回true，新添加的Entry的value将覆盖集合中原有Entry的value，但key不会覆盖；如果这两个Entry的key通过equals比较返回false，新添加的Entry将与集合中原有Entry形成Entry链，而且新添加的Entry位于Entry链的头部。即：key的hashCode()决定位置，位置相同则由key通过equals()比较值决定采用覆盖行为（返回true），还是生成Entry链（返回false）。

当创建HashMap时，有一个默认的负载因子（load factor），其默认值为0.75。这是时间和空间上的一种折中：增大负载因子可以减少Hash表（即Entry数组）所占用的内存空间，但会增加查询数据的时间开销，而查询是最频繁的操作（HashMap的get和put方法都要用到查询）；减小负载因子会提高查询的性能，但会增加Hash表所占用的内存空间。负载因子的作用是，当load factor（= HashMap中的数据量/HashMap总容量）达到指定值时，总容量自动扩展1倍。

3、TreeSet、TreeMap

与HashSet完全类似，TreeSet里绝大部分方法都是直接调用TreeMap的方法实现的。对TreeMap而言，它采用“红黑树”的排序二叉树来保存Map中的每个Entry——每个Entry被当做“红黑树”的一个节点对待，这就意味着，TreeMap通过循环查找到合适位置并添加元素、取出元素的性能都比HashMap低。但TreeMap、TreeSet相比HashMap、HashSet的优势在于：TreeMap中所有的Entry总是按key根据指定排序规则保持有序状态，TreeSet中所有元素总是根据指定排序规则保持有序状态。

类比而言，HashMap、HashSet的存储方式类似“妈妈放东西”，不同东西放在不同位置，需要时可以快速找到。TreeSet、TreeMap的存储方式类似“体育课站队”，第一个人（相当于元素）自成一队，以后每添加一个人都要找到这个人应该插入的位置（使这个位置前的所有人比新插入的人矮，这个位置后的所有人比新插入的高），然后在该位置插入即可。这样保证了该队伍总是由矮到高地排列。

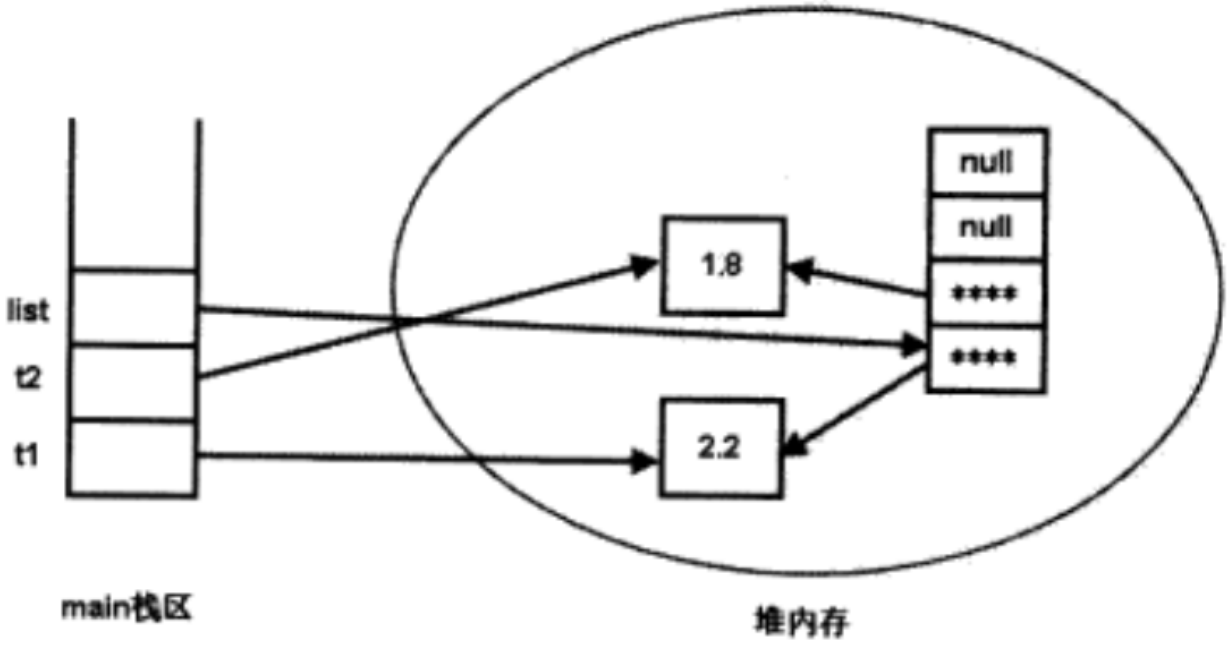
4、ArrayList

ArrayList底层是基于数组实现，每次创建ArrayList时传入的int参数就是它所包装的数组的长度；如果未传入int参数，那么ArrayList的初始长度为10，也就是它底层所封装的数组的长度为10。

注意：就像引用类型的数组一样，当把Java对象放入数组中，并不是把真正的Java对象放入数组，而是把对象的引用放入数组中，每个数组元素都是一个引用变量。如下程序：

```
public class ListTest {

	public static void main(String[] args) {
		Apple t1 = new Apple(2.2);
		Apple t2 = new Apple(1.8);

		List<Apple> list = new ArrayList<>(4);
		list.add(t1);
		list.add(t2);

		System.out.println(list.get(0) == t1);
		System.out.println(list.get(1) == t2);
	}
}

class Apple {
	double weight;

	public Apple(double w) {
		this.weight = w;
	}
}
```

当执行完两条add语句后，系统内存分配如下所示：



因为指向的对象相同，所以程序打印两个true。

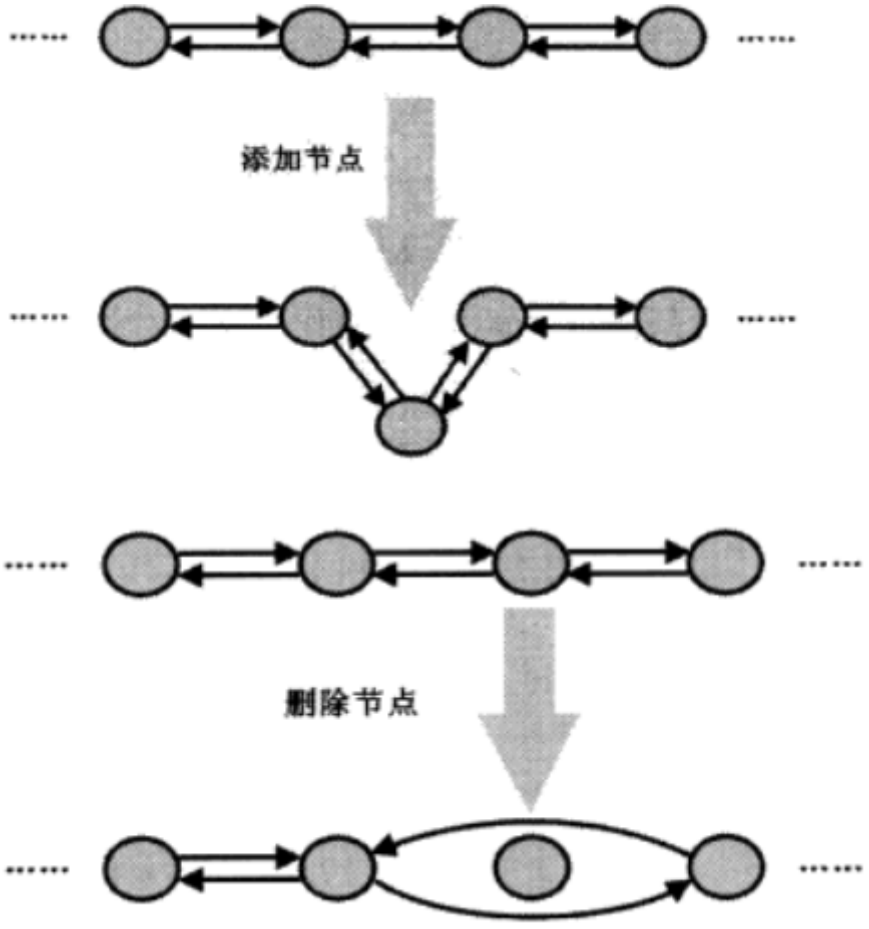
5、ArrayList和LinkedList的性能分析和使用场景

ArrayList是一种顺序存储的线性表，底层采用数组来保存每个集合的元素。LinkedList则是一种链式存储的线性表，其本质是一个双向链表，它不仅实现了List接口，还实现了

Deque接口。也就是说LinkedList既可以当双向链表使用，也可以当队列使用，可以当成栈使用。（Deque代表双端队列，既具有队列的特征，又具有栈的特征）

对于ArrayList而言，当程序向集合中添加、删除元素时，ArrayList底层都需要对数组进行“整体搬家”，如果添加元素导致集合长度将要超过底层数组长度，ArrayList必须创建一个长度为原来长度1.5倍的数组，再由垃圾回收机制回收原有数组，因此性能非常差。但如果程序调用get(int index)方法取出集合中的元素时，性能和数组几乎相同——非常快！

由于LinkedList采用双向链表保存集合元素，因此在添加集合元素的时候，只要对链表进行如下图所示的操作即可添加一个新的节点。因此LinkedList可以非常方便地在指定节点之前插入新节点，但若要在特定位置index处插入元素，则LinkedList必须一个一个元素地搜索，直到找到第index个元素为止。即便如此，当程序调用add(int index,Object obj)时，LinkedList方法的性能依然高于ArrayList。



就经验来讲，ArrayList的性能总体上由于LinkedList。因此绝大多数场景考虑使用ArrayList集合，但如果程序要经常添加、删除元素，尤其是需要调用add(E e)方法向集合中添加元素时，则应该考虑使用LinkedList。

五、Java的内存回收

1、对象的状态转换

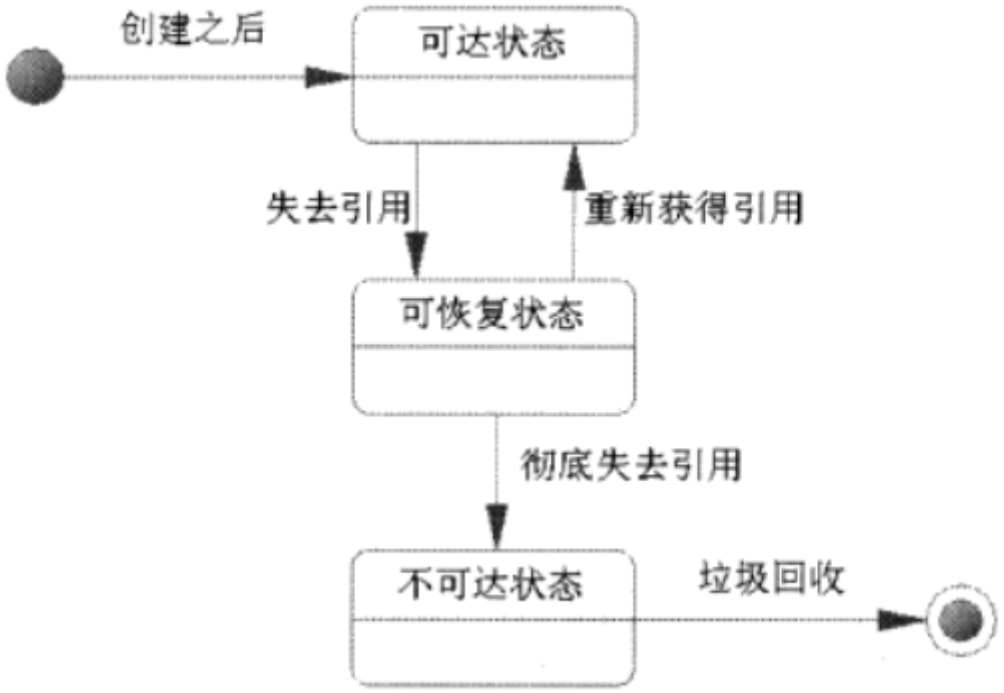
只要有引用变量引用一个对象，这个对象就不会被垃圾回收机制回收。JVM的垃圾回收机制采用有向图方式来管理内存中的对象，因此可以方便地解决循环引用的问题，比如3个对象相互引用。采用有向图管理内存中的对象具有较高的精度，但缺点是效率低。

当一个对象在堆内存中运行时，根据它在对应有向图中的状态，可以把它所处的状态分为3种：

可达状态：对象被创建后有一个以上的引用变量引用它，在有向图中可从起始顶点导航到该对象，则处于可达状态。

可恢复状态：程序中某个对象不再有任何引用变量引用它，它将进入可恢复状态，此时从有向图的起始点不能导航到该对象。在回收该对象之前，系统会调用可恢复状态的对象的finalize方法进行资源清理，如果系统在调用此方法重新让一个以上引用变量引用该对象，则这个对象会再次变为可达状态；否则，该对象将进入不可达状态。

不可达状态：当对象的所有关联被切断，且系统调用所有对象的finalize方法依然没有使该对象变成可达状态，那这个对象将永久性失去引用，最有变成不可达状态。只有当一个对象处于不可达状态时，系统才会真正回收该对象所占有的资源。



这里需要注意，一个对象可以被一个方法局部变量所引用，也可以被其他类的类变量引用，或者被其他对象的实例变量所引用。当某个对象被其他类的类变量引用时，只有该类被销毁后，该对象才会进入可恢复状态；当某个对象被其他对象的实例变量引用时，只有当引用该对象的对象被销毁或变成不可达状态后，该对象才会进入不可达状态。

2、四种对象的引用方式（强引用、软引用、弱引用、虚引用）

强引用：广泛使用的引用类型，被强引用所引用的Java对象不会被垃圾回收机制回收，即使系统内存非常紧张；即使有些Java对象以后永远都不会被用到，JVM也不会回收被强引用所引用的Java对象。另外，定义字符串时，系统会缓存这个字符串直接量（会使用强引用来引用它），系统不会回收被缓存的字符串常量。

软引用：需要通过SoftRefrence类来实现，当一个对象只具有软引用时，它有可能被垃圾回收机制回收。对于只有软引用的对象而言，当系统内存空间足够时，它不会被系统回收，程序也可使用该对象；当系统内存空间不足时，系统将会回收它。

弱引用：与软引用相似，区别在于弱引用所引用对象的生存期更短。弱引用通过WeakRefrence来实现，弱引用和软引用很像，但弱引用的引用级别更低。对于只有弱引用的对象而言，当系统垃圾回收机制运行时，不管系统内存是否足够，总会回收该对象所占用的内存。当然，比不是立即回收，而是等到系统垃圾回收机制运行时才会被回收。所以，弱引用具有很大的不确定性，因为垃圾回收机制的运行不受程序员控制，因此程序获取弱引用所引用的Java对象时必须小心空指针异常——通过弱引用所获取的Java对象可能是null。

虚引用：软引用和弱引用可以单独使用，但虚引用不能单独使用，必须和引用队列联合使用。虚引用的主要作用就是跟踪对象被垃圾回收的状态，程序可以通过检查与虚引用关联的引用队列中是否已经包含指定的虚引用，从而了解虚引用所引用的对象是否即将被回收。因为虚引用的对象被释放前，将把它对应的虚引用添加到它的关联的引用队列中，这使得可以在对象被回收之前采取行动。

使用这些引用类可以避免在程序执行期间将对象留在内存中，如果以软引用、弱引用或虚引用的方式引用对象，垃圾回收器就能够随意释放对象。如果希望尽可能减小程序在其生命周期中所占用的内存大小，这些引用类就很有好处。

3、Java的内存泄漏

定义：存在无用的内存没有被回收回来。比如程序中有一些Java对象，他们处于可达状态，但程序以后永远都不会再访问它们，那它们所占用的内存空间也不会被回收，它们所占用的空间也会产生内存泄漏。

例如：ArrayList中要删除最后一个元素，此时index会减一，如果不执行将删除位置的引用变量置为null，那么被删除的对象将一直处于可达状态，而对ArrayList而言永远不会访问到它，垃圾回收机制又不会回收它，这就产生了内存泄漏。

4、垃圾回收机制

垃圾回收机制主要完成两件事：

跟踪并监控每个Java对象，当某个对象处于不可达状态时，回收该对象所占用的内存；
清理内存分配、回收过程中产生的内存碎片。
5、内存管理的小技巧
（1）尽量使用直接量，少用new。

（2）使用StringBuilder和StringBuffer进行字符串拼接。

（3）尽早释放无用对象的引用。

（4）尽量少使用静态变量。因为类的静态成员变量，其生命周期与类同步，在类不被卸载的情况下，类对应的Class对象会常驻内存，直到程序运行结束，其静态成员变量也同样如此。如下程序：

```
class Person{
    static Object obj = new Object();
}
```

根据前面介绍的分代回收机制，JVM会将程序中Person类的信息存入Permanent代，也就是说，Person类、obj引用变量都将存放在Permanent代里，这将导致obj对象一直有效，从而使得obj所引用的Object得不到回收。

（5）避免在经常调用的方法、循环中创建Java对象

（6）缓存经常使用的对象

经常被使用的对象可以考虑用缓存池保存起来，当下次需要时就可以直接拿出来使用。典型的缓存就是数据库连接池，里边缓存了大量的数据库连接，每次程序需要访问数据库时都可以直接取出数据库连接。

实现缓存时通常有两种方式：

使用HashMap进行缓存；
直接使用某些开源的缓存项目。
（7）尽量不要使用finalize方法

前边介绍，一个对象失去引用之后，垃圾回收器准备回收该对象前会先调用对象的finalize方法来执行资源清理。所以很多程序员会考虑用此方法进行资源清理。但事实上这样做是非

常拙劣的选择，在垃圾回收器本身已经严重制约应用程序性能的情况下，如果再选择使用finalize方法进行资源清理，无疑是一种火上浇油的行为，这样会导致回收器的负担更大，导致程序运行效率更差。

（8）考虑使用SoftReference

当程序要创建长度很大的数组时，可以考虑使用SoftRefrence来包装数组元素，而不是直接让数组元素来引用对象。SoftRefrence是个很好的选择：当内存足够时，它的功能等同于普通引用；当内存不足时，它会牺牲自己，释放软引用所引用的对象。所以，程序取出SoftRefrence所引用的Java对象之后，应该显示判断该对象是否为null，以考虑是否要重建该对象。

六、表达式中的陷阱

1、StringBuilder和StringBuffer

进行字符串拼接时，通常应该优先考虑StringBuilder，StringBuilder和StringBuffer唯一的区别在于，StringBuffer是线程安全的，也就是说StringBuffer类里绝大数方法都增加了synchronized修饰符，以使得保证该方法线程安全，但会降低该方法的执行效率。在没有多线程的环境下，应该优先使用StringBuilder来表示字符串。

2、复合赋值运算符的陷阱

复合赋值运算符包含了一个隐式的类型转换，也就是说，下面两句并不等价。

```
a = a + 5;
a += 5;
```

实际上，a += 5;等价于a = (a的类型)(a + 5);，也就是说，复合赋值运算符会自动将它计算的结果值强制类型转换成其左侧变量的的类型。如果结果的类型与该变量的类型相同，那么这个转型不会造成任何影响。如果结果值的类型比该变量的类型要大，那么复合赋值运算符将会执行一次强制类型转换，这个强制类型转换将有可能导致高位“截断”，如下程序：

```
public class CompositeAssign {

    public static void main(String[] args) {
        short st = 5;
        st += 10;// 没有任何问题，系统执行隐式的类型转换
        System.out.println(st);
        st += 90000;// 出现问题，隐式类型转换，将会发生溢出
        System.out.println(st);
```

```
    }
}
```

执行第一个+=后，st的值是15，执行第二个+=会引起高位“截断”，第二个+=行代码相当于：

st = (short)(st + 90000);，即st = (short)90015;。问题是：short类型的变量只能接受-32768—32767之间的整数，因此上面程序会将90015高位“截断”，程序最后输出是24479。

容易导致计算结果被高位“截断”的情况有以下几种，需要注意：

将复合赋值运算符运用于btye，short或char等类型的变量。
将复合赋值运算符运用于int类型的变量，而表达式右侧是long、float、double类型的值。
将复合赋值运算符运用于float类型的变量，而表达式右侧是double类型的值。
3、多线程的陷阱
（1）线程的创建和启动

从JDK1.5开始，Java提供了3种方式创建、启动多线程：

继承Thread类来创建线程类，重写run()方法作为线程执行体；
实现Runnable接口来创建线程类，重写run()方法作为线程执行体；
实现Callable接口来创建线程类，重写call()方法作为线程执行体。
其中，第一种方式最差，它有两种不足：

线程类继承了Thread类，无法继承其他类。
因为每条线程都是一个Thread子类的实例，因此多个线程之间共享数据比较麻烦。
对于第二种、第三种方式，它们的本质是一样的，只是Callable接口里包含的call()方法既可以声明抛出异常，也可以拥有返回值。