**[【JDK优化】 Integer 自动打包机制的优化](http://www.iteye.com/topic/614440)**

我们首先来看一段代码：

**Java代码  [收藏代码](javascript:void())**

1. Integer i=100;
2. Integer j=100;
3. System.out.println(i==j);  //true
4. Integer i=200;
5. Integer j=200;
6. System.out.println(i==j);  //false

差不多的两段代码，怎么结果完全不同呢。我们分两步来说明这个问题：

首先 Integer i=100; 编译器会自动将int类型常数100包装成Interger，采用的是Integer.valueOf(100)； 这个方法。

然后我们看看valueOf(int)这个方法的源代码：

**Java代码  [收藏代码](javascript:void())**

1. /\*
2. \* 返回一个表示指定的 int 值的 Integer 实例。如果不需要新的 Integer 实例，则
3. \* 通常应优先使用该方法，而不是构造方法 Integer(int)，因为该方法有可能通过
4. \* 缓存经常请求的值而显著提高空间和时间性能。
5. \* @param  i an <code>int</code> value.
6. \* @return a <tt>Integer</tt> instance representing <tt>i</tt>.
7. \* @since  1.5
8. \*/
9. **public** **static** Integer valueOf(**int** i) {
10. **final** **int** offset = 128;
11. **if** (i >= -128 && i <= 127) { // must cache
12. **return** IntegerCache.cache[i + offset];
13. }
14. **return** **new** Integer(i);
15. }
17. /\*
18. \* IntegerCache内部类
19. \* 其中cache[]数组用于存放从-128到127一共256个整数
20. \*/
21. **private** **static** **class** IntegerCache {
22. **private** IntegerCache(){}
24. **static** **final** Integer cache[] = **new** Integer[-(-128) + 127 + 1];
26. **static** {
27. **for**(**int** i = 0; i < cache.length; i++)
28. cache[i] = **new** Integer(i - 128);
29. }
30. }

原来如此，当-128=<i<=127的时候，返回的是IntegerCache中的数组的值；当 i>127 或 i<-128 时，返回的是Integer类对象。 这就好解释：

**Java代码  [收藏代码](javascript:void())**

1. Integer i=100;
2. Integer j=100;
3. System.out.println(i==j);  //(1)

此时的 i=IntegerCache.cache[228]，因此 Integer引用i中存储的是cache数组第228号元素的地址。同理j也是同一个cache数组的第228号元素的地址(因为cache是Integer的静态数组，只有一个)。i==j比较的是引用地址，因此相等。

**Java代码  [收藏代码](javascript:void())**

1. Integer i=200;
2. Integer j=200;
3. System.out.println(i==j);  //(2)

此时的 i=new Integer(200);  同样j=new Integer(200) 。两次都在堆中开辟了Integer的对象。i 和 j 中存储的堆得对象地址是完全不同的。i==j 自然不相等了。

那么这样做有什么意义呢？ 我们来看看API的解释：

**Java代码  [收藏代码](javascript:void())**

1. 返回一个表示指定的 **int** 值的 Integer 实例。如果不需要新的 Integer 实例，
2. 则通常应优先使用该方法，而不是构造方法 Integer(**int**)，因为该方法有可能
3. 通过缓存经常请求的值而显著提高空间和时间性能。

假如我们在编程时大量需要值为100的Integer对象，如果只能通过new来创建的话。是不是需要在堆中开辟大量值一样的Integer对象呢！这是相当不划算的。既然如此，Java中的字符串常量池的应用是不是可以提醒我们点什么呢？是的，IntegerCache.cache就相当于这样一个字符串常量池。 当我们需要Integer i=100的时候，直接从cache中取出第[100+128]号元素的地址赋值引用i，再次需要Integer j=100时，还是直接去这个地址赋给j ..... 是不是省去了再堆中不停的创建对象的代价了(空间，时间上的消耗都很大)。