Java 基础（1）：深入解析基本类型

一、基本类型的简介

1. 基本类型的两条准则

- Java 中，如果对整数不指定类型，默认时 int 类型，对小数不指定类型，默认是 double 类型。
- 基本类型由小到大，可以自动转换，但是由大到小，则需要强制类型转换。

2. 所占的字节数

byte： 1 个字节；
char： 2 个字节；
short： 2 个字节；
int： 4 个字节；
long： 8 个字节；
float： 4 个字节；(6 位小数，指数是：10^-38~10^38; 范围：)
double： 8 个字节；
char：Java 中用 "\u 四位十六进制的数字 (即使在注释中出现\u，后面如果
跟的不是 4 个 16 进制的数字，也会报错)"表示将字符转换成对应的 unicode 编码；也可以用字符来赋值如： char c="\u0000" ，char 的默认初始化值，unicode 的 null 字符。

3. 基本类型的后缀

long ： l 或 L
float： f 或 F；
double： d 或 D

二、类型转换

正如前面所说的，类型由大到小，是必需强制转换。但这并不意味着需要用户手动强制转换 —— 也就是隐式转换。隐式转换 说的透彻点就是由编译器来进行强制转换，不需要用户再去写强制转换的代码。下面的前两个小点所说的便是特殊的隐式类型转换。

本小节所讨论的类型转换是不包括类型由小到大的转换，讨论的是其他比较容易让人迷

惑的类型转换。

1. int 类型的字面常量转换成比 int 类型低的变量类型

所谓的字面常量就是值的本身，如 5、7、“aa”等等。我们先看个例子：

```
public static void main(String[] args) {
    int a = 8;  //8 是字面常量
    byte b = 9;  //9 是字面常量
    char c = 9+5;//常量表达式
    short s = (short) (c+10); //变量表达式，需要显式强制转换
}
```

上面的代码是经过编译的，是正确的。b 是 byte 类型，但 b=9 不需要显式地手动强制转换，这是因为 9 是字面常量，是由 JVM 自动完成。

我们再来看一下 c=9+5，c 是 char 类型，9+5 得到结果是 int 类型，但也不需要显式地手动强制转换。这是因为 9+5 是常量表达式，所以在编译期间已经由编译器计算出结果了，即经过编译后，相当于 c=14，也是字面常量，所以可以隐式转换。同理，short s = (short)(c+10); 之所以不能隐式转换，就是因为表达式不是常量表达式，包含了变量，只能在运行期间完成，所以就要手动强制转换。

整形字面常量隐式转换的限制：

- 整形字面常量的大小超出目标类型所能表示的范围时，要手动强制类型转换。

```
byte b = 128;//编译错误，128 超出 byte 类型所能表示的范围
byte c = (byte)128;//编译通过
```

- 对于传参数时，必须要显式地进行强制类型转换，明确转换的类型。

编译器之所以这样要求，其实为了避免**方法重载出现的隐式转换**与**小类型自动转大类型**发生冲突。

```
public static void main(String[] args) {
    shortMethod(8);//编译错误
    shortMethod((short)8); //编译通过
    longMethod(8);//编译通过，因为这是小类型变成大类型，是不需要强制类型转换的
}
public static void shortMethod(short c){
    System.out.println(c);
}
public static void longMethod(short l){
    System.out.println(l);
}
```

- char 类型的特殊情况 ：下面再细讲

2. 复合运算符的隐式转换

复合运算符（+=、-=、*=、/=、%=）是可以将右边表达式的类型自动强制转换成左边的类型。

```
public static void main(String[] args) {
    int a = 8;
    short s = 5;
    s += a;
    s += a+5;
}
```

s+=a、s+=a+5;的表达式计算结果都是 int 类型，但都不需要手动强制转换。其实，如果是反编译这段代码的 class 文件，你会发现 s+=a;，其实是被编译器处理成了：

```
s=(short)(s+a)
```

也就是说对于所有的复合运算的隐式类型转换，其实是编译器自动添加类型转换的代码。

所以，相对于整形字面常量的隐式转换，复合运算符的隐式转换则没有任何限制因为前者只能在编译器期间发生，后者则是编译器实实在在的补全了类型转换的代码。

3. 特殊的 char 类型

char 类型在基本类中是一个比较特殊的存在。这种特殊性在于 **char 类型是一个无符号类型，所以 char 类型与其他基本类型不是子集与父集间的关系（其他类型都是有符号的类型）。也就是说，char 类型与 byte、short 之间的转换都需要显式的强制类型转换（小类型自动转换成大类型失败）。**

同时，**由于 char 类型是一个无符号类型，所以对于整形字面常量的隐式转换的限制，不仅包括字面常量数值的大小不能超出 2 个字节，还包括字面常量数值不能为负数。**

```
byte b = 2;
 char c = 2;//编译通过
      c = 100000000000;//编译不通过，超出 char 类型的范围
 char d = -2//字面常量为负数，编译不通过
      d = (char)-100;//编译通过
 char f = (char)b; //编译通过，必须显式的强制类型转换
      f = b;//编译不通过，不能隐式转换
 int  i = c;//编译通过，可以不需要强制类型转换
 short s = (short) c;//编译通过，必须显式地强制类型转换
```

char 类型是无符号的类型，这种无符号也体现在在其转换成 int 类型时，也就是说，**char 类型在扩展时，也是按无符号的方式扩展，扩展位填 0。**我们来看一个例子：

```
public static void main(String[] args) {
    short s = -5;
```

```
    char c = (char)s;
    System.out.println(c==s);  //false
    System.out.println("(int)c = "+(int)c); //转换成 int 类型，值为 65531
    System.out.println("(short)c = "+(short)c); //-5
    System.out.println("(int)s = "+(int)s);//-5
}
```

运行结果：

```
false
 (int)c = 65531
 (short)c = -5
 (int)s = -5
```

从上面的结果发现，char 类型的 c 与 short 类 s 其实存储字节码内容是一样的，但由于前者是无符号，所以扩展成 int 类型的结果是 65531，而不是 -5。运算符==比较的就是他们扩展成 int 类型的值，所以为 fasle。

对 char 类型的类型转换，可以总结成以下几点：

- char 类型与 byte、short 的相互转换，都需要显式地强类型制转换。
- 对于数值是负数的，都需要进行显式地强制类型转换，特别是在整形字面常量的隐式转换中。
- char 类型转换成 int、long 类型是符合小类型转大类型的规则，即无需要强制类型转换。

4. 运算结果的类型

在 Java 中，一个运算结果的类型是与表达式中类型最高的相等，如：

```
char cc = 5;
float dd = 0.6f+cc;//最高类型是 float，运算结果是 float
float ee = (float) (0.6d+cc);//最高类型是 double，运算结果也是 double
int aa = 5+cc;//最高类型是 int，运算结果也为 int
```

但是，**对于最高类型是 byte、short、char 的运算来说，则运行结果却不是最高类型，而是 int 类型。**看下面的例子，c、d 运算的最高类型都是 char，但运算结果却是 int，所以需要强制类型转换。

```
byte b = 2;
 char a = 5;
 char c = (char) (a+b);//byte+char，运算结果的类型为 int，需要强制类型转换
 int  e = a+b;//编译通过，不需要强制类型转换，可以证明是 int
 char d = (char) (a+c);//char+char，
 short s1 = 5;
 short s2 = 6;
 short s3 =(short)s1+s2;
```

综上所述，java 的运算结果的类型有两个性质：

- 运算结果的类型必须是 int 类型或 int 类型以上。
- 最高类型低于 int 类型的，运算结果都为 int 类型。否则，运算结果与表达式中最高类型一致。

三、浮点数类型

1. 浮点类型的介绍

我们都知道 ,long 类型转换成 float 类型是不需要强制类型转换的 ,也就是说相对于 float 类型，long 类型是小类型，存储的范围要更小。然而 float 只占了 4 个字节，而 long 却占了 8 个字节，long 类型的存储空间要比 float 类型大。这究竟是怎么一回事，我们接下来将细细分析。

浮点数使用 IEEE（电气和电子工程师协会）格式。 浮点数类型使用 符号位、指数、有效位数（尾数）来表示。要注意一下，尾数的最高

在 java 中，float 和 double 的结构如下：

类型	符号位	指数域	有效位域
float	1 位	8 位	23 位
double	1 位	11 位	52 位

符号位： 0 为正，1 为负；

指数域： 无符号的，float 的偏移量为 127（即 float 的指数范围是-126~127，），double

有效位域： 无符号的；

2. 浮点类型的两个需要注意的地方

1) 存储的小数的数值可能是模糊值

```
public static void main(String[] args) {
    double d1 = 0.1;
    double d2 = 0.2;
    System.out.println(d1+d2 == 0.3);
    System.out.println(d1+d2);
}
```

运行结果：

false

0.30000000000000004

上述的运算结果并不是错误。这是因为无法用二进制来准确地存储的 0.3，这是一个无限循环的值，与 10 进制的 1/3 很相似。不只是 0.3，**很多小数都是无法准确地用浮点型表示，**

其实这是由小数的十进制转成二进制的算法所决定的，十进制的小数要不断乘 2，知道最后的结果为整数才是最后的二进制值，但这有可能怎么也得不到整数，所以最后得到的结果可能是一个无限值，浮点型就无法表示了。

但是对于整数来说，在浮点数的有效范围内，则都是精确的。同样，也是由于转换算法：十进制的整数转成二进制的算法是不断对 2 求余数，所以 不会存在无限值的情况；

2) 浮点数的有效位及精度

浮点型所能表示的有效位是有限的，所以哪怕是整数，只要超出有效位数，也只能存储相似值，也就是该数值的最低有效位将会丢失,从而造精度丢失。

float 类型的二进制有效位是 24 位，对应十进制的 7 ~ 8 位数字；double 类型的二进制 53 位，对应十进制的 10 ~ 11 位数字。

double、float 类型 所能表示的范围比 int、long 类型表示的范围要广，也浮点类型属于大类型。但是，并不能完美地表示整型，浮点类型的精度丢失会造成一些问题。

```
public static void main(String[] args) {
    int a = 3000000;
    int b = 30000000;
    float f1 = a;
    float f2 = b;
    System.out.println("3000000==3000001 "+(f1==f1+1));
    System.out.println("30000000==30000001 "+(f2==f2+1));
    System.out.println("3000000 的有效二进制位数："+
Integer.toBinaryString(a).length());
    System.out.println("30000000 的有效二进制位数："+
Integer.toBinaryString(b).length());
}
```

运行结果：

3000000 == 3000001 false

30000000 == 30000001 true

3000000 的有效二进制位数： 22

30000000 的有效二进制位数： 25

上面的例子很好体现了精度丢失所带来的后果：30000000==30000001 的比较居然为 true 了。而造成这种结果的原因就是 30000000 的有效二进制位数是 25 位，超出了 float 所能表示的有效位 24 位，最后一位就被舍去，所以就造成在刚加的 1 也被舍去，因此 30000000 的加一操作前后的浮点型表示是一样的。

当然，**并不是超出浮点型的有效位就不能精确表示，其实，主要看的是最高有效位与最低非 0 有效位之间的 “间隙”，如果间隙的在浮点型的有效位数内，自然可以精确表示，因为舍去的低有效位都是 0，自然就无所谓了。**如果上面的例子的浮点型用的是 double 就不会丢失精度了，因为 double 的精度是 52 位。

3) 解决浮点型精度丢失的问题

浮点型带来精度丢失的问题是很让人头痛的，所以一般情况下，在程序中是不会使用 float、double 来存储比较大的数据。而商业计算往往要求结果精确。《Effactive Java》书中有一句话：

float 和 double 类型的主要设计目标是为了科学计算和工程计算

JDK 为此提供了两个高精度的大数操作类给我们：BigInteger、BigDecimal。