## 高效java笔记

### 一、避免创建不必要的对象

**1. 最好能重用对象而不是在每次需要的时候就创建一个相同功能的新对象。**

如果对象是不可变的(immutable)，它就始终可以被重用。如下示例：

String s= new String("stringette"};//DON'T DO THIS!

该语句每次被执行的时候都创建一个新的String实例，但是这些创建对象的动作全都是不必要的。传递给String构造器的参数(“stringette”)本身就是一个String实例，功能方面等同与构造器创建的所有对象。如果这种用法是在一个循环中，或者是在一个被频繁调用的方法中。就会创建出成下千上万不必要的String实例。直接写成如下形式：String s = “stringette”。

**2. 要优先使用基本类型而不是装箱基本类型，无意识的自动装箱。**

有一种创建多余对象的新方法，称作自动装箱（autoboxing），它允许程序员将从本类型和装箱基本类型混用，按需要自动装箱和拆箱。自动装箱使得基本类型和装箱基本类型之间的差别变得模糊起来，但是并没有完全消除。它们在语义上还有着微妙的差别，在性能上也有着比较明显的差别。考虑下面的程序，它计算所有sum的总和。

Public static void main(String[] args) {

Long sum = 0L;

for(long i = 0; i < Integer.MAX\_VALUE; i++) {

Sum += i;

}

System.out.println(sum);

}

这段程序结论没有问题，但是比实际情况要更慢一些，只因为sum被声明成Long而不是long，意味着程序构造了大约231个多余的Long实例。将sum的声明从Long改成long ，运行时间从43秒减少到6.8秒。

**3. 不应该依赖终结方法来更新重要的持久状态。**

为类提供终结方法，可能会随意地延迟其实例的回收过程。调试一个长期运行的GUI应用程序的时候，该应用程序出现OutOfMemoryError错误而死掉。分析表明，该应用程序死掉的时候，其终结方法队列中有数千个图形对象正在等待被终结和回收。但是，终结方法线程的优先级比该应用程序的其他线程的要低得多。所以，图形对象的终结速度达不到它们进入队列的速度。Java语言规范并不保证哪个线程将会执行终结方法，所以，除了不使用终结方法之外，并没有很轻便的办法能够避免这样的问题。

### 二、覆盖equals时需要遵循的约定

1 .**类的每个实例本质上都是唯一的。**对厂代表活动实体而不是值(value)的类来说确实如此，如Thread。Object提供的equals实现对于这些类来说正是正确的行为。

**2.不关心类是否提供了“逻辑相等”的测试功能。**例如，java.util.Random覆盖了equals，以检查两个Random实例是否产生相同的随机数序列，在这样的情况下，从Object继承得到的equals实现已经足够了。

**3.超类已经覆盖了equals，从超类继承过来的行为对于子类也是合适的。**例如，人多数的Set实现都从AbstractSet继承equals实现，List实现从AhstractList继承equals实现。

**4.类是私有的或是包级私有的，可以确定它的equals方法永远不会被调用。**在这种情况下，无疑是应该覆盖equals方法的，以防它被意外调用。

**equals方法实现的等价关系：**

1)自反性(reflexive )，对于任何非null的引用值x， x.equals(x]须返回true。

2)对称性(symmetric)，对于任何非null的引用值x和y，当且仅当y.equals(x)返回true时，x.equals(y)必须返回true。

3)传递性(transitive)。对于任何非null的引用值x，y和z，如果x.equals(y)返回 true，并且y.equals(c)也返回true，那么x.equals(z)也必须返true。

4)一致性(consistent)。对于任何非null的引用值x和y，只要equals的比较操作在对象中所用的信息没有被修改，多次调用x.equals(y)就会一致地返回true，或者一致地返回false。

**实现高质量equals的需要注意的问题：**

1)使用==操作符检查“参数是否为这个对象的引用”。

2)使用instan}eof操作符检查“参数是否为正确的类型”。

3)把参数转换成正确的类型。因为转换之前进行过instancenf测试，所以确保会成功。

4)对于类中的每个”关键(significant)"域，检查参数中的域是否与该对象中对应的域相匹配。

确保比较的对象是对称的、传递的、一致的。

### 三、使可变性最小化

为了使类成为不可变，要遵循下面五条规则:

**1. 不要提供任何会修改对象状态的方法。**

**2. 保证类不会被扩展。**这样可以防止粗心或者恶意的子类假装对象的状态已经改变，从而破坏该类的不可变行为。为了防止子类化，一般做法是使这个类成为final的。

**3. 使所有的域都是final的。**通过系统的强制方式，可以清楚地表明你的意图。而且，如果一个指向新创建实例的引用在缺乏同步机制的情况下，从一个线程被传递到另一个线程，就必需确保正确的行为。

**4. 使所有的域都成为私有的。**这样可以防止客户端获得访问被域引用的可变对象的权限，并防止客户端直接修改这些对象。

**5. 确保对于任何可变组件的互斥访问。**如果类具有指向可变对象的域，则必须确保该类的客户端无法获得指向这些对象的引用。并且，永远不要用客户端提供的对象引用来初始化这样的域，也不要从任何访问方法中返回该对象引用。

### 四、接口优于抽象类

现有的类可以很容易被更新，以实现新的接口。如果这些方法尚不存在，需要做的就只是增加必要的方法。然后在类的声明中增加一个implements子句。

接口是定义mixin(混合类型)的理想选择。类除了实现它的“基本类型( primary type )”之外，还可以实现这个mixin类型，以表明它提供了某些可供选择的行为。接口允许我们构造非层次结构的类型框架。类型层次对于组织某些事情是非常合适的，但是其他有些事物并不能被整齐地组织成一个严格的层次结构。

### 作业

|  |
| --- |
| public class ThreadTest extends Thread {  private static int threadCount = 0;  private static final int MAX\_NUM = 30;  public ThreadTest(String name) {  this.setName(name);  }  @Override  public void run() {  while (threadCount < MAX\_NUM) {  switch (threadCount % 3) {  case 0:  if ("A".equals(this.getName())) {  printName();  }  break;  case 1:  if ("B".equals(this.getName())) {  printName();  }  break;  case 2:  if ("C".equals(this.getName())) {  printName();  }  break;  }  }  }  private void printName() {  System.out.print(getName());  threadCount++;  }  public static void main(String[] args) {  new ThreadTest("A").start();  new ThreadTest("B").start();  new ThreadTest("C").start();  }  } |