JAVA**的基本类型**

基本类型，或者叫做内置类型，是 JAVA 中不同于类的特殊类型。它们是我们编程中使用最频繁的类型，因此面试题中也总少不了它们的身影，在这篇文章中我们将从面试中常考的几 个方面来回顾一下与基本类型相关的知识。

基本类型共有九种，它们分别都有相对应的包装类。

对于基本类型 void 以及它的包装类 java.lang.Void，我们都无法直接进行操作。基本类型可以分为三类，字符类型 char，布尔类型 boolean 以及数值类型 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。对于数值类型的基本类型的取值范围，我们无需强制去记忆，因 为它们的值都已经以常量的形式定义在对应的包装类中了。请看下面的例子：

**Java 代码 **

System.out.println("最小值：Short.MIN\_VALUE=" +



System.out.println("包装类：java.lang.Short");



System.out.println("基本类型：short 二进制位数：" + Short.SIZE);

// short



System.out.println();







Byte.MAX\_VALUE);

System.out.println("最大值：Byte.MAX\_VALUE=" +



System.out.println("最小值：Byte.MIN\_VALUE=" + Byte.MIN\_VALUE);



System.out.println("包装类：java.lang.Byte");



System.out.println("基本类型：byte 二进制位数：" + Byte.SIZE);



// byte



**public static void** main(String[] args) {



 **public class** PrimitiveTypeTest {

Short.MIN\_VALUE);

 System.out.println("最大值：Short.MAX\_VALUE=" +

Short.MAX\_VALUE);

 System.out.println();



 // int

 System.out.println("基本类型：int 二进制位数：" + Integer.SIZE);

 System.out.println("包装类：java.lang.Integer");

 System.out.println("最小值：Integer.MIN\_VALUE=" +

Integer.MIN\_VALUE);

 System.out.println("最大值：Integer.MAX\_VALUE=" +

Integer.MAX\_VALUE);

 System.out.println();



 // long

 System.out.println("基本类型：long 二进制位数：" + Long.SIZE);

 System.out.println("包装类：java.lang.Long");

 System.out.println("最小值：Long.MIN\_VALUE=" +

Long.MIN\_VALUE);

 System.out.println("最大值：Long.MAX\_VALUE=" +

Long.MAX\_VALUE);

 System.out.println();



 // float

 System.out.println("基本类型：float 二进制位数：" + Float.SIZE);

 System.out.println("包装类：java.lang.Float");

 System.out.println("最小值：Float.MIN\_VALUE=" +

Float.MIN\_VALUE);

 System.out.println("最大值：Float.MAX\_VALUE=" +

Float.MAX\_VALUE);

 System.out.println();



 // double

 System.out.println("基本类型：double 二进制位数：" + Double.SIZE);

 System.out.println("包装类：java.lang.Double");

 System.out.println("最小值：Double.MIN\_VALUE=" +

Double.MIN\_VALUE);

 System.out.println("最大值：Double.MAX\_VALUE=" +

Double.MAX\_VALUE);

 System.out.println();



 // char

 System.out.println("基本类型：char 二进制位数：" + Character.SIZE);

 System.out.println("包装类：java.lang.Character");

 // 以数值形式而不是字符形式将 Character.MIN\_VALUE 输出到控制台

 }

}



+ (**int**) Character.MAX\_VALUE);



System.out.println("最大值：Character.MAX\_VALUE="



// 以数值形式而不是字符形式将 Character.MAX\_VALUE 输出到控制台



+ (**int**) Character.MIN\_VALUE);



System.out.println("最小值：Character.MIN\_VALUE="



**Java 代码 **

System.out.println("基本类型：short 二进制位数：" + Short.SIZE);



// short



System.out.println();





Byte.MAX\_VALUE);

System.out.println("最大值：Byte.MAX\_VALUE=" +



System.out.println("最小值：Byte.MIN\_VALUE=" + Byte.MIN\_VALUE);



System.out.println("包装类：java.lang.Byte");



System.out.println("基本类型：byte 二进制位数：" + Byte.SIZE);



// byte



**public static void** main(String[] args) {



 **public class** PrimitiveTypeTest {

 System.out.println("包装类：java.lang.Short");

 System.out.println("最小值：Short.MIN\_VALUE=" +

Short.MIN\_VALUE);

 System.out.println("最大值：Short.MAX\_VALUE=" +

Short.MAX\_VALUE);

 System.out.println();



 // int

 System.out.println("基本类型：int 二进制位数：" + Integer.SIZE);

 System.out.println("包装类：java.lang.Integer");

 System.out.println("最小值：Integer.MIN\_VALUE=" +

Integer.MIN\_VALUE);

 System.out.println("最大值：Integer.MAX\_VALUE=" +

Integer.MAX\_VALUE);

 System.out.println();



 // long

 System.out.println("基本类型：long 二进制位数：" + Long.SIZE);

 System.out.println("包装类：java.lang.Long");

 System.out.println("最小值：Long.MIN\_VALUE=" +

Long.MIN\_VALUE);

 System.out.println("最大值：Long.MAX\_VALUE=" +

Long.MAX\_VALUE);

 System.out.println();



 // float

 System.out.println("基本类型：float 二进制位数：" + Float.SIZE);

 System.out.println("包装类：java.lang.Float");

 System.out.println("最小值：Float.MIN\_VALUE=" +

Float.MIN\_VALUE);

 System.out.println("最大值：Float.MAX\_VALUE=" +

Float.MAX\_VALUE);

 System.out.println();



 // double

 System.out.println("基本类型：double 二进制位数：" + Double.SIZE);

 System.out.println("包装类：java.lang.Double");

 System.out.println("最小值：Double.MIN\_VALUE=" +

Double.MIN\_VALUE);

 System.out.println("最大值：Double.MAX\_VALUE=" +

Double.MAX\_VALUE);

 System.out.println();



 // char

 System.out.println("基本类型：char 二进制位数：" + Character.SIZE);

 }

}



+ (**int**) Character.MAX\_VALUE);



System.out.println("最大值：Character.MAX\_VALUE="



// 以数值形式而不是字符形式将 Character.MAX\_VALUE 输出到控制台



+ (**int**) Character.MIN\_VALUE);



System.out.println("最小值：Character.MIN\_VALUE="



// 以数值形式而不是字符形式将 Character.MIN\_VALUE 输出到控制台



System.out.println("包装类：java.lang.Character");



运行结果：

 基本类型：byte 二进制位数：8

 包装类：java.lang.Byte

 最小值：Byte.MIN\_VALUE=-128

 最大值：Byte.MAX\_VALUE=127

 基本类型：short 二进制位数：16

 包装类：java.lang.Short

 最小值：Short.MIN\_VALUE=-32768

 最大值：Short.MAX\_VALUE=32767

 基本类型：int 二进制位数：32

 包装类：java.lang.Integer

 最小值：Integer.MIN\_VALUE=-2147483648

 最大值：Integer.MAX\_VALUE=2147483647

 基本类型：long 二进制位数：64

 包装类：java.lang.Long

 最小值：Long.MIN\_VALUE=-9223372036854775808

 最大值：Long.MAX\_VALUE=9223372036854775807

 基本类型：float 二进制位数：32

 包装类：java.lang.Float

 最小值：Float.MIN\_VALUE=1.4E-45

 最大值：Float.MAX\_VALUE=3.4028235E38

 基本类型：double 二进制位数：64

 包装类：java.lang.Double

 最小值：Double.MIN\_VALUE=4.9E-324

 最大值：Double.MAX\_VALUE=1.7976931348623157E308

 基本类型：char 二进制位数：16

 包装类：java.lang.Character

 最小值：Character.MIN\_VALUE=0

 最大值：Character.MAX\_VALUE=65535

Float 和 Double 的最小值和最大值都是以科学记数法的形式输出的，结尾的“E+数字”表示

E 之前的数字要乘以10的多少倍。比如3.14E3就是3.14×1000=3140，3.14E-3就是

3.14/1000=0.00314。

大家将运行结果与上表信息仔细比较就会发现 float、double 两种类型的最小值与

Float.MIN\_VALUE、Double.MIN\_VALUE 的值并不相同，这是为什么呢？实际上

Float.MIN\_VALUE 和Double.MIN\_VALUE 分别指的是float 和double 类型所能表示的最小正数。也就是说存在这样一种情况，0到±Float.MIN\_VALUE 之间的值 float 类型无法表示，

0到±Double.MIN\_VALUE 之间的值 double 类型无法表示。这并没有什么好奇怪的，因为这些范围内的数值超出了它们的精度范围。

基本类型存储在栈中，因此它们的存取速度要快于存储在堆中的对应包装类的实例对象。从

Java5.0（1.5）开始，JAVA 虚拟机（Java Virtual Machine）可以完成基本类型和它们对应包装类之间的自动转换。因此我们在赋值、参数传递以及数学运算的时候像使用基本类型一 样使用它们的包装类，但这并不意味着你可以通过基本类型调用它们的包装类才具有的方法。另外，所有基本类型（包括 void）的包装类都使用了 final 修饰，因此我们无法继承它们扩展新的类，也无法重写它们的任何方法。

各种数值类型之间的赋值与转换遵循什么规律呢？我们来看下面这个例子：

**Java 代码 **

// 编译器会做范围检查，如果赋予的值超出了范围就会报错



**byte** byte\_a = 1;



// 给 byte 类型变量赋值时，数字后无需后缀标识



**public static void** main(String[] args) {



 **public class** PrimitiveTypeTest {

 // byte byte\_b = 1000;

 // 把一个 long 型值赋值给 byte 型变量，编译时会报错，即使这个值没有

超出 byte 类型的取值范围

 // byte byte\_c = 1L;



 // 给 short 类型变量赋值时，数字后无需后缀标识

 **short** short\_a = 1;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // short short\_b = 70000;

 // 把一个 long 型值赋值给 short 型变量，编译时会报错，即使这个值没

有超出 short 类型的取值范围

 // byte short\_c = 1L;



 // 给 short 类型变量赋值时，数字后无需后缀标识

 **int** int\_a = 1;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // int int\_b = 2200000000;

 // 把一个 long 型值赋值给 int 型变量，编译时会报错，即使这个值没有

超出 int 类型的取值范围

 // int int\_c = 1L;



 // 可以把一个 int 型值直接赋值给 long 型变量，数字后无需后缀标识

 **long** long\_a = 1;

 // 如果给 long 型变量赋予的值超出了 int 型值的范围，数字后必须加 L

（不区分大小写）标识

 **long** long\_b = 2200000000L;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // long long\_c = 9300000000000000000L;



 // 可以把一个 int 型值直接赋值给 float 型变量

 **float** float\_a = 1;

 // 可以把一个 long 型值直接赋值给 float 型变量

 **float** float\_b = 1L;

 // 没有 F（不区分大小写）后缀标识的浮点数默认为 double 型的，不能

将它直接赋值给 float 型变量

 // float float\_c = 1.0;

 // float 型数值需要有一个 F（不区分大小写）后缀标识

 **float** float\_d = 1.0F;

 // 把一个 double 型值赋值给 float 型变量，编译时会报错，即使这个值

没有超出 float 类型的取值范围

 // float float\_e = 1.0D;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // float float\_f = 3.5000000E38F;



 // 可以把一个 int 型值直接赋值给 double 型变量

 **double** double\_a = 1;

 // 可以把一个 long 型值直接赋值给 double 型变量

 **double** double\_b = 1L;

 // 可以把一个 float 型值直接赋值给 double 型变量

 **double** double\_c = 1F;

 // 不带后缀标识的浮点数默认为 double 类型的，可以直接赋值

 **double** double\_d = 1.0;

 // 也可以给数字增加一个 D（不区分大小写）后缀标识，明确标出它是

double 类型的

 **double** double\_e = 1.0D;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // double double\_f = 1.8000000000000000E308D;



 // 把一个 double 型值赋值给一个 byte 类型变量，编译时会报错，即使

这个值没有超出 byte 类型的取值范围

 // byte byte\_d = 1.0D;

 // 把一个 double 型值赋值给一个 short 类型变量，编译时会报错，即使

这个值没有超出 short 类型的取值范围

 // short short\_d = 1.0D;

 // 把一个 double 型值赋值给一个 int 类型变量，编译时会报错，即使这

个值没有超出 int 类型的取值范围

 // int int\_d = 1.0D;

 // 把一个 double 型值赋值给一个 long 类型变量，编译时会报错，即使

这个值没有超出 long 类型的取值范围

 // long long\_d = 1.0D;



 }

}



// char char\_f = 70000;



// 编译器会做范围检查，如果赋予的值超出了范围就会报错



// char char\_e = 1.0D;



这个值没有超出 char 类型的取值范围

// 把一个 double 型值赋值给一个 char 类型变量，编译时会报错，即使



// char char\_d = 1.0F;



值没有超出 char 类型的取值范围

// 把一个 float 型值赋值给一个 char 类型变量，编译时会报错，即使这个



// char char\_c = 1L;



值没有超出 char 类型的取值范围

// 把一个 long 型值赋值给一个 char 类型变量，编译时会报错，即使这个



**char** char\_b = 1;



// 也可以用一个 int 型数值初始化 char 型变量



**char** char\_a = 'a';



// 可以用字符初始化一个 char 型变量



**Java 代码 **

**byte** byte\_a = 1;



// 给 byte 类型变量赋值时，数字后无需后缀标识



**public static void** main(String[] args) {



 **public class** PrimitiveTypeTest {

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // byte byte\_b = 1000;

 // 把一个 long 型值赋值给 byte 型变量，编译时会报错，即使这个值没有

超出 byte 类型的取值范围

 // byte byte\_c = 1L;



 // 给 short 类型变量赋值时，数字后无需后缀标识

 **short** short\_a = 1;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // short short\_b = 70000;

 // 把一个 long 型值赋值给 short 型变量，编译时会报错，即使这个值没

有超出 short 类型的取值范围

 // byte short\_c = 1L;



 // 给 short 类型变量赋值时，数字后无需后缀标识

 **int** int\_a = 1;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // int int\_b = 2200000000;

 // 把一个 long 型值赋值给 int 型变量，编译时会报错，即使这个值没有

超出 int 类型的取值范围

 // int int\_c = 1L;



 // 可以把一个 int 型值直接赋值给 long 型变量，数字后无需后缀标识

 **long** long\_a = 1;

 // 如果给 long 型变量赋予的值超出了 int 型值的范围，数字后必须加 L

（不区分大小写）标识

 **long** long\_b = 2200000000L;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // long long\_c = 9300000000000000000L;



 // 可以把一个 int 型值直接赋值给 float 型变量

 **float** float\_a = 1;

 // 可以把一个 long 型值直接赋值给 float 型变量

 **float** float\_b = 1L;

 // 没有 F（不区分大小写）后缀标识的浮点数默认为 double 型的，不能

将它直接赋值给 float 型变量

 // float float\_c = 1.0;

 // float 型数值需要有一个 F（不区分大小写）后缀标识

 **float** float\_d = 1.0F;

 // 把一个 double 型值赋值给 float 型变量，编译时会报错，即使这个值

没有超出 float 类型的取值范围

 // float float\_e = 1.0D;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // float float\_f = 3.5000000E38F;



 // 可以把一个 int 型值直接赋值给 double 型变量

 **double** double\_a = 1;

 // 可以把一个 long 型值直接赋值给 double 型变量

 **double** double\_b = 1L;

 // 可以把一个 float 型值直接赋值给 double 型变量

 **double** double\_c = 1F;

 // 不带后缀标识的浮点数默认为 double 类型的，可以直接赋值

 **double** double\_d = 1.0;

 // 也可以给数字增加一个 D（不区分大小写）后缀标识，明确标出它是

double 类型的

 **double** double\_e = 1.0D;

 // 编译器会做范围检查，如果赋予的值超出了范围就会报错

 // double double\_f = 1.8000000000000000E308D;



 // 把一个 double 型值赋值给一个 byte 类型变量，编译时会报错，即使

这个值没有超出 byte 类型的取值范围

 // byte byte\_d = 1.0D;

 // 把一个 double 型值赋值给一个 short 类型变量，编译时会报错，即使

这个值没有超出 short 类型的取值范围

 // short short\_d = 1.0D;

 // 把一个 double 型值赋值给一个 int 类型变量，编译时会报错，即使这

个值没有超出 int 类型的取值范围

 // int int\_d = 1.0D;

 // 把一个 double 型值赋值给一个 long 类型变量，编译时会报错，即使

这个值没有超出 long 类型的取值范围

 // long long\_d = 1.0D;

 }

}



// char char\_f = 70000;



// 编译器会做范围检查，如果赋予的值超出了范围就会报错



// char char\_e = 1.0D;



这个值没有超出 char 类型的取值范围

// 把一个 double 型值赋值给一个 char 类型变量，编译时会报错，即使



// char char\_d = 1.0F;



值没有超出 char 类型的取值范围

// 把一个 float 型值赋值给一个 char 类型变量，编译时会报错，即使这个



// char char\_c = 1L;



值没有超出 char 类型的取值范围

// 把一个 long 型值赋值给一个 char 类型变量，编译时会报错，即使这个



**char** char\_b = 1;



// 也可以用一个 int 型数值初始化 char 型变量



**char** char\_a = 'a';



// 可以用字符初始化一个 char 型变量





从上面的例子中我们可以得出如下几条结论：

 未带有字符后缀标识的整数默认为 int 类型；未带有字符后缀标识的浮点数默认为

double 类型。

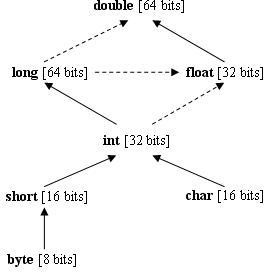
 如果一个整数的值超出了 int 类型能够表示的范围，则必须增加后缀“L”（不区分大小写，建议用大写，因为小写的 L 与阿拉伯数字1很容易混淆），表示为 long 型。

 带有“F”（不区分大小写）后缀的整数和浮点数都是 float 类型的；带有“D”（不区分大小写）后缀的整数和浮点数都是 double 类型的。

 编译器会在编译期对 byte、short、int、long、float、double、char 型变量的值进行检查，如果超出了它们的取值范围就会报错。

 int 型值可以赋给所有数值类型的变量；long 型值可以赋给 long、float、double 类型的变量；float 型值可以赋给 float、double 类型的变量；double 型值只能赋给 double 类型变量。

下图显示了几种基本类型之间的默认逻辑转换关系：



图中的实线表示无精度损失的转换，而虚线则表示这样的转换可能会损失一定的精度。如果 我们想把一个能表示更大范围或者更高精度的类型，转换为一个范围更小或者精度更低的类 型时，就需要使用强制类型转换（Cast）了。不过我们要尽量避免这种用法，因为它常常引发错误。请看下面的例子，如果不运行代码，你能预测它的结果吗？

**Java 代码 **

 }

}



System.out.println(b);



// b 的值会是什么呢？



**short** b = (**short**) a;



**int** a = 123456;



**public static void** main(String[] args) {



 **public class** PrimitiveTypeTest {

**Java 代码 **

 }

}



System.out.println(b);



// b 的值会是什么呢？



**short** b = (**short**) a;



**int** a = 123456;



**public static void** main(String[] args) {



 **public class** PrimitiveTypeTest {

运行结果：

 -7616

***运算符对基本类型的影响***

当使用+、-、\*、/、%运算符对基本类型进行运算时，遵循如下规则：

 只要两个操作数中有一个是 double 类型的，另一个将会被转换成 double 类型，并且结果也是 double 类型；

 否则，只要两个操作数中有一个是 float 类型的，另一个将会被转换成 float 类型， 并且结果也是 float 类型；

 否则，只要两个操作数中有一个是 long 类型的，另一个将会被转换成 long 类型， 并且结果也是 long 类型；

 否则，两个操作数（包括 byte、short、int、char）都将会被转换成 int 类型，并且结果也是 int 类型。

当使用+=、-=、\*=、/=、%=、运算符对基本类型进行运算时，遵循如下规则：

* 运算符右边的数值将首先被强制转换成与运算符左边数值相同的类型，然后再执行 运算，且运算结果与运算符左边数值类型相同。

了解了这些，我们就能解答下面这个常考的面试题了。请看： **引用**

short s1=1;s1=s1+1;有什么错？short s1=1;s1+=1;有什么错？

乍一看，觉得它们都应该没有错误，可以正常运行。我们来写个例子试试：

**Java 代码 **

 }

}



s1 += 1;



// 这一行代码也没有报错



s1 = 1 + 1;



// 这一行代码没有报错



// s1 = s1 + 1;



// 这一行代码会报编译错误



**short** s1 = 1;



**public static void** main(String[] args) {



 **public class** PrimitiveTypeTest {

**Java 代码 **

// 这一行代码也没有报错



s1 = 1 + 1;



// 这一行代码没有报错



// s1 = s1 + 1;



// 这一行代码会报编译错误



**short** s1 = 1;



**public static void** main(String[] args) {



 **public class** PrimitiveTypeTest {

 }

}



s1 += 1;



从例子中我们可以看出结果了。利用上面列举的规律，也很容易解释。在 s1=s1+1;中，s1+1 运算的结果是 int 型，把它赋值给一个 short 型变量 s1，所以会报错；而在 s1+=1;中，由于是 s1是 short 类型的，所以1首先被强制转换为 short 型，然后再参与运算，并且结果也是

short 类型的，因此不会报错。那么，s1=1+1;为什么不报错呢？这是因为1+1是个编译时可以确定的常量，“+”运算在编译时就被执行了，而不是在程序执行的时候，这个语句的效果等同于 s1=2，所以不会报错。前面讲过了，对基本类型执行强制类型转换可能得出错误的结果，因此在使用+=、-=、\*=、/=、%=等运算符时，要多加注意。

当使用“==”运算符在基本类型和其包装类对象之间比较时，遵循如下规则：

 只要两个操作数中有一个是基本类型，就是比较它们的数值是否相等。

 否则，就是判断这两个对象的内存地址是否相等，即是否是同一个对象。

下面的测试例子则验证了以上的规则：

**Java 代码 **

Integer Integer\_int = 0;



// Integer 类型用 int 类型初始化



Integer Integer\_Integer = **new** Integer(0);



// Integer 类型用 Integer 类型初始化



**int** int\_Integer = **new** Integer(0);



// int 类型用 Integer 类型初始化



**int** int\_int = 0;



// int 类型用 int 类型初始化



**public static void** main(String[] args) {



 **public class** EqualsTest {



 System.out.println("int\_int == int\_Integer 结果是："

 + (int\_int == int\_Integer));

 System.out.println("Integer\_Integer == Integer\_int 结果是："

 + (Integer\_Integer == Integer\_int));

 System.out.println();

 System.out.println("int\_int == Integer\_Integer 结果是："

 + (int\_int == Integer\_Integer));

 System.out.println("Integer\_Integer == int\_int 结果是："

 + (Integer\_Integer == int\_int));

 System.out.println();



 // boolean 类型用 boolean 类型初始化

 **boolean** boolean\_boolean = **true**;

 // boolean 类型用 Boolean 类型初始化

 **boolean** boolean\_Boolean = **new** Boolean(**true**);

 // Boolean 类型用 Boolean 类型初始化

 Boolean Boolean\_Boolean = **new** Boolean(**true**);

 // Boolean 类型用 boolean 类型初始化

 Boolean Boolean\_boolean = **true**;



 System.out.println("boolean\_boolean == boolean\_Boolean 结果是：

"

 + (boolean\_boolean == boolean\_Boolean));

：

：

 }

}



+ (Boolean\_Boolean == boolean\_boolean));

"



System.out.println("Boolean\_Boolean == boolean\_boolean 结果是



+ (boolean\_boolean == Boolean\_Boolean));

"



System.out.println("boolean\_boolean == Boolean\_Boolean 结果是



System.out.println();



+ (Boolean\_Boolean == Boolean\_boolean));

"



System.out.println("Boolean\_Boolean == Boolean\_boolean 结果是：



**Java 代码 **

// Integer 类型用 int 类型初始化



Integer Integer\_Integer = **new** Integer(0);



// Integer 类型用 Integer 类型初始化



**int** int\_Integer = **new** Integer(0);



// int 类型用 Integer 类型初始化



**int** int\_int = 0;



// int 类型用 int 类型初始化



**public static void** main(String[] args) {



 **public class** EqualsTest {

 Integer Integer\_int = 0;



 System.out.println("int\_int == int\_Integer 结果是："

 + (int\_int == int\_Integer));

 System.out.println("Integer\_Integer == Integer\_int 结果是："

 + (Integer\_Integer == Integer\_int));

 System.out.println();

 System.out.println("int\_int == Integer\_Integer 结果是："

 + (int\_int == Integer\_Integer));

 System.out.println("Integer\_Integer == int\_int 结果是："

 + (Integer\_Integer == int\_int));

 System.out.println();



 // boolean 类型用 boolean 类型初始化

 **boolean** boolean\_boolean = **true**;

 // boolean 类型用 Boolean 类型初始化

 **boolean** boolean\_Boolean = **new** Boolean(**true**);

 // Boolean 类型用 Boolean 类型初始化

 Boolean Boolean\_Boolean = **new** Boolean(**true**);

 // Boolean 类型用 boolean 类型初始化

 Boolean Boolean\_boolean = **true**;



 System.out.println("boolean\_boolean == boolean\_Boolean 结果是：

"

|  |  |  |
| --- | --- | --- |
|    |  | + (boolean\_boolean == boolean\_Boolean));  System.out.println("Boolean\_Boolean == Boolean\_boolean 结果是： |
| " |  |  |
|  |  | + (Boolean\_Boolean == Boolean\_boolean)); |
|  |  | System.out.println(); |
|  |  | System.out.println("boolean\_boolean == Boolean\_Boolean 结果是： |
| " |  |  |
|  |  | + (boolean\_boolean == Boolean\_Boolean)); |
|  |  | System.out.println("Boolean\_Boolean == boolean\_boolean 结果是： |
| " |  |  |
|  |  | + (Boolean\_Boolean == boolean\_boolean)); |
|  | } |  |
|  } |  |  |

运行结果：

 int\_int == int\_Integer 结果是：true

 Integer\_Integer == Integer\_int 结果是：false

 int\_int == Integer\_Integer 结果是：true

 Integer\_Integer == int\_int 结果是：true

 boolean\_boolean == boolean\_Boolean 结果是：true

 Boolean\_Boolean == Boolean\_boolean 结果是：false

 boolean\_boolean == Boolean\_Boolean 结果是：true

 Boolean\_Boolean == boolean\_boolean 结果是：true

为了便于查看，上例中变量命名没有采用规范的方式，而是采用了“变量类型”+“\_”+“初始化 值类型”的方式。

***Math.round()方法***

java.lang.Math 类里有两个 round()方法，它们的定义如下：

**Java 代码 **

 }

//other code



 }



 **public static long** round(**double** a) {

//other code



 **public static int** round(**float** a) {

**Java 代码 **

 }

//other code



 }



 **public static long** round(**double** a) {

//other code



 **public static int** round(**float** a) {

它们的返回值都是整数，且都采用四舍五入法。运算规则如下：

 如果参数为正数，且小数点后第一位>=5，运算结果为参数的整数部分+1。

 如果参数为负数，且小数点后第一位>5，运算结果为参数的整数部分-1。

 如果参数为正数，且小数点后第一位<5；或者参数为负数，且小数点后第一位<=5， 运算结果为参数的整数部分。

我们可以通过下面的例子来验证：

**Java 代码 **

 **public class** MathTest {

System.out.println("正数：Math.round(11.68)=" +







System.out.println("正数：Math.round(11.46)=" +









**public static void** main(String[] args) {



System.out.println("小数点后第一位=5");



System.out.println("正数：Math.round(11.5)=" + Math.round(11.5));



System.out.println("负数：Math.round(-11.5)=" +

Math.round(-11.5));

System.out.println();

System.out.println("小数点后第一位<5");



Math.round(11.46));



System.out.println("负数：Math.round(-11.46)=" +

Math.round(-11.46));

System.out.println();

System.out.println("小数点后第一位>5");



Math.round(11.68));



System.out.println("负数：Math.round(-11.68)=" +

Math.round(-11.68));



}

 }

**Java 代码**

 **public class** MathTest {

 **public static void** main(String[] args) {

 System.out.println("小数点后第一位=5");

 System.out.println("正数：Math.round(11.5)=" + Math.round(11.5));

 System.out.println("负数：Math.round(-11.5)=" +

Math.round(-11.5));

 System.out.println();



 System.out.println("小数点后第一位<5");

 System.out.println("正数：Math.round(11.46)=" +

Math.round(11.46));

 System.out.println("负数：Math.round(-11.46)=" +

Math.round(-11.46));

 System.out.println();



 System.out.println("小数点后第一位>5");

 System.out.println("正数：Math.round(11.68)=" +

Math.round(11.68));

 System.out.println("负数：Math.round(-11.68)=" +

Math.round(-11.68));

 }

 }

运行结果：

 小数点后第一位=5

 正数：Math.round(11.5)=12

 负数：Math.round(-11.5)=-11

 小数点后第一位<5

 正数：Math.round(11.46)=11

 负数：Math.round(-11.46)=-11

 小数点后第一位>5

 正数：Math.round(11.68)=12

 负数：Math.round(-11.68)=-12

根据上面例子的运行结果，我们还可以按照如下方式总结，或许更加容易记忆：

 参数的小数点后第一位<5，运算结果为参数整数部分。

 参数的小数点后第一位>5，运算结果为参数整数部分绝对值+1，符号（即正负）不变。

 参数的小数点后第一位=5，正数运算结果为整数部分+1，负数运算结果为整数部分。

但是上面的结论仍然不是很好记忆。我们来看看 round()方法的内部实现会给我们带来什么启发？我们来看这两个方法内部的代码：

**Java 代码 **

 }



 **public static long** round(**double** a) {

 **return** (**long**)floor(a + 0.5d);

 }

**return** (**int**)floor(a + 0.5f);



 **public static int** round(**float** a) {

**Java 代码 **

 }

**return** (**int**)floor(a + 0.5f);



 **public static int** round(**float** a) {

 }

**return** (**long**)floor(a + 0.5d);





 **public static long** round(**double** a) {

看来它们都是将参数值+0.5后交与 floor()进行运算，然后取返回值。那么 floor()方法的作用又是什么呢？它是取一个小于等于参数值的最大整数。比如经过 floor()方法运算后，如果参数是10.2则返回10，13返回13，-20.82返回-21，-16返回-16等等。既然是这样，我们就可以用一句话来概括 round()方法的运算效果了：

* Math 类的 round()方法的运算结果是一个<=(参数值+0.5)的最大整数。

***switch 语句***

哪些类型可以用于 switch 语句的判断呢？我们做个测试就知道了：

**Java 代码 **

System.out.println("byte 可以用于 switch 语句");



**case** 0:



**switch** (byte\_n) {



**byte** byte\_n = 0;



// byte



**public static void** main(String[] args) {





}



A, B



**static enum** enum\_e {



// 枚举类型，Java5.0以上版本可用



 **public class** MathTest {



**break**;



}

 </