java中，基本数据类型一共有8种，详细信息如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 大小 | 范围 | 默认值 |
| byte | 8 | -128 - 127 | 0 |
| short | 16 | -32768 – 32767 | 0 |
| int | 32 | -2147483648-2147483647 | 0 |
| long | 64 | -9233372036854477808-9233372036854477807 | 0 |
| float | 32 | -3.40292347E+38-3.40292347E+38 | 0.0f |
| double | 64 | -1.79769313486231570E+308-1.79769313486231570E+308 | 0.0d |
| char | 16 | \u0000 - u\ffff | \u0000 |
| boolean | 16 | true/false | false |

Java语言是一种面向对象的语言，但是Java中的基本数据类型却是不面向对象的，这在实际使用时存在很多的不便，为了解决这个不足，设计者将每个基本数据类型单独封装成一个类，这八个和基本数据类型对应的类统称为包装类（Wrapper Class）。

|  |  |  |
| --- | --- | --- |
| INDEX | 基本数据类型 | 包装类 |
| 1 | int | Integer |
| 2 | char | Character |
| 3 | float | Float |
| 4 | double | Double |
| 5 | byte | Byte |
| 6 | short | Short |
| 7 | long | Long |
| 8 | boolean | Boolean |

## 内置数据类型

Java语言提供了八种基本类型。六种数字类型（四个整数型（默认是int 型），两个浮点型（默认是double 型）），一种字符类型，还有一种布尔型。

byte：

* byte数据类型是8位、有符号的，以二进制补码表示的整数；（256个数字），占1字节
* 最小值是-128（-2^7）；
* 最大值是127（2^7-1）；
* 默认值是0；
* byte类型用在大型数组中节约空间，主要代替整数，因为byte变量占用的空间只有int类型的四分之一；
* 例子：byte a = 100，byte b = -50。

short：

* short数据类型是16位、有符号的以二进制补码表示的整数，占2字节
* 最小值是-32768（-2^15）；
* 最大值是32767（2^15 - 1）；
* Short数据类型也可以像byte那样节省空间。一个short变量是int型变量所占空间的二分之一；
* 默认值是0；
* 例子：short s = 1000，short r = -20000。

int：

* int数据类型是32位、有符号的以二进制补码表示的整数；占3字节
* 最小值是-2,147,483,648（-2^31）；
* 最大值是2,147,485,647（2^31 - 1）；
* 一般地整型变量默认为int类型；
* 默认值是0；
* 例子：int a = 100000, int b = -200000。

long：

* long数据类型是64位、有符号的以二进制补码表示的整数；占4字节
* 最小值是-9,223,372,036,854,775,808（-2^63）；
* 最大值是9,223,372,036,854,775,807（2^63 -1）；
* 这种类型主要使用在需要比较大整数的系统上；
* 默认值是0L；
* 例子： long a = 100000L，int b = -200000L。

long a=111111111111111111111111(错误，整数型变量默认是int型)

long a=111111111111111111111111L(正确，强制转换)

float：

* float数据类型是单精度、32位、符合IEEE 754标准的浮点数；占4字节    -3.4\*E38- 3.4\*E38。。。浮点数是有舍入误差的
* float在储存大型浮点数组的时候可节省内存空间；
* 默认值是0.0f；
* 浮点数不能用来表示精确的值，如货币；
* 例子：float f1 = 234.5f。
* float f=6.26(错误  浮点数默认类型是double类型)
* float f=6.26F（转换正确，强制）
* double d=4.55(正确)

double：

* double数据类型是双精度、64位、符合IEEE 754标准的浮点数；
* 浮点数的默认类型为double类型；
* double类型同样不能表示精确的值，如货币；
* 默认值是0.0d；
* 例子：double d1 = 123.4。

boolean：

* boolean数据类型表示一位的信息；
* 只有两个取值：true和false；
* 这种类型只作为一种标志来记录true/false情况；
* 默认值是false；
* 例子：boolean one = true。

char：

* char类型是一个单一的16位Unicode字符；用 ‘’表示一个字符。。java 内部使用Unicode字符集。。他有一些转义字符  ，2字节
* 最小值是’\u0000’（即为0）；
* 最大值是’\uffff’（即为65,535）；可以当整数来用，它的每一个字符都对应一个数字

**1.什么是自动拆箱与自动装箱**

自动装箱：把基本类型用它们对应的包装类包装起来，使它们具有对象的特质，可以调用所对应的包装类所定义的方法，**比如toString()等。**

举个例子：

Integer i0 = new Integer(0);

Integer i1 = 2;

Integer i1\_ = Integer.valueOf(2);

上面的三行代码第一行是最基本的创建一个integer对象的方式。第二行代码就是我们这里要讲的自动装箱。而第三行代码就是第二行代码的本质，也就是说，当你使用自动装箱来得到一个引用数据类型时，jvm实际上调用了valueOf()方法，稍后我们会去研究一下java源码。

自动拆箱：跟自动装箱的方向相反，将Integer及Double这样的包装类的对象重新简化为基本类型的数据。

举个例子：

1.System.out.println(i1+2);

这句代码就使用了自动拆箱。i1是我们上面通过自动装箱得到的一个integer对象，而**这个对象是不能直接进行四则运算的**，但是我们却给它+2，这样就必须将integer对象转变为基本数据类型（int），**这个过程就是自动拆箱的过程**。

p.s.所谓自动，就是说这个过程并不需要程序员去完成，而是jvm自动完成的，jvm会在编译期根据语法决定是否进行装箱和拆箱动作。   
另外，自动拆箱与自动装箱的jdk1.5才引入的新特性，所以如果你的jdk版本低于1.5的话，是不可以这样写的。

**2.为什么会有自动拆箱与自动装箱**

为什么java要提供这样一个功能呢？我的理解是这样的：   
1.因为懒。假如没有自动拆箱与自动装箱，那么我们的代码是这样的：

Integer i = new Integer(2);//假如需要一个integer的对象i，值为2

int b=i.intValue();//又需要一个int型的值，大小与i相等

但是，有了自动拆箱与装箱，我们就可以这么写：

**Integer i = 2;**

**int b = i;**

是不是省了不少事，而且看起来代码更简洁了呢？

2.**自动装箱的过程其实可以起到节约内存的作用**。我们先看一个例子：

Integer a = 1;

Integer b = 1;

Integer c = 144;

Integer d = 144;

Integer a1 = new Integer(1);

Integer b1 = new Integer(1);

System.out.println(a == b); //true

System.out.println(a.equals(b)); //true

System.out.println(a1 == b1); //false

System.out.println(a1.equals(b1)); //true

System.out.println(c == d); //false

System.out.println(c.equals(d)); //true

是不是很奇怪，为什么第7行为true而第12行为false呢?这是因为，在**自动装箱时对于值从–128到127之间的值，它们被装箱为Integer对象后，会存在内存中被重用，始终只存在一个对象（享元模式）**。而如果超过了从–128到127之间的值，被装箱后的Integer对象并不会被重用，即相当于每次装箱时都新建一个 Integer对象。

那么，为什么要这么设计呢？一般来说，**小数字的使用频率很高，将小数字保存起来，让其始终仅有一个对象可以节约内存，提高效率。**

这其实用到了一种叫做享元模式的设计模式，感兴趣的可以去研究一下这个设计模式。

**3.怎么使用自动拆箱与自动装箱（+自动装箱池）**

使用方式通过上面的例子大家应该也都清楚了，自动拆箱与装箱实际上就是jvm帮我们去调用一些函数，这样可以使我们省不少事，代码也会看起来更简洁一些，不过在这里还有一点需要强调，先看代码：

Integer a = null;

int b = a;

这么写完全是符合java语法规范的，编译也可以正常通过，但是很明显，运行的时候回抛出空指针异常。所以在这里提醒大家，在使用自动拆箱时，一定要确保包装类的引用不为空。

**4.源码剖析**

上面提到了几个包装类的方法，我们一Integer类为例，来看一看java源码是什么样子的。首先是valueOf()方法：

public static Integer valueOf(int i) {

if (i >= IntegerCache.low && i <= IntegerCache.high)

// 没有设置的话，IngegerCache.high 默认是127

return IntegerCache.cache[i + (-IntegerCache.low)];

return new Integer(i);

}

上面讲到，在自动装箱时对于值从–128到127之间的值，它们被装箱为Integer对象后，会存在内存中被重用。现在明白是为什么了吧，在调用valueOf()方法的时候，会判断你所给的数是不是在**IntegerCache.low 和 i <= IntegerCache.high之间，如果是，那么他就在内存中生成唯一的对象**，当你第二次想要生成它的时候，他会把第一次所生成对象的地址给你，不会重新生成。**而不在这个范围里的数，你每次所生成的对象都是不同的。**

自动装箱池的大小是怎么定义的呢，Integer.java中有这样一个内部类

private static class IntegerCache {

static final int low = -128;

static final int high;

static final Integer cache[];

static {

// high value may be configured by property

int h = 127;

String integerCacheHighPropValue =

sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");

if (integerCacheHighPropValue != null) {

try {

int i = parseInt(integerCacheHighPropValue);

i = Math.max(i, 127);

// Maximum array size is Integer.MAX\_VALUE

h = Math.min(i, Integer.MAX\_VALUE - (-low) -1);

} catch( NumberFormatException nfe) {

// If the property cannot be parsed into an int, ignore it.

}

}

high = h;

cache = new Integer[(high - low) + 1];

int j = low;

for(int k = 0; k < cache.length; k++)

cache[k] = new Integer(j++);

// range [-128, 127] must be interned (JLS7 5.1.7)

assert IntegerCache.high >= 127;

}

private IntegerCache() {}

}

IntegerCache类（这个是jdk1.8的源码）定义了Integer自动装箱池的大小。从源码中我们可以看到，**下界是写死的**，就是-128，但是上界却是由参数integerCacheHighPropValue解码得来的，这就表明，其实我们可以通过改变integerCacheHighPropValue值的大小来自定义自动装箱池的大小，当然，一般没人会去改它。

**Integer自动装箱池的范围是-128~127   
Byte,Short,Long范围是-128~127   
Character范围是0~127   
Float和Double没有自动装箱池**

**就是在此范围内，==是两个对象是相等的。**

**5.总结**

Java通过自动装箱和拆箱的机制，节省了部分内存开销和创建对象的开销，提高了效率同时简化了代码。在使用该机制的时候，需要注意以下几点：   
1.在进行==比较的时候，在自动装箱池范围内的数据的引用是相同的，范围外的是不同的。   
2。在自动拆箱时，要保证包装类的引用不为空。