表达式中的陷阱

本文知识点来自于李刚《Java程序员的基本修养》第五章
Author: Yi
Date: Dec/27/2015

- 字符串陷阱
- 表达式类型陷阱
- 输入法导致的陷阱
- 注释字符必须合法
- 转义字符陷阱
- 泛型可能引起的陷阱
- 正则表达式陷阱
- 多线程陷阱

1:字符串陷阱

字符串是我们在程序中经常打交道的对象，但是往往字符串存在很多陷阱，让我们很是头痛，在这里借助这本书帮助我们扫盲一下常见的字符串陷阱。 Let’s Go!!

1. 先来看一句再普通不过的Java语句：

```
String str = new String("str");
```

常见问题是，上面语句创建了几个字符串对象。答案是两个：一个是字符串的直接量；一个是由new String构造器构造的字符串对象。在Java中，字符串的直接量JVM会用一个字符串池进行缓存，而且一般情况下字符串池中的对象不会被垃圾回收机制回收，当程序需要再次使用该字符串时，直接从字符串池中找出来重新使用就行。但值得注意的是，这个字符串必须是直接量赋值，而不能是new。这一点之前已经反复提及，这里不再说明。

再看一个新的例子：

```
public class Test {
    public static void main(String[] args) {
        String s1 = "abc";
        String s2 = "ab" + "c";
        System.out.println(s1 == s2);
    }
}
这个结果会输出true；原因在之前的“宏替换”笔记中已经有说明。当字符串的赋值操作没有涉及到方法的调用，而只是字符串连接，那么在编译阶段该字符串就会确定，JVM也会把这个字符串放在字符串池中。
```

还记得在第四章中，提到的内存优化小技巧中，对于基本数据类型的包装类，尽量使用直接量赋值，而不要用new方法，这样会保证较好性能。

2. 不可变字符串
   我们知道字符串是不可变的(immutable)。一个字符串被创建之后就不能再改变了。来看这个小例子：

```
public class Test {
    public static void main(String[] args) {
        String s1 = "a";
        s1 = s1 + "b";
        s1 = s1 + "c";
    }
}
看起来好像字符串是s1一直在改变，但是我们要清楚的认识到，s1只是一个引用变量而已，它指向的是存放在堆中的字符串对象，在上面连续的几次字符串连接过程中，改变的只是s1这个引用变量的指向对象而已，而实际上“a”和“ab”这两个字符串对象依然存在在堆中，但是没有任何引用指向它，但是又因为字符串是存放在字符串池中的，而一般情况下JVM不会回收存放在字符串池中的字符串，所以会造成内存泄漏。
```

既然字符串是不可变的，Java还提供了可变的字符串。这就是StringBuffer和StringBuilder。***StringBuffer和StringBuilder的区别是StringBuffer是线程安全，而StringBuilder不是线程安全。***

字符串比较：
字符串比较是我们在码码过程中经常用到的。如果我们要判定两个字符串对象是否相同，直接用`==`就可以了，但是如果我们要判定字符串的字符序列是否相同，就必须使用String重写的`equals()`方法。同样的字符串还可以比较大小，字符串比较大小的规则是，把比较的字符串按左对齐，然后按照字母表的顺序比较大小。 比如说：`abcx`和`ax`相比较，按左对齐，第二个字符x比b大，因此`ax`大于`abcx`。在字符串比较大小的过程中，调用的方法是compareTo()方法。

2:表达式陷阱

Java是一种强类型语言，其包含两个特性：

- 所有的变量必须声明，声明同时必须规定变量类型
- 一旦某个变量类型确定下来，那么它只能存储该类型的数据

  1. **Java表达式类型的自动提升**
     Java规定，当一个操作表达式中包含多种基本类型数据时，整个算术表达式的基本数据类型都会进行提升。 Java语言提升规则如下：

     - byte, char, short类型数据将会被提升到int
     - 整个算术表达式的基本数据类型都会被提升到表达式中最高级的基本数据类型.
       Java中基本数据类型等级是：
       double > float > long > int > short > byte
       double > float > long > int > char

  2.
     ```
     short var = 5;
     var = var - 2;
     在后一个语句中，var会自动提升到int数据类型，然后把一个int数据类型赋值给short数据类型，会造成精度损失，编译出错。在eclipse中实验得到，凡是高等级的赋值给低等级的变量，编译都会出错。
     ```

     因此下面这个例子会造成编译错误：

  3. **复合赋值运算符的陷阱**
     在上面我们举了一个数据类型提升编译出错到例子；如何避免：

     ```
     将
     short var = 5;
     var = var - 2;
     改为：
     short var = 5;
     var  -=  2;
     即可
     ```

- **因此复合赋值运算可能出现的错误，也就是陷阱：**

  ```
  short var = 10;
  var += 90000;
  在这种情况下，编译起会通过该语句的编译，但是由于short的取值范围是－32768～32767。因此var此时的值90015已经超出了short精度范围，因此强制类型转换会造成高位“截断”。
  ```

  因此我们知道`var = var - 2;`和`var -= 2;`并不等价。在Java中，复合赋值运算都包含了一种隐式的类型转换，也就是说`var -= 2;`等价于`var = (short)(var - 2);`。尽管我们知道复合赋值运算比较简洁而且在一定程度上保证了类型的一致性，但是我们也要注意到复合赋值运算存在的潜在危险，也就是高位“截断”。看个例子：

  - 当复合赋值运算运用于byte， short 和 char基本数据类型时，尤其注意高位“截断”；
  - 当复合赋值运算表达式的左边为int类型时，而右边是long，float和double数据类型时；
  - 当复合赋值运算表达式的左边为float类型时，而右边的数据类型是double数据类型

3:输入法陷阱

输入法导致的陷阱主要在于全角状态和半角状态的区别。因此如果在编译过程中出现非法字符的编译错误，那么找出程序中的全角符号，逐个删除即可。

4:注释字符的陷阱和转义字符的陷阱

一般来说在Java中注释不会造成错误，我们常以为程序会跳过注释部分。其实不然，程序不会跳过注释部分如果注释中包含非法字符。 在unicode码中，我们通常是可以使用`\uXXXX`来表示字符，而后面所跟的四个`X`取值范围是`“0～F”`。

5:泛型引起的陷阱

6:正则表达式陷阱

以上两种陷阱不经常见，在平时码代码过程中，我们也很少按照书本所说陷阱那样去操作。因此在此省略。

7:多线程陷阱

1. 不要调用run()方法
   这个错误我最近还犯过。一定要记得在多线程中，启动新线程的方法是调用start()而不是run()。如果调用run()，那么你只是在单纯的调用run()方法而已。
   Java提供了三种方法来实现多线程：

   - 继承Thread类，重写run()方法来执行线程主体；
   - 实现Runnable接口，重写run()方法来执行线程主体
   - 实现Callable接口，重写call()方法来执行线程主体

2. 第一种方案效果最差，因为在Java中只存在单继承模式，继承了Thread之后，你就不能再继承其他类了。第二种和第三种方案效果一致。
3. 静态方法的同步
   Java提供关键字synchronized来为方法提供同步机制。synchronized可以用来修饰方法也可以用来修饰代码块。被synchronized修饰的方法叫做同步方法，synchronized修饰的代码块叫做同步代码块。Java语言规定，任何线程进入同步方法或者同步代码块之前，必须获得同步方法或者同步代码块的同步监视器，也就是锁。对于同步代码块而言，程序员必须显示的制定同步监视器；对于非静态的方法，同步监视器监视的是this，也就是调用改方法的堆中的对象；而对于静态方法而言，同步监视的是这个类。
   看个例子：

   ```
   静态方法的synchronized关键字获取该类的锁
   public static synchronized void test1() {//Do something}
   非静态方法synchronized关键字获取方法调用对象本身的锁。
   public void test2() {
   synchronized(this) {
       //Do something
   }
   }
   获取类的锁和调用对象本身的锁是互不影响的，二者可以同时进行。
   ```

4. 多线程执行环境
   在单一线程中执行时，不会出现资源竞争，因而往往在单一线程中不会出现data corruption。但如果存在多线程对同一资源存在竞争的可能性，这种情况下就会出现意想不到的我们不希望发生的事情。因此在多线程环境下，对于那些可能存在资源竞争的资源修改方法，我们最好使用同步监视器把它监视起来，即上锁。