Java 基础（2）：自增自减与贪心规则

引言

JDK 中提供了自增运算符++，自减运算符--。这两个操作符各有两种使用方式：前缀式（++a,--a），后缀式（a++，a--）。可能说到这里，说不得有读者就会吐槽说，前后缀式都挺简单的，前缀式不就是先进行+1（或-1），然后再使用该值参与运算嘛，后缀式则相反。有必要长篇大论吗

前后缀式的区别确实是这样，最起码表面上理解起来是这样，但是更深入的理解就不是这么简单了，甚至严重影响到你的程序的正确性。不信，接下去看吧！

1.前缀式与后缀式的真正区别

在 Java 中 ,运算是从左往右计算的 ,并且按照运算符的优先级 ,逐一计算出表达式的值，并用这个值参与下一个表达式的运算，如：1+2+3，其实是先计算出 1+2 表达式的值为 3，再参与下一个表达式的运算(1+2)+3，即 3+3。再如判断 if(a+2==3)。如此类推。

a++是一个表达式，那么 a++就会有一个表达式的计算结果，这个计算结果就是 a 的旧值（加 1 前的值）。相对的，++a 表达式的计算结果 a 加 1 后的值。所以，**自增的前缀形式与后缀形式的本质区别是：表达式的值（运算结果） 是加 1 前的变量的值还是加 1 后的变量的值（自减也是如此）。并不是先加 1 与 后加 1 的区别，或者说，前后缀形式都是先加 1（减 1）的，才得到表达式的值，再参与下一步运算。因为这是一个表达式，必须先计算完表达式的运算，最后才会得到表达式的值。**

我们来看一个面试经常遇到的问题：

```
int a = 5;
a = a++;
//此时 a 的值是多少？
```

有猜到 a 的值吗？我们用上面所学到的分析一下：a=a++可以理解成以下几个步骤：

1) 计算 a 自加 1，即 a=a+1
2) 计算表达式的值，因为这是后缀形式，所以 a++表达式的值就是加 1 前 a 的值(值为 5)；
3) 将表达式的值赋值给 a，即 a=5。

所以最后 a 的值是 5。

同理，如果改成 a = ++a;，则 a 的值是 6。这是因为++a 表达式的值是加 1 后的 a，即为 6。

2.自增自减是包含两个两个操作，不是线程安全的

自增、自减运算符本质上不是一个计算操作，而是两个计算操作。以 a++为例，这个运算将会编译器解析成：a=a+1，即包含两个单目运算符（ +、=),一个单目运算符的计算操作可以看作是一个原子性操作。a++的步骤可以描述成 ：

1) 先取 a 加 1，将结果存储在临时空间；
2) 将结果赋给 a。所以，自增自减运算符包含两个操作：一个加 1（减 1）的操作和一个赋值的操作

这个原理好像对我的编程没有用吧？不是的 ,在单线程的环境下 ,你可以不管这个细节。但在多线程的情况下，你就得时刻牢记这个细节了。要知道，自增自减不是原子性操作，也就是说不是线程安全的运算。因此，在多线程下，如果你要对共享变量实现自增自减操作，就要加锁，或者使用 JDK 提供的原子操作类（如 AtomincInteger，AtomicLong 等）提供的原子性自增自减方。

来看个例子，验证一下。下面的例子提供三个静态变量（一个是原子操作类），创建了 10 个线程，每个线程都对这三个变量以不同的方式进行加 1 操作,并循环 1000 次。

```
public class MyTest {
    static int a = 0;
    static int b = 0;
    //原子性操作类
    static AtomicInteger atomicInt = new AtomicInteger(0);
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 10; i++) {//创建 10 个线程
            Thread t = new Thread() {
                @Override
                public void run() {
                    for (int j = 0; j < 1000; j++) {//计算 1000 次
                        a = a + 1;
                        b++;
                        atomicInt.incrementAndGet();//自增的原子性方法
                    }
                }
            };
            t.start();
        }
        // 判断当前的活动线程是不是只有 main 线程，以确保 10 个计算线程执行完成。
        while (Thread.activeCount() > 1) {
```

```
            Thread.sleep(1000);
        }
        System.out.println("a=a+1 在多线程下的结果是：" + a);
        System.out.println("b++在多线程下的结果是：" + b);
        System.out.println("原子操作类 AtomicInteger 在多线程下的结果是：" +
atomicInt.get());
    }
}
```

运行结果：

```
a=a+1 在多线程下的结果是：8883
b++在多线程下的结果是：8974
原子操作类 AtomicInteger 在多线程下的结果是：10000
```

从运行的结果可以看出 ,a=a+1、b++不是线程安全的 ,没有计算出正确的结果 10000。也就是说这两个表达式都不是原子性操作。事实上，它们都包含了两个计算操作。

3.由 a+++b 表达式引起的思考

看到这个表达式，真的很让人疑惑：编译器是怎么解析的，是解析成

```
a++ + b
```

还是

```
a+ ++b
```

真纠结，干脆直接在编译器上跑一趟，看看结果吧！

```
int a = 5;
int b = 5;
int c=a+++b;
System.out.println("a 的值是: "+a);
System.out.println("b 的值是: "+b);
```

运行结果：

```
a 的值是: 6
b 的值是: 5
```

从结果可以确认，a+++b 其实是解析成了 a++ +b，为什么要这样结合呢？其实有两点原因：

- Java 中的运算是从左往右进行的；
- java 编译器有一种规则——贪心规则。**也就是说，编译器会尽可能多地结合有效的符号。**

那么，a+++b 这样的结合方式就可以解释了
但是这种结合是：尽可能多的结合，而不管这样的结合是否合法。如：

```
a--b
```

会被编译器解析成

```
a-- b
```

尽管这是不合法，但编译器还是这样处理了，这就导致编译不通过，产生编译错误。

从上面的分析来看，贪心规则在编程中也不好利用。那么，贪心规则的主要目的是为了什么？

贪心规则的主要目的就是为了分析 String 字符串，看看下面的例子就会明白：

```
String s = "\17";
System.out.println("\\17 转义字符的值是："+s+"  长度是："+s.length());
s = "\171";
System.out.println("\\171 转义字符的值是："+s+"  长度是："+s.length());
s = "\1717";
System.out.println("\\1717 转义字符的值是："+s+"  长度是："+s.length());
s = "\17178";
System.out.println("\\17178 转义字符的值是："+s+"  长度是："+s.length());
```

运行结果：

```
\17 转义字符的值是：    长度是：1
\171 转义字符的值是：y    长度是：1
\1717 转义字符的值是：y7    长度是：2
\17178 转义字符的值是：y78    长度是：3
```

“\17” 经转义得到一个特殊字符 “” 。而“\171” 转义后也得到一个字符 “y”。但 “\1717”、“\17178” 得到的字符串大于 1 ,不再是一个字符 ,分别是 “y7”、“y78”。

也就是说,“\1717” 字符串只转义了“\171” 部分 ,再链接 “7” 部分。“\17178” 字符串只转义了 “\171” 部分,再连接 “78”。

那为什么 “\171” 为什么不转义 ” \17 “ 部分，再链接 ” 1 “ 呢，而是作为一个整体进行转义的字符串呢？

这就是 ” 贪心规则 “ 所决定的。八进制的转义字符的取值范围是 \0~\377。所以解析 ” \171 “ 字符串时，编译器尽可能多地结合字符成一个转移字符，” \171 “ 还在取值范围内，所以就是一个字符。但 ” \1718” 字符串，最多结合前 4 个字符成一个有效的转义字符 “\171” ,而 “\1717” 已经超出取值范围 ,不是有效字符 ,所以最后解析成 “\171” + "7" 的结果。“17178” 也是如此。

总结

- 编译器在分析字符时，会尽可能多地结合成有效字符，但有可能会出现语法错误。
- 贪心规则是有用的，特别编译器是对转义字符的处理。