八大基本数据类型

四种整形

四种整形分别是byte,short,int,long，所占用的字节数分别是，1，2，4，8，计算机中一个字节是8bit。这四种整形都是有符号位数。

类　型	存储需求	取值范围
int	4 字节	— 2 147 483 648 ~ 2 147 483 647（正好超过 20 亿）
short	2 字节	— 32 768 ~ 32 767
long	8 字节	— 9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807
byte	1 字节	— 128 ~ 127

为什么byte占用一个字节8bit，可以表示-128到+127?

简单来说，就是计算机把byte类型中的数据在内存中表示为，机器说的补码形式，补码的规则是，整数的补码符号位为0，负数的补码符号位为1，因此剩余七位用来表示 数据值，正数0 1111111表示+127,负数的最小值计算机内部并表示方式为 1 0000000。推荐阅读：计算机组成原理的数据的运算与表示这一章，关于计算机内部数据的存储和表示。 下图是四位数的补码表示

1 00	1 01	1 10	1 11	0 00	0 01	0 10	0 11
-4	-3	-2	-1	0	1	2	3

```
byte a = Byte.MAX_VALUE; #
a+=1;
System.out.println(a); //输出-128
```

上述基本类型的整数都有默认值0，注意如果是局部变量没有初始化是没有默认值。静态变量或者类的非静态变量默认值是0

```
public class test {
    private  static  int intDefault;
    int intDefault2;
    public static void main(String[] args) {
        System.out.println(test.intDefault+"    "+ new test().intDefault2);
    }
}
```

整形数据的源码解析

待更新

两种浮点型

表 3-2　浮点类型

类　型	存储需求	取值范围
float	4 字节	大约 ±3.402 823 47E + 38F（有效位数为 6 ~ 7 位）
double	8 字节	大约 ±1.797 693 134 862 315 70E + 308（有效位数为 15 位）

推荐阅读：《计算机组成原理》的数据的运算和表示这一章，IEEE 754标准的浮点数标准。

一种char字符型

char表示按个字符，通常表示字符常量，一般采用Unicode编码单元进行标识十六进制，范围\u0000到 \uffff占用两个字节。

一种boolean类型

true or false 一般虚拟机底层实现是用int类型的1或者0表示

```
 boolean b=true;
 字节码层面实现如下。
Code:
    0: iconst_1
    1: istore_1
    2: return
```

本文主要介绍Java中的自动拆箱与自动装箱的有关知识。

基本数据类型

基本类型，或者叫做内置类型，是Java中不同于类(Class)的特殊类型。它们是我们编程中使用最频繁的类型。

Java是一种强类型语言，第一次申明变量必须说明数据类型，第一次变量赋值称为变量的初始化。

Java基本类型共有八种，基本类型可以分为三类：

> 字符类型 `char`
>
> 布尔类型 `boolean`
>
> 数值类型 `byte` 、 `short` 、 `int` 、 `long` 、 `float` 、 `double` 。

数值类型又可以分为整数类型 `byte` 、 `short` 、 `int` 、 `long` 和浮点数类型 `float` 、 `double` 。

Java中的数值类型不存在无符号的，它们的取值范围是固定的，不会随着机器硬件环境或者操作系统的改变而改变。

实际上，Java中还存在另外一种基本类型 `void` ，它也有对应的包装类 `java.lang.Void` ，不过我们无法直接对它们进行操作。

基本数据类型有什么好处

我们都知道在Java语言中， `new` 一个对象是存储在堆里的，我们通过栈中的引用来使用这些对象；所以，对象本身来说是比较消耗资源的。

对于经常用到的类型，如int等，如果我们每次使用这种变量的时候都需要new一个Java对象的话，就会比较笨重。所以，和C++一样，Java提供了基本数据类型，这种数据的变量不需要使用new创建，他们不会在堆上创建，而是直接在栈内存中存储，因此会更加高效。

整型的取值范围

Java中的整型主要包含 `byte` 、 `short` 、 `int` 和 `long` 这四种，表示的数字范围也是从小到大的，之所以表示范围不同主要和他们存储数据时所占的字节数有关。

先来个简答的科普，1字节=8位（bit）。java中的整型属于有符号数。

先来看计算中8bit可以表示的数字：

```
最小值：10000000 （-128）(-2^7)
最大值：01111111（127）(2^7-1)
```

整型的这几个类型中，

- byte：byte用1个字节来存储，范围为-128(-2^7)到127(2^7-1)，在变量初始化的时候，byte类型的默认值为0。
- short：short用2个字节存储，范围为-32,768 (-2^15)到32,767 (2^15-1)，在变量初始化的时候，short类型的默认值为0，一般情况下，因为Java本身转型的原因，可以直接写为0。
- int：int用4个字节存储，范围为-2,147,483,648 (-2^31)到2,147,483,647 (2^31-1)，在变量初始化的时候，int类型的默认值为0。
- long：long用8个字节存储，范围为-9,223,372,036,854,775,808 (-2^63)到9,223,372,036, 854,775,807 (2^63-1)，在变量初始化的时候，long类型的默认值为0L或0l，也可直接写为0。

超出范围怎么办

上面说过了，整型中，每个类型都有一定的表示范围，但是，在程序中有些计算会导致超出表示范围，即溢出。如以下代码：

```
int i = Integer.MAX_VALUE;
int j = Integer.MAX_VALUE;
```

```
        int k = i + j;
        System.out.println("i (" + i + ") + j (" + j + ") = k (" + k + ")");
```

输出结果：i (2147483647) + j (2147483647) = k (-2)

这就是发生了溢出，溢出的时候并不会抛异常，也没有任何提示。所以，在程序中，使用同类型的数据进行运算的时候，**一定要注意数据溢出的问题。**

包装类型

Java语言是一个面向对象的语言，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，在设计类时为每个基本数据类型设计了一个对应的类进行代表，这样八个和基本数据类型对应的类统称为包装类(Wrapper Class)。

包装类均位于java.lang包，包装类和基本数据类型的对应关系如下表所示

基本数据类型	包装类
byte	Byte
boolean	Boolean
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

在这八个类名中，除了Integer和Character类以后，其它六个类的类名和基本数据类型一致，只是类名的第一个字母大写即可。

为什么需要包装类

很多人会有疑问，既然Java中为了提高效率，提供了八种基本数据类型，为什么还要提供包装类呢?

这个问题，其实前面已经有了答案，因为Java是一种面向对象语言，很多地方都需要使用对象而不是基本数据类型。包装类有三个作用 一 实现基本类型之间的转换 ， 二是便于函数传值 ， 三就是在一些地方要用到Object的时候方便将基本数据类型装换。比如，在集合类中，我们是无法将int 、double等类型放进去的。因为集合的容器要求元素是Object类型。
实现类型转换有，Integer抓换成String类型。new Integer(10).toString(),进制转换Integer.toBinaryString(d)

int和Integer的默认值不同，int是0，Ingeger是null当表示一个学生的成绩时候，0和null含义不同

为了让基本类型也具有对象的特征，就出现了包装类型，它相当于将基本类型“包装起来”，使得它具有了对象的性质，并且为其添加了属性和方法，丰富了基本类型的操作。

拆箱与装箱

那么，有了基本数据类型和包装类，肯定有些时候要在他们之间进行转换。比如把一个基本数据类型的int转换成一个包装类型的Integer对象。

我们认为包装类是对基本类型的包装，所以，把基本数据类型转换成包装类的过程就是打包装，英文对应于boxing，中文翻译为装箱。

反之，把包装类转换成基本数据类型的过程就是拆包装，英文对应于unboxing，中文翻译为拆箱。

在Java SE5之前，要进行装箱，可以通过以下代码：

```
Integer i = new Integer(10);
```

自动拆箱与自动装箱

在Java SE5中，为了减少开发人员的工作，Java提供了自动拆箱与自动装箱功能。

自动装箱: 就是将基本数据类型自动转换成对应的包装类。

自动拆箱：就是将包装类自动转换成对应的基本数据类型。

```
Integer i =10;  //自动装箱
int b= i;     //自动拆箱
```

`Integer i=10` 可以替代 `Integer i = new Integer(10);`，这就是因为Java帮我们提供了自动装箱的功能，不需要开发者手动去new一个Integer对象。

自动装箱与自动拆箱的实现原理

既然Java提供了自动拆装箱的能力，那么，我们就来看一下，到底是什么原理，Java是如何实现的自动拆装箱功能。

我们有以下自动拆装箱的代码：

```
public static  void main(String[]args){
    Integer integer=1; //装箱
    int i=integer; //拆箱
}
```

对以上代码进行反编译后可以得到以下代码：

```
public static  void main(String[]args){
    Integer integer=Integer.valueOf(1);
    int i=integer.intValue();
}
```

从上面反编译后的代码可以看出，int的自动装箱都是通过 `Integer.valueOf()` 方法来实现的，Integer的自动拆箱都是通过 `integer.intValue` 来实现的。如果读者感兴趣，可以试着将八种类型都反编译一遍 ， 你会发现以下规律：

> 自动装箱都是通过包装类的 `valueOf()` 方法来实现的.自动拆箱都是通过包装类对象的 `xxxValue()` 来实现的。

哪些地方会自动拆装箱

我们了解过原理之后，在来看一下，什么情况下，Java会帮我们进行自动拆装箱。前面提到的变量的初始化和赋值的场景就不介绍了，那是最简单的也最容易理解的。

我们主要来看一下，那些可能被忽略的场景。

场景一、将基本数据类型放入集合类

我们知道，Java中的集合类只能接收对象类型，那么以下代码为什么会不报错呢？

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i ++){
    li.add(i);
}
```

将上面代码进行反编译，可以得到以下代码：

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2){
    li.add(Integer.valueOf(i));
}
```

以上，我们可以得出结论，当我们把基本数据类型放入集合类中的时候，会进行自动装箱。

场景二、包装类型和基本类型的大小比较

有没有人想过，当我们对Integer对象与基本类型进行大小比较的时候，实际上比较的是什么内容呢？看以下代码：

```
Integer a=1;
System.out.println(a==1?"等于":"不等于");
Boolean bool=false;
System.out.println(bool?"真":"假");
```

对以上代码进行反编译，得到以下代码：

```
Integer a=1;
System.out.println(a.intValue()==1?"等于":"不等于");
Boolean bool=false;
System.out.println(bool.booleanValue?"真":"假");
```

可以看到，包装类与基本数据类型进行比较运算，是先将包装类进行拆箱成基本数据类型，然后进行比较的。

场景三、包装类型的运算

有没有人想过，当我们对Integer对象进行四则运算的时候，是如何进行的呢？看以下代码：

```
Integer i = 10;
Integer j = 20;

System.out.println(i+j);
```

反编译后代码如下：

```
Integer i = Integer.valueOf(10);
Integer j = Integer.valueOf(20);
System.out.println(i.intValue() + j.intValue());
```

我们发现，两个包装类型之间的运算，会被自动拆箱成基本类型进行。

场景四、三目运算符的使用

这是很多人不知道的一个场景，作者也是一次线上的血淋淋的Bug发生后才了解到的一种案例。看一个简单的三目运算符的代码：

```
boolean flag = true;
Integer i = 0;
int j = 1;
int k = flag ? i : j;
```

很多人不知道，其实在 `int k = flag ? i : j;` 这一行，会发生自动拆箱。反编译后代码如下：

```
boolean flag = true;
Integer i = Integer.valueOf(0);
int j = 1;
int k = flag ? i.intValue() : j;
System.out.println(k);
```

```
boolean flag = true;
Integer i = null;
int j = 1;
int k = flag ? i : j;这里i拆箱导致NPE
System.out.println(k);
```

这其实是三目运算符的语法规范。当第二，第三位操作数分别为基本类型和对象时，其中的对象就会拆箱为基本类型进行操作。

因为例子中， `flag ? i : j;` 片段中，第二段的i是一个包装类型的对象，而第三段的j是一个基本类型，所以会对包装类进行自动拆箱。如果这个时候i的值为 `null` ，那么就会发生NPE。（自动拆箱导致空指针异常）

场景五、函数参数与返回值

这个比较容易理解，直接上代码了：

```
//自动拆箱
public int getNum1(Integer num) {
 return num;
}
//自动装箱
public Integer getNum2(int num) {
 return num;
}
```

```
场景6 switch中使用包装类
Integer num=new Integer(2);

switch (num){ //num这里自动拆箱
    case 1:
      System.out.println("1");break;
    case 2:
      System.out.println("2");break;
 }
```

自动拆装箱与缓存

Java SE的自动拆装箱还提供了一个和缓存有关的功能，我们先来看以下代码，猜测一下输出结果：

```
public static void main(String... strings) {

    Integer integer1 = 3;
    Integer integer2 = 3;

    if (integer1 == integer2)
        System.out.println("integer1 == integer2");
    else
        System.out.println("integer1 != integer2");

    Integer integer3 = 300;
    Integer integer4 = 300;
```

```
        if (integer3 == integer4)
            System.out.println("integer3 == integer4");
        else
            System.out.println("integer3 != integer4");
    }
```

我们普遍认为上面的两个判断的结果都是false。虽然比较的值是相等的，但是由于比较的是对象，而对象的引用不一样，所以会认为两个if判断都是false的。在Java中，==比较的是对象应用，而equals比较的是值。所以，在这个例子中，不同的对象有不同的引用，所以在进行比较的时候都将返回false。奇怪的是，这里两个类似的if条件判断返回不同的布尔值。

上面这段代码真正的输出结果：

```
integer1 == integer2
integer3 != integer4
```

原因就和Integer中的缓存机制有关。在Java 5中，在Integer的操作上引入了一个新功能来节省内存和提高性能。整型对象通过使用相同的对象引用实现了缓存和重用。

> 适用于整数值区间-128 至 +127。
>
> 只适用于自动装箱。使用构造函数创建对象不适用。

具体的代码实现可以阅读Java中整型的缓存机制一文，这里不再阐述。

我们只需要知道，当需要进行自动装箱时，如果数字在-128至127之间时，会直接使用缓存中的对象，而不是重新创建一个对象。

其中的javadoc详细的说明了缓存支持-128到127之间的自动装箱过程。最大值127可以通过 `-XX:AutoBoxCacheMax=size` 修改。

实际上这个功能在Java 5中引入的时候,范围是固定的-128 至 +127。后来在Java 6中，可以通过 `java.lang.Integer.IntegerCache.high` 设置最大值。

这使我们可以根据应用程序的实际情况灵活地调整来提高性能。到底是什么原因选择这个-128到127范围呢？因为这个范围的数字是最被广泛使用的。 在程序中，第一次使用Integer的时候也需要一定的额外时间来初始化这个缓存。

在Boxing Conversion部分的Java语言规范(JLS)规定如下：

如果一个变量p的值是：

```
-128至127之间的整数(§3.10.1)

true 和 false的布尔值 (§3.10.3)

‘\u0000’至 ‘\u007f’之间的字符(§3.10.4)
```

本文将介绍Java中Integer的缓存相关知识。这是在Java　5中引入的一个有助于节省内存、提高性能的功能。首先看一个使用Integer的示例代码，从中学习其缓存行为。接着我们将为什么这么实现以及他到底是如何实现的。你能猜出下面的Java程序的输出结果吗。如果你的结果和真正结果不一样，那么你就要好好看看本文了。

```
package com.javapapers.java;

public class JavaIntegerCache {

    public static void main(String... strings) {

        Integer integer1 = 3;
        Integer integer2 = 3;

        if (integer1 == integer2)
            System.out.println("integer1 == integer2");
        else
            System.out.println("integer1 != integer2");

        Integer integer3 = 300;
        Integer integer4 = 300;

        if (integer3 == integer4)
            System.out.println("integer3 == integer4");
        else
            System.out.println("integer3 != integer4");

    }
}
```

我们首先认为上面的两个判断的结果都是false。虽然比较的值是相等的，但是由于比较的是对象，而对象的引用不一样，所以会认为两个if判断都是false的。在Java中， == 比较的是对象应用，而 equals 比较的是值。所以，在这个例子中，不同的对象有不同的引用，所以在进行比较的时候都将返回false。奇怪的是，这里两个类似的if条件判断返回不同的布尔值。

上面这段代码真正的输出结果：

```
integer1 == integer2
integer3 != integer4
```

Java中Integer的缓存实现

在Java 5中，在Integer的操作上引入了一个新功能来节省内存和提高性能。整型对象通过使用相同的对象引用实现了缓存和重用。

> 适用于整数值区间-128 至 +127。
>
> 只适用于自动装箱。使用构造函数创建对象不适用。

Java的编译器把基本数据类型自动转换成封装类对象的过程叫做 自动装箱 ，相当于使用 valueOf 方法：

```
Integer a = 10; //this is autoboxing
Integer b = Integer.valueOf(10); //under the hood
```

现在我们知道了这种机制在源码中哪里使用了，那么接下来我们就看看JDK中的 valueOf 方法。下面是 JDK 1.8.0 build 25 的实现：

```
/**
     * Returns an {@code Integer} instance representing the specified
     * {@code int} value.  If a new {@code Integer} instance is not
     * required, this method should generally be used in preference to
     * the constructor {@link #Integer(int)}, as this method is likely
     * to yield significantly better space and time performance by
     * caching frequently requested values.
     *
     * This method will always cache values in the range -128 to 127,
     * inclusive, and may cache other values outside of this range.
     *
     * @param  i an {@code int} value.
     * @return an {@code Integer} instance representing {@code i}.
     * @since  1.5
     */
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high) -128到127之间用缓存的包装对象
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i); 大于127之后new新的缓存对象
    }
```

在创建对象之前先从IntegerCache.cache中寻找。如果没找到才使用new新建对象。

IntegerCache Class

IntegerCache是Integer类中定义的一个 `private static` 的内部类。接下来看看他的定义。体现了内部类用途：1.杜绝被别的外部类调度或实例化　　2.只为其依附的外部类服务 。私有静态内部类（private static class）

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[]; //存放包装类Integer对象-128 到127数组
    static {类加载时候执行类的clinit方法，该方法主要是初始化类的静态变量和执行静态代码块。因此
    IntegerCache和Integer在类加载时候执行了static代码块，并且通过for循环初始化包装类-128-127


        // 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); //这里是解析出自定义的最大值缓存上线。
               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++);//类加载时候初始化Integer缓存

              // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
```

其中的javadoc详细的说明了缓存支持-128到127之间的自动装箱过程。最大值127可以通过 `-XX:AutoBoxCacheMax=size` 修改。 缓存通过一个for循环实现。从低到高并创建尽可能多的整数并存储在一个整数数组中。这个缓存会在Integer类第一次被使用的时候被初始化出来。以后，就可以使用缓存中包含的实例对象，而不是创建一个新的实例(在自动装箱的情况下)。

实际上这个功能在Java 5中引入的时候,范围是固定的-128 至 +127。后来在Java 6中，可以通过 `java.lang.Integer.IntegerCache.high` 设置最大值。这使我们可以根据应用程序的实际情况灵活地调整来提高性能。到底是什么原因选择这个-128到127范围呢？因为这个范围的数字是最被广泛使用的。 在程序中，第一次使用Integer的时候也需要一定的额外时间来初始化这个缓存。

Java语言规范中的缓存行为

在 Boxing Conversion (http://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.7)部分的Java语言规范(JLS)规定如下：

如果一个变量p的值是：

-128至127之间的整数(§3.10.1)

true 和 false的布尔值 (§3.10.3)

‘\u0000’ 至 ‘\u007f’ 之间的字符(§3.10.4)

中时，将p包装成a和b两个对象时，可以直接使用a==b判断a和b的值是否相等。

其他缓存的对象

这种缓存行为不仅适用于Integer对象。我们针对所有的整数类型的类都有类似的缓存机制。

有ByteCache用于缓存Byte对象

有ShortCache用于缓存Short对象

有LongCache用于缓存Long对象

有CharacterCache用于缓存Character对象

`Byte`，`Short`，`Long` 有固定范围: -128 到 127。对于 `Character`，范围是 0 到 127。除了 `Integer` 以外，这个范围都不能改变。

范围内的时，将p包装成a和b两个对象时，可以直接使用a==b判断a和b的值是否相等。

自动拆装箱带来的问题

当然，自动拆装箱是一个很好的功能，大大节省了开发人员的精力，不再需要关心到底什么时候需要拆装箱。但是，他也会引入一些问题。

> 包装对象的数值比较，不能简单的使用 `==` ，虽然-128到127之间的数字可以，但是这个范围之外还是需要使用 `equals` 比较。
>
> 前面提到，有些场景会进行自动拆装箱，同时也说过，由于自动拆箱，如果包装类对象为null，那么自动拆箱时就有可能抛出NPE。三目运算符
>
> 如果一个for循环中有大量拆装箱操作，会浪费很多资源。

```
Integer sum = 0;
for(int i=0; i<100; i++){
  sum+=i;频繁发生拆箱装箱
}
```

参考资料

Java的自动拆装箱

```
public boolean equals(Object obj) {
        if (obj instanceof Integer) {
            return value == ((Integer)obj).intValue();//包装类内部封装的基本数据类型value
        }
        return false;
    }
```