第五章

转换和上下文

用 Java 编程语言编写的每个表达式要么不产生结果(§15.1)，要么具有可在编译时推导的类型(§15.3)。当表达式出现在大多数上下文中时，它必须与该上下文中所期望的类型兼容;这种类型称为目标类型。为了方便起见，表达式与其周围上下文的兼容性通过两种方式实现:

- 首先，对于一些被称为多元表达式(§15.2)的表达式，推导出的类型可能会受到目标类型的影响。同一个表达式在不同的上下文中可以有不同的类型。
- 其次，在推导出表达式的类型之后，有时可以执行从表达式类型到目标类型的隐式转换。

如果两种策略都不能生成适当的类型，则会发生编译时错误。

确定表达式是否是多元表达式的规则，如果是，其类型和在特定上下文中的兼容性取决于上下文的类型和表达式的形式。除了影响表达式的类型外，目标类型在某些情况下还可能影响表达式的运行时行为，以便生成适当类型的值。

类似地，决定目标类型是否允许隐式转换的规则取决于上下文的类型、表达式的类型，在一种特殊情况下，还取决于常量表达式的值(§15.29)。从类型 S 到类型 T 的转换允许在编译时将类型 S 的表达式当作是类型 T 来处理。 在某些情况下，这将需要在运行时执行相应的操作来检查转换的有效性，或将表达式的运行时值转换为适合新类型 T 的形式。

例子 5.0-1. 编译时和运行时转换

- 从 Object 类型到 Thread 类型的转换需要运行时检查，以确保运行时值实际上是 Thread 类或其子类之一的实例;如果不是，则抛出异常。
- 从 Thread 类到 Object 类的转换不需要运行时操作；Thread 是 Object 的子类，因此 Thread 类型表达式产生的任何引用都是 Object 类型的有效引用值。
- 从 int 类型到 long 类型的转换需要将 32 位整型值的运行时符号扩展为 64 位长整型表示形式。没有信息丢失。
- 从 double 类型到 long 类型的转换需要将 64 位浮点值转换为 64 位整数表示形式。根据实际的运行时值，可能会丢失信息。

Java 编程语言中可能的转换可以分为几个大类:

- 相等转换

- 拓宽原生转换
- 缩窄原生转换
- 拓宽引用转换
- 缩窄引用转换
- 装箱转换
- 拆箱转换
- 未检查转换
- 捕获转换
- 字符串转换

在六种转换上下文中，多元表达式可能会受到上下文的影响，或者会发生隐式转换。

每种上下文都有不同的多元表达式类型规则，允许在上面的某些类别中进行转换，但不允许在其他类别中进行转换。这些上下文是:

- 赋值上下文(§5.2，§15.26)，其中表达式的值被绑定到一个命名变量。原生和引用类型可能会变宽，值可能会被装箱或拆箱，某些原生常量表达式可能会变窄。 也可能发生未经检查的转换。

- 严格调用上下文(§5.3，§15.9，§15.12)，其中参数被绑定到构造函数或方法的形参。 可能会发生扩展原语、扩展引用和未检查的转换。

- 松散调用上下文(§5.3，§15.9，§15.12)，其中，与严格调用上下文一样，实参被绑定到形式参数。如果仅使用严格调用上下文无法找到适用的声明，则方法或构造函数调用可以提供此上下文。除了扩展和未检查转换外，此上下文还允许进行装箱和拆箱转换。

- 字符串上下文(§5.4，§15.18.1)，其中任何类型的值都被转换为 String 类型的对象。

- 强制转换上下文(§5.5)，将表达式的值转换为强制转换操作符显式指定的类型(§15.16)。强制转换上下文比赋值或松散调用上下文更具有包容性，允许除字符串转换外的任何特定转换，但对引用类型的某些强制转换会在运行时检查其正确性。

- 数字上下文(§5.6)，其中数字运算符或其他一些运算数字的表达式的操作数可以扩展为通用类型。

术语“转换”还用于描述在特定上下文中允许的任何转换，但并不具体。 例如，我们说一个局部变量的初始化式的表达式受“赋值转换”的约束，这意味着将根据赋值上下文的规则隐式地为该表达式选择特定的转换。另一个例子是，我们说一个表达式经历了“强制转换”，这意味着该表达式的类型将在强制转换上下文中被允许进行转换。

例子 5.0-2. 不同上下文中的转换

```
class Test {
    public static void main(String[] args) {
        // Casting conversion (5.5) of a float literal to
        // type int. Without the cast operator, this would
        // be a compile-time error, because this is a
        // narrowing conversion (5.1.3):
        int i = (int)12.5f;

        //     String     conversion     (5.4)     of     i's     int     value:
        System.out.println("(int)12.5f==" + i);

        // Assignment conversion (5.2) of i's value to type
        // float. This is a widening conversion (5.1.2):
        float f = i;

        // String conversion of f's float value:
        System.out.println("after float widening: " + f);

        // Numeric promotion (5.6) of i's value to type
        // float. This is a binary numeric promotion.
        // After promotion, the operation is float*float:
        System.out.print(f);
        f = f * i;

        // Two string conversions of i and f:
        System.out.println("*" + i + "==" + f);

        // Invocation conversion (5.3) of f's value
        // to type double, needed because the method Math.sin
        // accepts only a double argument:
        double d = Math.sin(f);

        // Two string conversions of f and d: System.out.println("Math.sin(" + f +
        ")==" + d);
    }
}
```


程序输出如下:

```
(int)12.5f==12
after float widening: 12.0
12.0*12==144.0
Math.sin(144.0)==-0.49102159389846934
```

5.1 转换类型

Java 编程语言中具体的类型转换分为 12 种。

5.1.1 相等转换

任何类型都允许从一种类型转换为同一类型。

这可能看起来微不足道，但它有两个实际后果。首先，总是允许表达式具有所需的类型，从而允许这样一个简单的规则:每个表达式都要进行转换，如果只是简单的相等转换的话。其次，它意味着为了清晰起见，允许程序包含冗余强制转换操作符。

5.1.2 拓宽原生转换

19 种特定的原生类型转换称为拓宽原生转换:

- byte 到 short, int, long, float, 或 double
- short 到 int, long, float, 或 double
- char 到 int, long, float, 或 double
- int 到 long, float, 或 double
- long 到 float 或 double
- float 到 double

在以下情况下，扩展原生转换不会丢失数值的整体大小信息，其中数值会完全保留:

- 从一个整数类型到另一个整数类型
- 从 byte, short, 或 char 到浮点类型
- 从 int 到 double
- 从 float 到 double

从 int 到 float, 或者从 long 到 float,或者从 long 到 double 的拓宽原生转换,可能导致精度损失, 也就是说，结果可能会丢失一些值中最不重要的位。在这种情况下，得到的浮点值将是整数值的正确四舍五入版本，使用四舍五入舍入策略（§15.4）。

将有符号整数值扩展到整型 T 的转换仅仅是符号扩展整数值的二进制补码表示以填充更宽的格式。

字符到整数类型 T 的扩展转换 0 扩展了字符值的表示，以填充更宽的格式。

根据 IEEE 754 从整数格式转换为二进制浮点格式的规则，会发生从 int 到 float 或从 long 到 float 或从 int 到 double 或从 long 至 double 的扩展转换。

根据IEEE 754在二进制浮点格式之间转换的规则确定，发生从float到double的扩展转换。

尽管可能会出现精度损失，但拓宽原生转换不会导致运行时异常（§11.1.1）。

例子 5.1.2-1. 拓宽原生转换

```
class Test {
    public static void main(String[] args) {
        int big = 1234567890;
        float approx = big;
        System.out.println(big - (int)approx);
```

```
            }
        }
```

程序打印:

```
-46
```

因此指示在从 int 类型转换为 float 类型期间信息丢失，因为 float 类型的值不精确到九位有效数字。

5.1.3 缩窄原生转换

22 种原生类型的特定转换称为缩窄原生转换:

- short 到 byte 或 char
- char 到 byte 或 short
- int 到 byte, short, 或 char
- long 到 byte, short, char, 或 int
- float 到 byte, short, char, int, 或 long
- double 到 byte, short, char, int, long, 或 float

缩窄原生转换可能会丢失有关整体数值大小的信息，还可能会丢失精度和范围。

将有符号整数缩窄转换为整数类型 T 只需要丢弃除 n 个最低位以外的所有位，其中 n 用于表示类型 T 的位数。除了可能丢失有关数值大小的信息外，这还可能导致结果值的符号与输入值的符号不同。

char 到整型 T 的缩窄转换同样简单地丢弃了除 n 个最低位以外的所有位，其中 n 是用于表示类型 T 的位数。 除了可能丢失有关数值大小的信息外，这可能导致结果值为负数，即使字符表示 16 位无符号整型值。

将浮点数缩窄转换为整型 T 需要两个步骤:

1. 在第一步中，如果 T 为 long 类型，则将浮点数转换为 long 类型;如果 T 为 byte、short、char 或 int 类型，则将浮点数转换为 int 类型，如下所示:

    - 如果浮点数是 NaN(§4.2.3)，第一步转换的结果是 int 或 long 0。
    - 否则，如果浮点数不是无穷大，则使用向零四舍五入策略将浮点值四舍五入为整数值 V(§4.2.4)。有两种情况:

        a. 如果 T 是 long 类型的值，这个整数值可以表示为 long 类型的值，那么第一步的结果就是 long 类型的值 V。

        b. 否则，如果这个整数可以表示为 int，那么第一步的结果就是 int 类型值 V。

    - 否则，以下两种情况之一必须为真:

a. 该值必须非常小(一个较小的负值或负无穷大)，并且第一步的结果是 int 或 long 类型的最小可表示值。

b. 该值必须非常大(大数量级的正值或正无穷大)，第一步的结果是 int 或 long 类型的最大可表示值。

2. 在第二步:

- 如果 T 是 int 或 long，转换的结果是第一步的结果。
- 如果T是byte、char或short，转换的结果是将第一步的结果收窄转换为类型T(§5.1.3)的结果。

根据 IEEE 754 关于二进制浮点格式之间转换的规则，使用四舍五入舍入策略（§15.4），从 double 到 float 的转换范围缩小。这种转换可能会失去精度，但也会失去范围，导致从非零双精度浮点数转换为浮点零，或从有限双精度浮点数转换为浮点无穷。double 类型的 NaN 转换为 float 类型的 NaN，double 类型的无穷大转换为同样有符号的 float 类型的无穷大。

尽管可能会发生溢出、下溢或其他信息丢失的情况，但收窄原生转换永远不会导致运行时异常(§11.1.1)。

例子 5.1.3-1. 收窄原生转换

```
class Test {
    public static void main(String[] args) {
        float fmin = Float.NEGATIVE_INFINITY;
        float fmax = Float.POSITIVE_INFINITY;
        System.out.println("long: " + (long)fmin +
                           ".." + (long)fmax);
        System.out.println("int: " + (int)fmin +
                           ".." + (int)fmax);
        System.out.println("short: " + (short)fmin +
                           ".." + (short)fmax);
        System.out.println("char: " + (int)(char)fmin +
                           ".." + (int)(char)fmax);
        System.out.println("byte: " + (byte)fmin +
                           ".." + (byte)fmax);
    }
}
```

程序产生输出:

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

char、int 和 long 的结果是意料之中的，产生了该类型的最小和最大可表示值。

byte 和 short 的结果会丢失有关数字值的符号和大小的信息，也会丢失精度。结果可以通过检查最小和最

大 int 的低位来理解。最小的整数的十六进制是 0x80000000，最大的整数是 0x7fffffff。 这解释了 short 的结果，即这些值的低 16 位，即 0x0000 和 0xffff；它解释了 char 的结果，这也是这些值的低 16 位，即 '\u0000'和'\uffff'；它解释了 byte 的结果，也就是这些值的低 8 位，即 0x00 和 0xff。

例子 5.1.3-2. 丢失信息的收窄原生转换

```
class Test {
    public static void main(String[] args) {
        // A narrowing of int to short loses high bits:
        System.out.println("(short)0x12345678==0x" +
        Integer.toHexString((short)0x12345678));
        // An int value too big for byte changes sign and magnitude:
        System.out.println("(byte)255==" + (byte)255);
        // A float value too big to fit gives largest int value:
        System.out.println("(int)1e20f==" + (int)1e20f);
        // A NaN converted to int yields zero:
        System.out.println("(int)NaN==" + (int)Float.NaN);
        // A double value too large for float yields infinity:

        System.out.println("(float)-1e100==" + (float)-1e100);
        // A double value too small for float underflows to zero:
        System.out.println("(float)1e-50==" + (float)1e-50);
    } }
```

程序产生输出:

```
(short)0x12345678==0x5678
(byte)255==-1
(int)1e20f==2147483647
(int)NaN==0
(float)-1e100==-Infinity (float)1e-50==0.0
```


5.1.4 拓宽和缩窄原生转换

以下转换结合了拓宽原生转换和缩窄原生转换:

• byte 到 char

首先，通过拓宽原生转换将 byte 转换为 int(§5.1.2)，然后通过缩窄原生转换将得到的 int 转换为 char(§5.1.3)。

5.1.5 拓宽引用转换

从任意引用类型 S 到任意引用类型 T 的拓宽引用转换存在，只要 S 是 T 的子类型(§4.10)。

拓宽引用转换在运行时从不需要特殊操作，因此也不会在运行时抛出异常。它们仅仅在于将引用视为具有某种可以在编译时被证明正确的其他类型。

null 类型不是引用类型(§4.1)，因此从 null 类型到引用类型的拓宽引用转换不存在。但是，许多转换上下文显式地允许将 null 类型转换为引用类型。

5.1.6 收窄引用转换

收窄引用转换将引用类型 S 的表达式视为不同引用类型 T 的表达式，其中 S 不是 T 的子类型。所支持的类型对在§5.1.6.1 中定义。与拓宽引用转换不同，收窄引用转换的类型不必直接相关。但是，当可以静态证明没有值可以同时为两种类型时，存在一些限制，禁止在某些类型对之间进行转换。

收窄引用转换可能需要在运行时进行测试，以验证类型 S 的值是否是类型 T 的合法值。然而，由于在运行时缺乏参数化的类型信息，一些转换无法通过运行时测试完全验证;它们在编译时被标记(§5.1.6.2)。对于可以通过运行时测试完全验证的转换，以及涉及参数化类型信息但仍可以在运行时部分验证的某些转换，如果测试失败，则抛出 ClassCastException(§5.1.6.3)。

5.1.6.1 允许的收窄引用转换

如果以下所有条件都为真，则存在从引用类型 S 到引用类型 T 的收窄引用转换:

- S 不是 T 的子类型 (§4.10)
- 如果存在一个参数化类型 X，它是 T 的超类型，和一个参数化类型 Y，它是 S 的超类型，这样 X 和 Y 的擦除是相同的，那么 X 和 Y 是不可证明的不同(§4.5)。

  使用来自 java.util 包的类型为例，不存在从 ArrayList<String>到 ArrayList<Object>的收窄引用转换，反之亦然，因为类型参数 String 和 Object 可证明是不同的。出于同样的原因，从 ArrayList<String>到 List<Object>之间不存在收窄引用转换，反之亦然。拒绝可证明的不同类型是一个简单的静态门，以防止“愚蠢的”收窄引用转换。

- 适用于下列情况之一:

  1. S 是一个类或接口类型，T 是一个类或接口类型，S 命名的类或接口与 T 命名的类或接口没有不相交(“不相交”的定义如下)。
  2. S 是类类型 Object 或接口类型 java.io.Serializable 或 Cloneable(数组实现的唯一接口(§10.8)), T 是数组类型。
  3. S 是一个数组类型 SC[]，即 SC 类型组件的数组; T 为数组类型 TC[]，即 TC 类型组件的数组; 存在从 SC 到 TC 的收窄引用转换。
  4. S 是一个类型变量，从 S 的上界到 T 存在一个收窄引用转换。
  5. T 是一个类型变量，从 S 到 T 的上界，要么存在拓宽引用转换，要么存在收窄引用转换。
  6. S 是一个交集类型 S_1 & ... & S_n, 对于所有的 i (1 ⩽ i ⩽ n), 存在从 S_i 到 T 的拓宽引用转换或收窄引用转换。
  7. T 是一个交集类型 T_1 & ... & T_n, 对于所有的 i(1 ⩽ i ⩽ n), 存在从 S 到 T_i 的拓宽引用转换或收窄引用转换。

如果可以静态地确定一个类或接口与另一个类或接口没有共同的实例(除了空值)，则该类或接口与另一个类或接口是不相交的。不相交的规则如下:

- 名为 C 的类与名为 I 的接口是不相交的，如果 (i) 不满足 C <: I, 并且 (ii) 适用于下列情况

之一:

- C 类定义为 final。
- C 是封闭的，C 的所有允许的直接子类都与 I 不相交。
- C 是可自由扩展的(§8.1.1.2)，I 是封闭的，C 与 I 的所有允许的直接子类和子接口是不相交的。

- 如果 C 与 I 不相交，则名为 I 的接口与名为 C 的类不相交。
- 一个名为 C 的类与另一个名为 D 的类是不相交的如果 (i) 不满足 C <: D, 并且 (ii) 不满足 D <: C。
- 一个名为 I 的接口与另一个名为 J 的接口不相交如果 (i) 不满足 I <: J, 并且 (ii) 不满足 J <: I, 并且 (iii) 适用于下列情况之一:
  - I 是封闭的，I 的所有允许的直接子类和子接口都与 J 不相交。
  - J 是封闭的，I 与 J 的所有允许的直接子类和子接口是不相交的。

  类是否为 final 对类是否与接口分离影响最大。考虑以下声明:

  ```
  interface I {} final class C {}
  ```

  因为 C 类是 final 类，并且没有实现 I，所以不可能有 C 的实例同时也是 I 的实例，所以 C 和 I 是不相交的。因此，没有从 C 到 I 的收窄引用转换。

  相比之下，考虑以下声明:

  ```
  interface J {} class D {}
  ```

  即使 D 类没有实现 J, D 的实例仍然可能是 J 的实例，例如，如果发生以下声明:

  ```
  class E extends D implements J {}
  ```

  因此，D 与 J 并没有不相交，D 到 J 之间存在一个收窄引用转换。

  上面的最后一个条款意味着两个可自由扩展的接口(§9.1.1.4)并不是不相交的。

5.1.6.2 检查和未检查的收窄引用转换

收窄引用转换是检查的或未检查的。这些术语指的是 Java 虚拟机验证转换的类型正确性的能力。

如果没有检查收窄引用转换，那么 Java 虚拟机将无法完全验证其类型的正确性，可能导致堆污染(§4.12.2)。要将此标记给程序员，未经检查的收窄引用转换会导致编译时未检查的警告，除非被@SuppressWarnings(§9.6.4.5)抑制。 相反，如果收缩引用转换没有未经检查;Java 虚拟机将能够完全验证其类型的正确性，因此在编译时不会给出警告。

未经检查的收窄引用转换如下:

• 从类型 S 到参数化类或接口类型 T 的收窄引用转换是未检查的，除非以下至少有一个为真:

- 所有 T 的类型参数是无边界通配符。
- T <: S, S 除了 T 没有子类型 X，其中 X 的类型参数不包含在 T 的类型参数中。

• 从类型 S 到类型变量 T 的收窄引用转换未检查。

• 如果存在 $T_i(1 \leqslant i \leqslant n)$，S 不是 T_i 的子类型并且从 S 到 T_i 的收窄引用转换是未检查的，那么从类型 S 到交集类型 $T_1 \& \ldots \& T_n$ 的收窄引用转换是未检查的。

5.1.6.3 运行时收窄引用转换

所有检查过的收缩引用转换都需要在运行时进行有效性检查。主要来说，这些转换是对没有参数化的类和接口类型。

有些未检查的收窄引用转换需要在运行时进行有效性检查。这取决于未检查的收窄引用转换是完全未检查还是部分未检查。部分未检查的收窄引用转换需要在运行时进行有效性检查，而完全未检查的收窄引用转换则不需要。

> 这些术语指的是转换中涉及的类型在被视为原始类型时的兼容性。如果转换在概念上是“上推”，那么转换是完全未检查的;不需要运行时测试，因为转换在 Java 虚拟机的非泛型类型系统中是合法的。相反地，如果转换在概念上是“向下转换”，那么转换是部分未检查的;即使在 Java 虚拟机的非泛型类型系统中，也需要运行时检查来测试转换中涉及的(原始)类型的兼容性。
>
> 使用 java.util 包中的类型举例，从 ArrayList<String>到 Collection<T>的转换是完全未检查的，因为 在 Java 虚拟机里，(原始) 类型 ArrayList 是 (原始) 类型 Collection 的子类型。相反地，从 Collection<T> 到 ArrayList<String>的转换是部分未检查的，因为 在 Java 虚拟机里，(原始) 类型 Collection 不是 (原始) 类型 ArrayList 的子类型。

未检查收窄引用转换的分类如下:

• 从 *S* 到一个非交集类型 *T* 的未检查收窄引用转换是完全未检查的，如果 |S| <: |T|.

否则，是部分未检查的。

• 从 S 到交集类型 $T_1 \& \ldots \& T_n$ 的未检查收窄引用转换是完全未检查的，如果 , 对所有 i (1 ⩽ i ⩽ n), S <: T_i 或者从 S 到 T_i 的收窄引用转换是完全未检查的。

否则，是部分未检查的。

已检查或部分未检查收窄引用转换的运行时有效性检查如下:

• 如果运行时的值为 null，则允许转换。

• 否则，设 R 为该值所指向的对象的类，设 T 为要转换到的类型的擦除(§4.6)。然后:

- 如果 R 是一个普通类(不是数组类):

> 如果 T 是一个类类型，那么 R 必须是与 T 相同的类(§4.3.4)或 T 的子类，否则抛出 ClassCastException 异常。

> 如果 T 是一个接口类型，那么 R 必须实现接口 T(§8.1.5)，否则抛出 ClassCastException 异常。

> 如果 T 是数组类型，则抛出 ClassCastException 异常。

- 如果 R 是一个接口:

  注意，当这些规则第一次应用于任何给定的转换时，R 不可能是接口，但如果规则是递归应用的，R 可能是接口，因为运行时引用的值可能引用元素类型为接口类型的数组.

> 如果 T 是一个类类型，那么 T 必须是 Object(§4.3.2)，否则抛出 ClassCastException 异常。

> 如果 T 是一个接口类型，那么 R 必须是与 T 相同的接口或 T 的子接口，否则抛出 ClassCastException 异常。

> 如果 T 是数组类型，则抛出 ClassCastException 异常。

> 如果 R 是一个表示数组类型 RC[]的类，即一个包含 RC 类型组件的数组:

> 如果 T 是一个类类型，那么 T 必须是 Object(§4.3.2)，否则抛出 ClassCastException 异常。

> 如果 T 是接口类型，那么 T 必须是 java.io.seriaiizabie 或 Cloneable(数组实现的唯一接口)类型，否则抛出 ClassCastException 异常。

> 如果 T 是一个数组类型 TC[]，即 TC 类型组件的数组，则抛出 ClassCastException 异常，除非 TC 和 RC 是相同的原生类型，或者 TC 和 RC 是引用类型，并且被允许递归地使用这些运行时规则。

如果转换是针对交集类型 T_1 & ... & T_n, 那么对所有 i ($1 \leqslant i \leqslant n$), 从 S 转换到 T_i所需的任何运行时检查也是转换到交集类型所需的。

5.1.7 装箱转换

装箱转换将原生类型的表达式视为相应引用类型的表达式。具体来说，以下 9 种转换称为装箱转换:

- 从类型 boolean 到类型 Boolean
- 从类型 byte 到类型 Byte
- 从类型 short 到类型 Short
- 从类型 char 到类型 Character
- 从类型 int 到类型 Integer

- 从类型 long 到类型 Long
- 从类型 float 到类型 Float
- 从类型 double 到类型 Double
- 从 null 类型到 null 类型

  这个规则是必要的，因为条件运算符(§15.25)对其操作数的类型进行装箱转换，并使用其结果进行进一步的计算。

在运行时，装箱转换过程如下:

- 如果 p 是 boolean 类型的值，那么装箱转换将 p 转换为 Boolean 类型的引用 r，这样 r.booleanValue () == p
- 如果 p 是 byte 类型的值，那么装箱转换将 p 转换为 Byte 类型的引用 r，这样 r.byteValue() == p
- 如果 p 是 char 类型的值，那么装箱转换将 p 转换为类型为 Character 的引用 r，这样 r. charValue () == p
- 如果 p 是 short 类型的值，那么装箱转换将 p 转换为 Short 类型的引用 r，这样 r.shortValue() == p
- 如果 p 是 int 类型的值，那么装箱转换将 p 转换为类型为 Integer 的引用 r，这样 r.intValue() == p
- 如果 p 是 long 类型的值，那么装箱转换将 p 转换为类型为 Long 的引用 r，这样 r.longValue() == p
- 如果 p 是 float 类型的值，那么:
  - 如果 p 不是 NaN，则装箱转换将 p 转换为类型为 Float 的引用 r，这样 r.p floatvalue() 就会计算为 p
  - 否则，装箱转换将 p 转换为类型为 Float 的引用 r，这样 r.isNaN()的计算结果为 true
- 如果 p 是 double 类型，那么:
  - 如果 p 不是 NaN，装箱转换将 p 转换为类型为 Double 的引用 r，这样 r.doulevalue() 就会计算为 p
  - 否则，装箱转换将 p 转换为类型为 Double 的引用 r，这样 r.isNaN()的计算结果为 true

• 如果 p 是任何其他类型的值，装箱转换等价于相等转换(§5.1.1)。

如果被装箱的值 p 是一个 bool 型、byte 型、char 型、short 型、int 型或 long 型的常量表达式(§15.29)的求值结果，并且结果是真、假;一个范围在'\u0000'到'\u007f'内的字符，或

者一个范围在-128 到 127 之间的整数，那么让 a 和 b 是 p 的任意两次装箱转换的结果。那么 a 总是等于 b。

> 理想情况下，装箱原生值将总是产生相同的引用。在实践中，使用现有的实现技术可能不可行。上述规则是一种务实的折衷，要求将某些公共值始终装箱到不可区分的对象中。实现可以延迟或立即缓存这些值。对于其他值，该规则不允许程序员对装箱值的一致性进行任何假设。这允许(但不要求)共享部分或所有这些引用。
>
> 这确保了在大多数情况下，行为将是理想的，而不会造成不适当的性能损失，特别是在小型设备上。例如，内存限制较少的实现可能会缓存所有 char 和 short 值，以及-32K 到+32K 范围内的 int 和 long 值。

如果需要分配包装类(Boolean、Byte、Character、Short、Integer、Long、Float 或 Double)的新实例，且可用存储空间不足，则装箱转换可能会导致 OutOfMemoryError。

5.1.8 拆箱转换

拆箱转换将引用类型的表达式视为相应原生类型的表达式。具体来说，以下八种转换被称为拆箱转换:

- 从类型 Boolean 到类型 boolean
- 从类型 Byte 到类型 byte
- 从类型 Short 到类型 short
- 从类型 Character 到类型 char
- 从类型 Integer 到类型 int
- 从类型 Long 到类型 long
- 从类型 Float 到类型 float
- 从类型 Double 到类型 double

在运行时，拆箱转换的过程如下:

- 如果 r 是 Boolean 类型的引用，那么拆箱转换将 r 转换为 r.booleanValue()
- 如果 r 是 Byte 类型的引用，则拆箱转换将 r 转换为 r.byteValue()
- 如果 r 是 Character 类型的引用，则拆箱转换将 r 转换为 r.charvalue ()
- 如果 r 是 Short 类型的引用，则拆箱转换将 r 转换为 r.shortValue()
- 如果 r 是 Integer 类型的引用，那么拆箱转换将 r 转换为 r. intvalue ()
- 如果 r 是 Long 类型的引用，则拆箱转换将 r 转换为 r. longvalue ()
- 如果 r 是 Float 类型的引用，则拆箱转换将 r 转换为 r.floatValue()
- 如果 r 是 Double 类型的引用，那么拆箱转换将 r 转换为 r.doubleValue()
- 如果 r 为 null, 拆箱转换抛出 NullPointerException 异常

如果一个类型是数字类型(§4.2)，则称为可转换为数字类型，或者它是一个引用类型，可以通过拆箱转换转换为数字类型。

如果类型是整型，则称该类型可转换为整型，或者该类型是可以通过拆箱转换转换为整型的引用类型。

5.1.9 未检查转换

G 是一个有 n 个类型参数的泛型类型声明。

存在从原始类型或接口类型(§4.8)G 到任何形式为 $G<T_1,...,T_n>$的参数化类型的未检查转换。

存在从原始数组类型 $G[]^k$ 到任何形式为 $G<T_1,...,T_n>[]^k$ 的数组类型的未检查转换。(符号$[]^k$ 表示 K 维的数组类型)

使用未检查转换会导致编译时未检查警告，除非所有类型参数 T(1⩽i⩽n)都是无边界的通配符(§4.5.1)，或者该警告被@SuppressWarnings(§9.6.4.5)抑制。

> 未检查转换用于实现在引入泛型类型之前编写的遗留代码与经过转换以使用泛型(我们称之为泛型化过程)的库的平滑互操作。在这种情况下(最明显的是，java.util 中的集合框架的客户端)，遗留代码使用原始类型(例如，Collection 而不是 Collection<string>)。原始类型的表达式作为参数传递给库方法，这些方法使用这些类型的参数化版本作为它们对应的形式参数的类型。
>
> 在使用泛型的类型系统下，不能显示这样的调用是静态安全的。拒绝这样的调用将使大量现有代码失效，并阻止它们使用更新版本的库。这反过来又会阻碍库供应商利用泛型。为了防止这种不受欢迎的事件转变，可以将原始类型转换为原始类型所引用的泛型类型声明的任意调用。虽然这种转换是不合理的，但作为对实用性的让步，它是可以容忍的。在这种情况下会发出未经检查的警告。

5.1.10 捕获转换

G 是一个有 n 个类型参数 $A_1,...,A_n$ 的泛型类型声明(§8.1.2, §9.1.2)，对应参数边界为 $U_1,...,U_n$。

存在从参数化类型 $G<T_1,...,T_n>$ (§4.5)到参数化类型 $G<S_1,...,S_n>$的捕获转换，其中 $1 \leqslant i \leqslant n$:

- 如果 T_i 是一个通配符类型参数(§4.5.1) 形式为 ?, 那么 S_i 是一个新的类型变量上界为 $U_i[A_1:=S_1,...,A_n:=S_n]$ 下界为 null 类型(§4.1)。
- 如果 T_i 是一个通配符类型参数(§4.5.1) 形式为 ? extends B_i 那么 S_i 是一个新变量，上界为 glb(B_i, $U_i[A_1:=S_1,...,A_n:=S_n]$) 下界为 null 类型。

  glb($V_1,...,V_m$) 定义为 V_1 & ... & V_m。

  如果任意两个类（非接口）V_i 和 V_j，V_i 不是 V_j 的子类，反之亦然，那么会产生编译错误。
- 如果 T_i 是一个通配符类型参数(§4.5.1) 形式为 ? super B_i, 那么 S_i 是一个新变量，上界为 $U_i[A_1:=S_1,...,A_n:=S_n]$ 下界为 B_i。
- 否则, Si = Ti。

对于参数化类型以外的任何类型的捕获转换(§4.5)充当相等转换(§5.1.1)。

捕获转换不会递归应用。

捕获转换在运行时从不需要特殊操作，因此在运行时也不会引发异常。

捕获转换旨在使通配符更加有用。为了理解这个动机，让我们先看看方法 java.util.Collections.reverse():

```
public static void reverse(List<?> list);
```

该方法反转作为参数提供的列表。它适用于任何类型的列表，因此使用通配符类型 List<?>作为形式参数的类型是完全合适的。

现在考虑一下如何实现 reverse():

```
public static void reverse(List<?> list) { rev(list); }
private static <T> void rev(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size() - i - 1));
    }
}
```

实现需要复制列表，从副本中提取元素，并将它们插入到原始列表中。为了以一种类型安全的方式完成此操作，我们需要为传入列表的元素类型提供一个名称 T。我们在私有服务方法 rev()中执行此操作。这要求我们传递输入的参数 list，类型为 list<?>，作为 rev()的参数。总之, List<?>是一个未知类型列表。对于任何 T，它不是 List<T>的子类型。允许这样的子类型关系是不合理的。考虑方法:

```
public static <T> void fill(List<T> l, T obj)
```

下面的代码将破坏类型系统:

```
List<String> ls = new ArrayList<String>();
List<?> l = ls;
Collections.fill(l, new Object()); // not legal - but assume it was String s =
ls.get(0); // ClassCastException - ls contains
                              // Objects, not Strings.
```

因此，如果没有一些特殊的豁免，我们可以看到从 reverse()到 rev()的调用将被禁止。如果是这种情况，reverse()的作者将被迫将其签名写成:

```
public static <T> void reverse(List<T> list)
```

这是不可取的，因为它向调用方暴露了实现信息。更糟糕的是，API 的设计者可能会认为使用通配符的签名是 API 的调用者所需要的，并且直到后来才意识到排除了类型安全的实现。

从 reverse()到 rev()的调用实际上是无害的，但不能基于 List<?>和 List<T>之间的一般子类型关系来证明它的合理性。这个调用是无害的，因为传入的参数无疑是某种类型的列表(尽管是未知类型)。如果我们可以在类型变量 X 中捕获这个未知类型，我们就可以推断 T 为 X。这就是捕获转换的本质。当然，规范必须处理复杂的问题，比如不平凡的(可能是递归定义的)上界或下界，存在多个参数等。

数学上有经验的读者会希望将捕获转换与已建立的类型理论联系起来。 不熟悉类型理论的读者可以跳过此讨论—或者研究合适的文本，如 Benjamin Pierce 的《类型和编程语言》，然后重新阅读本节。

这里简要地总结了捕获转换与既定类型理论概念的关系。通配符类型是存在类型的一种受限形式。捕获转换大致对应于存在类型的值的打开。表达式 e 的捕获转换可以被认为是 e 在包含 e 的顶级表达式的作用域内的开放。

存在的经典 open 操作要求捕获的类型变量不能转义打开的表达式。与捕获转换对应的 open 总是在一个足够大的作用域内，以至于捕获的类型变量永远不能在该作用域外可见。该方案的优点是不需要 close 操作，正如 Atsushi Igarashi 和 Mirko virroli 在第 16 届欧洲面向对象编程会议(ECOOP 2002)会议上发表的关于参数类型的基于方差的子类型的论文中定义的那样。关于通配符的正式描述，请参阅 Mads Torgersen, Erik Ernst 和 Christian Plesner Hansen 在第 12 届面向对象编程基础研讨会(FOOL 2005)上的 Wild FJ。

5.1.11 字符串转换

通过字符串转换，任何类型都可以转换成 String 类型。

原始类型 T 的值 x 首先被转换为一个引用值，就好像把它作为一个参数赋给一个合适的类实例创建表达式(§15.9):

- 如果 T 是 boolean, 那么使用 new Boolean(*x*)。
- 如果 T 是 char, 那么使用 new Character(*x*)。
- 如果 T 是 byte, short, 或 int, 那么使用 new Integer(*x*)。
- 如果 T 是 long, 那么使用 new Long(*x*)。
- 如果 T 是 float, 那么使用 new Float(*x*)。
- 如果 T 是 double, 那么使用 new Double(*x*)。

然后通过字符串转换将该引用值转换为 String 类型。

现在只需要考虑引用值:

- 如果引用为 null, 它被转换为 "null"字符串 (四个 ASCII 字符 n, u, l, l)。
- 否则, 转换是通过不带参数地调用被引用对象的 toString 方法来执行的; 但是如果调用 toString 方法的结果为 null, 那么就使用字符串"null" 。

  toString 方法由原始类 Object 定义(§4.3.2)。许多类重写了它，特别是 Boolean、Character、Integer、Long、Float、Double 和 String。

5.1.12 被禁止的转换

任何不明确允许的转换都是禁止的。

5.2 赋值上下文

赋值上下文允许将表达式的值赋给一个变量(§15.26);表达式的类型必须转换为变量的类型。

赋值上下文允许使用以下方式之一:

• 相等转换 (§5.1.1)

• 拓宽原生转换 (§5.1.2)

• 拓宽引用转换 (§5.1.5)

• 拓宽引用转换后跟一个拆箱转换

• 拓宽引用转换后跟一个拆箱转换，再跟一个拓宽原生转换

• 装箱转换 (§5.1.7)

• 装箱转换后跟一个拓宽引用转换

• 拆箱转换 (§5.1.8)

• 拆箱转换后跟拓宽原生转换

如果应用上述转换后，得到的类型是原始类型(§4.8)，则可以应用未检查的转换(§5.1.9)。

此外，如果表达式是 byte、short、char 或 int 类型的常量表达式(§15.29):

• 如果变量的类型为 byte、short 或 char，并且常量表达式的值可以用变量的类型表示，则可以使用收窄原生转换。

• 如果变量的类型为 Byte、Short 或 Character，并且常量表达式的值分别以 byte、short 或 char 类型表示，则可以使用先收窄原语转换，然后进行装箱转换。

常量表达式的编译时收窄意味着如下代码:

```
byte theAnswer = 42;
```

是被允许的。如果不缩小范围，整数文字 42 具有 int 类型的事实将意味着需要转换为字节类型:

```
byte theAnswer = (byte)42;       // cast is permitted but not required
```


最后，可以将空类型的值（空引用是唯一这样的值）赋值给任何引用类型，从而生成该类型的空引用。

如果转换链包含两个不在子类型关系中的参数化类型（§4.10），则产生编译时错误。

这种非法链的一个例子是:

```
Integer, Comparable<Integer>, Comparable, Comparable<String>
```

链的前三个元素通过拓宽引用转换而相互关联，而最后一个条目通过未检查的转换从其前一个条目派生而来。但是，这不是有效的赋值转换，因为链包含两个参数化类型，即 Comparable<Integer > 和 Comparable < String > ，它们不是子类型。

如果表达式的类型可以通过赋值转换转换为变量的类型，我们称表达式（或其值）可赋值给变量，或者，等价地，表达式的类型与变量的类型是赋值兼容的。

赋值上下文中转换可能产生的异常是:

• 如果在应用上述转换后，结果值是一个对象，而不是变量类型的擦除（§4.6）的子类或子接口的实例，则为 ClassCastException。

这种情况只能由堆污染引起（§4.12.2）。在实践中，当字段的擦除类型或方法的擦除返回类型与其未擦除类型不同时，实现仅需要在访问参数化类型的对象的字段或方法时执行强制转换。

- 装箱转换导致的 OutOfMemoryError。
- 由于对空引用进行拆箱转换而导致的 NullPointerException。
- 涉及数组元素或字段访问的特殊情况下的 ArrayStoreException（§10.5，§15.26.1）。

例子 5.2-1. 原生类型的赋值

```
class Test {
    public static void main(String[] args) {
        short s = 12;      // narrow 12 to short
        float f = s;       // widen short to float
        System.out.println("f=" + f);
        char c = '\u0123';
        long l = c;        // widen char to long
        System.out.println("l=0x" + Long.toString(l,16));
        f = 1.23f;
        double d = f; // widen float to double
        System.out.println("d=" + d);
        }
      }
```

程序产生输出：

```
f=12.0 l=0x123 d=1.2300000190734863
```

但是，以下程序会产生编译时错误：

```
class Test {
    public static void main(String[] args) {

        short s = 123;
        char c = s; // error: would require cast
        s = c;         // error: would require cast
    }
}
```

因为并不是所有的 short 类型的值是 char 类型的值，也不是所有的 char 类型的值是 short 类型的值。

例子 5.2-2. 引用类型赋值

```
class Point { int x, y; }
class Point3D extends Point { int z; }
interface Colorable { void setColor(int color); }

class ColoredPoint extends Point implements Colorable {
      int color;
      public void setColor(int color) { this.color = color; }
    }
```

```
class Test {
    public static void main(String[] args) {
        // Assignments to variables of class type:
        Point p = new Point();
        p = new Point3D();
          // OK because Point3D is a subclass of Point
        Point3D p3d = p;
          // Error: will require a cast because a Point
          // might not be a Point3D (even though it is,
          // dynamically, in this example.)

        // Assignments to variables of type Object:
        Object o = p;           // OK: any object to Object
        int[] a = new int[3];
        Object o2 = a;          // OK: an array to Object

        // Assignments to variables of interface type:
        ColoredPoint cp = new ColoredPoint();
        Colorable c = cp;
          // OK: ColoredPoint implements Colorable

        // Assignments to variables of array type:
        byte[] b = new byte[4];
        a = b;
          // Error: these are not arrays of the same primitive type Point3D[]
        p3da = new Point3D[3];
        Point[] pa = p3da;
          // OK: since we can assign a Point3D to a Point p3da = pa;
          // Error: (cast needed) since a Point
          // can't be assigned to a Point3D
     }
}
```

下面的测试程序演示了引用值的赋值转换，但未能编译，如其注释中所述。 应将此示例与前一示例进行比较。

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        // Okay because ColoredPoint is a subclass of Point:
        p = cp;
        // Okay because ColoredPoint implements Colorable: Colorable c = cp;
        // The following cause compile-time errors because
        // we cannot be sure they will succeed, depending on
        // the run-time type of p; a run-time check will be
        // necessary for the needed narrowing conversion and
        // must be indicated by including a cast:
        cp = p; // p might be neither a ColoredPoint
                  // nor a subclass of ColoredPoint
        c = p; // p might not implement Colorable
```

```
    }
}
```

例子 5.2-3. 数组类型赋值

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }

class Test {
    public static void main(String[] args) {
    long[] veclong = new long[100];
    Object o = veclong;              // okay
        Long l = veclong;            // compile-time error
        short[] vecshort = veclong; // compile-time error Point[] pvec = new
        Point[100];
        ColoredPoint[] cpvec = new ColoredPoint[100];
        pvec = cpvec;                // okay
        pvec[0] = new Point();       // okay at compile time,
                                     // but would throw an
                                     // exception at run time cpvec = pvec; //
        compile-time error
    }

  }
```

在这个例子里:

- 不能将 veclong 的值赋给 Long 变量，因为 Long 是除 Object 之外的类类型。数组只能分配给兼容数组类型的变量，或者分配给 Object、Cloneable 或 java.io.Serializable 类型的变量。
- veclong 的值不能赋给 vecshort，因为它们是原生类型的数组，并且 short 和 long 不是同一原生类型。
- cpvec 的值可以赋给 pvec，因为任何可能是 ColoredPoint 类型表达式的值的引用都可以是 Point 类型变量的值。随后将新 Point 类型的值分配给 pvec 的组件将引发 ArrayStoreException（如果程序被纠正以便可以编译），因为 ColoredPoint 数组不能将 Point 实例作为组件的值。
- pvec 的值不能赋给 cpvec，因为不是每一个可能是 Point 类型表达式的值的引用都能正确地成为 ColoredPoint 类型变量的值。如果运行时的 pvec 值是对 Point[]实例的引用，并且允许赋值给 cpvec，则对 cpvec 组件的简单引用（如 cpvec[0]）可以返回一个 Point，而 Point 不是 ColoredPoint。因此，允许这种赋值将允许违反类型系统。可使用强制类型转换（§5.5，§15.16）确保 pvec 引用 ColoredPoint []:

```
cpvec = (ColoredPoint[])pvec; // OK, but may throw an
                                  // exception at run time
```

5. 3 调用上下文

调用上下文允许将方法或构造函数调用(§8.8.7.1，§15.9，§15.12)中的实参值赋给相应的形参。

严格调用上下文允许使用以下方式之一:

• 相等转换 (§5.1.1)

• 拓宽原生转换 (§5.1.2)

• 拓宽引用转换 (§5.1.5)

松散调用上下文允许更宽松的转换集，因为它们仅在使用严格调用上下文找不到适用声明的情况下用于特定调用。 松散调用上下文允许使用以下内容之一：

• 一个相等转换 (§5.1.1)

• 拓宽原生转换(§5.1.2)

• 拓宽引用转换 (§5.1.5)

• 拓宽引用转换后跟一个拆箱转换

• 拓宽引用转换后跟一个拆箱转换，再跟一个拓宽原生转换

• 装箱转换 (§5.1.7)

• 装箱转换后跟拓宽引用转换

• 拆箱转换 (§5.1.8)

• 拆箱转换后跟拓宽原生转换

如果在应用了为调用上下文列出的转换后，结果类型是原始类型（§4.8），则可以应用未经检查的转换（§5.1.9）。

空类型的值（空引用是唯一这样的值）可以赋值给任何引用类型。

如果转换链包含两个不在子类型关系中的参数化类型（§4.10），则产生编译时错误。

调用上下文中可能出现的异常是：

• 如果在应用上述类型转换后，结果值是一个对象，而不是相应形式参数类型的擦除（§4.6）的子类或子接口的实例，则为 ClassCastException。

• 装箱转换导致的 OutOfMemoryError。

• 由于对空引用进行拆箱转换而导致的 NullPointerException。

严格调用上下文和松散调用上下文都不包括整数常量表达式的隐式收窄，这在赋值上下文中是允许的。Java 编程语言的设计者认为，包括这些隐式收窄转换将增加重载解析规则的复杂性（§15.12.2）。

因此，程序:

```
class Test {
    static int m(byte a, int b) { return a+b; }
    static int m(short a, short b) { return a-b; }
    public static void main(String[] args) {
           System.out.println(m(12, 2));  // compile-time error
  }
}
```

导致编译时错误，因为整数文字 12 和 2 的类型为 int，因此在重载解析规则下，两个参数方法 m

都不匹配。包含整数常量表达式隐式收窄的语言需要额外的规则来解决此类情况。

5.4 字符串上下文

字符串上下文仅适用于二元+运算符的操作数，当另一个操作数是字符串时，该操作数不是字符串。

这些上下文中的目标类型始终为字符串，非字符串操作数的字符串转换（§5.1.11）始终发生。然后按照§15.18.1 的规定对+运算符进行计算。

5.5 强制转换上下文

强制转换上下文允许强制转换表达式（§15.16）的操作数转换为强制转换运算符显式命名的类型。与赋值上下文和调用上下文相比，强制转换上下文允许使用§5.1 中定义的更多转换，并允许这些转换的更多组合。

如果表达式是原生类型，则强制转换上下文允许使用以下内容之一：

- 相等转换 (§5.1.1)
- 拓宽原生转换(§5.1.2)
- 缩窄原生转换 (§5.1.3)
- 拓宽和缩窄原生转换(§5.1.4)
- 装箱转换(§5.1.7)
- 装箱转换跟一个拓宽引用转换 (§5.1.5)

如果表达式是引用类型，则强制转换上下文允许使用以下内容之一：

- 相等转换 (§5.1.1)
- 拓宽引用转换 (§5.1.5)
- 拓宽引用转换跟一个拆箱转换
- 拓宽引用转换跟一个拆箱转换，再跟一个拓宽原生转换
- 收窄引用转换 (§5.1.6)
- 收窄引用转换跟一个拆箱转换
- 拆箱转换(§5.1.8)
- 拆箱转换跟一个拓宽原生转换

如果表达式具有空类型，则表达式可以转换为任何引用类型。

如果转换上下文使用已检查或部分未检查的收窄引用转换（§5.1.6.2、§5.2.6.3），则将对表达式值的类执行运行时检查，可能导致 ClassCastException。否则，不执行运行时检查。

如果表达式可以通过强制转换而不是未检查的收窄引用转换转换为引用类型，则我们称该表达式（或其值）与引用类型向下转换兼容。

下表列出了在某些强制转换上下文中使用的转换。每个转换由一个符号表示：

- -表示不允许转换
- ≈表示相等转换（§5.1.1）
- ω表示拓宽原生转换（§5.1.2）
- η表示收窄原生转换 (§5.1.3)
- ωη表示拓宽和收窄原生转换 (§5.1.4)
- ⇑表示拓宽引用转换 (§5.1.5)
- ⇓表示收窄引用转换 (§5.1.6)
- ⊕表示装箱转换 (§5.1.7)
- ⊗表示拆箱转换 (§5.1.8)

在表中，符号之间的逗号表示强制转换上下文先使用一个转换，然后使用另一个转换。Object 类型表示除了 8 个包装类以外的任何引用类型，这些包装类是 Boolean, Byte, Short, Character, Integer, Long, Float, Double。

表 5.5-A. 强制转换到原生类型

To → **From ↓**	byte	short	char	int	long	float	double	boolean
byte	≈	ω	ωη	ω	ω	ω	ω	-
short	η	≈	η	ω	ω	ω	ω	-
char	η	η	≈	ω	ω	ω	ω	-
int	η	η	η	≈	ω	ω	ω	-
long	η	η	η	η	≈	ω	ω	-
float	η	η	η	η	η	≈	ω	-
double	η	η	η	η	η	η	≈	-
boolean	-	-	-	-	-	-	-	≈
Byte	⊗	⊗,ω	-	⊗,ω	⊗,ω	⊗,ω	⊗,ω	-
Short	-	⊗	-	⊗,ω	⊗,ω	⊗,ω	⊗,ω	-
Character	-	-	⊗	⊗,ω	⊗,ω	⊗,ω	⊗,ω	-
Integer	-	-	-	⊗	⊗,ω	⊗,ω	⊗,ω	-
Long	-	-	-	-	⊗	⊗,ω	⊗,ω	-
Float	-	-	-	-	-	⊗	⊗,ω	-
Double	-	-	-	-	-	-	⊗	-
Boolean	-	-	-	-	-	-	-	⊗
Object	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗

表 5.5-B. 强制转换到引用类型

To → / From ↓	Byte	Short	Character	Integer	Long	Float	Double	Boolean	Object
byte	⊕	-	-	-	-	-	-	-	⊕,⇑
short	-	⊕	-	-	-	-	-	-	⊕,⇑
char	-	-	⊕	-	-	-	-	-	⊕,⇑
int	-	-	-	⊕	-	-	-	-	⊕,⇑
long	-	-	-	-	⊕	-	-	-	⊕,⇑
float	-	-	-	-	-	⊕	-	-	⊕,⇑
double	-	-	-	-	-	-	⊕	-	⊕,⇑
boolean	-	-	-	-	-	-	-	⊕	⊕,⇑
Byte	≈	-	-	-	-	-	-	-	⇑
Short	-	≈	-	-	-	-	-	-	⇑
Character	-	-	≈	-	-	-	-	-	⇑
Integer	-	-	-	≈	-	-	-	-	⇑
Long	-	-	-	-	≈	-	-	-	⇑
Float	-	-	-	-	-	≈	-	-	⇑
Double	-	-	-	-	-	-	≈	-	⇑
Boolean	-	-	-	-	-	-	-	≈	⇑
Object	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓	≈

例子 5.5-1. 引用类型强制转换

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}
final class EndPoint extends Point {}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        Colorable c;
        // The following may cause errors at run time because
        // we cannot be sure they will succeed; this possibility
        // is suggested by the casts:
        cp = (ColoredPoint)p; // p might not reference an
```

```
                                // object which is a ColoredPoint
                                // or a subclass of ColoredPoint
        c = (Colorable)p; // p might not be Colorable
        // The following are incorrect at compile time because
        // they can never succeed as explained in the text:
        Long l = (Long)p;            // compile-time error #1
        EndPoint e = new EndPoint();
        c = (Colorable)e;            // compile-time error #2
    }
}
```

在这里，会出现第一个编译时错误，因为类类型 Long 和 Point 不相关(也就是说，它们不相同，而且都不是另一个的子类)，所以它们之间的强制转换总是会失败。

第二个编译时错误发生的原因是 EndPoint 类型的变量永远不能引用实现了 Colorable 接口的值。这是因为 EndPoint 是 final 类型，而 final 类型的变量总是持有与其编译时类型相同的运行时类型的值。因此，变量 e 的运行时类型必须完全是 EndPoint 类型，而 EndPoint 类型不实现 Colorable 接口。

例子 5.5-2. 数组类型强制转换

```
class Point {
   int x, y;
   Point(int x, int y) { this.x = x; this.y = y; }
   public String toString() { return "("+x+","+y+")"; }
}
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
     int color;
     ColoredPoint(int x, int y, int color) {
     super(x, y); setColor(color);
     }
     public void setColor(int color) { this.color = color; }
     public String toString() {
     return super.toString() + "@" + color;
     }
}

class Test {
   public static void main(String[] args) {
        Point[] pa = new ColoredPoint[4];
        pa[0] = new ColoredPoint(2, 2, 12);
        pa[1] = new ColoredPoint(4, 5, 24);
        ColoredPoint[] cpa = (ColoredPoint[])pa; System.out.print("cpa: {");
        for (int i = 0; i < cpa.length; i++)
            System.out.print((i == 0 ? " " : ", ") + cpa[i]);
        System.out.println(" }");
     }
}
```

程序编译正常，产生如下输出：

```
cpa: { (2,2)@12, (4,5)@24, null, null }
```

例子 5.5-3. 在运行时强制转换不兼容的类型

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
```

```
class ColoredPoint extends Point implements Colorable{
   int color;
   public void setColor(int color) { this.color = color; }
  }

class Test {
    public static void main(String[] args){
        Point[] pa = new Point[100];

        // The following line will throw a ClassCastException: ColoredPoint[] cpa
        = (ColoredPoint[])pa;
        System.out.println(cpa[0]);
        int[] shortvec = new int[2];
        Object o = shortvec;

        // The following line will throw a ClassCastException: Colorable c =
        (Colorable)o;
        c.setColor(0); } }
```

这个程序使用强制转换进行编译，但它在运行时抛出异常，因为类型不兼容。

5.6 数值上下文

数字上下文应用于算术运算符的操作数、数组创建和访问表达式、条件表达式以及 switch 表达式的结果表达式。

如果表达式是下列类型之一，则表达式出现在数值算术上下文中:

- 一元加操作符+、一元减操作符-或按位补操作符~的操作数(§15.15.3、§15.15.4、§15.15.5)。
- 乘法运算符*、/或%的操作数(§15.17)
- 数字类型加或减操作符的操作数+或-(§15.18.2)
- 移位操作符<<、>>或>>>的操作数(§15.19)。这些移位操作符的操作数被单独处理，而不是作为一组处理。Long 类型的移动距离(右操作数)不会将被移动的值(左操作数)提升为 long 类型。
- 数值比较运算符的操作数 <, <=, >, or >= (§15.20.1)
- 数值相等运算符的操作数== or != (§15.21.1)
- 整数按位操作符的操作数 &,^,or | (§15.22.1)

如果表达式是下列类型之一，则表达式出现在数值数组上下文中:

- 数组创建表达式中的维度表达式 (§15.10.1)
- 数组创建表达式中的索引表达式 (§15.10.3)

如果表达式是以下类型之一，则表达式出现在数值选择上下文中:

- 数值条件表达式的第二个或第三个操作数 (§15.25.2)
- 一个独立 switch 表达式(§15.28.1)的结果表达式，其中所有的结果表达式都可转换为数字类型

数字提升决定数字上下文中所有表达式的提升类型。选择提升类型，使得每个表达式都可以转换为提升类型，并且在算术运算的情况下，为提升类型的值定义运算。数值上下文中表达式的顺序对于数值提升并不重要。规则如下：

1. 如果任何表达式为引用类型，则应进行拆箱转换（§5.1.8）。

2. 接下来，根据以下规则，将拓宽原生转换（§5.1.2）和收窄原生转换（§5.1.3）应用于某些表达式：

   - 如果任何表达式的类型为 double，则提升后的类型为 double，其他非 double 类型的表达式将经历拓展原生转换为 double。
   - 否则，如果任何表达式的类型为 float，则提升后的类型为 float，其他非 float 类型的表达式将经历拓展原生转换为 float。
   - 否则，如果任何表达式的类型为 `long`，则提升后的类型为 `long`，其他非 long 类型的表达式将经历拓展原生转换为 long。
   - 否则，没有表达式的类型为 double，float，或 long。在这种情况下，上下文的类型决定了提升类型的选择方式。

     在数值算术上下文或数值数组上下文中，提升的类型为 int，任何不属于 int 类型的表达式都将经历拓展原生转换为 int。

     在数字选择上下文中，以下规则适用：

   - 如果任何表达式的类型为 int，并且不是常量表达式（§15.29），则提升后的类型为 int，其他非 int 类型的表达式将进行拓展原生转换为 int。

   - 否则，如果任何表达式都是 short 类型，并且所有其他表达式都是 short 类型或 byte 类型，或者是 int 类型的常量表达式，其值可在 short 类型中表示，则提升后的类型为 short，byte 表达式将经历到 short 的拓展原生转换，而 int 表达式将经历到 short 的收窄原生转换。

   - 否则，如果任何表达式都是 byte 类型，并且其他所有表达式都是 byte 类型或 int 类型的常量表达式，其值可在 byte 类型中表示，则提升后的类型是 byte，int 表达式将经历收窄原生转换为 byte。

   - 否则，如果任何表达式都是 char 类型，并且其他表达式都是 char 类型或 int 类型的常量表达式，其值可在 char 类型中表示，则提升的类型是 char，int 表达式将经历收窄原生转换为 char。

- 否则，提升的类型为 int，所有不属于 int 类型的表达式都将经历拓展原生转换为 int。

一元数值提升包括对数值算术上下文或数值数组上下文中出现的单个表达式应用数值提升。

二元数值提升包括对数值算术上下文中出现的一对表达式应用数值提升。

一般的数字提升包括对数字选择上下文中出现的所有表达式应用数字提升。

例子 5.6-1. 一元数值提升

```
class Test {
    public static void main(String[] args){
        byte b = 2;
        int[] a = new int[b]; // dimension expression promotion
        char c = '\u0001';
        a[c] = 1;              // index expression promotion
        a[0] = -c;             // unary - promotion
        System.out.println("a: " + a[0] + "," + a[1]);
        b = -1;
        int i = ~b;            // bitwise complement promotion
        System.out.println("~0x" + Integer.toHexString(b)
                           + "==0x" + Integer.toHexString(i));
        i = b << 4L;           // shift promotion (left operand)
        System.out.println("0x" + Integer.toHexString(b)
                           + "<<4L==0x" + Integer.toHexString(i));
  }
}
```

程序产生输出:

```
a: -1,1
~0x==0x0
0x<<4L==0x
```

例子 5.6-2. 二元数值提升

```
class Test {
    public static void main(String[] args){
        int i = 0;
        float f = 1.0f;
        double d = 2.0;
        // First int*float is promoted to float*float, then
        // float==double is promoted to double==double:
        if (i * f == d) System.out.println("oops");
        // A char&byte is promoted to int&int:
        byte b = 0x1f;
        char c = 'G';
        int control = c & b;
        System.out.println(Integer.toHexString(control));

        // Here int:float is promoted to float:float: f = (b==0) ? i : 4.0f;
        System.out.println(1.0/f);
    }
}
```

程序产生输出:

```
7
0.25
```

该示例通过屏蔽除字符的低 5 位以外的所有字符，将 ASCII 字符 G 转换为 ASCII control-G (BEL)。7 是这个控制字符的数值。