.

第十五章

# 表达式

程序中的大部分工作都是通过计算表达式来完成的，要么是为了计算表达式的副作用，例如对变量的赋值，要么是计算表达式的值，这些值可以用作更大表达式中的参数或操作数，要么是影响语句中的执行顺序，或者两者兼而有之。

本章规定了表达式的含义及其计算规则。

## 15.1 计算、表示和结果

当计算（执行）程序中的表达式时，结果表示三种情况之一：

- 一个变量(§4.12) (在 C 中，这将被称为左值)
- 一个值(§4.2, §4.3)
- 什么都不是(表达式为 void)

如果一个表达式表示一个变量，并且在进一步求值时需要一个值，则使用该变量的值。在这种情况下，如果表达式表示变量或值，我们可以简单地说成表达式的值。

当且仅当是调用不返回值的方法的方法调用(§15.12)，即被声明为 void 的方法(§8.4)时，表达式才表示无意义。这样的表达式只能用作表达式语句(§14.8)或作为 lambda 体的单个表达式(§15.27.2)，因为表达式可以出现的每一个其他上下文都需要该表达式来表示某种东西。方法调用的表达式语句或 lambda 体也可以调用产生结果的方法；在这种情况下，该方法返回的值被悄悄丢弃。

表达式的求值可能会产生副作用，因为表达式可能包含嵌入的赋值、递增运算符、递减运算符、方法调用，在 switch 表达式中还可能包含任意语句。

表达式出现在以下任一项中：

- 正在声明的某个类或接口的声明: 在字段初始化器、静态初始化器、实例初始化器、构造函数声明、方法声明或注解中。
- 模块、包或顶级类或接口声明上的注解。

## 15.2 表达式的形式

表达可以大致分为以下句法形式之一：

- 表达式名称 (§6.5.6)
- Primary 表达式(§15.8 - §15.13)
- 一元运算符表达式(§15.14 - §15.16)
- 二元运算符表达式 (§15.17 - §15.24, 和 §15.26)
- 三元运算符表达式 (§15.25)
- Lambda 表达式(§15.27)
- switch 表达式 (§15.28)

操作符之间的优先级由语法生成的层次结构来管理。优先级最低的运算符是 lambda 表达式的箭头(->)，后跟赋值运算符。因此，所有表达式在语法上都包含在 LambdaExpression 和 AssignmentExpression 非终结符中：

*Expression:*
*LambdaExpression*
*AssignmentExpression*



当某些表达式出现在特定上下文中时，它们被视为多元表达式。以下形式的表达式可以是多元表达式：

- 括号表达式 (§15.8.5)
- 类实例创建表达式(§15.9)
- 方法调用表达式(§15.12)
- 方法引用表达式(§15.13)
- 条件表达式(§15.25)
- lambda 表达式(§15.27)
- switch 表达式(§15.28)

确定这些形式之一的表达式是否为多元表达式的规则在指定这些表达式的形式的单独章节中给出。

不是多元表达式的表达式是独立表达式。当确定不是多元表达式时，独立表达式是上述形式的表达式，以及所有其他形式的所有表达式。所有其他形式的表达式都有一个独立的形式。

有些表达式有一个可以在编译时确定的值。这些都是常量表达式(§15.29)。

## 15.3 表达式的类型

如果表达式表示变量或值，则表达式在编译时具有已知的类型。独立表达式的类型完全可以从表达式的内容确定; 相反，多元表达式的类型可能会受到表达式的目标类型的影响(§5)。下面将对每种表达式分别解释确定表达式类型的规则。

表达式的值与表达式的类型是赋值兼容的(§5.2)，除非发生堆污染(§4.12.2)。

同样，存储在变量中的值总是与变量的类型兼容，除非发生堆污染。

换句话说，类型为 T 的表达式的值总是适合赋值给类型为 T 的变量。

注意，如果表达式的类型是命名类 C 的类类型，那么将 C 类声明为 final 或密封的(§8.1.1.2)会暗示表达式的值:

• 如果 C 是 final，那么表达式一定有一个值(i)是空引用，或者(ii)是一个类本身是 C 的对象，因为 final 类没有子类。

• 如果 C 是密封的，那么表达式一定有一个值是:(i)空引用，(ii)类本身是 C 的对象，或(iii)与 C 的一个允许的直接子类赋值兼容(§8.1.6)。

• 如果 C 是可自由扩展的，那么表达式可以保证有一个值是:(i)空引用，(ii)类本身是 C 的对象，或(iii)与 C 赋值兼容。



## 15.4 浮点表达式

浮点表达式是类型为 float 或 double 的表达式。float 类型的浮点表达式表示与 32 位 IEEE 754 binary32 格式所表示的值完全对应的值。double 类型的浮点表达式表示与 64 位 IEEE 754 binary64 格式所表示的值完全对应的值。

Java 编程语言中许多可用于形成浮点表达式的比较和数值运算符都对应于 IEEE 754 操作，作用于浮点值的转换也是如此(表 15.4-A)。

表 15.4-A. 与 IEEE 754 操作的联系

| 操作/转换 | IEEE 754 操作 |
| --- | --- |
| 数值比较操作符<,<=,>,和>=(§15.20.1) | compareQuietLess, compareQuietLessEqual,<br>compareQuietGreater,<br>compareQuietGreaterEqual |
| 数值相等操作符==和!=(§15.21.1) | compareQuietEqual, compareQuietNotEqual |
| 一元减法操作符(§15.15.4) | negate |

| | |
|---|---|
| 乘法运算符*和/(§15.17.1, §15.17.2) | multiplication, division |
| 加性运算符+和-(§15.18.2) | addition, subtraction |
| 从整型拓宽原生转换(§5.1.2) | convertFromInt |
| 收窄原生转换到整型(§5.1.3) | convertToIntegerTowardZero |
| float 和 double 之间的转换 | convertFormat |

浮点余数运算符%(§15.17.3)与 IEEE 754 余数运算不对应。

一些在 Java 编程语言中没有相应运算符的 IEEE 754 运算是通过 Math 和 StrictMath 类中的方法提供的，这些方法包括用于 IEEE 754 squareRoot 运算的 sqrt 方法、用于 IEEE 754 fusedMultiplyAdd 运算的 fma 方法以及用于 IEEE 754 余数运算的 IEEEremainder 方法。

Java 编程语言需要 IEEE 754 次正常浮点数和渐进式下溢的支持，这使得证明特定数值算法的理想特性变得更加容易。如果计算结果是次正常数，浮点运算不会“刷新到零”。

Java 编程语言的浮点运算符的结果必须与相同操作数上的相应 IEEE 754 运算的结果相匹配。对于有限结果，这意味着浮点结果的符号、有效数和指数必须都是 IEEE 754 规定的。

匹配符号、有效数和指数的要求排除了在不太精确地指定浮点行为时可能允许的某些转换。例如，-x 通常不能用（0.0-x）代替，因为如果 x 为-0.0，结果的符号将不同。 此外，不允许使用其他可能改变值的转换，例如用对融合乘法累加库方法的调用替换（a*b+c），除非可以证明结果相同。

浮点表达式的求值在任何情况下都不能使用比表达式类型所指示的精度更高或指数范围更大的中间结果。

溢出的浮点操作产生一个带符号的无穷大。

下溢的浮点运算产生一个低于正常值或带符号的零。

没有唯一数学定义结果的浮点操作产生 NaN。

所有使用 NaN 作为操作数的数值运算结果都是 NaN。

由于 NaN 是无序的，任何涉及一个或两个 NaN 的数值比较操作都会返回 false，任何涉及 NaN 的==比较都会返回 false，而涉及 NaN 的任何!=比较返回 true。

浮点运算是实数运算的近似。虽然实数有无限个，但特定的浮点格式只有有限的值。在 Java 编程语言中，舍入策略是一个函数，用于将给定格式的实数映射到浮点值。 对于浮点格式的可表示范围内的实数，实数线的连续段被映射为单个浮点值。在数值上等于浮点值的实数被映射到该浮点值; 例如，实数 1.5 以给定的格式映射到浮点值 1.5。Java 编程语言定义了两种舍入策略，如下所示:

• 四舍五入策略适用于所有浮点操作符，但(i)转换为整数值和(ii)浮点余数除外。在四舍五入策略下，不精确的结果必须四舍五入到最接近无限精确结果的可表示值; 如果两个最近的可表示值接近相等，则选择其最低有效位为零的值。

四舍五入策略对应于 IEEE 754 中二进制算法的默认四舍五入方向属性 roundTiesToEven。

roundTiesToEven 四舍五入方向属性在 1985 年版本的 IEEE 754 标准中被称为“四舍五入”模式。Java 编程语言中的舍入策略就是以这种舍入模式命名的。

• 向零舍入策略适用于:(i)浮点值到整数值的转换(§5.1.3)和(ii)浮点余数(§15.17.3)。在向零舍入策略下，不精确的结果被舍入到最接近的可表示值，该值的大小不大于无限精确的结果。对于转换为整数，向零舍入策略等效于舍弃小数有效位的截断。

向零舍入策略对应于 IEEE 754 中二进制算术的 roundTowardZero 舍入方向属性。

roundTowardZero 舍入方向属性在 1985 年版本的 IEEE 754 标准中称为“向零舍入”舍入模式。Java 编程语言中的舍入策略以这种舍入模式命名。

Java 编程语言要求每个浮点运算符将其浮点结果舍入到结果精度。如上所述，用于每个浮点运算符的舍入策略是四舍五入舍入策略或向零舍入策略。

Java1.0 和 1.1 要求严格计算浮点表达式。严格计算意味着每个 float 操作数对应于 IEEE 754 binary32 格式中可表示的值，每个 double 操作数对应 IEEE 754 binary64 格式中可表示的值，并且具有相应 IEEE 754 操作的每个浮点运算符与相同操作数的 IEEE 754 结果相匹配。

严格的计算提供了可预测的结果，但在 Java 1.0/1.1 时代常见的一些处理器系列的 Java 虚拟机实现中造成了性能问题。因此，在 Java1.2 到 JavaSE 16 中，JavaSE 平台允许 Java 虚拟机实现具有与每个浮点类型相关联的一个或两个值集。float 类型与 float 值集和 float 扩展指数值集相关联，而 double 类型与 double 值集和 double 扩展指数值集中相关联。float 值集对应于 IEEE 754 binary32 格式中可表示的值；float 扩展指数值集具有相同的精度位数，但指数范围更大。类似地，double 值集对应于 IEEE 754 binary64 格式中可表示的值；double 扩展指数值集具有相同的精度位数，但指数范围更大。默认情况下允许使用扩展的指数值集可以改善某些处理器系列的性能问题。

出于兼容性考虑，Java 1.2 允许程序员禁止实现使用扩展指数值集。程序员通过在类、接口或方法的声明上放置 strictfp 修饰符来表达这一点。strictfp 约束任何封闭表达式的浮点语义，以使用浮点表达式的浮点值集和双精度表达式的双精度值集，从而确保此类表达式的结果是完全可预测的。因此，由 strictfp 修改的代码具有与 Java 1.0 和 1.1 中指定的相同的浮点语义。

在 Java SE 17 和更高版本中，Java SE 平台始终要求严格计算浮点表达式。在实现严格计算时遇到性能问题的处理器家族的新成员不再有这种困难。此规范不再将 float 和 double 与上述四个值集相关联，并且 strictfp 修饰符不再影响浮点表达式的计算。出于兼容性考虑，虽然鼓励 Java 编译器就其过时状态向程序员发出警告，但在 Java SE 19(§3.8)中 strictfp 仍是一个关键字，并继续对其使用进行限制(§8.4.3、§9.4)。Java 编程语言的未来版本可能会重新定义或删除 strictfp 关键字。

## 15.5 表达式和运行时检查

如果表达式的类型是原生类型，则该表达式的值也属于该原生类型。

如果表达式的类型是引用类型，则在编译时不一定知道被引用对象的类，甚至值是否是对对象的引用而不是 null。在 Java 编程语言中有一些地方，引用对象的实际类以无法从表达式类型推断的方式影响程序执行。它们如下:

· 方法调用(§15.12)。用于调用 o.m（…）的特定方法是基于作为 o 类型的类或接口的一部分的方法选择的。对于实例方法，o 的运行时值引用的对象的类参与，因为子类可能重写

父类中已声明的特定方法，从而调用该重写方法。 (重写方法可以或可以不选择进一步调用原始重写的 m 方法。)

· instanceof 操作符 (§15.20.2)。类型为引用类型的表达式可以使用 instanceof 进行测试，以确定表达式的运行时值引用的对象的类是否可以转换为其他引用类型。A

· 强制类型转换 (§15.16)。操作数表达式的运行时值引用的对象的类可能与强制转换运算符指定的类型不兼容。对于引用类型，这可能需要运行时检查，如果在运行时确定的引用对象的类无法转换为目标类型，则会引发异常。

· 赋值给引用类型的数组组件 (§10.5, §15.13, §15.26.1). 如果 S 是 T 的子类型，则类型检查规则允许将数组类型 S[]视为 T[]的子类型，但这需要对数组组件的赋值进行运行时检查，类似于对强制转换执行的检查。

· 异常处理(§14.20). 只有当抛出的异常对象的类是 catch 子句的形式参数类型的实例时，catch 子句才会捕获异常。只有当抛出的异常对象的类是 catch 子句的形式参数类型的实例时，异常才被 catch 子句捕获。

对象的类不是静态已知的情况可能会导致运行时类型错误。

此外，在某些情况下，静态已知的类型在运行时可能不准确。在产生编译时未检查警告的程序中可能会出现这种情况。这样的警告是针对那些不能静态地保证安全的操作给出的，也不能立即进行动态检查，因为它们涉及到不可具体化的类型(§4.7)。因此，程序执行过程中后期的动态检查可能会检测到不一致性，并导致运行时类型错误。

运行时类型错误只能在以下情况下发生：

· 在转换中，当操作数表达式的值引用的对象的实际类与强制转换运算符指定的目标类型不兼容时（§5.5，§15.16）；在这种情况下，将抛出 ClassCastException。

· 在自动生成的强制转换中引入，以确保对不可具体化类型的操作的有效性（§4.7）。

· 在对引用类型的数组组件的赋值中，当要赋值的值所引用的对象的实际类与数组的实际运行时组件类型不兼容时（§10.5、§15.13、§15-26.1）；在这种情况下，将引发 ArrayStoreException。

· 当 try 语句的任何 catch 子句未捕获异常时（§14.20）；在这种情况下，遇到异常的控制线程首先尝试调用未捕获异常处理程序（§11.3），然后终止。

## 15.6 计算的正常和突然完成

每个表达式都有一个正常的计算模式，其中执行某些计算步骤。以下各节描述了每种表达式的正常计算模式。

如果执行所有步骤时没有引发异常，则表示表达式正常完成。

但是，如果表达式的求值引发异常，则称该表达式突然完成。突然完成总是有一个相关的

原因，这总是一个给定值的抛出。

运行时异常由预定义的操作符抛出，如下所示:

· 如果可用内存不足，则类实例创建表达式(§15.9.4)、数组创建表达式(§15.10.2)、方法引用表达式(§15.13.3)、数组初始化器表达式(§10.6)、字符串连接操作符表达式(§15.18.1)或 lambda 表达式(§15.27.4)都会抛出 OutOfMemoryError 错误。

· 如果任何维数表达式的值小于 0，则数组创建表达式(§15.10.2)会抛出 NegativeArraySizeException 异常。

· 如果数组引用表达式的值为 null，则数组访问表达式（§15.10.4）将抛出 NullPointerException。

· 如果数组索引表达式的值为负值或大于或等于数组长度，则数组访问表达式（§15.10.4）将引发 ArrayIndexOutOfBoundsException。

· 如果对象引用表达式的值为 null，则字段访问表达式（§15.11）将抛出 NullPointerException。

· 如果目标引用为 null，则调用实例方法的方法调用表达式（§15.12）将抛出 NullPointerException。

· 如果在运行时发现强制转换不允许，则强制转换表达式（§15.16）将抛出 ClassCastException。

· 如果右侧操作数表达式的值为零，整数除法（§15.17.2）或整数余数（§15-17.3）运算符将抛出 ArithmeticException。

· 对引用类型的数组组件（§15.26.1）、方法调用表达式（§15.12）或前缀或后缀增量（§15.14.2，§15.15.1）或减量运算符（§15.14.3，§15.15.2）的赋值都可能由于装箱转换（§5.1.7）而引发 OutOfMemoryError。

· 当要赋值的值与数组的组件类型不兼容时，对引用类型的数组组件的赋值（§15.26.1）将引发 ArrayStoreException（§10.5）。

· 如果没有 switch 标签与选择器表达式的值匹配，则 switch 表达式（§15.28）将抛出 IncompatibleClassChangeError。

如果发生异常导致方法体的执行突然完成，则方法调用表达式也可能导致抛出异常。

如果发生异常导致构造函数的执行突然完成，则类实例创建表达式也可能导致引发异常。

在表达式的求值过程中也可能出现各种链接和虚拟机错误。就其性质而言，此类错误难以预测和处理。

如果发生异常，则一个或多个表达式的计算可以在其正常计算模式的所有步骤完成之前终止；这种表达式被称作是突然完成的。

如果表达式的求值需要对子表达式求值，则子表达式的突然完成总是导致表达式本身立即突然完成，原因相同，并且不执行正常求值模式中的所有后续步骤。

术语“正常完成”和“突然完成”也适用于语句的执行（§14.1）。语句可能由于各种原因而突然完成，而不仅仅是因为抛出了异常。

## 15.7 计算顺序

Java 编程语言保证运算符的操作数看起来按照特定的求值顺序求值，即从左到右。

> 建议代码不要严重依赖本规范。当每个表达式最多包含一个副作用(作为其最外层的操作)时，并且当代码不确切地依赖于从左到右的表达式求值所产生的异常时，代码通常更清晰。

### 15.7.1 首先计算左操作数

在计算右操作数的任何部分之前，二元运算符的左操作数似乎已被完全计算。

如果运算符是复合赋值运算符(§15.26.2)，则左操作数的计算包括记住左操作数表示的变量，以及获取和保存该变量的值以用于隐含的二元运算。如果二元运算符的左操作数的计算突然完成，则似乎没有计算右操作数的任何部分。

**例子 15.7.1-1. 首先计算左操作数**

在下面的程序中，*运算符具有一个左操作数和一个右操作数，前者包含对变量的赋值，后者包含对同一变量的引用。引用产生的值将反映赋值首先发生的事实。

```
class Test1 {
    public static void main(String[] args) {
        int i = 2;
        int j = (i=3) * i;
        System.out.println(j);
    }
}
```

此程序生成以下输出：

```
9
```

*运算符的求值不允许生成 6 而不是 9。

**例子 15.7.1-2. 复合赋值运算符中的隐式左操作数**

在下面的程序中，在计算加运算符的右操作数之前，这两个赋值语句都获取并记住左操作数的值 9，此时变量设置为 3。

```
class Test2 {
    public static void main(String[] args) {
        int a = 9;
        a += (a = 3);  // first example
        System.out.println(a);
        int b = 9;
        b = b + (b = 3);  // second example
```

```
            System.out.println(b);
        }
    }
```

此程序生成以下输出：

```
12
12
```

不允许任何一种赋值(a 为复合，b 为简单)产生结果 6。

另请参阅§15.26.2 中的示例。

**例子 15.7.1-3. 突然完成左操作数的求值**

```
class Test3 {
    public static void main(String[] args) {
        int j = 1;
        try {
            int i = forgetIt() / (j = 2);
        } catch (Exception e) {
            System.out.println(e);
            System.out.println("Now j = " + j);
        }
    }

    static int forgetIt() throws Exception {
       throw new Exception("I'm outta here!");
    }
}
```



此程序生成以下输出：

```
java.lang.Exception: I'm outta here! Now j = 1
```

也就是说，运算符/的左操作数 forgetIt()在对右操作数求值及其嵌入的 2 到 j 赋值发生之前引发异常。

## 15.7.2 运算前计算操作数

Java 编程语言保证运算符的每个操作数（除了条件运算符&&、||和?:）在执行操作本身的任何部分之前都会被完全求值。

如果二元运算符是整数除/(§15.17.2)或整数余数%(§15.17.3)，则其执行可能会引发 ArithmeticException，但只有在二元运算符的两个操作数都已求值且仅当这些求值正常完成时才会引发此异常。

**例子 15. 7. 2-1. 运算数的运算前求值**

```
class Test {
    public static void main(String[] args) {
        int divisor = 0;
        try {
            int i = 1 / (divisor * loseBig());
        } catch (Exception e) {
```

```
                System.out.println(e);
            }
        }
        static int loseBig() throws Exception {
            throw new Exception("Shuffle off to Buffalo!");
        }
    }
```

此程序生成以下输出：

```
java.lang.Exception: Shuffle off to Buffalo!
```

而不是：

```
java.lang.ArithmeticException: / by zero
```

因为除法运算的任何部分，包括除以零异常的信号，可能看起来都不会在完成对 loseBig 的调用之前发生，即使该实现可能能够检测或推断除法运算肯定会导致除以零异常。

### 15.7.3 计算尊重括号和优先级

Java 编程语言遵循圆括号显式表示的求值顺序和操作符隐式表示的优先级顺序。

Java 编程语言的实现可能不会利用代数的特性，比如结合律，将表达式改写成更方便的计算顺序，除非它能证明替换表达式在值和可观察到的副作用上是等价的，即使是在多个线程执行的情况下(使用§17(线程和锁)中的线程执行模型)，可能涉及的所有可能的计算值。

在浮点计算的情况下，此规则也适用于无穷大值和非整数(NaN)值。

例如，!(x<y)不能重写为 x>=y，因为如果 x 或 y 为 NaN 或两者都为 NaN，这些表达式有不同的值。

具体地说，看起来与数学有关的浮点计算不太可能与计算有关。这种计算不能简单地重新排序。

例如，Java 编译器将 4.0*x*0.5 重写为 2.0*x 是不正确的；虽然舍入恰好不是这里的问题，但存在大的 x 值，第一个表达式会产生无穷大（因为溢出），但第二个表达式会生成有限的结果。

例如，测试程序：

```
class Test {
    public static void main(String[] args) {
         double d = 8E307;

        System.out.println(4.0 * d * 0.5);
        System.out.println(2.0 * d);
    }
}
```

打印：

```
Infinity
1.6E308
```

因为第一个表达式溢出，而第二个不溢出。

相反，在 Java 编程语言中，整数加法和乘法被证明是关联的。

例如，a+b+c，其中 a、b 和 c 是局部变量(这个简化的假设避免了涉及多个线程和 volatile 变量的问题)，无论计算为(a+b)+c 还是 a+(b+c)，都将始终生成相同的结果；如果表达式 b+c 在代码中附近出现，则智能 Java 编译器可能能够使用这个公共子表达式。

### 15.7.4 参数列表按从左到右的顺序计算

在方法或构造函数调用或类实例创建表达式中，参数表达式可以出现在圆括号中，用逗号分隔。每个参数表达式似乎在其右侧的任何参数表达式的任何部分之前被完全求值。

如果参数表达式的计算突然完成，则其右侧的任何参数表达式的任何部分似乎都没有被计算过。

**例子 15.7.4-1. 方法调用时的求值顺序**

```
class Test1 {
    public static void main(String[] args) {
        String s = "going, ";
        print3(s, s, s = "gone");
    }
    static void print3(String a, String b, String c) {
        System.out.println(a + b + c);
    }
}
```

此程序生成以下输出：

```
going, going, gone
```

因为将字符串“gone”赋值给 s 是在计算完 print3 的前两个参数之后进行的。

**例子 15.7.4-2. 参数表达式的突然结束**

```
class Test2 {
   static int id;
   public static void main(String[] args) {
      try {
         test(id = 1, oops(), id = 3);
      } catch (Exception e) {
         System.out.println(e + ", id=" + id);
      }
   }
   static int test(int a, int b, int c) {
      return a + b + c;
   }
   static int oops() throws Exception {
      throw new Exception("oops");
   }
}
```

此程序生成以下输出：

```
java.lang.Exception: oops, id=1
```

因为不执行 3 到 id 的赋值。

### 15.7.5 其他表达式的求值顺序

这些一般规则并未完全涵盖某些表达式的求值顺序，因为这些表达式有时可能会引发必须指定的异常条件。以下几种表达式的求值顺序详见：

- 类实例创建表达式(§15.9.4)
- 数组创建表达式 (§15.10.2)
- 数组访问表达式 (§15.10.4)
- 方法调用表达式(§15.12.4)
- 方法引用表达式 (§15.13.3)
- 涉及数组组件的赋值(§15.26)
- lambda 表达式 (§15.27.4)



## 15.8 Primary 表达式

主表达式包括大多数最简单类型的表达式，所有其他类型的表达式都是从这些表达式构造而来的：文字字面量、对象创建、字段访问、方法调用、方法引用和数组访问。带括号的表达式在语法上也被视为主表达式。

*Primary:*
*PrimaryNoNewArray*
*ArrayCreationExpression*

*PrimaryNoNewArray:*
*Literal*
*ClassLiteral*
`this`
*TypeName* `.` `this`
`(` *Expression* `)`
*ClassInstanceCreationExpression*
*FieldAccess*
*ArrayAccess*
*MethodInvocation*
*MethodReference*

Java 编程语言的这一部分语法在两个方面是不寻常的。首先，人们可能希望简单的名称(如局部变量和方

法参数的名称)是主表达式。出于技术原因，当引入后缀表达式时，名称与主表达式稍后被组合在一起(§15.14)。

技术上的原因与只允许使用一个令牌的预判来从左到右解析 Java 程序有关。考虑表达式(z[3])和(z[])。第一个是圆括号括起来的数组访问(§15.10.3)，第二个是类型转换的开始(§15.16)。当向前查找符号为[时，从左到右的解析将 z 减少为非终止符 Name。在强制转换的上下文中，我们不希望必须将名称缩减为 Primary，但是如果 Name 是 Primary 的备选项之一，那么在不提前查找两个标记到[后面的标记的情况下，我们就无法判断是否要进行缩减(也就是说，我们无法确定当前情况是圆括号数组访问还是强制转换)。这里提供的语法通过保持 Name 和 Primary 分离并允许在某些其他语法规则中(那些用于 ClassInstanceCreationExpression、MethodInvocation、ArrayAccess 和 PostfixExpression 的语法规则，但不允许 FieldAccess，因为它直接使用标识符)，从而避免了这个问题。这种策略有效地推迟了 Name 是否应该被视为 Primary 的问题，直到可以检查更多上下文。

第二个不寻常的特性避免了表达式“new int[3][3]”中潜在的语法歧义，这在 Java 中总是意味着一次创建多维数组，但如果没有适当的语法技巧，它也可能被解释为与“(new int[3])[3]”的意思相同。

通过将 Primary 的预期定义分解为 Primary 和 PrimaryNoNewArray，消除了这种不确定性。(这可以与将 Statement 拆分为 Statement 和 StatementNoShortIf(§14.5)来避免“悬空 else”的问题相比。)

### 15.8.1 词法字面量

字面量(§3.10)表示一个固定的、不变的值。

为了方便起见，以下是§3.10 的产品:

*Literal:*
*IntegerLiteral*
*FloatingPointLiteral*
*BooleanLiteral*
*CharacterLiteral*
*StringLiteral*
*TextBlock*
*NullLiteral*



字面量的类型如下所示:

- 以 L 或 l(ell)结尾的整数字面量(§3.10.1)的类型是 long (§4.2.1)。

  其他任何整数字面值的类型都是 int (§4.2.1)。
- 以 F 或 f 结尾的浮点字面量的类型是 float(§4.2.3)。

  其他任何浮点字面量的类型都是 double(§4.2.3)。
- 布尔字面量的类型 (§3.10.3)是 boolean (§4.2.5)。
- 字符字面量的类型(§3.10.4) 是 char (§4.2.1)。
- 字符串字面量的类型(§3.10.5) 或文本块的类型(§3.10.6)是 String (§4.3.3)。
- 空字面量 null 的类型(§3.10.8)是 null 类型 (§4.1);它的值是空引用。

词法字面量的求值总是正常完成。

### 15.8.2 类字面量

类字面量是一个表达式，由类、接口、数组类型或原生类型的名称或伪类型 void 后跟一个'.'和令牌 class 组成。

*ClassLiteral:*
*TypeName* {`[ ]`} `.` `class`
*NumericType* {`[ ]`} `.` `class`
`boolean` {`[ ]`} `.` `class` `void` `.` `class`

TypeName 必须表示可访问的类或接口（§6.6）。如果 TypeName 表示不可访问的类或接口，或表示类型变量，则是编译时错误。

C.class 的类型是 Class<C>, 其中 C 是类、接口或数组类型的名字(§4.3)。

p.class 的类型是 Class<B>, 其中 p 是原生类型的名字 (§4.2), B 是装箱转换后 p 类型表达式的类型(§5.1.7)。

void.class 的类型(§8.4.5) 是 Class<Void>。

根据当前实例的类的定义类加载器（§12.2）的定义，类字面量对命名类、接口、数组类型或原生类型（或 void）的 Class 对象求值。

### 15.8.3 `this`

关键字 this 可在以下上下文中用作表达式:

- 类的实例方法体中 (§8.4.3.2)
- 类的构造函数体中 (§8.8.7)
- 类的实例初始化器中 (§8.6)
- 类的实例变量初始化器中 (§8.3.2)
- 在接口的实例方法体中，也就是默认方法或非静态私有接口方法(§9.4)

当用作表达式时，关键字 this 表示一个值，该值是对调用实例方法的对象（§15.12）的引用，或对正在构造的对象的引用。在 lambda 体(§15.27.2)中由 this 表示的值与在周围上下文中由 this 表示的值是相同的。

关键字 this 也用于显式的构造函数调用语句(§8.8.7.1)，以及表示方法或构造函数的 receiver 参数(§8.4)。

如果 this 表达式发生在静态上下文中，则为编译时错误(§8.1.3)。设 C 为 this 表达式的最内部的封闭类或接口声明。如果 C 是泛型, 类型参数为 $F_1,...,F_n$, this 的类型是 $C<F_1,...,F_n>$。否则，this 的类型是 C。

在运行时，引用实际对象的类可以是 C 或 C 的子类（§8.1.5）。

例子 15.8.3-1. this 表达式

```
class IntVector { int[] v;
```

```
    boolean equals(IntVector other) {
        if (this == other)
            return true;
        if (v.length != other.v.length)
            return false;
        for (int i = 0; i < v.length; i++) {
            if (v[i] != other.v[i]) return false;
        }
        return true;
    }
}
```

这里，类 IntVector 实现了一个方法 equals，它比较两个向量。如果另一个向量与调用 equals 方法的向量对象相同，则检查可以跳过长度和值比较。equals 方法通过将对其他对象的引用与 this 进行比较来实现此检查。

### 15.8.4 限定的 this

任何词法封闭的实例(§8.1.3)都可以通过显式限定关键字 this 来引用。

设 n 是一个整数，使得 TypeName 表示类或接口的第 n 个词法封闭的类或接口声明，其声明立即包含限定的 this 表达式。

限定的 this 表达式 TypeName.this 是 this 的第 n 个词法封闭实例。

如果 TypeName 表示一个泛型类，带有类型参数 $F_1,...,F_n$, 限定 this 表达式的类型是 TypeName<$F_1,...,F_n$>。否则，限定 this 表达式的类型是 TypeName。

如果限定的 this 表达式出现在静态上下文中，则为编译时错误(§8.1.3)。

如果其声明立即包含限定的 this 表达式的类或接口不是 TypeName 或 TypeName 本身的内部类，则为编译时错误。

### 15.8.5 带括号的表达式

带括号的表达式是主表达式，其类型是包含的表达式的类型，运行时的值是包含的表达式的值。如果包含的表达式表示变量，则带括号的表达式也表示该变量。

括号的使用仅影响求值顺序，除了在拐角的情况下，其中(-2147483648)和(-9223372036854775808L)是合法的，而-(2147483648)和-(9223372036854775808L)是非法的。

这是因为十进制字面量 2147483648 和 9223372036854775808L 只允许作为一元减操作符的操作数(§3.10.1)。

特别地，表达式中括号的存在与否并不影响变量是否被明确赋值，为真时明确赋值，为假时明确赋值，明确未赋值，为真时明确未赋值，或为假时明确未赋值(§16(明确赋值))。

如果带括号的表达式出现在目标类型为 T 的特定类型的上下文中(§5(转换和上下文))，它所包含的表达式同样出现在目标类型为 T 的相同类型的上下文中。

如果所包含的表达式是一个多元表达式(§15.2)，则括号表达式也是一个多元表达式。否则，它是一个独立表达式。

如果包含的表达式与 T 兼容，则多元括号表达式与目标类型 T 兼容。

## 15.9 类实例创建表达式

类实例创建表达式用于创建作为类实例的新对象。

*ClassInstanceCreationExpression:*
*UnqualifiedClassInstanceCreationExpression*
*ExpressionName* . *UnqualifiedClassInstanceCreationExpression*
*Primary* . *UnqualifiedClassInstanceCreationExpression*

*UnqualifiedClassInstanceCreationExpression:*
`new` *[TypeArguments]*
*ClassOrInterfaceTypeToInstantiate* ( *[ArgumentList]* ) *[ClassBody]*

*ClassOrInterfaceTypeToInstantiate:*
*{Annotation} Identifier {. {Annotation} Identifier} [TypeArgumentsOrDiamond]*

*TypeArgumentsOrDiamond:*
*TypeArguments*
<>

为了方便起见，以下是§15.12 的产品:

*ArgumentList:*
*Expression {, Expression}*

类实例创建表达式指定要实例化的类，后面可能跟类型参数(§4.5.1)或钻石操作符(<>)，如果被实例化的类是泛型的(§8.1.2)，后跟构造函数的实际值参数列表(可能为空)。

如果类的类型参数列表为空——钻石操作符形式<>——则推断类的类型参数。在钻石操作符的“<”和“>”之间留出空白是合法的，尽管出于风格上的考虑强烈不建议这样做。

如果构造函数是泛型的(§8.8.4)，构造函数的类型参数同样可以被推断或显式传递。如果显式传递，给构造函数的类型参数会紧跟在关键字 new 之后。

如果类实例创建表达式向构造函数提供类型参数，但使用钻石操作符形式作为类的类型参数，则是编译时错误。

引入此规则是因为泛型类类型参数的推断可能会影响泛型构造函数类型参数的约束。

如果 TypeArguments 紧跟在 new 之后，或紧接在(之前，则如果任何类型参数是通配符，则它是编译时错误(§4.5.1)。

类实例创建表达式可以抛出的异常类型在§11.2.1 中规定。

类实例创建表达式有两种形式：

- 非限定类实例创建表达式以关键字 new 开头。

  非限定类实例创建表达式可用于创建类的实例，无论类是顶级类(§7.6)、成员类(§8.5、§9.5)、局部类(§14.3)还是匿名类(§15.9.5)。

- 限定类实例创建表达式以主表达式或 ExpressionName 开头。

  限定的类实例创建表达式允许创建内部成员类及其匿名子类的实例。

非限定类实例创建表达式和限定类实例创建表达式都可以选择以类主体结尾。这样的类实例创建表达式声明一个匿名类(§15.9.5)并创建它的一个实例。

如果类实例创建表达式对类的类型参数使用钻石表达式形式，并且它出现在赋值上下文或调用上下文中，则它是一个多元表达式(§15.2)(§5.2、§5.3)。

我们说，当类的实例由类实例创建表达式创建时，类被实例化。类实例化涉及确定要实例化的类(§15.9.1)、新创建的实例的封闭实例（如有）(§15.9.2)以及要调用以创建新实例的构造函数(§15.9.3)。

### 15.9.1 确定要实例化的类

如果 ClassOrInterfaceTypeToInstantiate 以 TypeArguments 结束（而不是<>），那么 ClassOrInterfaceTypeToInstantiate 必须表示格式良好的参数化类型 (§4.5)，否则会发生编译时错误。

如果 ClassOrInterfaceTypeToInstantiate 以 <> 结束，但是在 ClassOrInterfaceTypeToInstantiate 中的 Identifier 表示的类或接口不是泛型，那么会发生编译时错误。

如果类实例创建表达式在类体中结束，则被实例化的类是匿名类。然后：

- 如果类实例创建表达式是非限定的，那么：

  ClassOrInterfaceTypeToInstantiate 中的 Identifier 必须表示可访问且可自由扩展的类（§8.1.1.2），而不是枚举类，或者表示可访问并可自由扩展（§9.1.4）的接口。否则会发生编译时错误。

  如果 ClassOrInterfaceTypeToInstantiate 中的 Identifier 表示一个类 C，然后声明 C 的匿名直接子类。如果存在 TypeArguments，则 C 具有 TypeArugments 给出的类型参数；如果 < > 存在，则 C 将在§15.9.3 中推断其类型参数；否则，C 没有类型参数。子类的主体是类实例创建表达式中给出的 ClassBody。被实例化的类是匿名子类。

  如果 ClassOrInterfaceTypeToInstantiate 中的 Identifier 表示一个接口 I，然后声明一个实现 I 的 Object 的匿名直接子类。如果存在 TypeArguments，那么 I 有 TypeArugments 给出的类型参数；如果 < > 存在，则 I 将在§15.9.3 中推断其类型参数；否则，I 没有类型参数。子类的主体是类实例创建表达式中给出的 ClassBody。被实例化的类是匿名子类。

- 如果类实例创建表达式是限定的，那么：

  ClassOrInterfaceTypeToInstantiate 中的 Identifier 必须明确表示可访问、可自由扩展的内部类，而不是枚举类，并且是主表达式或 ExpressionName 的编译时类型的成员。否则会发生编译时错误。

  让 ClassOrInterfaceTypeToInstantiate 中的 Identifier 表示类 C。声明了 C 的匿名直接子类。如果存在 TypeArguments，则 C 具有 TypeArugments 给出的类型参数；如果 < > 存在，则 C 将在§15.9.3 中推断其类型参数；否则，C 没有类型参数。子类的主体是类实例创建表达式中给出的 ClassBody。被实例化的类是匿名子类。

如果类实例创建表达式不声明匿名类，那么：

- 如果类实例创建表达式是非限定的，那么：

  ClassOrInterfaceTypeToInstantiate 中的 Identifier 必须表示一个可访问的类，非抽象的，且不是枚举类。否则，会发生编译时错误。

  被实例化的类由 ClassOrInterfaceTypeToInstantiate 中的 Identifier 指定。如果存在 TypeArguments，则该类具有 TypeArugments 给出的类型参数；如果 < > 存在，则该类将在§15.9.3 中推断其类型参数；否则，该类没有类型参数。

- 如果类实例创建表达式是限定的，那么：

  ClassOrInterfaceTypeToInstantiate 必须明确表示可访问的、非抽象的、非枚举类的内部类，以及主表达式或 ExpressionName 的编译时类型成员。

  正在实例化的类由 ClassOrInterfaceTypeToInstantiate 中的 Identifier 指定。如果存在 TypeArguments，则该类具有 TypeArugments 给出的类型参数；如果 < > 存在，则该类将在§15.9.3 中推断其类型参数；否则，该类没有类型参数。

### 15.9.2 确定封闭实例

设 C 是被实例化的类，i 是被创建的实例。如果 C 是一个内部类，那么 i 可能有一个直接封闭的实例(§8.1.3)，确定如下：

- 如果 C 是匿名类，那么：
  - 如果类实例创建表达式发生在静态上下文中，则 i 没有直接封闭实例。
  - 否则, i 的直接封闭实例是 this。
- 如果 C 是一个内部局部类，那么：
  - 如果 C 发生在静态上下文，那么 i 没有直接封闭实例。
  - 否则，如果类实例创建表达式发生在静态上下文，那么会发生编译时错误。
  - 否则，设 O 为 C 的直接封闭类或接口声明，U 为类实例创建表达式的直接封闭类或接口声明。

如果 U 不是 O 本身或 O 的内部类，则会发生编译时错误。

设 n 是一个整数，使得 O 是 U 的第 n 个词法封闭类或接口声明。

i 的直接封闭实例是 this 的第 n 个词法封闭实例。

- 如果 C 是一个内部成员类，那么：
  - 如果类实例创建表达式是非限定的，那么：
    - 如果类实例创建表达式出现在静态上下文中，则会发生编译时错误。
    - 否则，如果 C 不是其声明词法上包含类实例创建表达式的任何类的成员，则会发生编译时错误。
    - 否则，让 O 是 C 是其成员的最内部的封闭类声明，让 U 是类实例创建表达式的直接封闭类或接口声明。

      如果 U 不是 O 本身或 O 的内部类，则会发生编译时错误。

      设 n 为整数，使得 O 是 U 的第 n 个词法封闭类或接口声明。

      i 的直接封闭实例是 this 的第 n 个词法封闭实例。
  - 如果类实例创建表达式是限定的，则 i 的直接封闭实例是主表达式或 ExpressionName 的值的对象。

如果 C 是一个匿名类，其直接超类 S 是一个内部类，那么 i 可能有一个关于 S 的直接封闭实例，确定如下：

- 如果 S 是一个内部局部类，那么：
  - 如果 S 发生在静态上下文中，那么 i 没有关于 S 的直接封闭实例。
  - 否则，如果类实例创建表达式发生在静态上下文中，则会发生编译时错误。
  - 否则，设 O 为 S 的直接封闭类或接口声明，设 U 为类实例创建表达式的直接封闭的类或接口声明。

    如果 U 不是 O 本身或 O 的内部类，则会发生编译时错误。

    设 n 为整数，使得 O 是 U 的第 n 个词法封闭类或接口声明。

    i 相对于 S 的直接封闭实例是 this 的第 n 个词法封闭实例。
- 如果 S 是一个内部成员类，那么：
  - 如果类实例创建表达式是非限定的，则：
    - 如果类实例创建表达式发生在静态上下文中，则会发生编译时错误。
    - 否则，如果 S 不是其声明包含类实例创建表达式的任何类的成员，则会发生编译时错误。

> 否则，设 O 为 S 为成员的最内部封闭类声明，并设 U 为类实例创建表达式的直接封闭类或接口声明。

如果 U 不是 O 本身或 O 的内部类，则会发生编译时错误。

设 n 为整数，使得 O 是 U 的第 n 个词法封闭类或接口声明。

i 相对于 S 的直接封闭实例是 this 的第 n 个词法封闭实例。

> 否则，将发生编译时错误。

- 如果类实例创建表达式是限定的，那么相对于 S 的 i 的直接封闭实例是主表达式或 ExpressionName 的值的对象。

### 15.9.3 选择构造函数及其参数

设 C 是被实例化的类。为了创建 C 的实例 i，在编译时根据以下规则选择 C 的构造函数。

首先，确定构造函数调用的实际参数：

- 如果 C 是具有直接超类 S 的匿名类，则：

  - 如果 S 不是内部类，或者如果 S 是静态上下文中出现的局部类，则构造函数的参数是类实例创建表达式（如果有）的参数列表中的参数，按它们在表达式中出现的顺序排列。

  - 否则，构造函数的第一个参数是 i 相对于 S 的直接封闭实例（§15.9.2），构造函数的后续参数是类实例创建表达式的参数列表中的参数（如果有），按照它们在类实例创建表达中出现的顺序排列。

- 如果 C 是局部类或私有内部成员类，则构造函数的参数是类实例创建表达式（如果有）的参数列表中的参数，按照它们在类实例创建中出现的顺序排列。

- 如果 C 是非私有内部成员类，则构造函数的第一个参数是 i 的直接封闭实例（§8.8.1, §15.9.2），其构造函数的后续参数是类实例创建表达式的参数列表中的参数（如果有），按照它们在类实例创建表达式中出现的顺序排列。

- 否则，构造函数的参数是类实例创建表达式（如果有）的参数列表中的参数，按照它们在表达式中出现的顺序排列。

其次，确定 C 的构造函数以及相应的 throws 子句和返回类型：

- 如果类实例创建表达式不使用<>, 那么：

  - 如果 C 不是匿名类，则：

    设 T 是由 C 表示的类型，后跟表达式中的任何类类型参数。§15.12.2 中规定的过程，经修改以处理构造函数，用于选择 T 的一个构造函数并确定其 throws 子句。

    如果 T 中没有唯一且最具体的构造函数既适用又可访问（§6.6），则会发生编译

时错误（如在方法调用中）。

否则，与所选构造函数对应的返回类型为 T。

- 如果 C 是匿名类，则：

  §15.12.2 中规定的过程（修改为处理构造函数）用于选择 C 的直接超类类型的构造函数之一，并确定其 throws 子句。

  如果在 C 的直接超类类型中没有唯一且最具体的构造函数既适用又可访问，则会发生编译时错误（如在方法调用中）。

  否则，选择 C 的匿名构造函数作为 C 的构造函数（§15.9.5.1）。其主体由 C 的直接超类类型中选择的构造函数的显式构造函数调用（§8.8.7.1）组成。

  所选构造函数的 throws 子句包括在 C 的直接超类类型中选择的构造函数的 throws 子句中的异常。

  与所选构造函数相对应的返回类型是匿名类类型。

- 如果类实例创建表达式使用 <>，那么：

  如果 C 不是匿名类，设 D 与 C 相同。如果 C 是匿名类，设 D 是由类实例创建表达式命名的 C 的超类或超接口。

  如果 D 是类，则设 $C_1,\cdots,C_n$ 是类 D 的构造函数。如果 D 是一个接口，则设 $C_1,\cdots,C_n$ 是包含 Object 类的零参数构造函数的单例列表（n=1）。

  为了重载解析和类型参数推断的目的，定义了方法 $m_1,\cdots,m_n$ 的列表。对于所有的 j ($1 \leqslant j \leqslant n$), $m_j$ 用 $c_j$ 定义如下：

  - 首先定义一个替换 $\theta_j$ 来实例化 $c_j$ 中的类型。

    令 $F_1, ..., F_p$ 为 D 的类型参数，令 $G_1, ..., G_q$ 是 cj 的类型参数(如果有的话)。让 $X_1, ..., X_p$ 和 $Y_1, ..., Y_q$ 是类型变量，具有不同的名称，不在 D 主体作用域内。

    $\theta_j$ 是$[F_1:=X_1, ..., F_p:=X_p, G_1:=Y_1, ..., G_q:=Y_q]$。

  - $m_j$ 的类型参数是 $X_1, ..., X_p, Y_1, ..., Y_q$。每个类型参数的边界(如果有的话)是在 D 或 $c_j$ 中对应的类型参数边界上应用 $\theta_j$。

  - $m_j$ 的返回类型是 $\theta_j$ 应用到 $D<F_1,\cdots,F_p>$。

  - $m_j$ 的参数类型列表(可能为空)对 $c_j$ 的参数类型应用 $\theta_j$。

  - $m_j$ 的抛出类型列表（可能为空）是应用于 $c_j$ 的抛出类型的 $\theta_j$。

  - $m_j$ 的修饰符为 $c_j$ 的修饰符。

  - $m_j$ 的名字是#m，这是一个自动生成的名字，不同于 D 中的所有构造函数和方法名称，由 $m_1, ..., m_n$ 共享。

- $m_j$ 的主体无关紧要。

为了选择构造函数，我们暂时考虑 $m_1$，…， $m_n$ 是 D 中的成员。$m_1$, ..., $m_n$ 中的一个被选择，由类实例创建表达式的参数表达式决定，使用§15.12.2 中规定的过程。

如果没有既适用又可访问的唯一最特定的方法，则会发生编译时错误。

否则，其中 $m_j$ 是选择的方法:

- 如果 C 不是匿名类，则 $c_j$ 被选为 C 的构造函数。

  所选构造函数的 throws 子句与为 $m_j$ 确定的 throws 子句相同。

  所选构造函数对应的返回类型是为 $m_j$ 确定的返回类型(§15.12.2.6)。

- 如果 C 是一个匿名类，那么选择 C 的匿名构造函数作为 C 的构造函数。它的主体由 $c_j$ 的显式构造函数调用(§8.8.7.1)组成。

  所选构造函数的 throws 子句包括为 $m_j$ 确定的 throws 子句中的异常。

  所选构造函数对应的返回类型是匿名类类型。

如果类实例创建表达式是一个多元表达式，那么它与目标类型的兼容性由§18.5.2.1 确定，使用 $m_j$ 作为选择的方法 m。

> 在最终确定类实例创建表达式的目标类型和所选构造函数对应的返回类型之前，可能会进行多次兼容目标类型的测试。例如，一个封闭的方法调用表达式可能需要测试类实例创建表达式与不同方法的形式参数类型的兼容性。

如果 C 是一个匿名类，那么它的直接超类类型或直接超接口类型就是为 $m_j$ 确定的返回类型(§15.12.2.6)。

如果直接超类类型或直接超接口类型，或其中的任何子表达式(“subexpression”包括参数化类型的类型参数、通配符类型参数的边界和数组类型的元素类型，但不包括类型变量的边界)具有以下形式之一，则为编译时错误:

- 未声明为类型参数的类型变量(如捕获转换产生的类型变量)。
- 一个交集类型。
- 类或接口类型，其中的类或接口声明不能从出现类实例创建表达式的类或接口访问。

如果类实例创建表达式的参数与其从调用类型派生的目标类型不兼容，则会出现编译时错误(§15.12.2.6)。

如果编译时声明适用于可变参数调用(§15.12.2.4)，那么构造函数的调用类型的最后一个形式参数类型是 $F_n[]$，如果在调用点无法访问 $F_n$ 的擦除类型，则是编译时错误。

如上所述，类实例创建表达式的类型是与所选构造函数对应的返回类型。

### 15.9.4 类实例创建表达式的运行时计算

在运行时，类实例创建表达式的计算如下所示。

首先，如果类实例创建表达式是限定类实例创建表达式，则计算限定主表达式。如果限定表达式的计算结果为 null，则引发 NullPointerException，并且类实例创建表达式突然完成。如果限定表达式突然完成，则类实例创建表达式也会出于同样的原因突然完成。

接下来，为新的类实例分配空间。如果分配对象的空间不足，则类实例创建表达式的求值会突然结束，并抛出 OutOfMemoryError。

新对象包含指定类及其所有超类中声明的所有字段的新实例。在创建每个新的字段实例时，会将其初始化为其缺省值(§4.12.5)。

接下来，从左到右计算构造函数的实际参数。如果任何参数计算突然完成，则不计算其右侧的任何参数表达式，并且出于相同原因突然完成类实例创建表达式。

接下来，调用指定类的选定构造函数。这会导致为类的每个超类调用至少一个构造函数。此过程可由显式构造函数调用语句(§8.8.7.1)指导，并在§12.5 中详细说明。

类实例创建表达式的值是对指定类新创建对象的引用。每次计算表达式时，都会创建一个新的对象。

**例子 15.9.4-1. 计算顺序和内存不足检测**

如果类实例创建表达式的计算发现没有足够的内存来执行创建操作，那么将抛出 OutOfMemoryError。此检查发生在任何参数表达式计算之前。

例如，测试程序:

```
class List {
   int value;
   List next;
   static List head = new List(0);
   List(int n) { value = n; next = head; head = this; }
}
class Test {
    public static void main(String[] args) {
       int id = 0, oldid = 0;
       try {
          for (;;) {
               ++id;
               new List(oldid = id);
          }
       } catch (Error e) {
           List.head = null;
           System.out.println(e.getClass() + ", " + (oldid==id));
       }
    }
}
```

打印:

```
class java.lang.OutOfMemoryError, false
```

因为在计算参数表达式 oldid = id 之前会检测到内存不足的情况。

与此相比，数组创建表达式的处理是在维数表达式的计算之后检测到内存不足的情况(§15.10.2)。

### 15.9.5 匿名类声明

匿名类由类实例创建表达式或以类体结尾的枚举常量隐式声明（§8.9.1）。

匿名类从不是抽象的 (§8.1.1.1)。

匿名类从不是密封的（§8.1.1.2），因此没有允许的直接子类（§8-1.6）。

由类实例创建表达式声明的匿名类永远不是 final 类（§8.1.1.2）。

由枚举常量声明的匿名类始终是 final。

> 非 final 的匿名类与强制转换相关，特别是强制转换运算符允许的收窄引用转换（§5.5）。另一方面，它与子类化无关，因为不可能声明匿名类的子类（匿名类不能用 extends 子句命名），尽管匿名类是非 final 类。

匿名类始终是内部类(§8.1.3)。

与局部类或接口一样(§14.3)，匿名类不是任何包、类或接口的成员(§7.1、§8.5)。

由类实例创建表达式声明的匿名类的直接超类类型或直接超接口类型由表达式(§15.9.1)给出，在选择构造函数时根据需要推断类型参数(§15.9.3)。如果给定了直接超接口类型，则直接超类类型为 Object。

由枚举常量声明的匿名类的直接超类类型是声明枚举类的类型。

类实例创建表达式或枚举常量的 ClassBody 声明匿名类的字段(§8.3)、方法(§8.4)、成员类(§8.5)、成员接口(§9.1.1.3)、实例初始化器(§8.6)和静态初始化器(§8.7)。匿名类的构造函数始终是隐式的(§15.9.5.1)。

如果带有 ClassBody 的类实例创建表达式使用钻石操作符(<>)作为要实例化的类的类型参数，那么对于 ClassBody 中声明的所有非私有方法，就好像方法声明是用@Override(§9.6.4.4)注解的一样。

> 当使用<>时，推断的类型参数可能不像程序员预期的那样。因此，匿名类的超类型可能不像预期的那样，在匿名类中声明的方法可能不像预期的那样重写超类型方法。将这些方法当作用@Override 注解的方法来处理(如果它们没有显式地用@Override 注解)，可以帮助避免静默的错误程序。

#### 15.9.5.1匿名构造函数

匿名类不能有显式声明的构造函数。相反，为匿名类隐式声明了一个匿名构造函数。具有直接超类 S 的匿名类 C 的匿名构造函数形式如下:

- 如果 S 不是内部类，或者 S 是发生在静态上下文中的局部类，那么匿名构造函数对于类实例创建表达式或声明 C 的枚举常量的每个实际参数都有一个形式参数。

类实例创建表达式或枚举常量的实际参数用于确定 S 的构造函数 x，如§15.9.3 所述。匿名构造函数的每个形式参数的类型都与 x 对应的形式参数相同。

匿名构造函数体由一个显式的构造函数调用(§8.8.7.1)组成，其形式为 super(…)，其中实际的参数是匿名构造函数的形参，按声明的顺序排列。要调用的超类构造函数是 x。

- 否则，匿名构造函数的第一个形参表示 i 的直接封闭实例相对于 S 的值(§15.9.2)。此参数的类型是直接包含 S 声明的类类型。

  对于声明匿名类的类实例创建表达式的每个实际参数，匿名构造函数都有一个额外的形式参数。第 n 个形式参数对应于第 n-1 个实际参数。

  类实例创建表达式的实际参数用于确定 S 的构造函数 x，如§15.9.3 所述。匿名构造函数的每个形式参数的类型都与 x 对应的形式参数相同。

  匿名构造函数体由 o.super(…)形式的显式构造函数调用组成，其中 o 是匿名构造函数的第一个形参，实际参数是构造函数的后续形参，按照声明的顺序。要调用的超类构造函数是 x。

在所有情况下，匿名构造函数的 throws 子句都列出了由匿名构造函数中包含的显式构造函数调用语句(如§15.9.3 所述)抛出的所有检查异常，以及由匿名类的任何实例初始化器或实例变量初始化器抛出的所有检查异常。

注意，匿名构造函数的签名可能引用一个不可访问的类型(例如，如果这种类型出现在超类构造函数 x 的签名中)。这本身不会在编译时或运行时导致任何错误。

## 15.10 数组创建和访问表达式

### 15.10.1 数组创建表达式

数组创建表达式用来创建新数组 (§10 (数组))。

*ArrayCreationExpression:*
  *ArrayCreationExpressionWithoutInitializer*
  *ArrayCreationExpressionWithInitializer*

*ArrayCreationExpressionWithoutInitializer:*
  `new` *PrimitiveType DimExprs [Dims]*
  `new` *ClassOrInterfaceType DimExprs [Dims]*

*ArrayCreationExpressionWithInitializer:*
  `new` *PrimitiveType Dims ArrayInitializer*
  `new` *ClassOrInterfaceType Dims ArrayInitializer*

*DimExprs:*
  *DimExpr {DimExpr}*

*DimExpr:*
*{Annotation}* [ *Expression* ]

为了方便起见，以下是§4.3 的产品:

*Dims:*
*{Annotation}* [ ] *{{Annotation}* [ ]*}*

数组创建表达式创建的对象是一个新数组，其元素是由 PrimitiveType 或 ClassOrInterfaceType 指定的类型。

如果 ClassOrInterfaceType 不表示可具体化类型(§4.7)，则会出现编译时错误。否则，ClassOrInterfaceType 可以命名任何命名的引用类型，甚至是抽象类类型(§8.1.1.1)或接口类型。

上述规则意味着，数组创建表达式中的元素类型不能是参数化类型，除非该参数化类型的所有类型参数都是无界通配符。

DimExpr 中每个维数表达式的类型必须是可转换为整型的类型(§5.1.8)，否则将发生编译时错误。

每个维数表达式都经历一元数字提升(§5.6)。 提升的类型必须为 int，否则将发生编译时错误。

数组创建表达式的类型是一种数组类型，可以通过数组创建表达式的副本表示，其中删除了 new 关键字、每个 DimExpr 表达式和数组初始化器。

例如，创建表达式的类型:

```
new double[3][3][]
```

是:

```
double[][][]
```

### 15.10.2 数组创建表达式的运行时求值

在运行时，数组创建表达式的计算行为如下:

- 如果没有维数表达式，则必须有数组初始化器。新分配的数组将使用§10.6 中描述的数组初始化器提供的值进行初始化。数组初始化器的值将成为数组创建表达式的值。
- 否则，不存在数组初始化器，并且:
  - 首先，从左到右对维数表达式求值。如果任何表达式计算突然完成，则不计算其右侧的表达式。
  - 接下来，检查维数表达式的值。如果任何 DimExpr 表达式的值小于 0，则抛出 NegativeArraySizeException。

- 接下来，为新数组分配空间。如果没有足够的空间分配数组，则数组创建表达式的求值会突然结束，并抛出 OutOfMemoryError。

- 然后，如果出现一个 DimExpr，则创建一个指定长度的一维数组，数组的每个组件初始化为其默认值(§4.12.5)。

- 否则，如果出现 n 个 DimExpr 表达式，那么数组创建将有效地执行一组深度为 n-1 的嵌套循环，以创建数组的隐含数组。

多维数组不需要在每一层都有相同长度的数组。

**例子 15.10.2-1. 数组创建评估**

在具有一个或多个维数表达式的数组创建表达式中，每个维数表达式在其右侧的任何维数表达式的任何部分之前全部计算。因此：

```
class Test1 {
    public static void main(String[] args) {
        int i = 4;
        int[][] ia = new int[i][i=3];
        System.out.println(
           "[" + ia.length + "," + ia[0].length + "]");
    }
}
```

打印：

```
[4,3]
```

因为在第二次维数表达式将 i 设置为 3 之前，第一维被计算为 4。

如果维数表达式的求值突然完成，则其右边的任何维数表达式的任何部分看起来都没有求值。因此：

```
class Test2 {
    public static void main(String[] args) {
        int[][] a = { { 00, 01 }, { 10, 11 } };
        int i = 99;
        try {
           a[val()][i = 1]++;
        } catch (Exception e) {
           System.out.println(e + ", i=" + i);
        }
    }
    static int val() throws Exception {
       throw new Exception("unimplemented");
    }
}
```

打印：

```
java.lang.Exception: unimplemented, i=99
```

因为将 i 赋值为 1 的嵌入赋值永远不会执行。

**例子 15.10.2-2. 多维数组创建**

声明:

```
float[][] matrix = new float[3][3];
```

在行为上等价于:

```
float[][] matrix = new float[3][];
for (int d = 0; d < matrix.length; d++)
  matrix[d] = new float[3];
```

以及:

```
Age[][][][][] Aquarius = new Age[6][10][8][12][];
```

等价与:

```
Age[][][][][] Aquarius = new Age[6][][][][];
for (int d1 = 0; d1 < Aquarius.length; d1++) {
    Aquarius[d1] = new Age[10][][][];
    for (int d2 = 0; d2 < Aquarius[d1].length; d2++){
        Aquarius[d1][d2] = new Age[8][][];
        for (int d3 = 0; d3 < Aquarius[d1][d2].length; d3++){
          Aquarius[d1][d2][d3] = new Age[12][];
        }
    }
}
```

将 d、d1、d2 和 d3 替换为未在本地声明的名称。因此，一个 new 表达式实际上创建了一个长度为 6 的数组、6 个长度为 10 的数组、6x10 = 60 个长度为 8 的数组、以及 6x10x8 = 480 个长度为 12 的数组。这个示例将第五维保留为包含实际数组元素(对 Age 对象的引用)的数组，只初始化为空引用。这些数组可以稍后由其他代码填充，例如:

```
Age[] Hair = { new Age("quartz"), new Age("topaz") }; Aquarius[1][9][6][9]
= Hair;
```

三角形矩阵可以通过以下方法创建:

```
float[][] triang = new float[100][];
for (int i = 0; i < triang.length; i++)
    triang[i] = new float[i+1];
```

如果数组创建表达式的计算结果发现没有足够的内存来执行创建操作，则会抛出 OutOfMemoryError。如果数组创建表达式没有数组初始化器，则只有在正常完成所有维数表达式的求值之后才会进行此检查。如果数组创建表达式确实有数组初始化器，那么当在求值变量初始化器表达式时分配引用类型的对象，或者为数组分配空间以保存(可能嵌套)数组初始化器的值时，就会发生 OutOfMemoryError。

例子 15.10.2-3. OutOfMemoryError 和维数表达式求值

```
class Test3 {
public static void main(String[] args) {
        int len = 0, oldlen = 0;
        Object[] a = new Object[0];
        try {
          for (;;) {
```

```
                    ++len;
                    Object[] temp = new Object[oldlen = len];
                    temp[0] = a;
                    a = temp;
                }
            } catch (Error e) {
                System.out.println(e + ", " + (oldlen==len));
            }
        }
    }
```

该程序产生以下输出：

```
java.lang.OutOfMemoryError, true
```

因为在计算维数表达式 oldlen=len 之后检测到内存不足的情况。

将其与类实例创建表达式（§15.9）进行比较，后者在计算参数表达式（§5.9.4）之前检测内存不足的情况。

### 15.10.3 数组访问表达式

数组访问表达式引用的变量是数组的组件。

*ArrayAccess:*
*ExpressionName* [ *Expression* ]
*PrimaryNoNewArray* [ *Expression* ]
*ArrayCreationExpressionWithInitializer* [ *Expression* ]

数组访问表达式包含两个子表达式，数组引用表达式(在左括号之前)和索引表达式(在括号内)。

> 注意，数组引用表达式可以是一个名称或任何不是数组创建表达式的主表达式，除非数组创建表达式有数组初始化器(§15.10.1)。

数组引用表达式的类型必须是数组类型(称为 T[]，数组的组件为 T 类型)，否则将发生编译时错误。

索引表达式经历一元数字提升(§5.6)。提升的类型必须为 int，否则将发生编译时错误。

数组访问表达式的类型是对 T 应用捕获转换(§5.1.10)的结果。

数组访问表达式的结果是一个 T 类型的变量，即由索引表达式的值选择的数组内的变量。

这个结果变量是数组的一个组件，即使数组引用表达式表示一个 final 变量，它也不会被认为是 final 变量。

### 15.10.4 数组访问表达式的运行时计算

在运行时，数组访问表达式的计算行为如下:

- 首先，计算数组引用表达式。如果这个求值突然结束，那么数组访问也会因为同样的原因突然结束，索引表达式也不会被求值。

- 否则，索引表达式将被求值。如果这个求值突然结束，那么数组访问也会因为同样的原因突然结束。
- 否则，如果数组引用表达式的值为空，则抛出 NullPointerException。
- 否则，数组引用表达式的值确实指向数组。如果索引表达式的值小于零，或大于或等于数组的长度，则会抛出 ArrayIndexOutOfBoundsException。
- 否则，数组访问的结果是数组中由索引表达式的值选择的 T 类型变量。

例子 15.10.4-1. 首先计算数组引用

在数组访问中，在对括号内的表达式的任何部分求值之前，括号左侧的表达式似乎已经完全求值。例如，在表达式 a[(a=b)[3]]中，表达式 a 在表达式(a=b)[3]之前被完全求值;这意味着在计算表达式(a=b)[3]时，将获取并记住 a 的原始值。然后，a 的原始值所引用的数组下标为另一个数组(可能是同一个数组)的第 3 元素的值，该数组曾被 b 引用，现在也被 a 引用。

因此，程序:

```
class Test1 {
    public static void main(String[] args) {
        int[] a = { 11, 12, 13, 14 };
        int[] b = { 0, 1, 2, 3 };
        System.out.println(a[(a=b)[3]]);
    }
}
```

打印:

```
14
```

因为怪物表达式的值等于 a[b[3]]或 a[3]或 14。

例子 15.10.4-2. 突然完成数组引用求值

如果括号左边表达式的求值突然完成，则括号内的表达式看起来没有任何部分被求值。因此,程序:

```
class Test2 {
    public static void main(String[] args) {
        int index = 1;
        try {
            skedaddle()[index=2]++;
        } catch (Exception e) {
            System.out.println(e + ", index=" + index);
        }
    }
    static int[] skedaddle() throws Exception {
        throw new Exception("Ciao");
    }
}
```

打印:

```
java.lang.Exception: Ciao, index=1
```

因为从未发生将 2 嵌入赋值到索引。

例子 15.10.4-3. null 数组引用

如果数组引用表达式生成 null 而不是对数组的引用，则只有在数组访问表达式的所有部分都已求值且这些求值正常完成后，才会在运行时引发 NullPointerException。因此，程序:

```
class Test3 {
    public static void main(String[] args) {
        int index = 1;
        try {
            nada()[index=2]++;
        } catch (Exception e) {
            System.out.println(e + ", index=" + index);

        }
    } static int[] nada() { return null; }
}
```

打印:

```
java.lang.NullPointerException, index=2
```

因为在检查空数组引用表达式之前，将 2 嵌入赋值到索引。作为一个相关的例子，程序:

```
class Test4 {
    public static void main(String[] args) {
        int[] a = null;
        try {
            int i = a[vamoose()];
            System.out.println(i);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    static int vamoose() throws Exception {
        throw new Exception("Twenty-three skidoo!");
    }
}
```

同样打印:

```
java.lang.Exception: Twenty-three skidoo!
```

NullPointerException 永远不会发生，因为在发生任何进一步的数组访问之前，必须完全计算索引表达式，这包括检查数组引用表达式的值是否为空。

## 15.11 字段访问表达式

字段访问表达式可以访问对象或数组的字段，其引用是表达式的值或特殊关键字 super 的值。

*FieldAccess:*

*Primary* `.` *Identifier*
`super` `.` *Identifier*
*TypeName* `.` `super` `.` *Identifier*

字段访问表达式的含义与限定名相同(§6.5.6.2)，但受限于表达式不能表示包、类类型或接口类型的事实。

也可以使用一个简单的名称(§6.5.6.1)来引用当前实例或当前类的字段。

### 15.11.1 使用 Primary 的字段访问

Primary 的类型必须是引用类型 T，否则将发生编译时错误。

字段访问表达式的含义确定如下:

- 如果标识符在 T 类型中命名了几个可访问的(§6.6)成员字段，那么字段访问是不明确的，并且会发生编译时错误。
- 如果标识符没有命名类型 T 中可访问的成员字段，那么该字段访问是未定义的，并发生编译时错误。
- 否则，标识符在类型 T 中命名单个可访问的成员字段，并且字段访问表达式的类型是捕获转换后成员字段的类型(§5.1.10)。

在运行时，字段访问表达式的计算结果如下:(假设程序在明确赋值分析方面是正确的，即在访问之前，每个空白的 final 变量都被明确赋值)

- 如果字段是静态的:
  - 对 Primary 表达式被求值，并丢弃结果。如果 Primary 表达式的求值突然完成，字段访问表达式也会出于同样的原因突然完成。
  - 如果字段是非空的 final 字段，则结果是 Primary 表达式类型的类或接口中指定的类变量的值。
  - 如果字段不是 final，或者是一个空白的 final，并且字段访问发生在类变量初始化器(§8.3.2)或静态初始化器(§8.7)中，则结果是一个变量，即 Primary 表达式类型的类中指定的类变量。
- 如果字段不是静态的:
  - 对 Primary 表达式求值。如果 Primary 表达式的求值突然完成，字段访问表达式也会出于同样的原因突然完成。
  - 如果 Primary 的值为空，则抛出 NullPointerException。
  - 如果字段是非空 final 字段，则结果是在 Primary 的值所引用的对象中找到的类型 T 中命名成员字段的值。

- 如果该字段不是 final 字段，或者是一个空白的 final 字段，并且字段访问发生在实例变量初始化器(§8.3.2)、实例初始化器(§8.6)或构造函数(§8.8)中，那么结果是一个变量，即在 Primary 的值所引用的对象中找到的类型 T 中命名的成员字段。

注意，在决定使用哪个字段时，只使用 Primary 表达式的类型，而不是运行时引用的实际对象的类。

例子 15.11.1-1. 字段访问的静态绑定

```
class S            { int x = 0; }
class T extends S { int x = 1; }
class Test1 {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.x=" + t.x + when("t", t));
        S s = new S();
        System.out.println("s.x=" + s.x + when("s", s));
        s = t;
        System.out.println("s.x=" + s.x + when("s", s));
    }
    static String when(String name, Object t) {
       return " when " + name + " holds a "
                        + t.getClass() + " at run time."; }
}
```

这个程序产生输出:

```
t.x=1   when tholds  aclass  Tat  runtime.
s.x=0   when sholds  aclass  Sat  runtime.
s.x=0   when sholds  aclass  Tat  runtime.
```

最后一行显示，实际上，被访问的字段不依赖于被引用对象的运行时类;即使 s 保存了对 T 类对象的引用，表达式 s.x 引用了 S 类的 x 字段，因为表达式 s 的类型是 S。T 类的对象包含两个名为 x 的字段，一个用于 T 类，另一个用于其超类 S。

这种字段访问的动态查找的缺乏使得程序可以通过简单的实现高效地运行。后期绑定和重写的功能是可用的，但只有在使用实例方法时才可用。考虑使用实例方法访问字段的相同示例:

```
class S            { int x = 0; int z() { return x; } }
class T extends S { int x = 1; int z() { return x; } }
class Test2 {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.z()=" + t.z() + when("t", t));
        S s = new S();
        System.out.println("s.z()=" + s.z() + when("s", s));
        s = t;
        System.out.println("s.z()=" + s.z() + when("s", s));
    }
    static String when(String name, Object t) {
            return " when " + name + " holds a "
                        + t.getClass() + " at run time.";
    }
}
```

现在输出是:

```
t.z()=1  when  tholdsa class  T at  runtime.
s.z()=0  when  sholdsa class  S at  runtime.
s.z()=1  when  sholdsa class  T at  runtime.
```

最后一行显示，被访问的方法确实依赖于被引用对象的运行时类;当 s 保存了对 T 类对象的引用时，表达式 s.z()引用了 T 类的 z 方法，尽管表达式 s 的类型是 S。类 T 的方法 z 重写类 S 的方法 z。

例子 15.11.1-2. Receiver 变量与静态字段访问无关

下面的程序演示了可以使用空引用来访问类(静态)变量而不会引起异常:

```
class Test3 {
    static String mountain = "Chocorua";
    static Test3 favorite(){
        System.out.print("Mount ");
        return null;
    }
    public static void main(String[] args) {
        System.out.println(favorite().mountain);
    }
}
```

它编译、执行和打印:

```
Mount Chocorua
```

即使 favorite()的结果为 null，也不会抛出 NullPointerException。“Mount”被打印出来表明 Primary 表达式在运行时确实被完全求值了，尽管事实上，只有它的类型，而不是它的值，被用来决定访问哪个字段(因为字段 mountain 是静态的)。



### 15.11.2 使用 super 访问超类成员

形式 super.Identifier 指向当前对象名为 Identifier 的字段名，但是当前对象被视为当前类的超类的实例。

形式 T.super.Identifier 指向与 T 对应的词法封闭实例的名为 Identifier 的字段,但该实例被视为 T 的超类的实例。

使用关键字 super 的形式可以在类声明的位置中使用，并且允许关键字 this 作为一个表达式(§15.8.3)。

如果在静态上下文中出现使用关键字 super 的字段访问表达式，则会出现编译时错误(§8.1.3)。

对于形式为 super.Identifier 的字段访问表达式:

- 如果字段访问表达式的直接封闭类或接口声明是类 Object 或接口，则为编译时错误。

对于形式为 T.super.Identifier 的字段访问表达式:

- 如果 T 是类 Object 或接口，则为编译时错误。

- 让 U 作为字段访问表达式的直接封闭类或接口声明。如果 U 不是 T 本身或 T 的内部类，则为编译时错误。

假设一个字段访问表达式 super.f 出现在类 C 中，C 的直接超类是类 S。如果 S 中的 f 可以从 C 类中访问(§6.6)，则将 super.f 视为类 S 的主体中的表达式 this.f。否则，将发生编译时错误。

因此，super.f 可以访问在类 S 中可访问的字段 f，即使该字段被类 C 中的字段 f 的声明所隐藏。

假设字段访问表达式 T.super.f 出现在类 C 中，并且由 T 表示的类的直接超类是其完全限定名为 S 的类。如果 S 中的 f 可以从 C 访问，则 T.super.f 被视为是类 S 主体中的表达式 this.f。否则，将发生编译时错误。

因此，T.super.f 可以访问在类 S 中可访问的字段 f，即使该字段被类 T 中的字段 f 的声明所隐藏。

**例子 15.11.2-1. super 表达式**

```
interface I           {  int x=  0;}
class T1 implements   I { int x=  1;}
class T2 extends T1   {  int x=  2;}
class T3 extends T2 {
   int x = 3;
   void test() {
         System.out.println("x=\t\t"          +  x);
         System.out.println("super.x=\t\t" + super.x);
         System.out.println("((T2)this).x=\t" + ((T2)this).x);
         System.out.println("((T1)this).x=\t" + ((T1)this).x);
         System.out.println("((I)this).x=\t" + ((I)this).x);
   }
}
class Test {
   public static void main(String[] args) {
      new T3().test();
    }
}
```

此程序生成以下输出：

```
x=              3
super.x=        2
((T2)this).x=   2
((T1)this).x=   1
((I)this).x= 0
```

在类 T3 中，当 x 有包访问权限，表达式 super.x 与表达式 ((T2)this).x 有相同的效果。请注意，由于访问超类的受保护成员有困难，所以没有在强制转换方面指定 super.x。

## 15.12 方法调用表达式

方法调用表达式用于调用类或实例方法。

*MethodInvocation:*
  *MethodName* （ *[ArgumentList]* ）

*TypeName* . *[TypeArguments] Identifier* ( *[ArgumentList]* )
*ExpressionName* . *[TypeArguments] Identifier* ( *[ArgumentList]* ) *Primary* . *[TypeArguments] Identifier* ( *[ArgumentList]* ) `super` . *[TypeArguments] Identifier* ( *[ArgumentList]* )
*TypeName* . `super` . *[TypeArguments] Identifier* ( *[ArgumentList]* )

*ArgumentList:*
*Expression {, Expression}*

由于方法重载的可能性，在编译时解析方法名比解析字段名要复杂得多。在运行时调用方法也比访问字段复杂，因为可能会重写实例方法。

确定方法调用表达式将调用的方法涉及几个步骤。以下三节描述了方法调用的编译时处理。方法调用表达式类型的确定在§15.12.3 中规定。

方法调用表达式可以抛出的异常类型在§11.2.1 中规定。

如果在 MethodInvocation 中发生在(之前的最右边的"."的左边的名字不能被归类为 TypeName 或 ExpressionName(§6.5.2)，那将是一个编译时错误。

如果 TypeArguments 位于 Identifier 的左侧，则如果任何类型参数为通配符，则为编译时错误（§4.5.1）。

如果以下所有条件均为真，则方法调用表达式为多元表达式：

- 调用出现在赋值上下文或调用上下文中 (§5.2, §5.3)。
- 如果调用是限定的（即，除第一种调用外的任何形式的 MethodInvocation），则调用将省略 Identifier 左侧的 TypeArguments。
- 由以下小节确定的要调用的方法是泛型的（§8.4.4），并具有至少一个方法类型参数的返回类型。

否则，方法调用表达式是独立表达式。

### 15.12.1 编译时步骤 1:确定要搜索的类型

在编译时处理方法调用的第一步是确定要调用的方法的名称，以及搜索该名称的方法定义的类型。

方法的名称由 MethodName 或 Identifier 指定，该标识符直接位于 MethodInvocation 的左括号之前。

对于要搜索的类型，有六种情况需要考虑，具体取决于 MethodInvocation 左括号之前的形式：

- 如果形式为 MethodName，即仅为 Identifier，则：

  如果 Identifier 出现在具有该名称的方法声明作用域内（§6.3，§6.4.1），则：

- 如果有一个封闭的类或接口声明，该方法是该类或接口的成员，那么让 E 成为最内部的这种类或接口。要搜索的类型是 E.this 的类型（§15.8.4）。

  这种搜索策略被称为“梳子规则”。它在查找封闭类及其超类层次结构中的方法之前，有效地查找嵌套类的超类层次结构中的方法。详见§6.5.7.1。

- 否则，由于一个或多个单一静态导入或静态按需导入声明，方法声明可能在作用域内。没有要搜索的类型，因为要调用的方法是以后确定的(§15.12.2.1)。

- 如果形式为 TypeName . [TypeArguments] Identifier, 那么要搜索的类型就是 TypeName 表示的(可能是原始的)类型。

- 如果形式为 ExpressionName . [TypeArguments] Identifier，那么要搜索的类型是 ExpressionName 表示的变量的声明类型 T，如果 T 是类或接口类型，或者如果 T 是类型变量，则为 T 的上界。

- 如果形式为 Primary . [TypeArguments] Identifier, 那么设 T 为 Primary 表达式的类型。如果 T 是类或接口类型，则要搜索的类型是 T；如果 T 是类型变量，则要搜索的类型是 T 的上限。

  如果 T 不是引用类型，则是编译时错误。

- 如果形式为 super . [TypeArguments] Identifier, 那么要搜索的类型是其声明包含方法调用的类的直接超类类型。

  设 E 为直接包含方法调用的类或接口声明。如果 E 是类 Object 或接口，则是编译时错误。

- 如果形式为 TypeName . super . [TypeArguments] Identifier, 那么:

  - 如果 TypeName 既不表示类也不表示接口，则是编译时错误。

  - 如果 TypeName 表示类 C，则要搜索的类型是 C 的直接超类类型。

    如果 C 不是方法调用的词法封闭类声明，或者如果 C 是 Object 类，则是编译时错误。

    设 E 为直接包含方法调用的类或接口声明。如果 E 是类 Object，则是编译时错误。

  - 否则, TypeName 表示接口 I。

    设 E 为直接包含方法调用的类或接口声明。如果 I 不是 E 的直接超接口，或者如果存在 E，J 的其他直接超类或直接超接口（例如 J 是 I 的子类或子接口），则这是编译时错误。

    要搜索的类型是 I 类型，即 E 的直接超接口类型。

    TypeName.super 语法是重载的：传统上，TypeName 指的是一个词法封闭的类声明，而目标是这个类的超类，就好像调用是词法封闭类声明中的一个非限定 super。

```
class Superclass {
  void foo() { System.out.println("Hi"); }
}

class Subclass1 extends Superclass {
  void foo() { throw new UnsupportedOperationException(); }

  Runnable tweak = new Runnable() {
    void run() {
          Subclass1.super.foo(); // Gets the 'println' behavior
    }
  };
}
```

为了支持在超接口中调用默认方法，TypeName 还可以引用当前类或接口的直接超接口，目标是该超接口。

```
interface Superinterface {
  default void foo() { System.out.println("Hi"); }
}

class Subclass2 implements Superinterface {
    void foo() { throw new UnsupportedOperationException(); }

    void tweak() {
        Superinterface.super.foo(); // Gets the 'println' behavior
    }
}
```

没有语法支持这些形式的组合，也就是说，调用词法封闭类声明的超接口方法，就好像调用在词法封闭类声明中的形式是 InterfaceName.super。

```
class Subclass3 implements Superinterface {
void foo() { throw new UnsupportedOperationException(); }

    Runnable tweak = new Runnable() {
      void run() {
           Subclass3.Superinterface.super.foo(); // Illegal
      }
    };
}
```

一种解决方法是在词法封闭类声明中引入一个私有方法，该方法执行接口 super 调用。

### 15.12.2 编译时步骤 2:确定方法签名

第二步在上一步中确定的类型中搜索成员方法。此步骤使用方法的名称和参数表达式来定位可访问和适用的方法，即可以对给定参数正确调用的声明。

这样的方法可能不止一种，在这种情况下，选择最具体的一种。方法的最特定的描述符(签名加上返回类型)是在运行时用于执行方法分派的描述符。

某些包含隐式类型化 lambda 表达式(§15.27.1)或不精确的方法引用(§15.13.1)的参数表达式会被适用性测试忽略，因为它们的含义在调用的目标类型被选中之前无法确定。另一方面，只有参数表达式(而不是调用的目标类型)会影响适用性测试，即使方法调用表达式是多元表达式。

确定适用性的过程始于确定潜在的可适用的方法(§15.12.2.1)。然后，为了确保与 Java SE 5.0 之前的 Java 编程语言兼容，该过程分为三个阶段:

1. 第一个阶段执行重载解析，不允许装箱或拆箱转换，也不允许使用可变参数方法调用。如果在这一阶段没有找到适用的方法，那么处理将继续到第二阶段。

   这保证了 Java SE 5.0 之前在 Java 编程语言中有效的任何调用不会因为引入了可变参数方法、隐式装箱和/或拆箱而被认为是不明确的。然而，可变参数方法的声明(§8.4.1)可以改变为给定方法调用表达式所选择的方法，因为可变参数方法在第一阶段被视为固定参数方法。例如，在一个已经声明了 m(Object)的类中声明 m(Object…)会导致 m(Object)不再被某些调用表达式选择(比如 m(null))，因为 m(Object[])更具体。

2. 第二阶段执行重载解析，同时允许装箱和拆箱，但仍然不允许使用可变参数方法调用。如果在这一阶段没有找到适用的方法，那么处理将继续到第三阶段。

   这就确保了，如果方法是通过固定参数方法调用的，就不会通过可变参数方法调用选择方法。

3. 第三个阶段允许将重载与可变参数方法、装箱和拆箱相结合。

如果一个方法是通过严格调用(第一阶段，§15.12.2.2)、松散调用(第二阶段，§15.12.2.3)或可变参数调用(第三阶段，§15.12.2.4)中的一个来适用，那么它就是适用的。对于泛型方法(§8.4.4)，判断一个方法是否适用需要对类型参数进行分析。类型参数可以显式或隐式传递;如果它们是隐式传递的，那么必须从参数表达式中推断类型参数的边界(§18(类型推断))。

如果在适用性测试的三个阶段中的一个阶段已经确定了几种适用的方法，那么将选择最具体的一个，如§15.12.2.5 所规定的。

为了检查适用性，调用参数的类型通常不能作为分析的输入。这是因为:

- 方法调用的参数可以是多元表达式。
- 在没有目标类型的情况下，不能类型化多元表达式。
- 在知道参数的目标类型之前，必须完成重载解析。

相反，适用性检查的输入是参数本身的列表。可以检查参数与潜在目标类型的兼容性，即使参数的最终类型未知。

注意，重载解析与目标类型无关。这有两个原因:

- 首先，它使用户模型更容易访问，更不容易出错。方法名的意义(即与名称对应的声明)对于程序的意义来说太基础了，不能依赖于微妙的上下文提示。(相比之下，其他多元表达式可能会根据目标类型有不同的行为;但是，行为的变化总是有限的，而且本质上是相等的，而对于任意一组共享名称和参数个数的方法的行为，不能做出这样的保证。)
- 第二，它允许其他属性-例如方法是否为多元表达式（§15.12）或如何对条件表达式进行分类（§15.25）-取决于方法名称的含义，甚至在已知目标类型之前。

**例子 15.12.2-1. 方法的适用性**

```
class Doubler {
                    static int two()      { return two(1); }
            private static int two(int i) { return 2*i;    }
}
class Test extends Doubler {
```

```
    static long two(long j) { return j+j; }

    public static void main(String[] args) {
        System.out.println(two(3));
        System.out.println(Doubler.two(3)); // compile-time error
    }
}
```

对于类 Doubler 中的方法调用 two(1)，有两个名为 two 的可访问方法，但只有第二个方法适用，因此它是在运行时调用的方法。

对于类 Test 中的方法调用 two(3)，有两个适用的方法，但只有类 Test 中的一个是可访问的，因此这是在运行时调用的方法(参数 3 被转换为类型 long)。

对于方法调用 Doubler.two(3)，将在类 Doubler 而不是类 Test 中搜索名为 two 的方法；唯一适用的方法是不可访问的，因此此方法调用会导致编译时错误。

另一个例子是：

```
class ColoredPoint {
    int x, y;
    byte color;

    void setColor(byte color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        byte color = 37;
        cp.setColor(color);
        cp.setColor(37); // compile-time error
    }
}
```

在这里，第二次调用 setColor 时会出现编译时错误，因为在编译时找不到适用的方法。文本字面量 37 的类型为 int，不能通过调用转换将 int 转换为 byte。赋值转换用于变量 color 的初始化，它执行常量从类型 int 到 byte 的隐式转换，这是允许的，因为值 37 足够小，可以用类型 byte 表示；但调用转换不允许这样的转换。

然而，如果方法 setColor 被声明为接受 int 而不是 byte，那么两个方法调用都将是正确的；第一个调用将被允许，因为调用转换确实允许从 byte 到 int 的拓宽转换。但是，在 setColor 的正文中需要进行收窄类型转换：

```
void setColor(int color) { this.color = (byte)color; }
```

下面是一个重载二义性的示例。考虑程序：

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    static void test(ColoredPoint p, Point q) {
        System.out.println("(ColoredPoint, Point)");
    }
    static void test(Point p, ColoredPoint q) {
        System.out.println("(Point, ColoredPoint)");
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
```

```
        test(cp, cp); // compile-time error
    }
}
```

此示例在编译时生成错误。问题是，有两个 test 声明是可应用和可访问的，并且没有一个比另一个更具体。因此，方法调用是模棱两可的。

如果增加 test 的第三个定义：

```
static void test(ColoredPoint p, ColoredPoint q) {
    System.out.println("(ColoredPoint, ColoredPoint)");
}
```

那么它将比其他两个更具体，并且方法调用将不再模棱两可。

例子 15.12.2-2. 方法选择过程中未考虑返回类型

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    static int test(ColoredPoint p) {
       return p.color;
    }
    static String test(Point p) {
       return "Point";
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        String s = test(cp); // compile-time error
    }
}
```

在这里，方法 test 最具体的声明是接受 ColoredPoint 类型的参数的声明。由于该方法的结果类型为 int，因此会发生编译时错误，因为无法通过赋值转换将 int 转换为 String。此示例显示，方法的结果类型不参与解析重载方法，因此不选择返回 String 的第二个 test 方法，即使它的结果类型允许示例程序正确编译也是如此。

**例子 15.12.2-3. 选择最具体的方法**

最具体的方法是在编译时选择的；它的描述符确定在运行时实际执行的方法。如果将新方法添加到类中，则使用类的旧定义编译的源代码可能不会使用新方法，即使重新编译会导致选择此方法。

因此，例如，考虑两个编译单元，一个用于 Point 类：

```
package points;
public class Point {
    public int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return toString(""); }
    public String toString(String s) {
        return "(" + x + "," + y + s + ")";
    }
}
```

一个用于 ColoredPoint 类：

```
package points;
public class ColoredPoint extends Point {
    public static final int RED = 0, GREEN = 1, BLUE = 2;
```

```
        public static String[] COLORS = { "red", "green", "blue" };
        public byte color;
        public ColoredPoint(int x, int y, int color) {
            super(x, y);
            this.color = (byte)color;
        }

        /** Copy all relevant fields of the argument into this ColoredPoint object. */
        public void adopt(Point p) { x = p.x; y = p.y; }

        public String toString() {
            String s = "," + COLORS[color];
            return super.toString(s);
        }
    }
```

现在考虑使用 ColoredPoint 的第三个编译单元:

```
    import points.*;
    class Test {
        public static void main(String[] args) {
           ColoredPoint cp =
              new ColoredPoint(6, 6, ColoredPoint.RED);
           ColoredPoint cp2 =
              new ColoredPoint(3, 3, ColoredPoint.GREEN);
           cp.adopt(cp2);
           System.out.println("cp: " + cp);
        }
    }
```

输出为:

```
    cp: (3,3,red)
```

编写类 Test 代码的程序员希望看到 green 这个词，因为实际的参数 ColoredPoint 有一个 color 字段，而 color 似乎是一个“相关字段”。(当然，包 points 的文档应该更加精确！)

顺便说一句，请注意，用于 adopt 的方法调用的最具体的方法(实际上是唯一适用的方法)具有一个签名，该签名指示一个参数的方法，并且该参数的类型为 Point。该签名成为 Java 编译器生成的类 Test 的二进制表示的一部分，并在运行时由方法调用使用。

假设程序员报告了这个软件错误，并且 points 包的维护者经过深思熟虑后决定通过向 ColoredPoint 类添加一个方法来纠正它:

```
        public void adopt(ColoredPoint p) {
           adopt((Point)p);
           color = p.color;
        }
```

如果程序员随后使用 ColoredPoint 的新二进制文件运行用于 Test 的旧二进制文件，则输出仍为:

```
        cp: (3,3,red)
```

因为用于 Test 的旧二进制文件仍然具有与方法调用 cp.adopt(cp2)相关联的描述符“一个参数，其类型是 Point;void”。 如果重新编译用于 Test 的源代码，Java 编译器将发现现在有两个适用的 adopt 方法，并且更具体的一个方法的签名是“一个参数，其类型为 ColoredPoint;void”；然后运行该程序将产生所需的输出:

```
cp: (3,3,green)
```

考虑到这些问题，points 包的维护者可以修复 ColoredPoint 类，使其既能处理新编译的代码，也能处理旧的代码，方法是为旧的 adopt 方法添加防御代码，以使旧的代码仍然对 ColoredPoint 参数调用它:

```
public void adopt(Point p) {
   if (p instanceof ColoredPoint)
      color = ((ColoredPoint)p).color;
   x = p.x; y = p.y;
}
```

理想情况下，只要源代码所依赖的代码发生更改，就应该重新编译源代码。然而，在由不同组织维护不同类的环境中，这并不总是可行的。注意类演化问题的防御性编程可以使升级后的代码更加健壮。关于二进制兼容性和类型演化的详细讨论见§13(二进制兼容性)。

#### 15.12.2.1 确定可能适用的方法

通过编译时步骤 1（§15.12.1）确定的类型，搜索可能适用于该方法调用的所有成员方法；从超类和超接口继承的成员包含在此搜索中。

此外，如果方法调用表达式的形式是 MethodName-即，单个 Identifier -然后搜索可能适用的方法也会检查所有成员方法，这些成员方法是通过方法调用发生时编译单元的单个静态导入声明和静态按需导入声明导入的（§7.5.3，§7.4.4），并且在方法调用出现时未被遮蔽。

当且仅当以下所有条件均为真时，成员方法可能适用于方法调用:

- 成员的名称与方法调用中方法的名称相同。
- 方法调用所在的类或接口可以访问该成员（§6.6）。

  成员方法在方法调用时是否可访问取决于成员声明中的访问修饰符（public、protected、没有修饰符（包访问）或 private），以及编译时步骤 1 确定的类或接口对成员的继承，以及方法调用出现的位置。

- 如果成员是一个具有 n 个参数的固定参数方法，并且对于所有 i（1≤ i≤ n），方法调用的第 i 个参数可能与方法的第 i 参数的类型兼容，如下所述。

- 如果成员是带有 n 个参数的可变参数方法，那么对于所有 i (1 ≤ i ≤ n-1), 方法调用的第 i 个参数可能与方法的第 i 个参数的类型兼容;并且，该方法的第 n 个参数的类型为 T[]，以下情况之一为真:

  - 方法调用的参数个数为 n -1。
  - 方法调用的参数个数为 n，并且方法调用的第 n 个参数可能与 T 或 T[]兼容。
  - 方法调用的参数个数为 m，其中 m>n，并且对于所有 i (n ≤ i ≤ m), 方法调用的第 i 个参数可能与 T 兼容。

- 如果方法调用包含显式类型参数，并且成员是泛型方法，那么类型参数的数量等于方法的类型参数的数量。

  此子句意味着非泛型方法可能适用于提供显式类型参数的调用。事实上，它可能会被证明是适用的。在这种情况下，类型参数将被忽略。

这一规则源于兼容性问题和可替代性原则。由于接口或超类可以独立于其子类型进行泛型化，我们可以用非泛型方法重写泛型方法。但是，重写(非泛型)方法必须适用于对泛型方法的调用，包括显式传递类型参数的调用。否则，子类型将不能替代其泛型超类型。

如果搜索没有产生至少一个可能适用的方法，则会发生编译时错误。

根据以下规则，表达式可能与目标类型兼容:

- lambda 表达式(§15.27) 可能与函数式接口类型 T 兼容(§9.8) 如果以下都是真的:
  - 函数类型 T(§9.9)的参数数量与 lambda 表达式的参数数量相同。
  - 如果 T 的函数类型有一个 void 的返回值，那么 lambda 主体要么是一个语句表达式(§14.8)，要么是一个 void 兼容的块(§15.27.2)。
  - 如果 T 的函数类型具有(非空)返回类型，则 lambda 主体要么是表达式，要么是与值兼容的块(§15.27.2)。
- 方法引用表达式(§15.13)可能与函数接口类型 T 兼容，如果 T 的函数类型的参数数量为 n，当方法引用表达式以参数数量为 n 的函数类型为目标时，至少存在一个可能适用的方法(§15.13.1)，并且下列条件之一成立:
  - 方法引用表达式具有形式 ReferenceType :: [TypeArguments] Identifier 并且至少一个可能使用的方法是(i)静态方法并且参数个数为 n，或者(ii)非静态方法并且参数个数为 n-1。
  - 方法引用表达式有一些其他形式并且至少一个可能使用的方法不是静态的。
- 如果类型变量是候选方法的类型参数，则 lambda 表达式或方法引用表达式可能与类型变量兼容。
- 如果括号表达式（§15.8.5）所包含的表达式可能与类型兼容，则该表达式可能与该类型兼容。
- 如果条件表达式（§15.25）的第二个和第三个操作数表达式都可能与某个类型兼容，则该表达式可能与该类型兼容。
- 如果 switch 表达式（§15.28）的所有结果表达式都可能与某个类型兼容，则该表达式可能与该类型兼容。
- 类实例创建表达式、方法调用表达式或独立形式的表达式（§15.2）可能与任何类型兼容。

潜在适用性的定义超出了基本的算术检查，还考虑了函数式接口目标类型的存在和“形状”。 在某些涉及类型参数推断的情况下，出现为方法调用参数的 lambda 表达式在重载解析之后才能正确类型化。这些规则允许仍然考虑 lambda 表达式的形式，丢弃可能导致歧义错误的明显不正确的目标类型。

15.12.2.2阶段 1：识别通过严格调用适用的匹配参数方法

参数表达式被认为与可能适用的方法 m 的适用性相关，除非它具有以下形式之一:

- 隐式类型的 lambda 表达式 (§15.27.1)。
- 不精确的方法引用表达式 (§15.13.1)。
- 如果 m 是一个泛型方法，并且方法调用没有提供显式类型参数，则显式类型化的 lambda 表达式或确切的方法引用表达式，其对应的目标类型(从 m 的签名派生)是 m 的类型参数。
- 显式类型化的 lambda 表达式，其主体是与适用性无关的表达式。
- 显式类型化的 lambda 表达式，其主体为块，其中至少有一个结果表达式与适用性无关。
- 括号表达式（§15.8.5），其包含的表达式与适用性无关。
- 一种条件表达式（§15.25），其第二或第三操作数与适用性无关。

设 m 是一个可能适用的方法(§15.12.2.1)，其形式参数类型为 $F_1$, ..., $F_n$，参数个数为 n，且设 $e_1$, ..., $e_n$ 是方法调用的实际参数表达式。那么：

- 如果 m 是泛型方法，并且该方法调用不提供显式类型参数，则该方法的适用性按照 §18.5.1 中的规定进行推断。
- 如果 m 是一个泛型方法并且方法调用提供显示类型参数, 那么设 $R_1$,⋯,$R_p$ ($p \geqslant 1$)是 m 的类型参数, 设 $B_l$ 为 $R_l$ ($1 \leqslant l \leqslant p$)的声明边界, 设 $U_1$,⋯,$U_p$ 是方法调用中给出的显示类型参数。那么 m 适用于严格调用，如果以下两个都为真:
  - 对于 $1 \leqslant i \leqslant n$, 如果 $e_i$ 与适用性相关，那么 $e_i$ 在严格的调用上下文中与 $F_i[R_i:=U_i, ..., R_p:=U_p]$ 兼容(§5.3)。
  - 对于 $1 \leqslant l \leqslant p$, $U_l <: B_l [R_1:=U_1,\cdots,R_p:=U_p]$。
- 如果 m 不是泛型方法，则如果对于 $1 \leqslant i \leqslant n$，$e_i$ 在严格调用上下文中与 $F_i$ 兼容(§5.3)，或者 $e_i$ 与适用性无关，则 m 可通过严格调用来应用。

如果未找到通过严格调用可应用的方法，则搜索适用的方法将继续进行阶段 2(§15.12.2.3)。

否则，从严格调用适用的方法中选择最具体的方法(§15.12.2.5)。

> 隐式类型化 lambda 表达式或不精确的方法引用表达式的含义在解析目标类型之前足够模糊，因此包含这些表达式的参数被认为与适用性无关;它们会被简单地忽略(除了它们的期望值)，直到重载解析完成。

12.2.2.3阶段 2：确定松散调用适用的匹配参数方法

设 m 是一个可能适用的方法(§15.12.2.1)，其形式参数类型为 $F_1$, ..., $F_n$，参数个数为 n，且设 $e_1$, ..., $e_n$ 是方法调用的实际参数表达式。然后：

- 如果 m 是泛型方法，并且该方法调用不提供显式类型参数，则该方法的适用性按照 §18.5.1 中的规定进行推断。
- 如果 m 是泛型方法并且方法调用提供显式类型参数, 那么设 $R_1$,⋯,$R_p$ ($p > 1$)为 m 的类型

参数, 设 $B_l$ 为 $R_l$ (1 ⩽ l ⩽ p)的声明边界, 设 $U_1$, ..., $U_p$ 是方法调用中给出的显式类型参数。如果满足以下两个条件，则 m 可通过松散调用来应用:

- 对于 1 ⩽ i ⩽n, 如果 $e_i$ 与适用性相关 (§15.12.2.2), 那么 $e_i$ 在松散调用上下文中与 $F_i[R_1:=U_1, ..., R_p:=U_p]$兼容。
- 对于 1 ⩽ l ⩽ p, $U_l <: B_l [ R_1:=U_1,\cdots,R_p:=U_p]$。

- 如果 m 不是泛型方法，则 m 可通过松散调用来应用，如果对于 1⩽i⩽n，$e_i$ 在松散调用上下文中与 $F_i$ 兼容(§5.3)，或者 $e_i$ 与适用性无关。

如果没有找到通过松散调用可应用的方法，则搜索适用的方法继续进行阶段 3(§15.12.2.4)。

否则，从松散调用适用的方法中选择最具体的方法(§15.12.2.5)。

*15.12.2.4* 第 3 阶段：确定适用可变参数调用的方法

可变参数方法具有形式参数类型 $F_1$, ..., $F_{n-1}$, $F_n$[], 设方法的第 i 个可变参数类型定义如下:

- 对于 i ⩽ n-1, 第 i 个可变参数类型是 $F_i$。
- 对于 i ⩾ n, 第 i 个可变参数类型是 $F_n$。

设 m 是具有可变参数的可能适用方法(§15.12.2.1)，$T_1$, ..., $T_k$ 是 m 的前 k 个可变参数类型，$e_1$, ..., $e_k$ 是方法调用的实际参数表达式，那么:

- 如果 m 是泛型方法，并且该方法调用不提供显式类型参数，则该方法的适用性按照 §18.5.1 中的规定进行推断。
- 如果 m 是泛型方法，且方法调用提供了显式类型参数，则设 $R_1$, ..., $R_p$ (p ⩾1)为 m 的类型参数，$B_l$ 为 $R_l$ (1⩽l⩽p)的声明边界，$U_1$, ..., $U_p$ 为方法调用中给出的显式类型参数。如果满足以下条件，则 m 可通过可变参数调用来应用:
  - 对于 1 ⩽ i ⩽k, 如果 $e_i$ 与适用性相关(§15.12.2.2)，则 $e_i$ 在松散调用的上下文中与 $T_i[R_1:=U_1, ..., R_p:=U_p]$兼容(§5.3)。
  - 对于 1 ⩽ l ⩽ p, $U_l <: B_l[ R_1:=U_1, ..., R_p:=U_p]$。
- 如果 m 不是泛型方法，则如果对于 1⩽i⩽k，$e_i$ 在松散调用上下文中与 $T_i$ 兼容(§5.3)，或者 $e_i$ 与适用性无关，则 m 可通过可变参数调用来应用。

如果找不到可变参数调用适用的方法，则会发生编译时错误。

否则，从可变参数调用所适用的方法中选择最具体的方法(§15.12.2.5)。

*15.12.2.5* 选择最具体的方法

如果多个成员方法既可访问又适用于方法调用，则有必要选择一个成员方法来为运行时方法分派提供描述符。Java 编程语言使用的规则是选择最具体的方法。

非正式的直觉是，如果第一个方法处理的任何调用都可以传递给另一个方法，而不会出现

编译时错误，那么一个方法比另一个方法更具体。在显式类型的 lambda 表达式参数(§15.27.1)或可变参数调用(§15.12.2.4)等情况下，允许灵活地使一个签名适应另一个签名。

对于具有参数表达式 $e_1, ..., e_k$ 的调用，如果下列条件之一为真，则一种适用的方法 $m_1$ 比另一种适用的方法 $m_2$ 更具体：

- $m_2$ 是泛型，而对于参数表达式 $e_1, ..., e_k$，通过§18.5.4 推断 $m_1$ 比 $m_2$ 更具体。
- $m_2$ 不是泛型，并且 $m_1$ 和 $m_2$ 可通过严格或松散调用来应用，其中 $m_1$ 具有形式参数类型 $S_1, ..., S_n$，$m_2$ 具有形式参数类型 $T_1,\cdots,T_n$，对于所有 i($1 \leqslant i \leqslant n$，n=k)以及参数 $e_i$，类型 $S_i$ 比 $T_i$ 更具体。
- $m_2$ 不是泛型，并且 $m_1$ 和 $m_2$ 可通过可变参数调用来应用，并且其中 $m_1$ 的前 k 个可变参数类型是 $S_1, ..., S_k$，$m_2$ 的前 k 个可变参数类型是 $T_1, ..., T_k$，对于所有 i($1 \leqslant i \leqslant k$)以及参数 $e_i$，类型 $S_i$ 比 $T_i$ 更具体。另外，如果 $m_2$ 具有 k+1 个参数，则 $m_1$ 的第 k+1 个可变参数类型是 $m_2$ 的第 k+1 个可变参数类型的子类型。

上述条件是一种方法可能比另一种方法更具体的唯一情况。

对于任何表达式，类型 S 比类型 T 更具体如果 S <: T (§4.10)。

如果满足以下所有条件，则对于表达式 e，函数式接口类型 S 比函数式接口类型 T 更具体：

- S 接口既不是 T 接口的超接口，也不是 T 接口的子接口。

  如果 S 或 T 是交集类型，则 S 的任何接口不是 T 的任何接口的超接口或子接口。(交集类型的“接口”在这里指的是在交集中显示为(可能是参数化的)接口类型的接口集。)

- 设 $MT_S$ 为 S 捕获的函数类型，设 $MT_T$ 为 T 的函数类型。$MT_S$ 和 $MT_T$ 必须具有相同的类型参数(如果有的话)(§8.4.4)。

- 设 $P_1, ..., P_n$ 为 $MT_S$ 的形式参数类型，适应 $MT_T$ 的类型参数。设 $P_1', ..., P_n'$ 为 S 的函数类型的形式参数类型（没有捕获），适应 $MT_T$ 的类型参数。设 $Q_1, ..., Q_n$ 为 $MT_T$ 的形式参数类型。那么，对所有 i ($1 \leqslant i \leqslant n$), $Q_i <: P_i$ 并且 $Q_i = P_i'$。

  通常，该规则断言从 S 和 T 派生的形式参数类型是相同的。但是在 S 是通配符参数化类型的情况下，为了允许捕获变量发生在形式参数类型中，检查更加复杂:首先，T 的每个形式参数类型必须是 S 捕获对应的形式参数类型的子类型;其次，在将通配符映射到它们的边界(§9.9)之后，得到的函数类型的形式参数类型是相同的。

- 设 $R_S$ 为 $MT_S$ 的返回类型，适应 $MT_T$ 的类型参数，设 $R_T$ 为 $MT_T$ 的返回类型。下列条件之一必须为真:

  - e 是一个显式类型化的 lambda 表达式(§15.27.1)，下面有一个是正确的:

    > $R_T$ 为 void。

    > $R_S <: R_T$

    > $R_S$ 和 $R_T$ 是函数式接口类型，至少有一个结果表达式，对于 e 的每个结果表达式，

$R_S$比 $R_T$更具体。

带块体的 lambda 表达式的结果表达式的定义见§15.27.2;带表达式体的 lambda 表达式的结果表达式就是表达式体本身。

> $R_S$是原生类型,$R_T$是引用类型,而且至少有一个结果表达式，e 的每个结果表达式都是一个原生类型的独立表达式(§15.2)。

> $R_S$ 是引用类型, $R_T$ 是原生类型, 至少有一个结果表达式，e 的每个结果表达式要么是引用类型的独立表达式，要么是多元表达式。

- e 是一个确切的方法引用表达式(§15.13.1)，并且下列情况之一是正确的:

  > $R_T$为 void。

  > $R_S$ <: $R_T$

  > $R_S$是一个原生类型，$R_T$是一个引用类型，方法引用的编译时声明的返回类型是原生类型。

  > $R_S$ 是一个引用类型，$R_T$ 是一个原生类型，方法引用的编译时声明的返回类型是一个引用类型。

- e 是带括号的表达式，其中一个条件递归地应用于包含的表达式。
- e 是一个条件表达式，对于第二个和第三个操作数，递归地应用其中一个条件。
- e 是一个 switch 表达式，对于它的每个结果表达式，递归地应用这些条件中的一个。

当且仅当 $m_1$比 $m_2$更具体且 $m_2$不比 $m_1$更具体时，方法 $m_1$严格地比另一方法 $m_2$更具体。

如果一个方法是可访问和可应用的，并且没有其他可访问和可应用的方法是严格更具体的,则称该方法对于方法调用是最大特定的。

如果只有一种最大特定方法，那么这个方法就是最特定的方法;它必然比任何其他可访问可应用的方法更具体。然后，按照§15.12.3 的规定，对它进行进一步的编译时检查。

可能没有一个方法是最特定的，因为有两个或更多的方法是最大特定的。在这种情况下:

- 如果所有最大特定方法都有重写等效签名(§8.4.2)，并且其中恰好有一个是具体的(也就是说，既不是抽象的，也不是缺省的)，那么它就是最具体的方法。
- 否则，如果所有最大特定方法都具有重写等效签名，并且所有最大特定方法都是抽象的或默认的，并且这些方法的声明具有相同的擦除参数类型，并且根据下面的规则至少有一个最大特定方法是首选的，那么最特定的方法将在首选的最大特定方法子集中任意选择。最具体的方法被认为是抽象的。

  最大特定方法是首选的如果它有:

- 每个最大特定方法的签名的子签名的签名;而且
- 返回类型 R(可能是 void)，其中 R 与每个最大化特定方法的返回类型相同，或者 R 是一个引用类型，也是每个最大化特定方法返回类型的子类型(在适应任何类型参数后(§8.4.4)，如果两个方法有相同的签名)。

如果根据上述规则不存在首选方法，则首选最大特定方法，如果它:

- 拥有一个签名，它是每个最大特定方法的签名的子签名;而且
- 对于每个最大特定方法都是返回类型可替换的(§8.4.5)。

最特定方法的抛出异常类型来自于最大特定方法的 throws 子句，如下所示:

1. 如果最特定的方法是泛型的，则 throws 子句首先适应最特定方法的类型参数(§8.4.4)。

   如果最特定的方法不是泛型的，但至少有一个最大特定方法是泛型的，则首先擦除 throws 子句。

2. 然后，抛出的异常类型包括满足以下约束条件的所有类型 E:
   - 在一个 throws 子句中提到了 E。
   - 对于每个 throws 子句，e 是该子句中命名的某种类型的子类型。

这些从一组重载方法派生出单一方法类型的规则也被用来标识函数式接口的函数类型(§9.9)。

- 否则，方法调用是不明确的，并发生编译时错误。

15.12.2.6 方法调用类型

最具体的可访问和适用的方法的调用类型是一种方法类型(§8.2)，它表示调用参数的目标类型、调用的结果(返回类型或 void)和调用的异常类型。确定如下:

- 如果选择的方法是泛型的，并且方法调用没有提供显式的类型参数，则调用类型被推断为§18.5.2 中指定的。

在这种情况下，如果方法调用表达式是一个多元表达式，那么它与目标类型的兼容性由§18.5.2.1 确定。

在最终确定方法调用表达式的目标类型和调用类型之前，可能会进行多次与目标类型的兼容性测试。例如，一个封闭的方法调用表达式可能需要测试更深层的方法调用表达式，以便与不同方法的形式参数类型兼容。

- 如果选择的方法是泛型的，并且方法调用提供了显式的类型参数，设 $P_i$ 为方法的类型参数，设 $T_i$ 是为方法调用提供的显式类型参数 ($1 \leqslant i \leqslant p$)。那么:
  - 如果未经检查的转换对方法适用是必要的，然后，通过对方法类型的参数类型应用替换$[P_1:=T_1, \ldots, P_p:=T_p]$来获得调用类型的参数类型，调用类型的返回类型和抛出类型由方法类型的返回类型和抛出类型的擦除给出。

- 如果未检查的转换不是方法适用所必需的，则通过将替换[$P_1$:=$T_1$, ..., $P_p$:=$T_p$]应用于方法的类型来获得调用类型。

- 如果选择的方法是非泛型的, 那么:
  - 如果未检查的转换是方法适用的必要条件，调用类型的参数类型是方法类型的参数类型，返回类型和抛出类型由方法类型的返回类型和抛出类型的擦除给出。
  - 否则，如果选择的方法是 Object 类的 getClass 方法(§4.3.2)，调用类型与方法的类型相同，只是返回类型为 Class<? extends |T|>，其中 T 是搜索的类型，由§15.12.1 确定，|T|表示 T 的擦除(§4.6)。
  - 否则, 调用类型和方法类型相同。

### 15.12.3 编译时步骤 3：选择的方法合适吗？

如果一个方法调用有一个最具体的方法声明，那么它被称为该方法调用的编译时声明。

如果方法调用的参数与其从编译时声明的调用类型派生的目标类型不兼容，则为编译时错误。

如果该编译时声明适用于可变参数调用，那么当该方法的调用类型的最后一个形式参数类型是 $F_n$[]时，如果在调用点无法访问 $F_n$ 的擦除类型(§6.6)，则会出现编译时错误。

如果编译时声明为 void，那么方法调用必须是顶级表达式(即表达式语句中的 Expression 或 for 语句的 ForInit 或 ForUpdate 部分中的表达式)，否则将发生编译时错误。这样的方法调用不会产生值，因此只能在不需要值的情况下使用。

此外，编译时声明是否合适可能取决于左括号前的方法调用表达式的形式，如下所示:

- 如果形式为 MethodName – 也就是说，只是一个 Identifier - 而编译时声明是一个实例方法，那么:
  - 如果方法调用发生在静态上下文中(§8.1.3)，则会出现编译时错误。
  - 否则，让 T 成为用于搜索的类或接口(§15.12.1)。如果方法调用的最内部的封闭类或接口声明既不是 T，也不是 T 的内部类，则为编译时错误。
- 如果形式为 TypeName . [TypeArguments] Identifier, 那么编译时声明必须是静态的，否则将发生编译时错误。
- 如果形式为 ExpressionName . [TypeArguments] Identifier 或 Primary . [TypeArguments] Identifier, 那么编译时声明必须不是在接口中声明的静态方法，否则将发生编译时错误。
- 如果形式为 super . [TypeArguments] Identifier, 那么:
  - 如果编译时声明是抽象的，则为编译时错误。
  - 如果方法调用发生在静态上下文中，则为编译时错误。

- 如果形式为 TypeName . super . [TypeArguments] Identifier, 那么:
  - 如果编译时声明是抽象的，则为编译时错误。
  - 如果方法调用发生在静态上下文中，则为编译时错误。
  - 如果 TypeName 表示类 C，那么，如果直接包含方法调用的类或接口声明不是 C 或 C 的内部类，则会发生编译时错误。
  - 如果 TypeName 表示一个接口，设 E 为直接包含方法调用的类或接口声明。如果存在一个不同于编译时声明的方法，它重写了 E 的直接超类或直接超接口的编译时声明(§9.4.1)，则会发生编译时错误。

    在超接口重写祖父母接口中声明的方法的情况下，该规则防止子接口通过简单地将祖父母接口添加到它的直接超接口列表中来"跳过"重写。访问祖父母功能的适当方法是通过直接超接口，并且只有当该接口选择公开所需的行为时才可以这样做。（或者，程序员可以自由地定义额外的超接口，用超方法调用公开所需的行为。）

编译时参数类型和编译时结果如下所示:

- 如果方法调用的编译时声明不是签名多态方法，那么:
  - 编译时参数类型是编译时声明的形式参数的类型。
  - 编译时结果是编译时声明的调用类型的结果(§15.12.2.6)。
- 如果方法调用的编译时声明是签名多态方法，那么:
  - 编译时参数类型是实际参数表达式的类型。空字面量 null(§3.10.8)的参数表达式被视为 Void 类型。
  - 编译时结果如下所示:
    > 如果签名多态方法是 void 或者返回类型不是 Object，那么编译时结果就是编译时声明的调用类型的结果(§15.12.2.6)。
    > 否则，如果方法调用表达式是表达式语句，则编译时结果为 void。
    > 否则，如果方法调用表达式是强制转换表达式的操作数(§15.16)，则编译时的结果是对强制转换表达式的类型的删除(§4.6)。
    > 否则，编译时的结果是签名多态方法的返回类型 Object。

如果以下所有条件都为真，则该方法为签名多态方法:

- 它被声明在 java.lang.invoke.MethodHandle 类或 java.lang.invoke.VarHandle 类中。
- 它只有一个可变参数(§8.4.1)，其声明类型为 Object[]。
- 它是 native 的。

然后，将以下编译时信息与在运行时使用的方法调用关联起来:

- 方法名。
- 方法调用的限定类或接口 (§13.1)。
- 按顺序排列的参数数量和编译时参数类型。
- 编译时结果。
- 调用模式，计算如下:
  - 如果编译时声明具有静态修饰符，则调用模式为 static。
  - 否则，如果左括号前的方法调用部分的形式为 super . Identifier 或 TypeName . super . Identifier, 那么调用模式为 super。
  - 否则，如果方法调用的限定类或接口实际上是一个接口，那么调用模式就是 interface。
  - 否则, 调用模式是 virtual。

如果编译时声明的调用类型的结果不是 void，那么通过对编译时声明的调用类型的返回类型应用捕获转换(§5.1.10)来获得方法调用表达式的类型。

### 15.12.4 方法调用的运行时计算

在运行时，方法调用需要五个步骤。首先，可以计算目标引用。其次，对参数表达式求值。第三，检查要调用的方法的可访问性。第四，定位要执行的方法的实际代码。第五，创建一个新的激活帧，在必要时执行同步，并将控制转移到方法代码。

15.12.4.1 计算目标引用(如果需要)

根据方法调用的形式，有六种情况需要考虑:

- 如果形式为 MethodName – 也就是说，只是一个 Identifier – 那么:
  - 如果调用模式是 static,那么没有目标引用。
  - 否则，设 T 是该方法是其成员的封闭类型声明，n 是整数，使得 T 是其声明直接包含方法调用的类的第 n 个词法封闭类型声明。目标引用是 this 的第 n 个词法封闭的实例。

    如果 this 的第 n 个词法封闭实例不存在，则为编译时错误。
- 如果形式为 TypeName . [TypeArguments] Identifier, 那么没有目标引用。
- 如果形式为 ExpressionName . [TypeArguments] Identifier, 那么:
  - 如果调用模式是 static,那么没有目标引用。ExpressionName 被计算，但是结果被丢弃。
  - 否则, 目标引用是 ExpressionName 表示的值。

- 如果形式为 Primary . [TypeArguments] Identifier, 那么:
  - 如果调用模式是 static,那么没有目标引用。主表达式被求值，但结果随后被丢弃。
  - 否则，将计算主表达式，并将结果用作目标引用。

  在任何一种情况下，如果主表达式的求值突然完成，则任何参数表达式的任何部分似乎都没有被求值，并且由于相同的原因，方法调用突然完成。
- 如果形式为 super . [TypeArguments] Identifier,那么目标引用是 this 的值。
- 如果形式为 TypeName . super . [TypeArguments] Identifier, 那么如果 TypeName 表示一个类, 目标引用是 TypeName.this 的值; 否则，目标引用是 this 的值。

例子 15.12.4.1-1. 目标引用和静态方法

当目标引用被计算然后由于调用模式是 static 而被丢弃时，不会检查该引用以查看它是否为 null:

```
class Test1 {
    static void mountain() {
        System.out.println("Monadnock");
    }

    static Test1 favorite(){
       System.out.print("Mount ");
       return null;
    }
    public static void main(String[] args) {
       favorite().mountain();
    }
}
```

程序打印:

```
Mount Monadnock
```

favorite()返回 null, 然而没有 NullPointerException 被抛出。

例子 15.12.4.1-2. 方法调用期间的计算顺序

作为实例方法调用的一部分(§15.12)，有一个表示要调用的对象的表达式。在对方法调用的任何参数表达式的任何部分求值之前，此表达式似乎已完全求值。

因此，例如，在:

```
class Test2 {
    public static void main(String[] args) {
        String s = "one";
        if (s.startsWith(s = "two"))
                System.out.println("oops");
    }
}
```

首先计算出现在“.startsWith”之前的 s，然后是参数表达式 s=“two”。 因此，在将局部变量 s 更改为引用字符串“two”之前，对字符串“one”的引用将被记住为目标引用。因此，使用参数“two”为目标对象“one”调用 startswith 方法，因此调用的结果为 false，因为字符串“one”不是以“two”开头的。因此，测试程序不会打印“oops”。

15.12.4.2计算参数

根据被调用的方法是固定参数方法还是可变参数方法，参数列表的求值过程有所不同(§8.4.1)。

如果被调用的方法是可变参数方法 m，则它必须具有 n>0 个形参。对于某些 T，m 的 final 形式参数必须具有类型 T[]，并且必须使用 k≥0 实际参数表达式来调用 m。

如果使用 k≠n 个实际参数表达式调用 m，或者如果使用 k=n 个实际参数表达式调用 m，并且第 k 个参数表达式的类型与 T[]不兼容赋值，则参数列表($e_1$, ..., $e_{n-1}$, $e_n$, ..., $e_k$)的求值方式为($e_1$, ..., $e_{n-1}$, new |T[]| { $e_n$, ..., $e_k$ })，其中|T[]|表示 T[]的擦除(§4.6)。

> 前面的段落是为处理参数化类型和数组类型的交互而精心设计的，这些交互发生在 Java 虚拟机的擦除泛型中。也就是说，如果变量数组参数的元素类型 T 是不可具体化的，例如 List<String>，则必须特别注意数组创建表达式(§15.10)，因为创建的数组的元素类型必须是可具体化。通过擦除参数列表中 final 表达式的数组类型，可以保证获得可具体化的元素类型。然后，由于数组创建表达式出现在调用上下文中(§5.3)，从具有可具体化元素类型的数组类型转换为具有不可具体化元素类型的数组类型的未检查转换是可能的，特别是可变参数。在此转换时，需要 Java 编译器给出编译时未检查警告。Oracle 的 Java 编译器的引用实现将此未检查的警告标识为信息更丰富的未检查的泛型数组创建。

参数表达式(可能按照上面的描述重写)现在被求值以产生参数值。每个参数值恰好对应于该方法的 n 个形参中的一个。

参数表达式(如果有的话)按从左到右的顺序计算。如果任何参数表达式的求值突然结束，那么它右边的任何参数表达式似乎都没有求值，方法调用也会因为同样的原因突然结束。对于 1≤j≤n，计算第 j 个参数表达式的结果是第 j 个参数值。然后继续计算，使用参数值，如下所示。

15.12.4.3检查类型和方法的可访问性

在这一小节:

- 设 D 为包含方法调用的类。
- 设 Q 为方法调用的限定类或接口 (§13.1)。
- 设 m 为编译时确定的方法名(§15.12.3)。

Java 编程语言的实现必须确保，作为链接的一部分，类或接口 Q 是可访问的:

- 如果 Q 和 D 在同一个包中，那么 Q 是可访问的。
- 如果 Q 与 D 在不同的包中，并且它们的包在同一个模块中，并且 Q 是公共的或受保护的，那么 Q 是可访问的。
- 如果 Q 与 D 在不同的包中，并且它们的包在不同的模块中，并且 Q 的模块将 Q 的包导出到 D 的模块中，并且 Q 是公共的或受保护的，那么 Q 是可访问的。

> 如果 Q 是受保护的，那么它必然是一个嵌套的类或接口，因此在编译时，它的可访问性会受到包含其

声明的类和接口的可访问性的影响。但是，在链接期间，它的可访问性不受包含其声明的类和接口的可访问性的影响。此外，在链接期间，一个受保护的 Q 与一个公共的 Q 的可访问性相同。这些在编译时访问控制(§6.6)和运行时访问控制之间的差异是由于 Java 虚拟机的限制。

实现还必须确保，在链接期间，方法 m 仍然可以在 Q 或 Q 的超类或超接口中找到。如果 m 找不到，则会出现 NoSuchMethodError(这是 IncompatibleClassChangeError 的子类)。如果 m 可以找到，那么让 C 是声明 m 的类或接口。在链接过程中，实现必须确保 C 中 m 的声明对 D 是可访问的:

- 如果 m 是公共的，那么 m 是可访问的。
- 如果 m 是受保护的，那么 m 是可访问的当且仅当(i) 要么 D 和 C 在同一个包中，或者 D 是 C 的子类或者 C 本身；并且(ii)如果 m 是受保护的实例方法，那么 Q 必须是 D 的子类或者 D 本身。

  这是检查 m 时唯一涉及到 Q 的地方，因为受保护的实例方法只能通过与调用者类型一致的限定类或接口调用。
- 如果 m 有包访问权限, 那么 m 是可访问的当且仅当 D 和 C 在同一个包中。
- 如果 m 是私有的，那么 m 是可访问的当且仅当 D 是 C，或者 D 包含 C，或者 C 包含 D，或者 C 和 D 都包含在第三方类或接口中。

如果 Q 或 m 不可访问，则会出现 IllegalAccessError(§12.3)。

如果调用模式是接口，那么实现必须检查目标引用类是否仍实现指定的接口。如果目标引用类仍然没有实现接口，则会发生 IncompatibleClassChangeError。

15.12.4.4 定位要调用的方法

如上节(§15.12.4.3)所述:

- 设 Q 为方法调用的限定类或接口(§13.1)。
- 设 m 是在 Q 或 Q 的超类或超接口中找到的方法。(注意 m 仅仅是前一节中方法的名称; 这是实际的声明。)
- 设 C 是声明 m 的类或接口。

定位要调用的方法的策略取决于调用模式:

- 如果调用模式是 static, 不需要目标引用，也不允许重写。要调用的是类或接口 C 的方法 m。
- 否则，将调用一个实例方法，并且存在一个目标引用。如果目标引用为空，此时将抛出 NullPointerException。否则，目标引用被称为指向一个目标对象，并将被用作所调用方法中关键字 this 的值。然后考虑调用模式的其他三种可能:
  - 如果调用模式是 super,则不允许重写。要调用的是类或接口 C 的方法 m。如果 m 是 abstract, 将抛出 AbstractMethodError。

- 否则, 如果调用模式是 virtual, q 和 m 共同表示签名多态方法（§15.12.3），那么目标对象是 java.lang.invoke.MethodHandle 或 java.lang.invoke.VarHandle 的实例。 目标对象封装状态，该状态与编译时与方法调用相关联的信息相匹配。Java 虚拟机规范、Java SE 19 版本和 Java SE 平台 API 中给出了这种匹配的详细信息。如果匹配成功，则直接立即调用由 java.lang.invoke.MethodHandle 实例引用的方法，或者直接立即访问由 java.lang.invoke.VarHandle 实例表示的变量，并且在这两种情况下都不执行§15.12.4.5 中的过程。

  如果匹配失败, 则抛出 java.lang.invoke.WrongMethodTypeException。

- 否则, 调用模式是 interface 或 virtual。

  如果类或接口 C 的方法 m 是私有的，那么它就是要调用的方法。

  否则, 重写可能发生。下面指定的动态方法查找用于定位要调用的方法。查找过程从类 R 开始，这是目标对象的实际运行时类。

  > 注意，对于调用模式 interface，R 必须实现 Q；对于调用模式 virtual，R 必须是 Q 或 Q 的子类。如果目标对象是数组，则 R 是表示数组类型的“类”。

动态方法查找的过程如下。假设 S 是要搜索的类，从 R 开始。然后：

1. 如果类 S 包含一个方法的声明，该方法从 R 重写类或接口 C 的方法 m(§8.4.8.1)，则该重写方法是要调用的方法，过程终止。
2. 否则，如果 S 具有超类，则使用 S 的直接超类代替 S 递归地执行该查找过程的步骤 1 和 2；要调用的方法(如果有的话)是该查找过程的递归调用的结果。
3. 如果前两步没有找到方法，则搜索 S 的超接口以寻找合适的方法。

   考虑一组具有以下属性的候选方法：(i) 每个方法都在 R 的(直接或间接)超接口中声明；(ii) 每个方法都有方法调用所需的名称和描述符；(iii) 每个方法都是非静态和非私有的；(iv) 对于每个方法，其中该方法的声明接口是 I，没有其他方法满足 (i) 通过(iii)在 I 的子接口中声明的。

   如果此集合包含默认方法，则其中一个方法是要调用的方法。否则，选择集合中的抽象方法作为要调用的方法。

动态方法查找可能会导致发生以下错误：

- 如果要调用的方法是 abstract, 则抛出 AbstractMethodError。
- 如果要调用的方法是 default, 并且在上述步骤 3 中的候选集合中出现多个默认方法, 则抛出 IncompatibleClassChangeError。
- 如果调用模式是 interface 并且要调用的方法既不是公共的，也不是私有的, 则抛出 IllegalAccessError。

如果程序中的所有类和接口都经过了一致的编译，则上面的过程(如果它终止时没有错误)将找到一个非抽象的、可访问的方法来调用。然而，如果情况并非如此，则可能会出现各种错误，如上文所述；Java 虚拟机规范 Java SE 19 版提供了关于这些情况下 Java 虚拟机行为的更多详细信息。

动态查找过程虽然在这里显式描述，但通常会隐式实现，例如，作为构建和使用每类方法调度表的副作用，或构建用于高效调度的其他每类结构的副作用。

**例子 15.12.4.4-1. 重写和方法调用**

```
class Point {
    final int EDGE = 20;
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy;
        if (Math.abs(x) >= EDGE || Math.abs(y) >= EDGE)
           clear();
    }
    void clear() {
        System.out.println("\tPoint clear");
        x = 0; y = 0;
    }
}
class ColoredPoint extends Point {
    int color;
    void clear() {
        System.out.println("\tColoredPoint clear");
        super.clear();
        color = 0;
    }
}
```



这里，子类 ColoredPoint 扩展了由其超类 Point 定义的清晰抽象。它通过使用自己的方法重写 clear 方法来实现，该方法使用 super.clear()的形式调用其超类的 clear 方法。

每当调用 clear 的目标对象是 ColoredPoint 时，就会调用此方法。 当 this 类为 ColoredPoint 时，甚至 Point 类的 move 方法也会调用 ColoredPoint 类的 clear 方法，如该测试程序的输出所示：

```
class Test1 {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println("p.move(20,20):");
        p.move(20, 20);

        ColoredPoint cp = new ColoredPoint();
        System.out.println("cp.move(20,20):");
        cp.move(20, 20);

        p = new ColoredPoint();
        System.out.println("p.move(20,20), p colored:");
        p.move(20, 20);
    }
}
```

即:

```
p.move(20,20):
        Point clear
cp.move(20,20):
        ColoredPoint clear
        Point clear
p.move(20,20), p colored:
        ColoredPoint clear
        Point clear
```

重写有时称为"延迟绑定自引用"；在本例中，这意味着在 Point.move（这实际上是 this.clear 的语法缩写）的主体中对 clear 的引用。move 调用被"延迟"选择（在运行时，基于 this 引用的对象的运行时类）的方法，而不是被"提早"选择（在编译时，仅基于 this 的类型）的方法。这为程序员提供了扩展抽象的强大方法，并且是面向对象编程中的一个关键思想。

例子 15.12.4.4-2. 使用 super 的方法调用

可以通过使用关键字 super 访问直接超类的成员来访问超类的重写实例方法，从而绕过包含方法调用的类中的任何重写声明。

当访问实例变量时，super 的含义与 this 的强制转换相同（§15.11.2），但这种等价性对于方法调用不成立。这可以通过以下示例来证明:

```
class T1 {
    String s() { return "1"; }
}
class T2 extends T1 {
    String s() { return "2"; }
}
class T3 extends T2 {
    String s() { return "3"; }
    void test() {
        System.out.println("s()=\t\t"          +  s());
        System.out.println("super.s()=\t" + super.s());
        System.out.println("((T2)this).s()=\t" + ((T2)this).s());
        System.out.println("((T1)this).s()=\t" + ((T1)this).s()); }
}
class Test2 {
    public static void main(String[] args) {
        T3 t3 = new T3();
        t3.test();
    }
}
```

其产生输出:

```
s()=            3
super.s()=      2
((T2)this).s()= 3
((T1)this).s()= 3
```

对类型 T1 和 T2 的强制转换不会更改所调用的方法，因为要调用的实例方法是根据 this 引用的对象的运行时类选择的。强制转换不会更改对象的类；它只检查类是否与指定类型兼容。

15.12.4.5 创建帧、同步和传输控制

某些类 S 中的方法 m 已被标识为要调用的方法。

现在，创建了一个新的激活帧，其中包含目标引用（如果有）和参数值（如果有），以及为要调用的方法的局部变量和栈提供的足够空间，以及实现可能需要的任何其他簿记信息（栈指针、程序计数器、对先前激活帧的引用等）。如果没有足够的内存来创建这样的激活帧，则会引发 StackOverflowError。

新创建的激活帧成为当前激活帧。这样做的效果是将参数值赋给方法的相应的新创建的方法的参数变量，并使目标引用和 this 一样可用(如果有目标引用的话)。在将每个参数值赋给其对应的参数变量之前，对其进行调用转换(§5.3)。

如果被调用的方法类型的擦除(§4.6)的签名不同于方法调用的编译时声明的类型的擦除(§15.12.3)，则如果任何变量值是一个对象，而该对象不是方法调用的编译时声明中相应形参类型的擦除的子类或子接口的实例，则抛出 ClassCastException。

如果方法 m 是 native 方法，但所需的 native、依赖于实现的二进制代码尚未加载或无法动态链接，则会引发 UnsatisfiedLinkError。

如果方法 m 未同步，则将控制转移到要调用的方法 m 的主体。

如果方法 m 是同步的，那么在转移控制之前必须锁定一个对象。在当前线程获得锁之前，不能进行进一步的操作。如果有目标引用，那么目标对象必须被锁定;否则类 S 的 Class 对象，方法 m 的类，必须被锁定。然后将控制转移到要调用的方法 m 的主体。当方法体执行完成时，无论是正常完成还是突然完成，对象会自动解锁。锁定和解锁的行为就像是将方法体嵌入到一个同步语句中一样。

例子 15.12.4.5-1. 调用的方法签名与编译时方法签名具有不同的擦除

考虑声明:

```
abstract class C<T> {
   abstract T id(T x);
}
class D extends C<String> {
   String id(String x) { return x; }
}
```

现在，给定一个调用:

```
C c = new D();
c.id(new Object()); // fails with a ClassCastException
```

被调用的实际方法 D.id()的擦除与编译时方法声明 C.id()的签名不同。前者接受 String 类型的参数，而后者接受 Object 类型的参数。在执行方法体之前，调用失败并引发 ClassCastException 异常。

只有当程序产生编译时未检查的警告时(§4.8、§5.1.6、§5.1.9、§8.4.1、§8.4.8.3、§15.13.2、§15.12.4.2、§15.27.3)时，才会出现这种情况。

实现可以通过创建桥接方法来实施这些语义。在上面的示例中，将在类 D 中创建以下桥接方法：

```
Object id(Object x) { return id((String) x); }
```

这是 Java 虚拟机实际将调用的方法，以响应上面所示的调用 c.id(new Object())，并且它将执行强制转换并根据需要失败。

## 15.13 方法引用表达式

方法引用表达式用于引用方法的调用，而不实际执行调用。某些形式的方法引用表达式还允许将类实例创建(§15.9)或数组创建(§15.10)视为方法调用。

*MethodReference:*
*ExpressionName* `::` *[TypeArguments] Identifier*
*Primary* `::` *[TypeArguments] Identifier*
*ReferenceType* `::` *[TypeArguments] Identifier*
`super` `::` *[TypeArguments] Identifier*
*TypeName* `.` `super` `::` *[TypeArguments] Identifier*
*ClassType* `::` *[TypeArguments]* `new`
*ArrayType* `::` `new`



如果 TypeArguments 出现在::的右侧，则如果任何类型参数是通配符，则它是编译时错误(§4.5.1)。

如果方法引用表达式的形式为 ExpressionName :: [TypeArguments] Identifier 或 Primary :: [TypeArguments] Identifier, 如果 ExpressionName 或 Primary 的类型不是引用类型，则是编译时错误。

如果一个方法引用表达式的形式为 super :: [TypeArguments] Identifier, 设 E 为直接包含方法引用表达式的类或接口声明。如果 E 是类 Object 或者 E 是接口，则是编译时错误。

如果一个方法引用表达式的形式为 TypeName . super :: [TypeArguments] Identifier, 那么:

- 如果 TypeName 表示类 C, 如果 C 不是当前类的词法封闭类，或者如果 C 是类 Object，则它是编译时错误。
- 如果 TypeName 表示接口 I, 然后，让 E 是直接包含方法引用表达式的类或接口声明。如果 I 不是 E 的直接超接口，或者如果存在 E，J 的其他直接超类或直接超接口，使得 J 是 I 的子类或子接口，则它是编译时错误。
- 如果 TypeName 表示类型变量，则会发生编译时错误。

如果一个方法引用表达式的形式为 super :: [TypeArguments] Identifier 或 TypeName . super :: [TypeArguments] Identifier, 如果表达式出现在静态上下文中，则为编译时错误(§8.1.3)。

如果一个方法引用表达式的形式为 ClassType :: [TypeArguments] new, 那么:

- ClassType 必须命名一个可访问(§6.6)、非抽象且不是枚举类的类，否则会发生编译时错误。
- 如果 ClassType 表示一个参数化类型(§4.5)，那么如果它的任何类型参数是通配符，那么它就是编译时错误。
- 如果 ClassType 表示原始类型(§4.8)，则如果 TypeArguments 出现在::之后，则是编译时错误。

如果方法引用表达式的形式为 ArrayType::New，则 ArrayType 必须表示可具体化的类型(§4.7)，否则会发生编译时错误。

实例方法(§15.12.4.1)的目标引用可以由方法引用表达式使用 ExpressionName、Primary 或 super 提供，也可以稍后在调用方法时提供。新的内部类实例的直接封闭实例(§15.9.2)由 this(§8.1.3)的词法封闭实例提供。

当一个类型的多个成员方法具有相同的名称时，或者当一个类具有多个构造函数时，根据方法引用表达式所针对的函数式接口类型来选择适当的方法或构造函数，如§15.13.1 中所述。

如果方法或构造函数是泛型的，则可以推断或显式提供适当的类型参数。类似地，方法引用表达式所提及的泛型类型的类型参数可以显式提供或推断。

方法引用表达式始终为多元表达式 (§15.2)。

如果方法引用表达式出现在程序中的某个位置，而不是在赋值上下文(§5.2)、调用上下文(§5.3)或转换上下文(§5.5)中，则是编译时错误。

方法引用表达式的求值会产生一个函数式接口类型的实例(§9.8)。这不会导致执行相应的方法；相反，执行可能会在稍后调用函数式接口的适当方法时执行。

下面是一些方法引用表达式，首先没有目标引用，然后有一个目标引用:

```
String::length                  // instance method
System::currentTimeMillis // static method
List<String>::size // explicit type arguments for generic type
List::size               // inferred type arguments for generic type
int[]::clone
T::tvarMember

System.out::println
"abc"::length
foo[x]::bar
(test ? list.replaceAll(String::trim) : list) :: iterator
super::toString
```

下面是更多的方法引用表达式:

```
String::valueOf            // overload resolution needed
Arrays::sort               // type arguments inferred from context
Arrays::<String>sort // explicit type arguments
```

下面是一些表示延迟创建对象或数组的方法引用表达式：

```
ArrayList<String>::new          // constructor for parameterized type
ArrayList::new                  // inferred type arguments
                           // for generic class
Foo::<Integer>new               //  explicit type arguments
                           // for generic constructor
Bar<String>::<Integer>new //        generic class, generic constructor
Outer.Inner::new                //  inner class constructor
int[]::new                      //  array creation
```

不能指定要匹配的特定签名，例如，Arrays::sort(int[])。相反，函数式接口提供了参数类型，这些参数类型用作重载解析算法的输入(§15.12.2)。这应该可以满足绝大多数用例；当出现需要更精确控制的罕见情况时，可以使用 lambda 表达式。

在类名称中，在分隔符(List<String>::size)之前使用类型参数语法会引发区分<作为类型参数括号和<作为小于运算符的解析问题。理论上，这并不比在强制转换表达式中允许类型参数差；然而，不同之处在于，仅当遇到(令牌时才出现强制转换用例；通过添加方法引用表达式，每个表达式的开始都可能是参数化类型。

### 15.13.1 方法引用的编译时声明

方法引用表达式的编译时声明是该表达式引用的方法。在特殊情况下，编译时声明实际上并不存在，而是表示类实例创建或数组创建的概念方法。编译时声明的选择取决于表达式所针对的函数类型，正如方法调用的编译时声明取决于调用的参数（§15.12.3）。

编译时声明的搜索反映了§15.12.1 和§15.2.2 中方法调用的过程，如下所示：

- 首先，确定要搜索的类型：
  - 如果方法引用表达式的形式为 ExpressionName::[TypeArguments] Identifier 或者 Primary::[TypeArguments] Identifier, 要搜索的类型是::令牌之前的表达式类型。
  - 如果方法引用表达式的形式为 ReferenceType :: [TypeArguments] Identifier, 要搜索的类型是应用于 ReferenceType 的捕获转换（§5.1.10）的结果。
  - 如果方法引用表达式的形式为 super :: [TypeArguments] Identifier, 要搜索的类型是方法引用表达式的直接封闭类或接口声明的超类类型。

    设 T 为直接包含方法引用表达式的类或接口声明。如果 T 是类 Object 或接口，则是编译时错误。
  - 如果方法引用表达式的形式为 TypeName . super :: [TypeArguments] Identifier, 那么如果 TypeName 表示类，则要搜索的类型是命名类的超类类型；否则，TypeName 表示要搜索的接口。

    如果 TypeName 既不是方法引用表达式的词法封闭类或接口声明，也不是方法引用表达的直接封闭类或接口声明的直接超接口，则是编译时错误。

    如果 TypeName 是类 Object，则是编译时错误。

    如果 TypeName 是接口，并且存在方法引用表达式 J 的直接封闭类或接口声明

的其他直接超类或直接超接口，则这是编译时错误，因此 J 是 TypeName 的子类或子接口。

- 对于其他两种形式 (involving :: new), 引用的方法是概念性的，没有要搜索的类型。

- 其次，给定具有 n 个参数的目标函数类型，确定一组潜在适用的方法：
  - 如果方法引用表达式的形式为 ReferenceType :: [TypeArguments] Identifier, 那么潜在适用的方法是：

    > 要搜索的类型的成员方法可能适用于方法调用（§15.12.2.1），该方法调用命名 Identifier，具有 n 个参数，具有类型参数 TypeArguments，并与方法引用表达式出现在同一类中；加上

    > 要搜索的类型的成员方法可能适用于方法调用，该方法调用命名 Identifier，具有 n-1 个参数，具有类型参数 TypeArguments，并与方法引用表达式出现在同一类中。

    考虑了两种不同的数字，n 和 n-1，以解释这种形式指向静态方法或实例方法的可能性。

  - 如果方法引用表达式的形式为 ClassType :: [TypeArguments] new, 然后，可能适用的方法是一组与 ClassType 的构造函数相对应的概念方法。

    如果 ClassType 是原始类型，但不是原始类型的非静态成员类型，则候选概念成员方法是那些在§15.9.3 中为类实例创建表达式指定的方法，该表达式使用<>省略类的类型参数。否则，候选概念成员方法是 ClassType 的构造函数，被视为具有返回类型 ClassType 的方法。

    在这些候选方法中，可能适用的方法是可能适用于方法调用的概念方法，该方法具有参数个数 n、类型参数 TypeArguments 并且出现在与方法引用表达式相同的类中。

  - 如果方法引用表达式的形式为 ArrayType :: new, 考虑的是一种单一的概念方法。该方法只有一个 int 类型的参数，返回 ArrayType，并且没有 throws 子句。如果 n=1，则这是唯一可能适用的方法；否则，没有可能适用的方法。

  - 对于所有其他形式, 可能适用的方法是要搜索的类型的成员方法，这些方法可能适用于方法调用，该方法调用命名为 Identifier，具有参数数量 n，具有类型参数 TypeArguments，并且出现在与方法引用表达式相同的类中。

- 最后，如果没有可能适用的方法，那么就没有编译时声明。

  否则，在给定具有参数类型 $P_1,\cdots,P_n$ 和一组可能适用的方法的目标函数类型的情况下，将按如下方式选择编译时声明：

  - 如果方法引用表达式的形式为 ReferenceType :: [TypeArguments] Identifier, 然后，对最具体的适用方法执行两次搜索。每一次搜索都在§15.12.2.2 至

§15.12.2.5 中进行了规定，并在下文进行了澄清。每次搜索都会产生一组适用的方法，并可能指定其中最特定的方法。在§15.12.2.4 规定的错误情况下，适用的方法集为空。对于§15.12.2.5 中规定的错误，没有最具体的方法。

在第一次搜索中，方法引用被视为具有类型 $P_1$, ..., $P_n$ 的参数表达式的调用。类型参数（如果有）由方法引用表达式给出。

在第二次搜索中，如果 $P_1$, ..., $P_n$ 非空并且 $P_1$ 是 ReferenceType 的子类型，然后将方法引用表达式视为具有 $P_2$, ..., $P_n$ 类型的参数表达式的方法调用表达式。如果 ReferenceType 是原始类型，并且存在这种类型的参数化，G<...>，是 $P_1$ 的超类型，要搜索的类型是应用于 G<...>的捕获转换（§5.1.10）的结果；否则，要搜索的类型与第一次搜索的类型相同。类型参数（如果有）由方法引用表达式给出。

如果第一次搜索产生了最特定的静态方法，并且第二次搜索产生的适用方法集不包含非静态方法，则编译时声明是第一次搜索的最特定方法。

否则，如果第一次搜索生成的适用方法集不包含静态方法，而第二次搜索生成了最特定的非静态方法，则编译时声明是第二次搜索的最特定方法。

否则，没有编译时声明。

- 对于所有其他形式的方法引用表达式，将执行一次最特定适用方法的搜索。搜索如§15.12.2.2 至§15.1.2.5 所述，澄清如下。

  方法引用被视为具有类型 $P_1$、…、$P_n$ 的参数表达式的调用；类型参数（如果有）由方法引用表达式给出。

  如果搜索导致§15.12.2.2 至§15.2.5 中规定的错误，或者如果最特定的适用方法是静态的，则没有编译时声明。

  否则，编译时声明是最特定的适用方法。

如果方法引用表达式的形式为 ReferenceType :: [TypeArguments] Identifier，编译时声明是静态的，ReferenceType 不是一个简单或限定名称(§6.2)，这是一个编译时错误。

如果方法引用表达式的形式为 super :: [TypeArguments] Identifier 或 TypeName . super :: [TypeArguments] Identifier，并且编译时声明是抽象的，这是一个编译时错误。

如果方法引用表达式的形式为 super :: [TypeArguments] Identifier 或 TypeName . super :: [TypeArguments] Identifier，并且方法引用表达式出现在静态上下文中（§8.1.3），这是一个编译时错误。

如果方法引用表达式的形式为 TypeName . super :: [TypeArguments] Identifier，并且 TypeName 表示类 C，并且方法引用表达式的直接封闭类或接口声明不是 C 或 C 的内部类，这是一个编译时错误。

如果方法引用表达式的形式为 TypeName . super :: [TypeArguments] Identifier，并且

TypeName 表示一个接口，并且存在一种不同于编译时声明的方法，该方法从类或接口的直接超类或直接超接口重写编译时声明，其声明直接包含方法引用表达式（§8.4.8，§9.4.1），这是一个编译时错误。

如果方法引用表达式的形式为 ClassType :: [TypeArguments] new 并且当确定§15.9.2 中规定的 ClassType 的封闭实例时，将发生编译时错误（将方法引用表达式视为非限定类实例创建表达式），这是一个编译时错误。

形式为 ReferenceType :: [TypeArguments] Identifier 的方法引用表达式可以用不同的方式解释。如果 Identifier 引用实例方法，则与标识符引用静态方法相比，隐式 lambda 表达式具有额外的参数。ReferenceType 可能具有两种适用的方法，因此上述搜索算法分别识别它们，因为每种情况都有不同的参数类型。

歧义的一个例子是：

```
interface Fun<T,R> { R apply(T arg); }

class C {
    int size() { return 0; }
    static int size(Object arg) { return 0; }

    void test() {
        Fun<C, Integer> f1 = C::size;
          // Error: instance method size()
          // or static method size(Object)?
    }
}
```

这种二义性不能通过提供比适用静态方法更具体的适用实例方法来解决：

```
interface Fun<T,R> { R apply(T arg); }

class C {
    int size() { return 0; }
    static int size(Object arg) { return 0; }
    int size(C arg) { return 0; }

    void test() {
        Fun<C, Integer> f1 = C::size;
          // Error: instance method size()
          // or static method size(Object)?
    }
}
```

搜索足够聪明，可以忽略所有适用方法（来自两个搜索）都是实例方法的二义性：

```
interface Fun<T,R> { R apply(T arg); }

class C {
    int size() { return 0; }
    int size(Object arg) { return 0; }
    int size(C arg) { return 0; }
    void test() {
       Fun<C, Integer> f1 = C::size;
         // OK: reference is to instance method size()
    }
}
```

为了方便起见，当泛型类型的名称用于引用实例方法（其中接收器成为第一个参数）时，目标类型用于确定类型参数。这便于使用 Pair::first 代替 Pair<String,Integer>::first。类似地，像 Pair::new 这样的方法引用被视为“钻石操作符”实例创建（new Pair<>()）。因为“钻石操作符”是隐式的，所以这种形式不实例化原始类型；事实上，没有办法表达对原始类型构造函数的引用。

对于某些方法引用表达式，无论目标函数类型如何，只有一种可能的编译时声明具有一种可能的调用类型(§15.12.2.6)。这样的方法引用表达式被认为是准确的。不准确的方法引用表达式称为不准确。

一个以 Identifier 结尾的方法引用表达式如果满足以下所有条件就是准确的:

- 如果方法引用表达式的形式为 ReferenceType :: [TypeArguments] Identifier，那么 ReferenceType 不表示原始类型。
- 要搜索的类型只有一个名称为 Identifier 的成员方法，该方法引用表达式出现在其中的类或接口可以访问该成员方法。
- 方法不是可变参数 (§8.4.1)。
- 如果方法是泛型(§8.4.4), 则方法引用表达式提供 TypeArguments。

一个形式为 ClassType :: [TypeArguments] new 的方法引用表达式是准确的，如果它满足以下所有条件:

- 由 ClassType 表示的类型不是原始类型, 或为原始类型的非静态成员类型。
- 由 ClassType 表示的类型只有一个构造函数，该构造函数对于出现方法引用表达式的类或接口是可访问的。
- 构造函数不是可变参数构造函数。
- 如果构造函数是泛型，那么方法引用表达式提供 TypeArguments。

形式为 ArrayType :: new 的方法引用表达式总是准确的。

### 15.13.2 方法引用的类型

如果 T 是一个函数式接口类型(§9.8)，并且表达式与派生自 T 的地面目标类型的函数类型一致，那么方法引用表达式在赋值上下文、调用上下文或强制转换上下文中与目标类型 T 兼容。

地面目标类型由 T 导出，如下所示:

- 如果 T 是通配符参数化函数式接口类型，则地面目标类型是 T 的非通配符的参数化（§9.9）。
- 否则，地面目标类型为 T。

如果下列两个都为真，则方法引用表达式与函数类型一致:

- 函数类型标识与引用对应的单个编译时声明。

- 以下之一为真:
  - 函数类型的结果为 void。
  - 函数类型的结果是 R，而将捕获转换(§5.1.10)应用于所选编译时声明的调用类型(§15.12.2.6)的返回类型的结果是 R'(其中 R 是可用于推断 R'的目标类型)，R 和 R'都不是 void，而且 R'在赋值上下文中与 R 兼容。

如果编译时声明要适用，必须进行未检查的转换，并且此转换将在调用上下文中导致未检查的警告，则会发生编译时未检查的警告，除非被@SuppressWarnings（§9.6.4.5）抑制。

如果要使上述返回类型 R'与函数类型的返回类型 R 兼容，必须进行未检查的转换，并且此转换将在赋值上下文中导致未检查的警告，则会出现编译时未检查的警告，除非被@SuppressWarnings 抑制。

如果方法引用表达式与目标类型 T 兼容，那么表达式的类型 U 是从 T 派生的地面目标类型。

如果 U 或 U 的函数类型所提到的类或接口在方法引用表达式所在的类或接口中不可访问(§6.6)，则会出现编译时错误。

对于 U 中的每个非静态成员方法 m，如果 U 的函数类型有 m 的签名的子签名，那么一个方法类型是 U 的函数类型的概念方法被称为重写 m，并且可能会发生§8.4.8.3 中规定的任何编译时错误或未检查的警告。

对于编译时声明的调用类型的 throws 子句中列出的每个检查异常类型 X，必须在 U 的函数类型的 throws 子句中提到 X 或 X 的超类，否则将发生编译时错误。

驱动兼容性定义的关键思想是，当且仅当等效 lambda 表达式(x, y, z) -> exp.<T1, T2>method(x, y, z)兼容时，方法引用是兼容的。(这是非正式的，并且存在一些问题，使得通过这样的重写来正式定义语义变得困难或不可能。)

这些兼容性规则为从一个函数式接口转换到另一个函数式接口提供了便利:

```
Task t = () -> System.out.println("hi");
Runnable r = t::invoke;
```

可以对实现进行优化，以便在传递 lambda 派生对象并转换为各种类型时，这不会导致围绕核心 lambda 主体的许多级别的适配逻辑。

与 lambda 表达式不同，方法引用可以与泛型函数类型(即具有类型参数的函数类型)一致。这是因为 lambda 表达式需要能够声明类型参数，而没有语法支持这一点;而对于方法引用，则不需要这样的声明。例如，下面的程序是合法的:

```
interface ListFactory {
    <T> List<T> make();
}

ListFactory lf = ArrayList::new;
List<String> ls = lf.make();
List<Number> ln = lf.make();
```

### 15.13.3 方法引用的运行时评估

在运行时，方法引用表达式的求值类似于类实例创建表达式的求值，因为正常完成会生成对对象的引用。方法引用表达式的计算不同于方法本身的调用。

首先，如果方法引用表达式以 ExpressionName 或 Primary 开头，则会计算此子表达式。如果子表达式的计算结果为 null，则会引发 NullPointerException，方法引用表达式会突然结束。如果子表达式突然完成，方法引用表达式也会因为同样的原因突然完成。

接下来，要么分配并初始化具有以下属性的类的新实例，要么引用具有以下属性的类的现有实例。如果要创建一个新实例，但是没有足够的空间来分配对象，那么方法引用表达式的求值会突然结束，从而抛出一个 OutOfMemoryError。

方法引用表达式的值是对具有以下属性的类实例的引用:

· 该类实现目标函数接口类型，如果目标类型是交集类型，则实现交集中提到的所有其他接口类型。

· 其中方法引用表达式的类型为 U，对于 U 中的每个非静态成员方法 m:

如果函数类型 U 有 m 签名的子签名，那么该类声明一个重写 m 的调用方法。 调用方法的主体调用所引用的方法、创建类实例或创建数组，如下所述。如果调用方法的结果不是 void，那么在任何必要的赋值转换之后，主体返回方法调用或对象创建的结果(§5.2)。

如果被重写的方法类型的擦除与 U 的函数类型的擦除签名不同，那么在方法调用或对象创建之前，调用方法的主体检查每个参数值是否是 U 的函数类型中相应参数类型擦除的子类或子接口的实例;如果不是，则抛出 ClassCastException。

· 这个类不重写函数式接口类型或上面提到的其他接口类型的其他方法，尽管它可能重写 Object 类的方法。

调用方法的主体取决于方法引用表达式的形式，如下所示:

· 如果形式为 ExpressionName :: [TypeArguments] Identifier 或 Primary :: [TypeArguments] Identifier, 然后，调用方法的主体具有编译时声明的方法调用表达式的效果，编译时声明是方法引用表达式的编译时声明。方法调用表达式的运行时求值见§15.12.4.3、§15.12.4.4 和§15.12.4.5，其中:

- 调用模式由§15.12.3 中规定的编译时声明派生而来。
- 目标引用是 ExpressionName 或 Primary 的值，这是在计算方法引用表达式时确定的。
- 方法调用表达式的参数是调用方法的形式参数。

· 如果形式为 ReferenceType :: [TypeArguments] Identifier, 调用方法的主体类似地具有编译时声明的方法调用表达式的效果，编译时声明是方法引用表达式的编译时声明。方法调用表达式的运行时求值见§15.12.4.3、§15.12.4.4 和§15.12.4.5，其中:

- 调用模式由§15.12.3 中规定的编译时声明派生而来。
- 如果编译时声明是一个实例方法，那么目标引用就是调用方法的第一个形式参数。否则，就没有目标引用。
- 如果编译时声明是一个实例方法，那么方法调用表达式的参数(如果有的话)是调用方法的第二个和后续的形式参数。否则，方法调用表达式的参数就是调用方法的形式参数。

· 如果形式为 super :: [TypeArguments] Identifier 或 TypeName . super :: [TypeArguments] Identifier, 调用方法的主体具有编译时声明的方法调用表达式的效果，编译时声明是方法引用表达式的编译时声明。方法调用表达式的运行时求值见§15.12.4.3、§15.12.4.4 和§15.12.4.5，其中:

- 调用模式是 super。
- 如果方法引用表达式以命名类的 TypeName 开始，在方法引用被求值的点，目标引用就是 TypeName.this 的值。否则，目标引用就是在方法引用被求值时 this 的值。
- 方法调用表达式的参数是调用方法的形式参数。

· 如果形式为 ClassType :: [TypeArguments] new, 调用方法的主体具有 new [TypeArguments] ClassType($A_1$, ..., $A_n$)形式的类实例创建表达式的效果，其中，参数 $A_1$, ..., $A_n$ 是调用方法的形式参数，而:

- 如§15.9.2 所述，新对象的封闭实例(如果有的话)是从方法引用表达式的位置派生出来的。
- 调用的构造函数是对应于方法引用的编译时声明的构造函数(§15.13.1)。

· 如果形式为 Type[]$^k$ :: new(k ⩾ 1), 那么调用方法的主体与 new Type [ size ] []$^{k-1}$ 格式的数组创建表达式具有相同的效果，其中 size 是调用方法的单个参数。(符号[]$^k$ 表示 k 个括号对的序列。)

如果调用方法的主体具有方法调用表达式的效果，则按照§15.12.3 的规定确定方法调用的编译时参数类型和编译时结果。为了确定编译时结果，如果调用方法的结果为 void，则方法调用表达式是表达式语句，如果调用的结果为非空，则为返回语句的 Expression。

> 方法引用表达式求值的时序比 lambda 表达式的时序更复杂(§15.27.4)。当方法引用表达式在::分隔符之前有一个表达式(而不是类型)时，会立即计算该子表达式。求值结果将一直存储到调用相应函数式接口类型的方法为止；此时，结果将用作调用的目标引用。这意味着只有当程序遇到方法引用表达式时，才会计算::分隔符之前的表达式，而不会在对函数式接口类型的后续调用中重新计算。
>
> 将此处的 null 处理与其在方法调用期间的处理进行对比是很有趣的。当一个方法调用表达式被求值时，Primary 表达式可能将调用的值限定为 null，但不会引发 NullPointerException 异常。当被调用的方法是静态的(尽管调用的语法建议使用实例方法)，就会发生这种情况。由于被 Primary 限定的方法引用表达式的适用方法被禁止为静态的(§15.13.1)，方法引用表达式的求值更简单-空 Primary 表达式总是引发 NullPointerException。

## 15.14 后缀表达式

后缀表达式包括后缀++和--操作符的使用。名称不被认为是主表达式(§15.8)，而是在语法中单独处理，以避免某些二义性。只有在后缀表达式的优先级级别上，它们才可以互换。

*PostfixExpression:*
*Primary*
*ExpressionName*
*PostIncrementExpression*
*PostDecrementExpression*

### 15.14.1 表达式名

表达式名的求值规则见§6.5.6。

#### 15.14.2 后缀自增操作符++

后跟++操作符的后缀表达式是后缀自增表达式。

*PostIncrementExpression:*
*PostfixExpression* ++

后缀表达式的结果必须是可转换为数值类型(§5.1.8)的变量，否则将发生编译时错误。

后缀递增表达式的类型就是变量的类型。后缀递增表达式的结果不是一个变量，而是一个值。

在运行时，如果操作数表达式的求值突然完成，则后缀递增表达式出于同样的原因突然完成，并且不发生递增。否则，将值 1 添加到变量的值中，并将总和存储回变量中。在加之前，对值 1 和变量的值进行二元数值提升(§5.6)。必要时，通过收窄原生转换(§5.1.3)和/或在变量被存储之前进行装箱转换(§5.1.7)，将总和收窄到变量的类型。后缀递增表达式的值是存储新值之前的变量值。

注意，上面提到的二元数值提升可能包括拆箱转换(§5.1.8)。

声明为 final 的变量不能递增，因为当将这种 final 变量的访问用作表达式时，结果是一个值，而不是一个变量。因此，它不能用作后自增操作符的操作数。

#### 15.14.3 后缀自减操作符--

后跟--操作符的后缀表达式是后缀自减表达式。

*PostDecrementExpression:*
*PostfixExpression* --

后缀表达式的结果必须是可转换为数值类型(§5.1.8)的变量，否则将发生编译时错误。

后缀递减表达式的类型就是变量的类型。后缀递减表达式的结果不是一个变量，而是一个

值。

在运行时，如果操作数表达式的求值突然完成，则后缀递减表达式出于同样的原因突然完成，并且没有发生递减。否则，将从变量的值中减去值 1，并将差值存储回变量中。在减法之前，对值 1 和变量的值进行二元数值提升(§5.6)。如果有必要，通过收窄原生转换(§5.1.3)和/或在变量被存储之前进行装箱转换(§5.1.7)来缩小差值。 后缀递减表达式的值是存储新值之前的变量值。

> 注意，上面提到的二元数值提升可能包括拆箱转换(§5.1.8)。

声明为 final 的变量不能进行递减，因为当将对 final 变量的访问用作表达式时，结果是值，而不是变量。因此，它不能用作后缀递减操作符的操作数。

## 15.15 一元操作符

操作符+、-、++、--、~、!和强制转换操作符(§15.16)称为一元操作符。 一元表达式要么是应用于操作数的一元操作符，要么是 switch 表达式(§15.28)。

*UnaryExpression:*
  *PreIncrementExpression*
  *PreDecrementExpression*
  + *UnaryExpression*
  - *UnaryExpression*
  *UnaryExpressionNotPlusMinus*

*PreIncrementExpression:*
  ++ *UnaryExpression*

*PreDecrementExpression:*
  -- *UnaryExpression*

*UnaryExpressionNotPlusMinus:*
  *PostfixExpression*
  ~ *UnaryExpression*
  ! *UnaryExpression*
  *CastExpression*
  *SwitchExpression*



带有一元运算符的表达式从右到左分组，因此-~x 的含义与-（~x）相同。

> 语法的这一部分包含一些技巧，以避免两种潜在的语法歧义。
>
> 第一个可能产生歧义的表达式是(p)+q，对于 C 或 C++程序员来说，它看起来像是对 q 进行一元+操作的类型 p 的强制转换，或者是两个量 p 和 q 的二进制相加。在 C 和 C++中，解析器通过在解析时执行有限的语义分析来处理这个问题，这样它就知道 p 是类型名还是变量名。
>
> Java 采用了不同的方法。+操作符的结果必须是数值型的，对数值进行强制转换所涉及的所有类型名

都是已知的关键字。因此，如果 p 是一个命名原生类型的关键字，那么(p)+q 只能作为一元表达式的强制转换才有意义。但是，如果 p 不是一个命名原生类型的关键字，那么(p)+q 只能作为二进制算术操作才有意义。类似的说明也适用于-操作符。语法将 CastExpression 拆分为多种情况来进行区分:

*CastExpression:*
( *PrimitiveType* ) *UnaryExpression*
( *ReferenceType {AdditionalBound}* ) *UnaryExpressionNotPlusMinus*
( *ReferenceType {AdditionalBound}* ) *LambdaExpression*

非终止符 UnaryExpression 包括所有一元运算符，但非终止符 UnaryExpressionNotPlusMinus 排除了也可以是二元运算符的所有一元运算符的使用，在 Java 中是+和-。

第二个潜在的歧义是，对于 C 或 C++程序员来说，表达式(p)++可能看起来要么是带括号的表达式的后缀自增，要么是强制转换的开始，例如，在(p)++q 中。与前面一样，C 和 C++的解析器知道 p 是类型的名称还是变量的名称。但是，在解析过程中只使用一个标记的提前查找而不进行语义分析的解析器将无法判断，当++是提前查找标记时，(p)是否应该被视为 Primary 表达式，还是作为 CastExpression 的一部分留到以后考虑。

在 Java 中，++操作符的结果必须是数字类型，对数字值进行强制转换所涉及的所有类型名都是已知的关键字。因此，如果 p 是一个命名原生类型的关键字，那么(p)++只能作为前缀递增表达式的强制转换才有意义，并且最好在++后面有一个操作数，比如 q。但是，如果 p 不是一个命名原生类型的关键字，那么(p)++只能作为 p 的后缀增量才有意义。类似的说明也适用于--操作符。因此，非终止符 UnaryExpressionNotPlusMinus 也不使用前缀操作符++和--。

### 15.15.1 前缀自增操作符++

前面带++操作符的一元表达式是前缀自增表达式。

一元表达式的结果必须是可转换为数值类型(§5.1.8)的变量，否则将发生编译时错误。

前缀增量表达式的类型是变量的类型。前缀增量表达式的结果不是一个变量，而是一个值。

在运行时，如果操作数表达式的求值突然完成，则前缀递增表达式出于同样的原因突然完成，并且没有发生递增。否则，将值 1 添加到变量的值中，并将总和存储回变量中。在加之前，对值 1 和变量的值进行二元数值提升(§5.6)。必要时，通过收窄原语转换(§5.1.3)和/或在变量被存储之前进行装箱转换(§5.1.7)，将总和收窄到变量的类型。前缀自增表达式的值是存储新值后的变量值。

注意，上面提到的二元数值提升可能包括拆箱转换(§5.1.8)。

声明为 final 的变量不能递增，因为当将这种 final 变量的访问用作表达式时，结果是一个值，而不是一个变量。因此，它不能用作前缀自增操作符的操作数。

### 15.15.2 前缀自减操作符--

前面有--操作符的一元表达式是前缀自减表达式。

一元表达式的结果必须是可转换为数值类型(§5.1.8)的变量，否则将发生编译时错误。

前缀递减表达式的类型是变量的类型。前缀递减表达式的结果不是一个变量，而是一个值。

在运行时，如果操作数表达式的求值突然完成，则前缀递减表达式出于同样的原因突然完

成，并且没有发生递减。否则，将从变量的值中减去值 1，并将差值存储回变量中。在减法之前，对值 1 和变量的值进行二元数值提升(§5.6)。如果有必要，通过收窄原生转换(§5.1.3)和/或在变量被存储之前进行装箱转换(§5.1.7)来缩小差值。前缀递减表达式的值是存储新值后的变量的值。

注意，上面提到的二元数值提升可能包括拆箱转换(§5.1.8)。

声明为 final 的变量不能进行递减，因为当将对 final 变量的访问用作表达式时，结果是值，而不是变量。因此，它不能用作前缀自减操作符的操作数。

### 15.15.3 一元加法操作符

一元+操作符的操作数表达式的类型必须是可转换为原生数值类型(§5.1.8)的类型，否则将发生编译时错误。

对操作数执行一元数字提升(§5.6)。一元加法表达式的类型是操作数的提升类型。一元加法表达式的结果不是一个变量，而是一个值，即使操作数表达式的结果是一个变量。

在运行时，一元加法表达式的值是操作数的提升值。

### 15.15.4 一元减法操作符

一元减法操作符的操作数表达式的类型必须是可转换为原生数值类型(§5.1.8)的类型，否则将发生编译时错误。

对操作数执行一元数字提升(§5.6)。

一元减法表达式的类型是操作数的提升类型。

在运行时，一元减号表达式的值是操作数提升值的算术取反。

对于整数值，取反等同于从零减去。Java 编程语言对整数使用二进制补码表示法，而二进制补码值的范围是不对称的，因此对最大负数 int 或 long 求反会产生相同的最大负数。在这种情况下会发生溢出，但不会引发异常。对于所有整数值 x，-x 等于(~x)+1。

对于浮点值，求反与从零减去不同，因为如果 x 是+0.0，那么 0.0-x 是+0.0，而-x 是-0.0。一元减法只是将浮点数的符号倒过来。特别关注个案:

- 如果操作数为 NaN，则结果为 NaN。(请记住，NaN 没有符号(§4.2.3)。)

  Java 编程语言尚未采用 2019 年版 IEEE 754 标准中更严格的要求，即对所有输入（包括 NaN）的符号位求反。

- 如果操作数是无穷大，则结果是符号相反的无穷大。
- 如果操作数为零，则结果为符号相反的零。

### 15.15.5 按位补操作符 ~

一元~运算符的操作数表达式的类型必须是可转换（§5.1.8）为原生整型的类型，否则会发生编译时错误。

对操作数执行一元数字提升（§5.6）。一元按位补表达式的类型是操作数的提升类型。

在运行时，一元按位补表达式的值是操作数提升值的按位补。在所有情况下，~x 等于（-x）-1。

### 15.15.6 逻辑补操作符 !

一元!操作符的操作符表达式的类型必须为 boolean 或 Boolean，否则会发生编译时错误。

一元逻辑补码表达式的类型为 boolean。

在运行时，如有必要，对操作数进行拆箱转换（§5.1.8）。如果（可能转换的）操作数值为 false，则一元逻辑补码表达式的值为 true；如果（可能转化的）操作值为 true，则一元逻辑补码表达式的值为 false。

## 15.16 强制转换表达式

强制转换表达式在运行时将一种数值类型的值转换为另一种数值型的类似值；或在编译时确认表达式的类型为 boolean；或在运行时检查引用值是否引用了一个对象，该对象的类与指定的引用类型或引用类型列表兼容，或者包含了原生类型的值。

*CastExpression:*
- ( *PrimitiveType* ) *UnaryExpression*
- ( *ReferenceType {AdditionalBound}* ) *UnaryExpressionNotPlusMinus*
- ( *ReferenceType {AdditionalBound}* ) *LambdaExpression*

为方便起见，此处显示了§4.4 中的以下产品：

*AdditionalBound:*
- & *InterfaceType*

圆括号及其包含的类型或类型列表有时称为强制转换运算符。

如果强制转换运算符包含类型列表，即 ReferenceType 后跟一个或多个 AdditionalBound 项，则以下所有条件都必须为真，否则会发生编译时错误：

- ReferenceType 必须表示一个类或接口类型。
- 所有列出的类型的擦除(§4.6)必须是两两不同的。
- 列出的两个类型不能是同一泛型接口的不同参数化的子类型。

强制转换表达式引入的强制转换上下文的目标类型(§5.5)是强制转换操作符中出现的 PrimitiveType 或 ReferenceType(如果后面没有 AdditionalBound 项)，或者是强制转换操作符中出现的 ReferenceType 和 AdditionalBound 项所表示的交集类型。

强制转换表达式的类型是对该目标类型应用捕获转换(§5.1.10)的结果。

类型转换可用于显式地"标记"lambda 表达式或具有特定目标类型的方法引用表达式。为了提供适当程度的灵活性，目标类型可以是表示交集类型的类型列表，前提是交集引入了函数式接口(§9.8)。

强制转换表达式的结果不是变量，而是值，即使操作数表达式的求值结果是变量。

如果操作数的编译时类型不能通过强制转换(§5.5)转换为强制转换操作符指定的目标类型，则会发生编译时错误。

否则，在运行时，通过强制转换将操作数值转换为由强制转换运算符指定的目标类型（如果需要）。

如果在运行时发现强制转换是不允许的，则抛出 ClassCastException。

> 某些强制转换会导致编译时出错。在编译时，可以证明某些强制转换在运行时总是正确的。例如，将类类型的值转换为其超类的类型总是正确的；这种强制转换在运行时不需要任何特殊操作。 最后，一些强制转换在编译时不能被证明总是正确的或总是不正确的。此类强制转换需要在运行时进行测试。详见§5.5。

## 15.17 乘法运算符

运算符*、/、和%称为乘法运算符。

*MultiplicativeExpression:*
*UnaryExpression*
*MultiplicativeExpression* * *UnaryExpression*
*MultiplicativeExpression* / *UnaryExpression*
*MultiplicativeExpression* % *UnaryExpression*

乘法运算符具有相同的优先级，并且在语法上是左结合的（它们从左到右分组）。

乘法运算符的每个操作数的类型必须是可转换（§5.1.8）为原生数字类型的类型，否则会发生编译时错误。

对操作数执行二元数字提升（§5.6）。

> 注意，二元数值提升可能包括拆箱转换（§5.1.8）。

乘法表达式的类型是其操作数的提升类型。

如果提升类型为 int 或 long，则执行整数运算。

如果提升类型为 float 或 double，则执行浮点运算。

### 15.17.1 乘法操作符 *

二元*运算符执行乘法，产生其操作数的乘积。

如果操作数表达式没有副作用，则乘法是交换运算。

当操作数都是相同类型时，整数乘法是关联的。

浮点乘法不是关联的。

如果整数乘法溢出，则结果是以某种足够大的二进制补码格式表示的数学乘积的低位。因此，如果发生溢出，则结果的符号可能与两个操作数值的数学乘积的符号不同。

浮点乘法的结果由 IEEE 754 算术规则决定:

- 如果任何一个操作数为 NaN，则结果为 NaN。
- 如果结果不是 NaN，则如果两个操作数具有相同的符号，则结果的符号为正，如果两个操作数具有不同的符号，则结果的符号为负。
- 无穷大乘以零得到 NaN。
- 把一个无穷大乘以一个有限值，得到一个有符号无穷大。该符号是由上述规则决定的。
- 在其余情况下，既不涉及无穷大，也不涉及 NaN，将计算精确的数学乘积。

  如果乘积的大小太大而不能表示，我们说运算溢出；然后结果是具有适当符号的无穷大。

  否则，使用四舍五入到最近的舍入策略将乘积舍入到最接近的可表示值(§15.4)。Java 编程语言要求支持 IEEE 754 定义的渐进式下溢。

尽管可能会发生信息上溢、下溢或丢失，但乘法运算符*的求值绝不会抛出运行时异常。

### 15.17.2 除法操作符 /

二元/运算符执行除法运算，产生其操作数的商。左操作数是被除数，右操作数是除数。

整数除法向 0 舍入。即为整数的操作数 n 和 d 经过二元数值提升(§5.6)产生的商是一个整数 q，其大小尽可能大，同时满足$|d \cdot q| \leqslant |n|$。当|n|≥|d|和 n、d 符号相同时，q 为正;当|n|≥|d|和 n、d 符号相反时，q 为负。

有一种特殊情况不满足这一规则:如果被除数是其类型的最大可能大小的负整数，而除数为-1，则发生整数溢出，结果等于被除数。尽管溢出，但在这种情况下没有抛出异常。另一方面，如果整数除法中除数的值为 0，则抛出 ArithmeticException。

浮点除法的结果由 IEEE 754 算法规则确定:

- 如果任何一个操作数为 NaN，则结果为 NaN。
- 如果结果不是 NaN，如果两个操作数的符号相同，则结果的符号为正，如果操作数的符号不同，则结果为负。
- 无穷大除以无穷大的结果是 NaN。
- 将无穷大除以有限值得到有符号无穷大。这个符号是由上述规则决定的。
- 用一个有限值除以一个无穷值，结果是有符号的零。这个符号是由上述规则决定的。
- 0 除以 0 的结果是 NaN;0 除以任何其他有限值，结果是有符号的 0。这个符号是由上述规则决定的。
- 非零有限值除以零会得到有符号无穷大。这个符号是由上述规则决定的。
- 在剩余的情况下，如果既不涉及无穷大也不涉及 NaN，则计算精确的数学商。

如果商的大小太大而无法表示，我们称操作溢出；结果是适当符号的无穷大。

否则，使用四舍五入到最近舍入策略（第 15.4 节），将商四舍五进到最接近的可表示值。Java 编程语言需要支持 IEEE 754 定义的逐渐下溢。

尽管可能会发生上出、下溢、除零或信息丢失，但浮点除法运算符/的求值不会引发运行时异常。

### 15.17.3 取余操作符 %

二元%运算符被称为从隐含除法得到其操作数的剩余部分；左操作数是被除数，右操作数是除数。

在 C 和 C++中，余数运算符只接受整数操作数，但在 Java 编程语言中，它也接受浮点操作数。

对于二元数值提升（§5.6）后为整数的操作数，余数运算会产生一个结果值，使得(a/b)*b+(a%b)等于 a。

即使在被除数是其类型的最大可能大小的负整数且除数为-1（余数为 0）的特殊情况下，该恒等式仍然成立。

根据这个规则，余数运算的结果只有被除数为负数时才能为负数，只有被除数为正数时才能为正数。此外，结果的大小总是小于除数的大小。

如果整数余数运算符的除数值为 0，则抛出 ArithmeticException 异常。

例子 15.17.3-1. 整数取余运算符

```
class Test1 {
    public static void main(String[] args) {
        int a = 5%3;  // 2
        int b = 5/3;  // 1
        System.out.println("5%3 produces " + a +
                           "  (note that 5/3 produces " + b + ")");

        int c = 5%(-3);  // 2
        int d = 5/(-3);  // -1
        System.out.println("5%(-3) produces " + c +
                           "  (note that 5/(-3) produces " + d + ")");

        int e = (-5)%3;  // -2
        int f = (-5)/3;  // -1
        System.out.println("(-5)%3 produces " + e +
                           "  (note that (-5)/3 produces " + f + ")");

        int g = (-5)%(-3);   // -2
        int h = (-5)/(-3);   // 1
        System.out.println("(-5)%(-3) produces " + g +
                           "  (note that (-5)/(-3) produces " + h + ")");
    }
}
```

这个程序产生输出:

```
5%3 produces 2 (note that 5/3 produces 1)
5%(-3) produces 2 (note that 5/(-3) produces -1)
(-5)%3 produces -2 (note that (-5)/3 produces -1)
(-5)%(-3) produces -2 (note that (-5)/(-3) produces 1)
```

由于Java编程语言中舍入策略的选择(§15.4)，由%操作符计算的浮点余数操作的结果与IEEE 754 中由余数操作计算的结果不同。IEEE 754 余数运算计算舍入除法的余数，而不是截断除法的余数，因此它的行为与通常的整数余数运算符不同。相反，Java 编程语言在浮点操作数上定义了%，其行为类似于整数余数操作符，使用向零舍入策略进行隐含除法;这可以与 C 库函数 fmod 相比较。IEEE 754 余数运算可由库例程 Math.IEEEremainder 或 StrictMath.IEEEremainder 计算。

浮点余数运算的结果由以下规则确定，除了如何计算隐含除法外，这些规则与 IEEE 754 算法匹配:

- 如果任何一个操作数为 NaN，则结果为 NaN。
- 如果结果不是 NaN，结果的符号等于被除数的符号。
- 如果被除数是无穷大，或者除数是零，或者两者都是，结果是 NaN。
- 如果被除数是有限的，除数是无穷大，结果等于被除数。
- 如果被除数是零，除数是有限的，结果等于被除数。
- 在剩余的情况下，既不涉及无穷大，也不涉及零，也不涉及 NaN，由被除数 n 除以除数 d 得到的浮点余数 r 由数学关系 r = n - (d · q)定义，其中 q 是仅当 n/d 为负时为负，当 n/d 为正时才为正的整数，并且其大小在不超过 n 和 d 的真实数学商的大小的情况下尽可能大。

浮点取余运算符%的求值从不引发运行时异常，即使右操作数为零。不能发生上溢、下溢或精度损失。

**例子 15.17.3-2. 浮点取余操作**

```
class Test2 {
    public static void main(String[] args) {
        double a = 5.0%3.0;     // 2.0
        System.out.println("5.0%3.0 produces " + a);

        double b = 5.0%(-3.0);  // 2.0
        System.out.println("5.0%(-3.0) produces " + b);

        double c = (-5.0)%3.0;  // -2.0
        System.out.println("(-5.0)%3.0 produces " + c);

        double d = (-5.0)%(-3.0);  // -2.0
        System.out.println("(-5.0)%(-3.0) produces " + d);

    }
}
```

这个程序产生输出:

```
5.0%3.0 produces 2.0
5.0%(-3.0) produces 2.0
(-5.0)%3.0 produces -2.0
(-5.0)%(-3.0) produces -2.0
```

## 15.18 加法运算符

运算符+和-称为加法运算符。

*AdditiveExpression:*
*MultiplicativeExpression*
*AdditiveExpression* + *MultiplicativeExpression*
*AdditiveExpression* - *MultiplicativeExpression*

加性操作符具有相同的优先级，并且在语法上是左结合的(它们从左到右分组)。

如果+操作符的任意一个操作数的类型为 String，则该操作为字符串连接。

否则，+操作符的每个操作数的类型必须是可转换为原生数值类型(§5.1.8)的类型，否则将发生编译时错误。

在每种情况下，二元-操作符的每个操作数的类型必须是可转换为原生数值类型(§5.1.8)的类型，否则将发生编译时错误。



### 15.18.1 字符串连接操作符 +

如果只有一个操作数表达式是 String 类型的，那么在运行时对另一个操作数进行字符串转换(§5.1.11)，产生一个字符串。

字符串连接的结果是对一个 String 对象的引用，该对象是两个字符串操作数的连接。在新创建的字符串中，左操作数的字符位于右操作数的字符之前。

String 对象是新创建的(§12.5)，除非表达式是常量表达式(§15.29)。

实现可以选择在一个步骤中执行转换和连接，以避免创建然后丢弃中间的 String 对象。为了提高重复字符串连接的性能，Java 编译器可以使用 StringBuffer 类或类似的技术来减少通过计算表达式创建的中间 String 对象的数量。

对于原生类型，实现还可以通过直接从原生类型转换为字符串来优化包装器对象的创建。

**例子 15.18.1-1. 字符串连接**

表达式的例子:

```
"The square root of 2 is " + Math.sqrt(2)
```

产生的结果是:

```
"The square root of 2 is 1.4142135623730952"
```

+操作符在语法上是左结合的，无论它是由类型分析确定来表示字符串连接还是数字相加。在某些情况下，需要小心谨慎才能得到想要的结果。例如，表达式:

```
a + b + c
```

始终被视为含义为:

```
(a + b) + c
```

因此表达式的结果:

```
1  + 2 + " fiddlers"
```

是:

```
"3 fiddlers"
```

但该表达式的结果:

```
"fiddlers " + 1 + 2
```

是:

```
"fiddlers 12"
```

**例子 15.18.1-2. 字符串连接和条件**

在这个有趣的小例子中:

```
class Bottles {
    static void printSong(Object stuff, int n) {
        String plural = (n == 1) ? "" : "s";
  loop: while (true) {
            System.out.println(n + " bottle" + plural
                    + " of " + stuff + " on the wall,");
            System.out.println(n + " bottle" + plural
                    + " of " + stuff + ";");
            System.out.println("You take one down "
                    + "and pass it around:");
            --n;
            plural = (n == 1) ? "" : "s";
            if (n == 0)
                break loop;
            System.out.println(n + " bottle" + plural
                    + " of " + stuff + " on the wall!");
            System.out.println();
        }
         System.out.println("No bottles of " + stuff
                                + " on the wall!");
    }

    public static void main(String[] args) {
        printSong("slime", 3);
    }
}
```

方法 printSong 将打印儿童歌曲的一个版本。stuff 的流行的值包括 "pop"和"beer"; n 最流行的值是 100。

以下是运行程序的结果：

```
3 bottles  of  slime onthe wall,
3 bottles  of  slime;
You take one  down andpass it around:
2 bottles  of  slime onthe wall!

2 bottles  of  slime onthe wall,
2 bottles  of  slime;
You take one down and pass it around:
1 bottle of slime on the wall!

1 bottle of slime on the wall,
1 bottle of slime;
You take one down and pass it around:
No bottles of slime on the wall!
```

在代码中，请注意，在适当的情况下，会有条件地生成单数“bottle”，而不是复数“bottle”;还要注意如何使用字符串连接操作符来中断长常量字符串：

```
"You take one down and pass it around:"
```

分成两部分，以避免源代码中出现不方便的长行。

### 15.18.2 数字类型的加法运算符（+和-）

二元+运算符在应用于两个数字类型的操作数时执行加法，生成操作数之和。

二元-运算符执行减法运算，产生两个数字操作数的差。

对操作数执行二元数值提升（§5.6）。

注意，二元数值提升可能包括拆箱转换（§5.1.8）。

数字操作数上的加法表达式的类型是其操作数的提升类型。

如果此提升类型为 int 或 long，则执行整数运算。

如果此提升类型为 float 或 double，则执行浮点运算。

如果操作数表达式没有副作用，则加法是一种交换运算。

当操作数都是同一类型时，整数相加是关联的。

浮点加法不是关联的。

如果整数加法溢出，则结果是以某种足够大的二进制补码格式表示的数学和的低阶位。如果发生溢出，则结果的符号与两个操作数值的数学和的符号不同。

浮点加法的结果由 IEEE 754 算术规则决定：

- 如果任何一个操作数为 NaN，则结果为 NaN。
- 符号相反的两个无穷大之和为 NaN。
- 同一符号的两个无穷大之和就是该符号的无穷大。

- 无穷大和有限值的和等于无穷大的操作数。
- 符号相反的两个零的和为正零。
- 同一符号的两个零的和就是该符号的零。
- 零和非零有限值之和等于非零操作数。
- 两个大小相同、符号相反的非零有限值之和为正零。
- 在剩余的情况下，如果既不涉及无穷大，也不涉及零或 NaN，并且操作数具有相同的符号或不同的大小，则计算精确的数学和。

  如果和的大小太大而无法表示，我们称操作溢出；结果是适当符号的无穷大。

  否则，使用四舍五入到最近舍入策略（第 15.4 节），将总和舍入至最接近的可表示值。Java 编程语言需要支持逐渐下溢。

二元-操作符在应用于两个数字类型的操作数时执行减法运算，产生其操作数的差值；左操作数是被减数，右操作数是减数。

对于整数减法和浮点减法，a-b 产生的结果总是与 a+(-b)的结果相同。

请注意，对于整数值，从零减去等同于求反。然而，对于浮点操作数，从零减去与求反不同，因为如果 x 是+0.0，则 0.0-x 是+0.0，但-x 是-0.0。

尽管可能会发生上溢、下溢或信息丢失，但数值加法运算符的计算从不会引发运行时异常。

## 15.19 移位操作符

运算符<<(左移)、>>(带符号右移)和>(无符号右移)称为移位运算符。移位运算符左操作数是要移位的值；右操作数指定移位距离。

*ShiftExpression:*
*AdditiveExpression*
*ShiftExpression* << *AdditiveExpression*
*ShiftExpression* >> *AdditiveExpression*
*ShiftExpression* >>> *AdditiveExpression*

移位操作符在语法上是左结合的(它们从左到右分组)。

一元数值提升(§5.6)分别对每个操作数执行。(不对操作数执行二元数值提升。)

如果移位运算符的每个操作数的类型在一元数值提升后不是原生整型，则为编译时错误。

移位表达式的类型是左操作数的提升类型。

如果左操作数的提升类型为 int，则只使用右操作数的最低五位作为移位距离。这就好比右操作数接受了掩码值为 0x1f(0b11111)的按位逻辑 AND 运算符&(§15.22.1)。因此，实际使用的移位距离始终在 0 到 31 的范围内(包括 0 和 31)。

如果左操作数的提升类型为 long，则只使用右操作数的最低六位作为移位距离。这就好比右操作数接受了掩码值为 0x3f(0b111111)的按位逻辑 AND 运算符&(§15.22.1)。因此，实际使用的移位距离始终在 0 到 63 的范围内(包括 0 和 63)。

在运行时，对左操作数的值的二进制补码整数表示执行移位操作。

n << s 的值为 n 左移 s 位;这相当于(即使发生溢出)乘以 2 的 s 次方。

n>> s 的值为 n 右移 s 位，高位带符号扩展。结果值为 floor(n / $2^s$)。对于 n 的非负值，这相当于截断整数除法，由整数除法运算符/计算，除以 2 的 s 次方。

n >>> s 的值为 n 右移 s 位，高位零扩展，其中:

- 如果 n 为正，则结果与 n>>s 的结果相同。
- 如果 n 为负并且左操作数的类型为 int，则结果等于表达式(*n* >> *s*) + (2 << ~*s*)的结果。
- 如果 n 为负，且左操作数的类型为 long，则结果等于表达式(n >> s) + (2L << ~s)。

  相加项(2 << ~*s*) 或 (2L << ~*s*) 取消传播的符号位。

  请注意，由于移位运算符的右操作数的隐式掩码，移位整数值时，~s 作为移位距离等于 31-s，而移位长值时，~s 等于 63-s。

## 15.20 关系运算符

数值比较运算符<、>、<=和>=以及 instanceof 运算符称为关系运算符。

*RelationalExpression:*
*ShiftExpression*
*RelationalExpression* < *ShiftExpression*
*RelationalExpression* > *ShiftExpression*
*RelationalExpression* <= *ShiftExpression*
*RelationalExpression* >= *ShiftExpression*
*InstanceofExpression*

关系运算符在语法上是左结合的(它们从左到右分组)。

然而，这一事实并不有用。例如，a<b<c 解析为(a<b)<c，这始终是编译时错误，因为 a<b 的类型始终是布尔值，而<不是布尔值上的运算符。

关系表达式的类型总是 boolean。

### 15.20.1 数值比较运算符 <, <=, >, 和 >=

数值比较运算符的每个操作数的类型必须是可转换(§5.1.8)为原生数值类型的类型，否则会发生编译时错误。

对操作数执行二元数值提升(§5.6)。

请注意，二元数值提升可能包括拆箱转换(§5.1.8)。

如果提升的操作数类型为 int 或 long，则执行带符号整数比较。

如果提升的类型是 float 或 double，则执行浮点比较。

由 IEEE 754 标准规范确定的浮点比较结果是:

- 如果任何一个操作数为 NaN，则结果为假。
- 除 NaN 之外的所有值都是有序的，负无穷小于所有有限值，正无穷大于所有有限值。
- 正零和负零被认为相等。

  例如, -0.0<0.0 为 false, 但是 -0.0<=0.0 为 true。

  但是请注意，方法 Math.min 和 Math.max 认为负 0 严格小于正 0。

根据浮点数的这些考虑，以下规则适用于整型操作数或非 NaN 的浮点操作数:

- 如果左操作数小于右操作数，<操作符生成的值为 true，否则为 false。
- 如果左操作数小于或等于右操作数，<=操作符生成的值为 true，否则为 false。
- 如果左操作数大于右操作数，则>操作符生成的值为 true，否则为 false。
- 如果左操作数大于等于右操作数，则>=操作符生成的值为 true，否则为 false。

### 15.20.2 `instanceof` 操作符

instanceof 表达式可以执行类型比较或模式匹配。

*InstanceofExpression:*
*RelationalExpression* `instanceof` *ReferenceType*
*RelationalExpression* `instanceof` *Pattern*

如果 instanceof 关键字右边的操作数是 ReferenceType，那么 instanceof 关键字就是类型比较操作符。

如果 instanceof 关键字右边的操作数是 Pattern，那么 instanceof 关键字就是模式匹配操作符。

当 instanceof 是类型比较操作符时，应用以下规则:

- 表达式 RelationalExpression 的类型必须是引用类型或空类型，否则将发生编译时错误。
- RelationalExpression 必须向下转换以兼容 ReferenceType(§5.5)，否则会发生编译时错误。
- 在运行时，类型比较操作符的结果如下所示:
  - 如果 RelationalExpression 的值是空引用(§4.1)，那么结果是 false。
  - 如果 RelationalExpression 的值不是空引用，那么如果该值可以被强制转换为 ReferenceType 而不引发 ClassCastException，结果为真，否则为假。

当 instanceof 是模式匹配操作符时，应用以下规则:

- 表达式 RelationalExpression 的类型必须是引用类型或空类型，否则将发生编译时错误。
- RelationalExpression 必须与 Pattern 兼容(§14.30.1)，否则会发生编译时错误。
- 如果 RelationalExpression 的类型是 Pattern 类型的子类型，那么会发生编译时错误。
- 在运行时，模式匹配操作符的结果如下所示:
  - 如果 RelationalExpression 的值是空引用，那么结果是 false。
  - 如果 RelationalExpression 的值不是 null 引用，那么如果该值与 Pattern 匹配，结果为 true，否则为 false。

    真实结果的一个副作用是，在 Pattern 中声明的模式变量将被初始化。

例子 15.20.2-1. 类型比较操作符

```
class Point { int x, y; }
class Element { int atomicNumber; }
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        Element e = new Element();
        if (e instanceof Point) {  // compile-time error
            System.out.println("I get your point!");
            p = (Point)e; // compile-time error
        }
    }
}
```

此程序导致两个编译时错误。强制转换(Point)e 是不正确的因为 Element 或其任何可能的子类（此处未显示）的实例都不可能是 Point 的任何子类的实例。由于完全相同的原因，instanceof 表达式不正确。另一方面，如果类 Point 是 Element 的子类(在这个例子中这是一个公认的奇怪的概念):

```
class Point extends Element { int x, y; }
```

然后就可以进行强制转换了，尽管它需要运行时检查，而且 instanceof 表达式将是合理且有效的。强制转换(Point)e 永远不会引发异常，因为如果 e 的值不能正确地强制转换为 Point 类型，它就不会被执行。

在 Java SE 16 之前，类型比较操作符的 ReferenceType 操作数必须是可具体化的(§4.7)。这防止了参数化类型的使用，除非它的所有类型参数都是通配符。Java SE 16 提升了这一要求，允许使用更多的参数化类型。例如，在下面的程序中，测试方法参数 x(静态类型 List<Integer>)在运行时是否具有更“精练”的参数化类型 ArrayList<Integer>是合法的:

```
import java.util.ArrayList;
import java.util.List;

class Test2 {
    public static void main(String[] args) {
        List<Integer> x = new ArrayList<Integer>();

        if (x instanceof ArrayList<Integer>) {  // OK
            System.out.println("ArrayList of Integers");
```

```
            }
            if (x instanceof ArrayList<String>) {   // error
                System.out.println("ArrayList of Strings");
            }
            if (x instanceof ArrayList<Object>) {   // error
                System.out.println("ArrayList of Objects");
            }
        }
    }
```

第一个 instanceof 表达式是合法的，因为有一个从 List<Integer>到 ArrayList<Integer>的强制类型转换。但是，第二个和第三个 instanceof 表达式都导致编译时错误，因为没有从 List<Integer>到 ArrayList<String>或 ArrayList<Object>的强制转换。

## 15.21 相等操作符

操作符==(等于)和!=(不等于)称为相等操作符。

*EqualityExpression:*
  *RelationalExpression*
  *EqualityExpression* `==` *RelationalExpression*
  *EqualityExpression* `!=` *RelationalExpression*

相等操作符在语法上是左结合的(它们从左到右分组)。

然而，这个事实从本质上来说是没有用的。例如，a==b==c 解析为(a==b)==c。
a==b 的结果类型总是布尔型，因此 c 必须是布尔型，否则将发生编译时错误。因此，a==b==c 不测试 a、b 和 c 是否都相等。

如果操作数表达式没有副作用，则相等操作符是可交换的。

相等操作符类似于关系操作符，只是它们的优先级较低。因此，只要 a<b 和 c<d 具有相同的真值，则 a<b==c<d 为真。

等式运算符可用于比较两个可转换为数值类型的操作数(§5.1.8)，或两个 boolean 或 Boolean 类型的操作数，或两个分别为引用类型或空类型的操作数。所有其他情况都会导致编译时错误。

相等表达式的类型总是布尔型。

在所有情况下，a!=b 产生与!(a==b)相同的结果。

### 15.21.1 数值相等运算符==和=

如果相等操作符的两个操作数都是数字类型，或者一个是数字类型而另一个可转换为数字类型(§5.1.8)，则对操作数进行二元数值提升(§5.6)。

请注意，二元数值提升可能包括拆箱转换(§5.1.8)。

如果提升的操作数类型为 int 或 long，则执行整数相等性测试。

如果提升的类型是 float 或 double，则执行浮点相等性测试。

浮点相等性测试按照 IEEE 754 标准的规则进行:

- 如果其中一个操作数为 NaN，则==的结果为假，而!=的结果为真。

  实际上，当且仅当 x 的值为 NaN 时，测试 x！=x 才为真。

  方法 Float.isNaN 和 Double.isNaN 可以用来测试值是否为 NaN。

- 正零和负零被认为是相等的。

  例如, -0.0==0.0 为 true。

- 否则，相等运算符会将两个不同的浮点值视为不相等。

  具体地说，有一个值表示正无穷大，一个值表示负无穷大；每个值都只与其自身相等，而每个值与所有其他值都不相等。

根据浮点数的这些注意事项，以下规则适用于整数操作数或非 NaN 的浮点操作数:

- 如果左操作数的值等于右操作数的值，则==运算符生成的值为 true；否则，结果为 false。

- 如果左操作数的值不等于右操作数的值，则！=运算符生成的值为 true；否则，结果为 false。



### 15.21.2 布尔相等运算符 == 和 !=

如果相等运算符的操作数都是布尔类型，或者如果一个操作数是 boolean 类型，而另一个操作数是 Boolean 类型，则该运算是布尔相等。

布尔相等运算符是关联的。

如果其中一个操作数是 Boolean 类型的，则对其进行拆箱转换(§5.1.8)。

如果操作数(在任何所需的拆箱转换之后)都为真或都为假，则==的结果为真；否则，结果为假。

如果操作数都为真或都为假，则！=的结果为 false；否则，结果为 true。

因此，当应用于布尔操作数时，！=的行为与^(§15.22.2)相同。

### 15.21.3 引用相等运算符 == 和 !=

如果相等运算符的操作数既是引用类型，也是空类型，则该操作是对象相等。

如果无法通过强制转换将其中一个操作数的类型转换为另一个操作数的类型，则为编译时错误(§5.5)。两个操作数的运行时值必然不相等(忽略两个值都为 null 的情况)。

在运行时，如果操作数值都为 null 或都引用相同的对象或数组，则==的结果为 true；否则，结果为 false。

如果两个操作数值都为 null 或都引用同一对象或数组，则！=的结果为 false；否则，结果

为 true。

虽然==可用于比较 String 类型的引用，但这样的相等性测试确定两个操作数是否引用相同的 String 对象。如果操作数是不同的 String 对象，则结果为 false，即使它们包含相同的字符序列(§3.10.5、§3.10.6)。可以通过方法调用 s.equals(t)测试两个字符串 s 和 t 的内容是否相等。

## 15.22 位操作符和逻辑操作符

按位运算符和逻辑运算符包括 AND 运算符&、异或运算符^和包含或运算符|。

*AndExpression:*
*EqualityExpression*
*AndExpression* & *EqualityExpression*

*ExclusiveOrExpression:*
*AndExpression*
*ExclusiveOrExpression* ^ *AndExpression*

*InclusiveOrExpression:*
*ExclusiveOrExpression*
*InclusiveOrExpression* | *ExclusiveOrExpression*

这些运算符具有不同的优先级，其中&的优先级最高，而|的优先级最低。

这些操作符在语法上都是左结合的(每个从左到右分组)。

如果操作数表达式没有副作用，则每个操作符都是可交换的。

每个运算符都是结合式的。

位操作符和逻辑操作符可用于比较两个数字类型的操作数或两个布尔类型的操作数。所有其他情况都会导致编译时错误。

### 15.22.1 整数按位操作符 &,^,和 |

如果操作符&、^或|的两个操作数都是可转换为原生整型的类型(§5.1.8)，则首先对操作数进行二元数值提升(§5.6)。

位操作符表达式的类型是操作数的提升类型。

对于&，结果值是操作数值的按位与。

对于^，结果值是操作数值的按位异或。

对于|，结果值是操作数值的按位或。

例如，表达式的结果:

```
0xff00 & 0xf0f0
```

是:

```
0xf000
```

表达式的结果:

```
0xff00 ^ 0xf0f0
```

是:

```
0x0ff0
```

表达式的结果:

```
0xff00 | 0xf0f0
```

是:

```
0xfff0
```

### 15.22.2 布尔逻辑操作符 &,^,和 |

如果操作符&、^或|的两个操作数都是 boolean 或 Boolean，则位操作符表达式的类型为 boolean。在所有情况下，操作数都需要进行拆箱转换(§5.1.8)。

对于&，如果两个操作数值都为 true，则结果值为 true;否则，结果为 false。

对于^，如果操作数值不同，则结果值为 true;否则，结果为 false。

对于|，如果两个操作数值都为 false，则结果值为 false;否则，结果为 true。

## 15.23 条件与操作符 &&

条件操作符&&类似于&(§15.22.2)，但只有当左操作数的值为 true 时，才计算其右操作数。

*ConditionalAndExpression:*
  *InclusiveOrExpression*
  *ConditionalAndExpression* `&&` *InclusiveOrExpression*

条件与操作符在语法上是左结合的(它从左到右分组)。

条件与操作符对于副作用和结果值是完全关联的。也就是说，对于任何表达式 a、b 和 c，对表达式((a) && (b)) && (c)求值产生的结果与表达式(a) && ((b) && (c))求值产生的结果相同，副作用相同，发生的顺序相同。

条件与操作符的每个操作数必须是 boolean 或 Boolean，否则将发生编译时错误。

条件与表达式的类型总是布尔型。

在运行时，先计算左操作数表达式;如果结果为 Boolean 类型，则进行拆箱转换(§5.1.8)。

如果结果值为 false，则条件与表达式的值为 false，并且不计算右操作数表达式。

如果左操作数的值为 true，则计算右表达式;如果结果为 Boolean 类型，则进行拆箱转换(§5.1.8)。结果值成为条件与表达式的值。

因此，&&对布尔操作数的计算结果与&相同。它的不同之处在于右操作数表达式是有条件地求值，而不是总是求值。

## 15.24 条件或操作符 ||

条件或运算符||类似于|(§15.22.2)，但只有当左操作数的值为 false 时，才计算其右操作数。

*ConditionalOrExpression:*
*ConditionalAndExpression*
*ConditionalOrExpression* || *ConditionalAndExpression*

条件或操作符在语法上是左结合的(它从左到右分组)。

条件或操作符与副作用和结果值完全关联。也就是说，对于任何表达式 a、b 和 c，计算表达式((a) || (b)) || (c)产生的结果与计算表达式(a) || ((b) || (c))产生的结果相同，副作用和发生顺序相同。

条件或操作符的每个操作数必须为 boolean 或 Boolean，否则将发生编译时错误。

条件或表达式的类型总是布尔型。

在运行时，先计算左操作数表达式;如果结果为 Boolean 类型，则进行拆箱转换(§5.1.8)。

如果结果值为 true，则条件或表达式的值为 true，右操作数表达式不计算。

如果左操作数的值为 false，则计算右表达式;如果结果为 Boolean 类型，则进行拆箱转换(§5.1.8)。结果值成为条件或表达式的值。

因此，对于 boolean 或 Boolean 操作数，||与|计算相同的结果。它的不同之处在于右操作数表达式是有条件地求值，而不是总是求值。

## 15.25 条件操作符 ? :

条件运算符?:使用一个表达式的布尔值来决定其他两个表达式中应该计算哪一个。

*ConditionalExpression:*
*ConditionalOrExpression*
*ConditionalOrExpression* ? *Expression* : *ConditionalExpression*
*ConditionalOrExpression* ? *Expression* : *LambdaExpression*

条件运算符在语法上是右结合的(它从右到左分组)。因此,a?b:c?d:e?f:g 和 a?b:(c?d:(e?f:g))一

样。

条件操作符有三个操作数表达式。？ 出现在第一个和第二个表达式之间，而:出现在第二个和第三个表达式之间。

第一个表达式必须是 boolean 或 Boolean 类型，否则将发生编译时错误。

如果第二个或第三个操作数表达式是 void 方法的调用，则是编译时错误。

> 事实上，根据表达式语句的语法（§14.8），条件表达式不允许出现在可能出现 void 方法调用的任何上下文中。

根据第二个和第三个操作数表达式，有三种条件表达式:布尔条件表达式、数值条件表达式和引用条件表达式。分类规则如下:

- 如果第二个和第三个操作数表达式都是布尔表达式，那么条件表达式就是布尔条件表达式。

  为了对条件进行分类，下面的表达式是布尔表达式:

  - 具有 boolean 或 Boolean 类型的独立形式的表达式(§15.2)。
  - 带括号的布尔表达式(§15.8.5)。
  - 一个 Boolean 类的类实例创建表达式(§15.9)。
  - 一种方法调用表达式(§15.12)，其选择的最特定的方法(§15.12.2.5)返回类型为 boolean 或 Boolean。

    > 注意，对于泛型方法，这是在实例化方法的类型参数之前的类型。

  - 布尔条件表达式。
  - switch 表达式(§15.28)，其结果表达式全部为布尔表达式。

- 如果第二个和第三个操作数表达式都是数值表达式，那么条件表达式就是数值条件表达式。

  为了对条件进行分类，以下表达式是数值表达式:

  - 一种独立形式(§15.2)的表达式，其类型可转换为数字类型(§4.2、§5.1.8)。
  - 带括号的数字表达式(§15.8.5)。
  - 可转换为数字类型的类的类实例创建表达式(§15.9)。
  - 一种方法调用表达式(§15.12)，其所选择的最特定的方法(§15.12.2.5)的返回类型可转换为数字类型。

    > 注意，对于泛型方法，这是在实例化方法的类型参数之前的类型。

  - 一个数值条件表达式。

- switch 表达式(§15.28)，其结果表达式全部为数值表达式。

· 否则，条件表达式为引用条件表达式。

确定条件表达式类型的过程取决于条件表达式的类型，如下几节所述。

下表通过给出第二个和第三个操作数的所有可能类型的条件表达式类型，总结了上面的规则。bnp(..)表示应用二元数字提升。“T | bnp(..)”形式用于其中一个操作数是 int 类型的常量表达式，可以在 T 类型中表示，如果操作数不能在 T 类型中表示，则使用二元数字提升。操作数类型 Object 表示除 null 类型和 8 个包装类型 Boolean, Byte, Short, Character, Integer, Long, Float, Double 之外的任何引用类型布尔、Byte、Short、Character、Integer、Long、Float、Double。

**表 15.25-A. 条件表达式类型(第 3 个基本操作数，第一部分)**

| **3rd →**<br>**2nd ↓** | byte | short | char | int |
|---|---|---|---|---|
| byte | byte | short | bnp(byte,char) | byte \| bnp(byte,int) |
| Byte | byte | short | bnp(Byte,char) | byte \| bnp(Byte,int) |
| short | short | short | bnp(short,char) | short \| bnp(short,int) |
| Short | short | short | bnp(Short,char) | short \| bnp(Short,int) |
| char | bnp(char,byte) | bnp(char,short) | char | char \| bnp(char,int) |
| Character | bnp(Character,byte) | bnp(Character,short) | char | char \| bnp(Character,int) |
| int | byte \| bnp(int,byte) | short \| bnp(int,short) | char \| bnp(int,char) | int |
| Integer | bnp(Integer,byte) | bnp(Integer,short) | bnp(Integer,char) | int |
| long | bnp(long,byte) | bnp(long,short) | bnp(long,char) | bnp(long,int) |
| Long | bnp(Long,byte) | bnp(Long,short) | bnp(Long,char) | bnp(Long,int) |
| float | bnp(float,byte) | bnp(float,short) | bnp(float,char) | bnp(float,int) |
| Float | bnp(Float,byte) | bnp(Float,short) | bnp(Float,char) | bnp(Float,int) |
| double | bnp(double,byte) | bnp(double,short) | bnp(double,char) | bnp(double,int) |
| Double | bnp(Double,byte) | bnp(Double,short) | bnp(Double,char) | bnp(Double,int) |
| boolean | lub(Boolean,Byte) | lub(Boolean,Short) | lub(Boolean,Character) | lub(Boolean,Integer) |
| Boolean | lub(Boolean,Byte) | lub(Boolean,Short) | lub(Boolean,Character) | lub(Boolean,Integer) |
| null | lub(null,Byte) | lub(null,Short) | lub(null,Character) | lub(null,Integer) |
| Object | lub(Object,Byte) | lub(Object,Short) | lub(Object,Character) | lub(Object,Integer) |

## 表 15.25-B. 条件表达式类型(第 3 个基本操作数，第二部分)

| 3rd → <br> 2nd ↓ | long | float | double | boolean |
|---|---|---|---|---|
| byte | bnp(byte,long) | bnp(byte,float) | bnp(byte,double) | lub(Byte,Boolean) |
| Byte | bnp(Byte,long) | bnp(Byte,float) | bnp(Byte,double) | lub(Byte,Boolean) |
| short | bnp(short,long) | bnp(short,float) | bnp(short,double) | lub(Short,Boolean) |
| Short | bnp(Short,long) | bnp(Short,float) | bnp(Short,double) | lub(Short,Boolean) |
| char | bnp(char,long) | bnp(char,float) | bnp(char,double) | lub(Character,Boolean) |
| Character | bnp(Character,long) | bnp(Character,float) | bnp(Character,double) | lub(Character,Boolean) |
| int | bnp(int,long) | bnp(int,float) | bnp(int,double) | lub(Integer,Boolean) |
| Integer | bnp(Integer,long) | bnp(Integer,float) | bnp(Integer,double) | lub(Integer,Boolean) |
| long | long | bnp(long,float) | bnp(long,double) | lub(Long,Boolean) |
| Long | long | bnp(Long,float) | bnp(Long,double) | lub(Long,Boolean) |
| float | bnp(float,long) | float | bnp(float,double) | lub(Float,Boolean) |
| Float | bnp(Float,long) | float | bnp(Float,double) | lub(Float,Boolean) |
| double | bnp(double,long) | bnp(double,float) | double | lub(Double,Boolean) |
| Double | bnp(Double,long) | bnp(Double,float) | double | lub(Double,Boolean) |
| boolean | lub(Boolean,Long) | lub(Boolean,Float) | lub(Boolean,Double) | boolean |
| Boolean | lub(Boolean,Long) | lub(Boolean,Float) | lub(Boolean,Double) | boolean |
| null | lub(null,Long) | lub(null,Float) | lub(null,Double) | lub(null,Boolean) |
| Object | lub(Object,Long) | lub(Object,Float) | lub(Object,Double) | lub(Object,Boolean) |

## 表 15.25-C. 条件表达式类型(引用第 3 个操作数，第一部分)

| **3rd →**<br>**2nd ↓** | Byte | Short | Character | Integer |
|---|---|---|---|---|
| byte | byte | short | bnp(byte,Character) | bnp(byte,Integer) |
| Byte | Byte | short | bnp(Byte,Character) | bnp(Byte,Integer) |
| short | short | short | bnp(short,Character) | bnp(short,Integer) |
| Short | short | Short | bnp(Short,Character) | bnp(Short,Integer) |
| char | bnp(char,Byte) | bnp(char,Short) | char | bnp(char,Integer) |
| Character | bnp(Character,Byte) | bnp(Character,Short) | Character | bnp(Character,Integer) |
| int | byte \| bnp(int,Byte) | short \| bnp(int,Short) | char \| bnp(int,Character) | int |
| Integer | bnp(Integer,Byte) | bnp(Integer,Short) | bnp(Integer,Character) | Integer |
| long | bnp(long,Byte) | bnp(long,Short) | bnp(long,Character) | bnp(long,Integer) |
| Long | bnp(Long,Byte) | bnp(Long,Short) | bnp(Long,Character) | bnp(Long,Integer) |
| float | bnp(float,Byte) | bnp(float,Short) | bnp(float,Character) | bnp(float,Integer) |
| Float | bnp(Float,Byte) | bnp(Float,Short) | bnp(Float,Character) | bnp(Float,Integer) |
| double | bnp(double,Byte) | bnp(double,Short) | bnp(double,Character) | bnp(double,Integer) |
| Double | bnp(Double,Byte) | bnp(Double,Short) | bnp(Double,Character) | bnp(Double,Integer) |
| boolean | lub(Boolean,Byte) | lub(Boolean,Short) | lub(Boolean,Character) | lub(Boolean,Integer) |
| Boolean | lub(Boolean,Byte) | lub(Boolean,Short) | lub(Boolean,Character) | lub(Boolean,Integer) |
| null | Byte | Short | Character | Integer |
| Object | lub(Object,Byte) | lub(Object,Short) | lub(Object,Character) | lub(Object,Integer) |

## 表 15.25-D. 条件表达式类型(引用第 3 个操作数，第二部分)

| 3rd → / 2nd ↓ | Long | Float | Double | Boolean |
|---|---|---|---|---|
| byte | bnp(byte,Long) | bnp(byte,Float) | bnp(byte,Double) | lub(Byte,Boolean) |
| Byte | bnp(Byte,Long) | bnp(Byte,Float) | bnp(Byte,Double) | lub(Byte,Boolean) |
| short | bnp(short,Long) | bnp(short,Float) | bnp(short,Double) | lub(Short,Boolean) |
| Short | bnp(Short,Long) | bnp(Short,Float) | bnp(Short,Double) | lub(Short,Boolean) |
| char | bnp(char,Long) | bnp(char,Float) | bnp(char,Double) | lub(Character,Boolean) |
| Character | bnp(Character,Long) | bnp(Character,Float) | bnp(Character,Double) | lub(Character,Boolean) |
| int | bnp(int,Long) | bnp(int,Float) | bnp(int,Double) | lub(Integer,Boolean) |
| Integer | bnp(Integer,Long) | bnp(Integer,Float) | bnp(Integer,Double) | lub(Integer,Boolean) |
| long | long | bnp(long,Float) | bnp(long,Double) | lub(Long,Boolean) |
| Long | Long | bnp(Long,Float) | bnp(Long,Double) | lub(Long,Boolean) |
| float | bnp(float,Long) | float | bnp(float,Double) | lub(Float,Boolean) |
| Float | bnp(Float,Long) | Float | bnp(Float,Double) | lub(Float,Boolean) |
| double | bnp(double,Long) | bnp(double,Float) | double | lub(Double,Boolean) |
| Double | bnp(Double,Long) | bnp(Double,Float) | Double | lub(Double,Boolean) |
| boolean | lub(Boolean,Long) | lub(Boolean,Float) | lub(Boolean,Double) | boolean |
| Boolean | lub(Boolean,Long) | lub(Boolean,Float) | lub(Boolean,Double) | Boolean |
| null | Long | Float | Double | Boolean |
| Object | lub(Object,Long) | lub(Object,Float) | lub(Object,Double) | lub(Object,Boolean) |

**表 15.25-E. 条件表达式类型(引用第三个操作数，第三部分)**

| 3rd → / 2nd ↓ | null | Object |
|---|---|---|
| byte | lub(Byte,null) | lub(Byte,Object) |
| Byte | Byte | lub(Byte,Object) |
| short | lub(Short,null) | lub(Short,Object) |
| Short | Short | lub(Short,Object) |
| char | lub(Character,null) | lub(Character,Object) |
| Character | Character | lub(Character,Object) |
| int | lub(Integer,null) | lub(Integer,Object) |
| Integer | Integer | lub(Integer,Object) |
| long | lub(Long,null) | lub(Long,Object) |
| Long | Long | lub(Long,Object) |
| float | lub(Float,null) | lub(Float,Object) |
| Float | Float | lub(Float,Object) |
| double | lub(Double,null) | lub(Double,Object) |
| Double | Double | lub(Double,Object) |
| boolean | lub(Boolean,null) | lub(Boolean,Object) |
| Boolean | Boolean | lub(Boolean,Object) |
| null | null | lub(null,Object) |
| Object | Object | Object |

在运行时，首先计算条件表达式的第一个操作数表达式。如有必要，将对结果执行拆箱转换。

然后，生成的布尔值用于选择第二个或第三个操作数表达式：

- 如果第一个操作数的值为真，则选择第二个操作数表达式。
- 如果第一个操作数的值为 false，则选择第三个操作数表达式。

然后，对所选操作数表达式求值，并将结果值转换为由下面陈述的规则确定的条件表达式的类型。

此转换可包括装箱或拆箱转换(§5.1.7、§5.1.8)。

对于条件表达式的该特定求值，不计算未选择的操作数表达式。

### 15.25.1 布尔条件表达式

布尔条件表达式是独立的表达式(§15.2)。

布尔条件表达式的类型按如下方式确定:

- 如果第二个和第三个操作数都是 Boolean 类型，则条件表达式的类型为 Boolean。
- 否则，条件表达式的类型为 boolean。

### 15.25.2 数值条件表达式

数值条件表达式是独立的表达式(§15.2)。

数值条件表达式的类型按如下方式确定:

- 如果第二个和第三个操作数具有相同的类型，则这就是条件表达式的类型。
- 如果第二个和第三个操作数中的一个是原生类型 T，而另一个操作数的类型是将装箱转换(§5.1.7)应用到 T 的结果，则条件表达式的类型是 T。
- 如果其中一个操作数的类型为 byte 或 Byte，而另一个操作数的类型为 short 或 Short，则条件表达式的类型为 short。
- 如果其中一个操作数是类型 T，其中 T 是 byte, short, 或 char，而另一个操作数是 int 类型的常量表达式(§15.29)，其值可用类型 T 表示，则条件表达式的类型是 T。
- 如果其中一个操作数是类型 T，其中 T 是 Byte, Short, 或 Character，而另一个操作数是 int 类型的常量表达式，其值可在类型 U 中表示，该类型 U 是对 T 应用拆箱转换的结果，则条件表达式的类型是 U。
- 否则，对第二个和第三个操作数应用一般数值提升(§5.6)，条件表达式的类型是第二个和第三个操作数的提升类型。

  请注意，数值提升可能包括拆箱转换(§5.1.8)。

### 15.25.3 引用条件表达式

如果引用条件表达式出现在赋值上下文或调用上下文，它就是多元表达式(§5.2. §5.3)。否则，它就是一个独立表达式。

多元引用条件表达式出现在目标类型为 T 的特定类型的上下文中，它的第二个和第三个操作数表达式同样出现在目标类型为 T 的相同类型的上下文中。

如果多元引用条件表达式的第二个和第三个操作数表达式与 T 兼容，则它与目标类型 T 兼容。

多元引用条件表达式的类型与其目标类型相同。

独立引用条件表达式的类型确定如下:

- 如果第二个和第三个操作数具有相同的类型(可能是 null 类型)，那么这就是条件表达式的类型。
- 如果第二个和第三个操作数中的一个的类型是 null 类型，而另一个操作数的类型是引用类型，那么条件表达式的类型就是该引用类型。
- 否则，第二个和第三个操作数分别为 $S_1$ 和 $S_2$ 类型。设 $T_1$ 是对 $S_1$ 应用装箱转换产生的类型，设 $T_2$ 是对 $S_2$ 应用装箱转换产生的类型。条件表达式的类型是对 lub($T_1$, $T_2$)应用捕获转换(§5.1.10)的结果。

因为引用条件表达式可以是多元表达式，所以它们可以"向下传递"上下文到它们的操作数。这允许 lambda 表达式和方法引用表达式作为操作数出现:

```
return ... ? (x -> x) : (x -> -x);
```

它还允许使用额外的信息来改进泛型方法调用的类型检查。在 Java SE 8 之前，该赋值类型良好:

```
List<String> ls = Arrays.asList();
```

但这不是:

```
List<String> ls = ... ? Arrays.asList() : Arrays.asList("a","b");
```

上述规则允许两个赋值都被视为类型良好。

请注意，引用条件表达式不必包含多元表达式作为操作数才能成为多元表达式。它是一个多元表达式，只是由于它出现的上下文。例如，在下面的代码中，条件表达式是一个多元表达式，并且每个操作数都被视为位于针对 Class<? super Integer>的赋值上下文中:

```
Class<? super Integer> choose(boolean b,
                              Class<Integer> c1,
                              Class<Number> c2) { return b ? c1 : c2;
}
```

如果条件表达式不是多元表达式，则会出现编译时错误，因为它的类型将是 lub(Class<Integer>, Class<Number>) = Class<? extends Number>，这与 choose 的返回类型不兼容。

## 15.26 赋值操作符

共有 12 个赋值操作符；它们在语法上都是右结合的(它们从右到左分组)。因此，a=b=c 意味着 a=(b=c)，它将 c 的值赋给 b，然后将 b 的值赋给 a。

*AssignmentExpression:*
  *ConditionalExpression*
  *Assignment*

*Assignment:*
  *LeftHandSide AssignmentOperator Expression*

*LeftHandSide:*
*ExpressionName*
*FieldAccess*
*ArrayAccess*

*AssignmentOperator:*
*(one of)*
= *= /= %= += -= <<= >>= >>>= &= ^= |=

赋值运算符的第一个操作数的结果必须是变量，否则会发生编译时错误。

这个操作数可以是一个命名变量，比如局部变量或当前对象或类的字段，也可以是一个计算变量，可以是通过字段访问(§15.11)或数组访问(§15.10.3)得到的结果。

赋值表达式的类型是捕获转换后的变量的类型(§5.1.10)。

在运行时，赋值表达式的结果是赋值发生后变量的值。赋值表达式的结果本身不是变量。

被声明为 final 的变量不能被赋值(除非它确实是未赋值的(§16(明确赋值))，因为当对此类 final 变量的访问用作表达式时，其结果是一个值，而不是一个变量，因此它不能被用作赋值操作符的第一个操作数。

### 15.26.1 简单赋值操作符 =

如果右操作数的类型与变量类型的赋值不兼容(§5.2)，则会发生编译时错误。

否则，在运行时，将以以下三种方式之一计算表达式。

如果左操作数表达式是字段访问表达式 e.f(§15.11)，可能包含在一对或多对括号中，则:

- 首先，对表达式 e 求值。如果 e 的求值突然结束，赋值表达式也会因为同样的原因而突然结束。
- 接下来，计算右操作数。如果右手表达式的计算突然完成，赋值表达式也会出于同样的原因突然完成。
- 然后，如果 e.f 表示的字段不是静态的，并且上述 e 的求值结果为空，则抛出 NullPointerException。
- 否则，由 e.f 表示的变量被赋予如上计算的右操作数的值。

如果左操作数是数组访问表达式(§15.10.3)，可能包含在一对或多对括号中，则:

- 首先，计算左操作数数组访问表达式的数组引用子表达式。如果此计算突然完成，则赋值表达式也会因同样的原因而突然完成；不计算(左操作数的数组访问表达式)索引子表达式和右操作数，也不进行赋值。
- 否则，计算左操作数数组访问表达式的索引子表达式。如果此计算突然完成，则赋值表达式出于相同的原因而突然完成，并且不计算右操作数也不发生赋值。

- 否则，右操作数将被求值。如果这个求值突然结束，那么赋值表达式也会因为同样的原因突然结束，没有赋值发生。
- 否则，如果数组引用子表达式的值为 null，则不发生赋值，并抛出 NullPointerException。
- 否则，数组引用子表达式的值确实指向一个数组。如果索引子表达式的值小于零，或大于或等于数组的长度，则不发生赋值，并引发 ArrayIndexOutOfBoundsException 异常。
- 否则，索引子表达式的值将用于选择数组引用子表达式的值所引用的数组的组件。

  这个组件是一个变量;将其类型称为 SC。同样，让 TC 是赋值操作符左操作数的类型，这是在编译时确定的。那么就有两种可能:

  - 如果 TC 是原生类型，那么 SC 必然与 TC 相同。

    右操作数的值被转换为所选数组组件的类型，转换的结果被存储到数组组件中。

  - 如果 TC 是引用类型，那么 SC 可能与 TC 不同，而是扩展或实现 TC 的类型。

    设 RC 为运行时右侧操作数的值所引用的对象的类。

    Java 编译器可能能够在编译时证明数组组件将完全属于 TC 类型（例如，TC 可能是 final 的）。但是，如果 Java 编译器无法在编译时证明数组组件的类型为 TC，则必须在运行时执行检查，以确保类 RC 与数组组件的实际类型 SC 是赋值兼容的（§5.2）。

    该检查类似于收窄转换（§5.5，§15.16），但如果检查失败，将引发 ArrayStoreException 而不是 ClassCastException。

    如果类 RC 不可分配给类型 SC，则不会发生赋值，并引发 ArrayStoreException。

    否则，右操作数的引用值将存储到所选数组组件中。

否则，需要三个步骤:

- 首先，计算左操作数以产生一个变量。如果这个求值突然结束，赋值表达式也会因为同样的原因突然结束;右操作数不计算，也不进行赋值。
- 否则，右操作数将被求值。如果这个求值突然结束，那么赋值表达式也会因为同样的原因突然结束，没有赋值发生。
- 否则，将右操作数的值转换为左侧变量的类型，并将转换结果存储到变量中。

例子 15.26.1-1. 数组组件的简单赋值

```
class ArrayReferenceThrow  extends RuntimeException { }
class IndexThrow           extends RuntimeException { }
class RightHandSideThrow   extends RuntimeException { }

class IllustrateSimpleArrayAssignment {
    static Object[] objects = { new Object(), new Object() };
    static Thread[] threads = { new Thread(), new Thread() };
```

```
static Object[] arrayThrow() {
    throw new ArrayReferenceThrow(); }
static int indexThrow() {
    throw new IndexThrow();
}
static Thread rightThrow() {
    throw new RightHandSideThrow();
}
static String name(Object q) {
    String sq = q.getClass().getName();
    int k = sq.lastIndexOf('.');
    return (k < 0) ? sq : sq.substring(k+1);
}
static void testFour(Object[] x, int j, Object y) {
    String sx = x == null ? "null" : name(x[0]) + "s";
    String sy = name(y);
    System.out.println();
    try {
        System.out.print(sx + "[throw]=throw => ");
        x[indexThrow()] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[throw]=" + sy + " => "); x[indexThrow()] = y;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]=throw => ");
        x[j] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]=" + sy + " => ");
        x[j] = y;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); } }

public static void main(String[] args) {
    try {
        System.out.print("throw[throw]=throw => "); arrayThrow()[indexThrow()]
        = rightThrow(); System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[throw]=Thread => ");
        arrayThrow()[indexThrow()] = new Thread();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]=throw => ");
        arrayThrow()[1] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]=Thread => ");
        arrayThrow()[1] = new Thread();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
```

```
            testFour(null, 1, new         Thread());
            StringBuffer());              StringBuffer());
            testFour(null, 9, new         Thread());
            Thread());                    StringBuffer());
            testFour(objects, 1, new      Thread());
            StringBuffer());
            testFour(objects, 1, new Thread());
            testFour(objects, 9, new StringBuffer());
            testFour(objects, 9, new Thread());
            testFour(threads, 1, new StringBuffer());
            testFour(threads, 1, new Thread());
            testFour(threads, 9, new StringBuffer());
            testFour(threads, 9, new Thread());
        }
    }
```

此程序生成以下输出：

```
throw[throw]=throw => ArrayReferenceThrow
throw[throw]=Thread => ArrayReferenceThrow
throw[1]=throw => ArrayReferenceThrow
throw[1]=Thread => ArrayReferenceThrow

null[throw]=throw => IndexThrow
null[throw]=StringBuffer => IndexThrow
null[1]=throw => RightHandSideThrow
null[1]=StringBuffer => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=Thread => IndexThrow
null[9]=throw => RightHandSideThrow
null[9]=Thread => NullPointerException

Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=StringBuffer => Okay!

Objects[throw]=throw => IndexThrow
Objects[throw]=Thread => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=Thread => Okay!

Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=StringBuffer => ArrayIndexOutOfBoundsException

Objects[throw]=throw => IndexThrow
Objects[throw]=Thread => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=Thread => ArrayIndexOutOfBoundsException

Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=StringBuffer => ArrayStoreException

Threads[throw]=throw => IndexThrow
```

```
Threads[throw]=Thread => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=Thread => Okay!

Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=StringBuffer => ArrayIndexOutOfBoundsException

Threads[throw]=throw => IndexThrow
Threads[throw]=Thread => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=Thread => ArrayIndexOutOfBoundsException
```

最有趣的例子是倒数第十三个:

```
Threads[1]=StringBuffer => ArrayStoreException
```

这表示试图将对 StringBuffer 的引用存储到其组件为 Thread 类型的数组中会引发 ArrayStoreException。代码在编译时是类型正确的：赋值的左侧类型为 Object[]，右侧类型为 Object。在运行时，方法 testFour 的第一个实际参数是对“Thread 数组”实例的引用，第三个实际参数是对 StringBuffer 类的实例的引用。

### 15.26.2 复合赋值运算符

形式为 E1 op= E2 的复合赋值表达式等价于 E1 = (T) ((E1) op (E2)), 其中 T 是 E1 的类型, 唯一不同的是，E1 只计算一次。

例如，以下代码是正确的:

```
short x = 3;
x += 4.6;
```

并导致 x 的值为 7，因为它等同于:

```
short x = 3;
x = (short)(x + 4.6);
```

在运行时，以以下两种方式之一计算表达式。

如果左操作数表达式不是数组访问表达式，则:

- 首先，计算左操作数以产生一个变量。如果此计算突然完成，则赋值表达式出于相同的原因而突然完成；不计算右操作数，也不进行赋值。
- 否则，保存左操作数的值，然后计算右操作数。如果此计算突然完成，则赋值表达式出于相同的原因而突然完成，并且不发生赋值。
- 否则，使用保存的左侧变量的值和右操作数的值来执行复合赋值运算符指示的二元运算。如果此操作突然完成，则赋值表达式出于相同的原因而突然完成，并且不发生赋值。
- 否则，将二元运算的结果转换为左侧变量的类型，并将转换结果存储到变量中。

如果左操作数表达式是数组访问表达式(§15.10.3)，则:

- 首先，计算左操作数数组访问表达式的数组引用子表达式。如果这个求值突然结束，赋值表达式也会因为同样的原因突然结束;索引子表达式(左操作数数组访问表达式的)和右操作数不计算，也没有赋值。
- 否则，计算左操作数数组访问表达式的索引子表达式。如果这个求值突然结束，赋值表达式也会因为同样的原因突然结束，右操作数不会被求值，也不会发生赋值。
- 否则，如果数组引用子表达式的值为 null，则不发生赋值，并抛出 NullPointerException。
- 否则，数组引用子表达式的值确实指向一个数组。如果索引子表达式的值小于零，或大于或等于数组的长度，则不发生赋值，并引发 ArrayIndexOutOfBoundsException 异常。
- 否则，索引子表达式的值将用于选择数组引用子表达式的值所引用的数组的组件。保存该组件的值，然后计算右操作数。如果这个求值突然结束，那么赋值表达式也会因为同样的原因突然结束，没有赋值发生。

  对于简单的赋值操作符，右操作数的求值发生在检查数组引用子表达式和索引子表达式之前，但是对于复合赋值操作符，右操作数的求值发生在这些检查之后。

• 否则，考虑在上一步中选择的数组组件，它的值已经保存。这个组件是一个变量;称之为类型 S。同样，让 T 是赋值操作符左操作数的类型，这是在编译时确定的。

- 如果 T 是原生类型，那么 S 必然与 T 相同。

  数组组件的保存值和右操作数的值用于执行复合赋值操作符所指示的二元运算。

  如果该操作突然完成（唯一的可能性是整数除以零-见§15.17.2），则赋值表达式出于相同原因突然完成，且不发生赋值。

  否则，二元运算的结果将转换为所选数组组件的类型，并且转换的结果将存储到数组组件中。

- 如果 T 是引用类型，则它必须是 String。因为类 String 是一个 final 类，所以 S 也必须是 String。

  因此，简单赋值运算符有时需要的运行时检查对于复合赋值运算符永远不需要。

  数组组件的保存值和右操作数的值用于执行复合赋值运算符（必须为+=）指示的二元运算（字符串连接）。如果此操作突然完成，则赋值表达式出于相同原因突然完成，并且不会发生赋值。

  否则，二元运算的字符串结果将存储到数组组件中。

例子 15.26.2-1. 数组组件的复合赋值

```
class  ArrayReferenceThrowextends  RuntimeException  { }
class  IndexThrow         extends  RuntimeException  { }
class  RightHandsideThrow extends  RuntimeException  { }

class IllustrateCompoundArrayAssignment {
    static string[] strings = { "simon", "Garfunkel" };
```

```
static double[] doubles = { Math.E, Math.PI };

static string[] stringsThrow() {
   throw new ArrayReferenceThrow();
}
static double[] doublesThrow() {
   throw new ArrayReferenceThrow();
}
static int indexThrow() {
   throw new IndexThrow();
}
static String stringThrow() {
   throw new RightHandSideThrow();
}
static double doubleThrow() {
   throw new RightHandSideThrow();
}
static String name(Object q) {
   String sq = q.getClass().getName();
   int k = sq.lastIndexOf('.');
   return (k < 0) ? sq : sq.substring(k+1);
}

static void testEight(String[] x, double[] z, int j) {
    String sx = (x == null) ? "null" : "Strings";
    String sz = (z == null) ? "null" : "doubles";
    System.out.println();
    try {
        System.out.print(sx + "[throw]+=throw => ");
        x[indexThrow()] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[throw]+=throw => ");
        z[indexThrow()] += doubleThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[throw]+=\"heh\" => ");
        x[indexThrow()] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[throw]+=12345 => ");
        z[indexThrow()] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]+=throw => ");
        x[j] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[" + j + "]+=throw => ");
        z[j] += doubleThrow();
        System.out.println("Okay!");
```

```
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]+=\"heh\" => ");
        x[j] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[" + j + "]+=12345 => ");
        z[j] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
}

public static void main(String[] args) {
    try {
        System.out.print("throw[throw]+=throw => ");
        stringsThrow()[indexThrow()] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[throw]+=throw => ");
        doublesThrow()[indexThrow()] += doubleThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[throw]+=\"heh\" => ");
        stringsThrow()[indexThrow()] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[throw]+=12345 => ");
        doublesThrow()[indexThrow()] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]+=throw => ");
        stringsThrow()[1] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]+=throw => ");
        doublesThrow()[1] += doubleThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]+=\"heh\" => ");
        stringsThrow()[1] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]+=12345 => ");
        doublesThrow()[1] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); } testEight(null,
    null, 1);
    testEight(null, null, 9);
    testEight(strings, doubles, 1);
    testEight(strings, doubles, 9);
}
```

```
}
```

该程序产生以下输出：

```
throw[throw]+=throw => ArrayReferenceThrow
throw[throw]+=throw => ArrayReferenceThrow
throw[throw]+="heh" => ArrayReferenceThrow
throw[throw]+=12345 => ArrayReferenceThrow
throw[1]+=throw => ArrayReferenceThrow
throw[1]+=throw => ArrayReferenceThrow
throw[1]+="heh" => ArrayReferenceThrow
throw[1]+=12345 => ArrayReferenceThrow

null[throw]+=throw => IndexThrow
null[throw]+=throw => IndexThrow
null[throw]+="heh" => IndexThrow
null[throw]+=12345 => IndexThrow
null[1]+=throw => NullPointerException
null[1]+=throw => NullPointerException
null[1]+="heh" => NullPointerException
null[1]+=12345 => NullPointerException

null[throw]+=throw => IndexThrow
null[throw]+=throw => IndexThrow
null[throw]+="heh" => IndexThrow
null[throw]+=12345 => IndexThrow
null[9]+=throw => NullPointerException
null[9]+=throw => NullPointerException
null[9]+="heh" => NullPointerException
null[9]+=12345 => NullPointerException

Strings[throw]+=throw => IndexThrow
doubles[throw]+=throw => IndexThrow
Strings[throw]+="heh" => IndexThrow
doubles[throw]+=12345 => IndexThrow
Strings[1]+=throw => RightHandSideThrow
doubles[1]+=throw => RightHandSideThrow
Strings[1]+="heh" => Okay!
doubles[1]+=12345 => Okay!

Strings[throw]+=throw => IndexThrow
doubles[throw]+=throw => IndexThrow
Strings[throw]+="heh" => IndexThrow
doubles[throw]+=12345 => IndexThrow
Strings[9]+=throw => ArrayIndexOutOfBoundsException
doubles[9]+=throw => ArrayIndexOutOfBoundsException
Strings[9]+="heh" => ArrayIndexOutOfBoundsException
doubles[9]+=12345 => ArrayIndexOutOfBoundsException
```

最有趣的案例是倒数第 11 和第 12 个：

```
Strings[1]+=throw => RightHandSideThrow
doubles[1]+=throw => RightHandSideThrow
```

它们是抛出异常的右侧实际抛出异常的情况；而且，它们是批次中唯一这样的情况。这表明右操作数的求值确实发生在检查空数组引用值和越界索引值之后。

例子 15.26.2-2. 计算复合赋值时，左边的值先保存，然后再计算右边的值

```
class Test {
    public static void main(String[] args) {
        int k = 1;
        int[] a = { 1 };
        k += (k = 4) * (k + 2);
        a[0] += (a[0] = 4) * (a[0] + 2);
        System.out.println("k==" + k + " and a[0]==" + a[0]);
    }
}
```

程序产生输出：

```
k==25 and a[0]==25
```

复合赋值运算符+=在计算其右操作数(k=4)*(k+2)之前保存 k 的值 1。此右操作数的求值将 4 赋给 k，计算 k+2 的值 6，然后将 4 乘以 6 得到 24。将其与保存的值 1 相加得到 25，然后由+=操作符将其存储到 k 中。同样的分析也适用于使用 a[0]的情况。

简而言之，这些语句：

```
k += (k = 4) * (k + 2);
a[0] += (a[0] = 4) * (a[0] + 2);
```

行为方式与以下语句完全相同：

```
k = k + (k = 4) * (k + 2);
a[0] = a[0] + (a[0] = 4) * (a[0] + 2);
```



## 15.27 Lambda 表达式

lambda 表达式类似于方法：它提供形式参数的列表和以这些参数表示的主体--表达式或块。

*LambdaExpression:*
*LambdaParameters* -> *LambdaBody*

lambda 表达式始终是多元表达式(§15.2)。

如果 lambda 表达式出现在程序中的某个位置，而不是在赋值上下文(§5.2)、调用上下文(§5.3)或强制转换上下文(§5.5)中，则是编译时错误。

对 lambda 表达式的求值会产生一个函数式接口的实例(§9.8)。 lambda 表达式的求值不会导致表达式体的执行；相反，这可能会在稍后调用函数式接口的适当方法时发生。

以下是 lambda 表达式的一些示例：

```
()->{}                  //No parameters; result is void
()->42                  //No parameters,expression body
()->null                //No parameters,expression body
()->{return 42;}        //No parameters,block body with return
```

```
()->{System.gc();}    //No parameters,void block body

() -> {                 // Complex block body with returns
  if (true) return 12;
  else {
    int result = 15;
    for (int i = 1; i < 10; i++) result *= i;
    return result;
  }
}

(int  x) -> x+1             //  Single  declared-type parameter
(int  x) -> {return x+1; }  //  Single  declared-type parameter
(x) -> x+1                  //  Single  inferred-type parameter
x ->  x+1                   //   Parentheses optional for
                             // single inferred-type parameter

(String s) -> s.length()      //  Singledeclared-type  parameter
(Thread t) -> { t.start(); }  //  Singledeclared-type  parameter
s -> s.length()               //  Singleinferred-type  parameter
t -> { t.start(); }           //  Singleinferred-type  parameter

(int x, int y) -> x+y // Multiple declared-type parameters
(x, y) -> x+y           // Multiple inferred-type parameters
(x, int y) -> x+y   // Illegal: can't mix inferred and declared types
(x, final y) -> x+y // Illegal: no modifiers with inferred types
```

这种语法的优点是最小化简单 lambda 表达式周围的括号噪声，当 lambda 表达式是方法的参数时，或当主体是另一个 lambda 表达式时，这尤其有益。它还清楚地区分了表达式和语句形式，从而避免了歧义或过度依赖";"标记。当需要一些额外的括号来在视觉上区分整个 lambda 表达式或其主体表达式时，括号自然得到支持(就像其他操作符优先级不明确的情况一样)。

该语法存在一些解析挑战。Java 编程语言总是需要任意的提前处理来区分类型和'('标记后的表达式:后面可能是强制类型转换或带括号的表达式。当泛型在类型中重用二元操作符'<'和'>'时，情况变得更糟。lambda 表达式引入了一种新的可能性:'('后面的标记可以描述类型、表达式或 lambda 参数列表。一些标记立即指示一个参数列表(注解, final);在其他情况下，某些模式必须解释为参数列表(一行中有两个名称，'，'没有嵌套在'<'和'>'中);有时，只有在')'之后遇到'->'时才能做出决定。考虑如何有效地解析这种情况的最简单方法是使用状态机:每个状态表示可能的解释(类型、表达式或参数)的一个子集，当状态机转换到一个状态(其中的集合是一个单例)时，解析器知道是哪种情况。然而，这并不能很好地映射到固定的提前查找语法。

没有特殊的空值形式: 带有零个参数的 lambda 表达式表示为()->…。明显的特殊情况语法 -> … 不起作用，因为它在参数列表和类型转换之间引入了歧义: (x) -> …。

lambda 表达式不能声明类型参数。虽然这样做在语义上是有意义的，但是自然的语法(在参数列表前面加上类型参数列表)引入了混乱的二义性。例如,考虑:

```
foo( (x) < y , z > (w) -> v )
```

这可以是 foo 的一个参数调用(泛型 lambda 转换为类型 x)，也可以是 foo 的两个参数调用，两个参数都是比较的结果，第二个参数将 z 与 lambda 表达式进行比较。(严格地说，lambda 表达式作为关系操作符> 的操作数是没有意义的，但这是构建语法的一个脆弱的假设。)

有一个涉及强制转换的歧义解决的先例，它本质上禁止在非原生强制转换(§15.15)之后使用-和+，但将该方法扩展到泛型 lambdas 将涉及对语法的侵入性更改。

### 15.27.1 Lambda 参数

lambda 表达式的形式参数(如果有的话)由逗号分隔的参数说明符的圆括号列表或逗号分隔的标识符的圆括号列表指定。在参数说明符列表中，每个参数说明符由可选修饰符组成，然后是类型(或 var)，然后是指定参数名称的标识符。在标识符列表中，每个标识符指定参数的名称。

如果 lambda 表达式没有形式参数，则在->和 lambda 体之前出现一对空圆括号。

如果 lambda 表达式只有一个形式参数，并且参数由标识符而不是参数说明符指定，则可以省略标识符周围的圆括号。

*LambdaParameters:*
（ *[LambdaParameterList]* ）
*Identifier*

*LambdaParameterList:*
*LambdaParameter {，LambdaParameter}*
*Identifier {，Identifier}*

*LambdaParameter:*
*{VariableModifier} LambdaParameterType VariableDeclaratorId*
*VariableArityParameter*

*LambdaParameterType:*
*UnannType*
`var`



为方便起见，以下是§8.4.1、§8.3 和§4.3 的产品:

*VariableArityParameter:*
*{VariableModifier} UnannType {Annotation}* `...` *Identifier*

*VariableModifier:*
*Annotation*
`final`

*VariableDeclaratorId:*
*Identifier [Dims]*

*Dims:*
*{Annotation}* `[ ]` *{{Annotation}* `[ ]`*}*

只有在由参数说明符指定的情况下，lambda 表达式的形参才能被声明为 final 或加注解。如果形参由标识符指定，则形参不是 final，并且没有注解。

lambda 表达式的形式参数可以是可变参数，由参数说明符中类型后面的省略号表示。lambda 表达式最多允许一个可变参数。如果可变参数出现在参数说明符列表中除最后一个位置外的任何位置，则会发生编译时错误。

lambda 表达式的每个形式参数都具有推断类型或声明类型：

• 如果形式参数由使用 var 的参数说明符指定，或者由标识符而不是参数说明符指定，则形式参数具有推断类型。该类型是从 lambda 表达式所针对的函数式接口类型中推断出来的（§15.27.3）。

• 如果形式参数由不使用 var 的参数说明符指定，则形式参数具有声明的类型。声明的类型确定如下：

- 如果形式参数不是可变参数，则如果 UnannType 和 VariableDeclaratorId 中未出现括号对，则声明的类型由 UnannType 表示，否则由§10.2 指定。
- 如果形式参数是可变参数，则声明的类型是§10.2 规定的数组类型。

  以下 lambda 参数列表之间没有区别：

  ```
  (int... x) -> BODY
  (int[] x) -> BODY
  ```

  无论函数式接口的抽象方法是固定参数还是可变参数，都可以使用 (这与方法重写的规则一致。) 由于 lambda 表达式从未被直接调用，在函数式接口使用 int[]的形式参数中使用 int…，可能不会对周围的程序产生影响。在 lambda 体中，可变参数被视为数组类型的参数。

所有形式参数都声明了类型的 lambda 表达式称为显式类型化。所有形式参数都具有推断类型的 lambda 表达式称为隐式类型化。没有形式参数的 lambda 表达式是显式类型化。

如果 lambda 表达式是隐式类型化，则其 lambda 主体将根据其出现的上下文进行解释。具体而言，主体中表达式的类型、主体抛出的检查异常以及主体中代码的类型正确性都取决于为形式参数推断的类型。这意味着形式参数类型的推断必须在“尝试”对 lambda 主体进行类型检查之前进行。

如果 lambda 表达式声明了具有声明类型的形式参数和具有推断类型的形式参数，则这是编译时错误。

该规则防止在形式参数中混合使用推断类型和声明类型，例如(x, int y) -> BODY 或(var x, int y) -> BODY。请注意，如果所有形式参数都有推断类型，语法将防止混合使用标识符和 var 参数说明符，例如(x, var y) -> BODY 或(var x, y) -> BODY。

关于形参声明的注解修饰符的规则见§9.7.4 和§9.7.5。

如果 final 不止一次作为形式参数声明的修饰符出现，则会出现编译时错误。

如果形式参数的 LambdaParameterType 为 var，而同一形式参数的 VariableDeclaratorId 具有一个或多个括号对，则会出现编译时错误。

形式参数的作用域和遮蔽在§6.3 和§6.4 中有规定。

对嵌套类或接口或嵌套 lambda 表达式的形参的引用是受限制的，见§6.5.6.1。

lambda 表达式声明两个具有相同名称的形参是编译时错误。(也就是说，它们的声明提到了相同的 Identifier。)

在 Java SE 8 中，使用_作为 lambda 参数的名称是被禁止的，并且不鼓励使用它作为其他变量类型的名称(§4.12.3)。在 Java SE 9 中，_是一个关键字(§3.9)，所以它不能在任何上下文中用作变量名。

如果在 lambda 表达式的主体内指定了声明为 final 的形式参数，则这是编译时错误。

当调用 lambda 表达式时（通过方法调用表达式（§15.12）），实际参数表达式的值在执行 lambda 主体之前初始化新创建的参数变量，每个参数变量都是声明的或推断的类型。出现在 LambdaParameter 中或直接出现在 LambdaParameterList 或 LambdaParameters 中的标识符可以用作 lambda 主体中的简单名称，以引用形式参数。

### 15.27.2 Lambda 体

lambda 体要么是单个表达式，要么是一个块(§14.2)。与方法体一样，lambda 体描述当调用发生时将执行的代码。

*LambdaBody:*
*Expression*
*Block*

与出现在匿名类声明中的代码不同，名称的含义以及出现在 lambda 体中的 this 和 super 关键字，以及引用声明的可访问性，与周围上下文中相同(只是 lambda 参数引入了新名称)。

在 lambda 表达式体中 this（包括显式和隐式）的透明性-也就是说，将其与周围的上下文一样对待-为实现提供了更大的灵活性，并防止主体中的非限定名称的含义依赖于重载解析。

实际上，lambda 表达式需要谈论自身(递归地调用自身或调用它的其他方法)是不常见的，而更常见的情况是希望使用名称来引用封闭类中原本会被遮蔽的对象(this，toString())。如果 lambda 表达式需要引用自身(就像通过 this 一样)，则应该使用方法引用或匿名内部类。

如果块中的每个 return 语句都具有 return;的形式，则块 lambda 体是 void 兼容的。

如果块 lambda 体不能正常完成(§14.22)，并且块中的每个返回语句都具有 return Expression 的形式，则块 lambda 体是值兼容的。

如果块 lambda 体既不是 void 兼容也不是值兼容，则为编译时错误。

在值兼容的块 lambda 体中，结果表达式是可能产生调用值的任何表达式。具体来说，对于每个主体中包含的形式为 return *Expression* 的语句来说， *Expression* 是一个结果表达式。

下面的 lambda 体是 void 兼容的：

```
() -> {}
() -> { System.out.println("done"); }
```

这些是值兼容的：

```
() -> { return "done"; }
() -> { if (...) return 1; else return 0; }
```

这些都是：

```
() -> { throw new RuntimeException(); }
() -> { while (true); }
```

这些都不是：

```
() -> { if (...) return "done"; System.out.println("done"); }
```

对 void/值兼容的处理和主体中名称的含义的共同作用是最小化给定上下文中对特定目标类型的依赖，这对实现和程序员理解都很有用。虽然在重载解析期间可以根据目标类型为表达式分配不同的类型，但非限定名称的含义和 lambda 主体的基本结构不变。

请注意，void/value 兼容定义不是严格的结构属性："可以正常完成"取决于常量表达式的值，这些表达式可能包括引用常量变量的名称。

按照§6.5.6.1 的规定，在 lambda 表达式中使用但未声明的任何局部变量、形式参数或异常参数必须是 final 的或实际 final 的（§4.12.4）。

在 lambda 主体中使用但未声明的任何局部变量必须在 lambda 主体之前明确赋值（§16（明确赋值）），否则会发生编译时错误。

关于变量使用的类似规则适用于内部类的主体（§8.1.3）。对实际 final 变量的限制禁止访问动态变化的局部变量，这些变量的捕获可能会带来并发问题。与 final 限制相比，它减轻了程序员的文书负担。

对实际 final 变量的限制包括标准循环变量，但不包括增强的 for 循环变量，这些变量在循环的每次迭代中被视为不同的变量(§14.14.2)。

以下 lambda 主体演示了实际 final 变量的使用。

```
void m1(int x) {
    int y = 1;
    foo(() -> x+y);
    // Legal: x and y are both effectively final.
}

void m2(int x) {
    int y;
    y = 1;
    foo(() -> x+y);
    // Legal: x and y are both effectively final.
}

void m3(int x) {
    int y;
    if (...) y = 1;
    foo(() -> x+y);
    // Illegal: y is effectively final, but not definitely assigned.
}

void m4(int x) {
    int y;
    if (...) y = 1;
    else y = 2;
    foo(() -> x+y);
    // Legal: x and y are both effectively final.
}
```

```
void m5(int x) {
    int y;
    if (...) y = 1;
    y = 2;
    foo(() -> x+y);
    // Illegal: y is not effectively final.
}

void m6(int x) {
    foo(() -> x+1);
    x++;
    // Illegal: x is not effectively final.
}

void m7(int x) {
    foo(() -> x=1); // Illegal: x is not effectively final.
}

void m8() {
    int y;
    foo(() -> y=1);
    // Illegal: y is not definitely assigned before the lambda.
}

void m9(String[] arr) {
    for (String s : arr) {
       foo(() -> s);
       // Legal: s is effectively final
       // (it is a new variable on each iteration)
    }
}

void m10(String[] arr) {
   for (int i = 0; i < arr.length; i++) {
      foo(() -> arr[i]);
      // Illegal: i is not effectively final
      // (it is not final, and is incremented)
   }
}
```



### 15.27.3 Lambda 表达式的类型

如果 T 是函数式接口类型(§9.8)，并且该表达式与从 T 派生的地面目标类型的函数类型一致，则 lambda 表达式在具有目标类型 T 的赋值上下文、调用上下文或强制转换上下文中兼容。

地面目标类型由 T 派生，如下所示：

- 如果 T 是通配符参数化的函数式接口类型，并且 lambda 表达式是显式类型化的，则将按照§18.5.3 中所述推断地面目标类型。

- 如果 T 是通配符参数化的函数式接口类型，并且 lambda 表达式是隐式类型化的，则地面目标类型是 T 的非通配符参数化(§9.9)。

- 否则，地面目标类型为 T。

如果满足以下所有条件，则 lambda 表达式与函数类型一致：

- 该函数类型没有类型参数。
- lambda 参数的数量与函数类型的参数类型的数量相同。
- 如果 lambda 表达式是显式类型化的，则其形式参数类型与函数类型的参数类型相同。
- 如果假定 lambda 参数的类型与函数类型的参数类型相同，则：
  - 如果函数类型的结果是 void，lambda 体要么是一个语句表达式(§14.8)，要么是一个 void 兼容的块。
  - 如果函数类型的结果是(非 void)类型 R，则 (i) lambda 主体是在赋值上下文中与 R 兼容的表达式，或者 (ii) lambda 主体是一个值兼容的块，并且每个结果表达式(§15.27.2)在赋值上下文中与 R 兼容。

如果 lambda 表达式与目标类型 T 兼容，则该表达式的类型 U 是从 T 派生的地面目标类型。

如果 U 或 U 的函数类型提到的任何类或接口不能从出现 lambda 表达式的类或接口访问(§6.6)，则是编译时错误。

对于 U 的每个非静态成员方法 m，如果 U 的函数类型具有 m 的签名的子签名，则其方法类型为 U 的函数类型的概念方法被视为重写 m，并且可能发生§8.4.8.3 中规定的任何编译时错误或未检查的警告。

如§11.2.3 所述，可在 lambda 表达式主体中引发的检查异常可能会导致编译时错误。

显式类型的 lambda 的参数类型需要与函数类型的参数类型完全匹配。虽然可以更加灵活，例如允许装箱或逆变，但这种通用性似乎是不必要的，并且与类声明中重写的工作方式不一致。在编写 lambda 表达式时，程序员应该确切地知道目标函数类型，因此程序员应该准确地知道必须重写什么签名。(方法引用不是这种情况，因此在使用它们时允许更大的灵活性。) 此外，参数类型的更大灵活性将增加类型推断和重载解析的复杂性。

请注意，虽然在严格的调用上下文中不允许装箱，但始终允许装箱 lambda 结果表达式-也就是说，结果表达式出现在赋值上下文中，而与包围 lambda 表达式的上下文无关。然而，如果显式类型的 lambda 表达式是重载方法的参数，则最具体的检查(§15.12.2.5)会首选避免装箱或拆箱 lambda 结果的方法签名。

如果 lambda 的主体是一个语句表达式(即允许单独作为语句存在的表达式)，则它与产生 void 的函数类型兼容;任何结果都被简单地丢弃。例如，以下两种情况都是合法的:

```
// Predicate has a boolean result
java.util.function.Predicate<String> p = s -> list.add(s);
// Consumer has a void result
java.util.function.Consumer<String> c = s -> list.add(s);
```

一般来说, 形式为 () -> expr 的 lambda 表达式，其中 expr 是语句表达式, 被解读为 () -> { return expr; } 或 () -> { expr; }, 取决于目标类型。

### 15.27.4 lambda 表达式的运行时求值

在运行时，lambda 表达式的计算类似于类实例创建表达式的计算，只要正常完成生成对

对象的引用。lambda 表达式的计算不同于 lambda 主体的执行。

分配并初始化具有以下属性的类的新实例，或者引用具有以下属性类的现有实例。如果要创建一个新实例，但没有足够的空间来分配对象，则 lambda 表达式的计算将通过抛出 OutOfMemoryError 突然完成。

这意味着对 lambda 表达式求值(或对 lambda 表达式进行序列化和反序列化)的结果的标识是不可预测的，因此标识敏感的操作(例如引用相等(§15.21.3)、对象锁定(§14.19)和 System.identityHashCode 方法)可能在 Java 编程语言的不同实现中产生不同的结果，甚至在同一实现中的不同 lambda 表达式计算上产生不同的结果。

lambda 表达式的值是对具有以下属性的类实例的引用:

· 该类实现目标函数接口类型，如果目标类型是交集类型，则实现交集中提到的所有其他接口类型。

· 其中 lambda 表达式有类型 U，对于 U 中的每个非静态成员方法 m:

如果 U 的函数类型有 m 的签名的子签名，那么该类声明一个重写 m 的方法。该方法的主体具有对 lambda 主体求值的效果(如果它是一个表达式)，或者对 lambda 主体执行的效果(如果它是一个块);如果预期有结果，则从方法返回。

如果被重写的方法类型的擦除在其签名上与 U 的函数类型的擦除不同，那么在求值或执行 lambda 体之前，方法体检查每个参数值是否是 U 的函数类型中相应参数类型擦除的子类或子接口的实例;如果不是，则抛出 ClassCastException。

· 这个类不重写目标函数式接口类型或上面提到的其他接口类型的其他方法，尽管它可能重写 Object 类的方法。

这些规则旨在为 Java 编程语言的实现提供灵活性，因为:

- 不需要在每次求值时都分配新对象。
- 由不同 lambda 表达式产生的对象不需要属于不同的类(例如，如果主体相同)。
- 求值产生的每个对象不需要属于同一个类(例如，捕获的局部变量可能是内联的)。
- 如果“现有实例”可用，则不必在以前的 lambda 求值时创建它（例如，它可能在封闭类的初始化过程中被分配）。

如果目标函数接口类型是 java.io.Serializable 的子类型，结果对象将自动成为可序列化类的实例。使从 lambda 表达式派生的对象可序列化可能会产生额外的运行时开销和安全影响，因此 lambda 派生的对象“默认情况下”不需要可序列化。

## 15.28 switch 表达式

switch 表达式根据表达式的值将控制权转移到多个语句或表达式中的一个;必须处理该表达式的所有可能值，并且所有语句和表达式必须为 switch 表达式的结果产生一个值。

*SwitchExpression:*
`switch` `(` *Expression* `)` *SwitchBlock*

Expression 称为选择器表达式。选择器表达式的类型必须是 char, byte, short, int, Character, Byte, Short, Integer, String,或枚举类型 (§8.9), 否则发生编译时错误。

switch 表达式和 switch 语句(§14.11)的主体都称为 switch 块。适用于所有 switch 块的一般规则，无论它们出现在 switch 表达式中还是 switch 语句中，见§14.11.1。为了方便起见，这里列出了§14.11.1 的以下产品:

*SwitchBlock:*
{ *SwitchRule {SwitchRule}* }
{ *{SwitchBlockStatementGroup} {SwitchLabel* **:}** }

*SwitchRule:*
*SwitchLabel* **->** *Expression* **;**
*SwitchLabel* **->** *Block*
*SwitchLabel* **->** *ThrowStatement*

*SwitchBlockStatementGroup:*
*SwitchLabel* **:** *{SwitchLabel* **:}** *BlockStatements*

*SwitchLabel:*
`case` *CaseConstant {*`,` *CaseConstant}*
`default`

*CaseConstant:*
*ConditionalExpression*



### 15.28.1 switch 表达式的 switch 块

除了 switch 块的一般规则(§14.11.1)之外，switch 表达式中还有更多的 switch 块规则。 也就是说，对于 switch 表达式的 switch 块，以下所有条件必须为真，否则会发生编译时错误:

- 如果选择器表达式的类型不是枚举类型，那么只有一个 default 标签与 switch 块相关联。
- 如果选择器表达式的类型是枚举类型，那么(i)与 switch 块关联的 case 常量集包括枚举类型的每个枚举常量，(ii)最多一个 default 标签与 switch 块关联。

  当 case 标签覆盖所有枚举常量时，default 标签是允许的，但不是必需的。

- 如果 switch 块由 switch 规则组成，那么任何 switch 规则块都不能正常完成(§14.22)。
- 如果 switch 块由 switch 标签语句组组成，则 switch 块中的最后一条语句不能正常完成，并且在最后一条 switch 标签语句组之后，switch 块中没有任何 switch 标签。

  与 switch 语句不同，switch 表达式不能有空的 switch 块。 此外，switch 表达式与 switch 语句的不同之处在于，switch 块中哪些表达式可能出现在箭头(->)的右边，也就是说，哪些表达式可以用作 switch 规则表达式。在 switch 表达式中，任何表达式都可以用作 switch 规则表达式，但在 switch 语句中，只能使用语句表达式(§14.11.1)。

switch 表达式的结果表达式如下所示:

- 如果 switch 块由 switch 规则组成，则依次考虑每个 switch 规则:
  - 如果 switch 规则的形式为 ... -> Expression ； 那么 Expression 是 switch 表达式的

结果表达式。

- 如果 switch 规则的形式为 ... -> Block，则直接包含在 Block 中的 yield 语句中且其 yield 目标是给定 switch 表达式的每个表达式都是 switch 表达式的结果表达式。

• 如果 switch 块由标记为 switch 的语句组组成，则 switch 块中的 yield 语句中直接包含的每个表达式(其 yield 目标是给定的 switch 表达式)都是 switch 表达式的结果表达式。

如果 switch 表达式没有结果表达式，则为编译时错误。

如果 switch 表达式出现在赋值上下文或调用上下文中，则它是多元表达式(§5.2、§5.3)。否则，它是一个独立的表达式。

在多元 switch 表达式出现在具有目标类型 T 的特定类型的上下文中的情况下，其结果表达式类似地出现在具有目标类型 T 的相同类型的上下文中。

如果其每个结果表达式都与目标类型 T 兼容，则多元 switch 表达式与目标类型 T 兼容。

多元 switch 表达式的类型与其目标类型相同。

独立 switch 表达式的类型按如下方式确定：

• 如果结果表达式都具有相同的类型(可能是空类型(§4.1))，那么这就是 switch 表达式的类型。

• 否则，如果每个结果表达式的类型为 boolean 或 Boolean，则对 Boolean 类型的每个结果表达式应用拆箱转换(§5.1.8)，并且 switch 表达式具有 boolean 类型。

• 否则，如果每个结果表达式的类型可转换为数值类型(§5.1.8)，则 switch 表达式的类型是应用于结果表达式的一般数值提升(§5.6)的结果。

• 否则，将装箱转换(§5.1.7)应用于每个具有原生类型的结果表达式之后，switch 表达式的类型是将捕获转换(§5.1.10)应用于结果表达式类型的最小上界(§4.10.4)的结果。

### 15.28.2 switch 表达式的运行时计算

通过首先计算选择器表达式来计算 switch 表达式。然后：

- 如果选择器表达式的计算突然完成，那么 switch 表达式也会出于同样的原因突然完成。
- 否则，如果选择器表达式的求值结果为空，则会引发 NullPointerException 异常，并因此原因突然结束整个 switch 表达式。
- 否则，如果选择器表达式的求值结果为类型 Character, Byte, Short, 或 Integer, 则应对其进行拆箱转换(§5.1.8)。如果此转换突然完成，则出于同样的原因，整个 switch 表达式也会突然完成。

如果选择器表达式的求值正常完成且结果不为 null，并且后续的拆箱转换(如果有的话)正常完成，那么 switch 表达式的求值将通过判断与 switch 块关联的 switch 标签是否匹配选择器

表达式的值(§14.11.1)继续进行。然后:

- 如果没有匹配的 switch 标签，则抛出 IncompatibleClassChangeError，并因此突然完成整个 switch 表达式。
- 如果 switch 标签匹配，则应用以下其中之一:
  - 如果是 switch 规则表达式的 switch 标签，则对表达式进行计算。如果求值的结果是一个值，那么 switch 表达式将以相同的值正常完成。
  - 如果是 switch 规则块的 switch 标签，则执行该块。如果该块正常完成，则 switch 表达式正常完成。
  - 如果是 switch 规则抛出语句的 switch 标签，则执行抛出语句。
  - 否则，switch 块中匹配 switch 标签之后的所有语句将依次执行。如果这些语句正常完成，则 switch 表达式正常完成。

如果在 switch 块中任何语句或表达式突然执行完成，它被处理如下:

- 如果一个表达式的执行突然结束，那么 switch 表达式也会因为同样的原因突然结束。
- 如果语句执行突然结束，因为 yield 值为 V，那么 switch 表达式正常完成，switch 表达式的值为 V。
- 如果一个语句的执行突然结束，而不是由于带有值的 yield，那么 switch 表达式也会因为同样的原因突然结束。

## 15.29 常量表达式

*ConstantExpression:*
*Expression*

常量表达式是表示原生类型值或 String 的表达式，它不突然完成，只使用以下语句组成:

- 原生类型字面量(§3.10.1, §3.10.2, §3.10.3, §3.10.4), 字符串字面量(§3.10.5), 和文本块(§3.10.6)
- 强制转换为原生类型和强制转换为 String 类型 (§15.16)
- 一元操作符 +, -, ~, 和 ! (不包括 ++ 或 --) (§15.15.3, §15.15.4, §15.15.5, §15.15.6)
- 乘法运算符 *, /, 和 % (§15.17)
- 加法运算法 + 和 - (§15.18)
- 移位运算符 <<, >>, 和 >>> (§15.19)
- 关系运算符 <, <=, >, 和 >= (不包括 instanceof) (§15.20)

- 相等运算符 == 和 != (§15.21)
- 按位和逻辑运算符 &,^,和 | (§15.22)
- 条件与运算符 && 和条件或运算符 || (§15.23, §15.24)
- 三元条件运算符 ? : (§15.25)
- 括号表达式(§15.8.5)，其包含的表达式是常量表达式。
- 引用常量变量的简单名称(§6.5.6.1)(§4.12.4)。
- 形式为 TypeName . Identifier 的限定名(§6.5.6.2)指向常量变量(§4.12.4)。

String 类型的常量表达式总是被“内嵌”，以便使用 String.intern 方法共享唯一的实例。

常量表达式用作 switch 语句和 switch 表达式(§14.11、§15.28)中的 case 标签，并且在赋值上下文(§5.2)和类或接口的初始化(§12.4.2)中具有特殊意义。它们还可以控制 while、do 或 for 语句正常完成的能力(§14.22)，以及使用数值操作数的条件运算符? :的类型。

**例子 15.29-1. 常量表达式**

```
true
(short)(1*2*3*4*5*6)
Integer.MAX_VALUE / 2
2.0 * Math.PI
"The integer " + Long.MAX_VALUE + " is mighty big."
```