[一、数据类型](#4852-1585320553745)

[1、基本类型](#5045-1585319739976)

[基本数据类型的存储方式—存在栈中](#3611-1585321487017)

[基本数据类型的存储方式—存在堆中](#8444-1585321708868)

[2、包装类型（引用类型）](#2025-1585320310585)

[自动拆箱与装箱](#1068-1585320595185)

[自动拆箱与装箱实现过程](#5345-1585320763368)

[3、缓存常量池](#2121-1585321019576)

**一、数据类型**

**1、基本类型**

八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型

byte/8：-2^7—2^7-1，默认值0，用在大型数组中节约空间，主要代替整数，因为该变量占用的空间只有 int 类型的四分之一

char/16：\u0000（即为0）—\uffff（即为65,535），char 数据类型可以储存任何字符

short/16：-2^15—2^15-1，默认值0，一个short变量是int型变量所占空间的二分之一

int/32：-2^31—2^31-1，默认值0

float/32：单精度，默认值0.0f，储存大型浮点数组的时候可节省内存空间，浮点数不能用来表示精确的值

long/64：-2^63—2^63 -1，默认值0L

double/64：双精度，默认值0.0d，浮点数的默认类型为double类型

boolean/~：true 和 false，默认值false

boolean 只有两个值：true、false，可以使用 1 bit 来存储，但是具体大小没有明确规定。JVM 会在编译时期将 boolean 类型的数据转换为 int，使用 1 来表示 true，0 表示 false。JVM 支持 boolean 数组，但是是通过读写 byte 数组来实现的。

**基本数据类型的存储方式—存在栈中**

public void(int a) { int i = 1; int j = 1; } 
方法中的i 存在虚拟机栈的局部变量表里，i是一个引用，j也是一个引用，它们都指向局部变量表里的整型值 1. int a是传值引用，所以a也会存在局部变量表。

**基本数据类型的存储方式—存在堆中**

class A{
int i = 1;
A a = new A();
}
i是类的成员变量。类实例化的对象存在堆中，所以成员变量也存在堆中。引用a存的是对象的地址，引用i存的是值，这个值1也会存在堆中。
基本数据类型的包装类型可以在常量池查找对应值的对象，找不到就会自动在常量池创建该值的对象。

[java] view plain copy
String s = new String("1");
s.intern();
String s2 = "1";
System.out.println(s == s2);
String s3 = new String("1") + new String("1");
s3.intern();
String s4 = "11";
System.out.println(s3 == s4);
输出结果为：
[java] view plain copy
JDK1.6以及以下：false false
JDK1.7以及以上：false true
JDK1.6查找到常量池存在相同值的对象时会直接返回该对象的地址。
JDK 1.7后，intern方法还是会先去查询常量池中是否有已经存在，如果存在，则返回常量池中的引用，这一点与之前没有区别，
区别在于，如果在常量池找不到对应的字符串，则不会再将字符串拷贝到常量池，而只是在常量池中生成一个对原字符串的引用。

**2、包装类型（引用类型）**

基本类型都有对应的包装类型，基本类型与其对应的包装类型之间的赋值使用自动装箱与拆箱完成。

对象、数组都是引用数据类型

所有引用类型的默认值都是null

一个引用变量可以用来引用任何与之兼容的类型

Integer x = 2; // 装箱 调用了 Integer.valueOf(2)
int y = x; // 拆箱 调用了 X.intValue()

**自动拆箱与装箱**

Java 5增加了自动装箱与自动拆箱机制，方便基本类型与包装类型的相互转换操作。在Java 5之前，如果要将一个int型的值转换成对应的包装器类型Integer，必须显式的使用new创建一个新的Integer对象，或者调用静态方法Integer.valueOf()

//在Java 5之前，只能这样做
Integer value = new Integer(10);
//或者这样做
Integer value = Integer.valueOf(10);
//直接赋值是错误的
//Integer value = 10;`
//在Java 5中，直接赋值是合法的，由编译器来完成转换`
Integer value = 10;
//在Java 5 中可以直接这么做`
Integer value = new Integer(10);`
int i = value;`

**自动拆箱与装箱实现过程**

//在自动装箱时，编译器调用包装类型的valueOf()方法；在自动拆箱时，编译器调用了相应的xxxValue()方法
class Auto
{
public static void main(String[] paramArrayOfString)
{
Integer localInteger = Integer.valueOf(10);​
​ int i = localInteger.intValue();​
​ Double localDouble = Double.valueOf(12.4D);​ double d = localDouble.doubleValue();​
}
}

**3、缓存常量池**

//Integer源码
public final class Integer extends Number implements Comparable<Integer> {
private final int value;
/\*Integer的构造方法，接受一个整型参数,Integer对象表示的int值，保存在value中\*/
public Integer(int value) {
this.value = value;
}
/\*equals()方法判断的是:所代表的int型的值是否相等\*/
public boolean equals(Object obj) {
if (obj instanceof Integer) {
return value == ((Integer)obj).intValue();
}
return false;
}
/\*返回这个Integer对象代表的int值，也就是保存在value中的值\*/
public int intValue() {
return value;
}
/\*\*
\* 首先会判断i是否在[IntegerCache.low,Integer.high]之间
\* 如果是，直接返回Integer.cache中相应的元素
\* 否则，调用构造方法，创建一个新的Integer对象
\*/
public static Integer valueOf(int i) {
assert IntegerCache.high >= 127;
if (i >= IntegerCache.low && i <= IntegerCache.high)
return IntegerCache.cache[i + (-IntegerCache.low)];
return new Integer(i);
}
/\*\*
\* 静态内部类，缓存了从[low,high]对应的Integer对象
\* low -128这个值不会被改变
\* high 默认是127，可以改变，最大不超过：Integer.MAX\_VALUE - (-low) -1
\* cache 保存从[low,high]对象的Integer对象
\*/
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) {
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);
}
high = h;
cache = new Integer[(high - low) + 1];
int j = low;
for(int k = 0; k < cache.length; k++)
cache[k] = new Integer(j++);
}
private IntegerCache() {}
}

1）Integer有一个实例域value，它保存了这个Integer所代表的int型的值，且它是final的，也就是说这个Integer对象一经构造完成，它所代表的值就不能再被改变。

2）Integer重写了equals()方法，它通过比较两个Integer对象的value，来判断是否相等。

3）重点是静态内部类IntegerCache，通过类名就可以发现：它是用来缓存数据的。它有一个数组，里面保存的是连续的Integer对象。

(a) low：代表缓存数据中最小的值，固定是-128。

(b) high：代表缓存数据中最大的值，它可以被该改变，默认是127。high最小是127，最大是Integer.MAX\_VALUE-(-low)-1，如果high超过了这个值，那么cache[ ]的长度就超过Integer.MAX\_VALUE了，也就溢出了。

(c) cache[]：里面保存着从[low,high]所对应的Integer对象，长度是high-low+1(因为有元素0，所以要加1)。

4）调用valueOf(inti)方法时，首先判断i是否在[low,high]之间，如果是，则复用Integer.cache[i-low]。比如，如果Integer.valueOf(3)，直接返回Integer.cache[131]；如果i不在这个范围，则调用构造方法，构造出一个新的Integer对象。

5）调用intValue()，直接返回value的值。 通过3）和4）可以发现，默认情况下，在使用自动装箱时，VM会复用[-128,127]之间的Integer对象。

Integer a1 = 1;
Integer a2 = 1;
Integer a3 = new Integer(1);
//会打印true，因为a1和a2是同一个对象,都是Integer.cache[129]
System.out.println(a1 == a2);
//false，a3构造了一个新的对象，不同于a1,a2
System.out.println(a1 == a3);

（1）Integer默认复用了[-128,127]这些对象，其中**高位置可以修改**；在启动 jvm 的时候，通过 -XX:AutoBoxCacheMax=<size> 来指定这个缓冲池的大小，该选项在 JVM 初始化的时候会设定一个名为 java.lang.IntegerCache.high 系统属性，然后 IntegerCache 初始化的时候就会读取该系统属性来决定上界

（2）Byte复用了全部256个对象[-128,127]；

（3）Short复用了[-128,127]这些对象；

（4）Long复用了[-128,127];

（5）Character复用了[0,127],Charater不能表示负数;