第九章

接口

接口声明定义了一个可以由一个或多个类实现的新接口。程序可以使用接口为其他不相关的类提供公共超类，并使相关类不必共享公共抽象超类。

接口没有实例变量，通常声明一个或多个抽象方法;否则，不相关的类可以通过为其抽象方法提供实现来实现接口。接口不能直接实例化。

顶层接口(§7.6)是直接在编译单元中声明的接口。

嵌套接口是在另一个类或接口声明体中声明的任何接口。嵌套接口可以是成员接口(§8.5、§9.5)，也可以是局部接口(§14.3)。

注解接口(§9.6)是一个使用不同语法声明的接口，旨在通过注解的反射表示来实现(§9.7)。

本章讨论所有接口的通用语义。特定于特定类型接口的细节将在专门讨论这些构造的小节中讨论。

接口可以声明为一个或多个其他接口的直接扩展，这意味着它继承了它扩展的接口的所有成员类和接口、实例方法和静态字段，但它可能重写或隐藏的任何成员除外。

可以声明一个类直接实现一个或多个接口（§8.1.5），这意味着该类的任何实例都实现了该接口指定的所有抽象方法。类必须实现其直接超类和直接超接口所实现的所有接口。这种（多）接口继承允许对象在不共享超类的情况下支持（多）公共行为。

与类不同，接口不能声明为 final。然而，接口可以声明为密封的（§9.1.1.4），以限制其子类和子接口。

声明类型为接口类型的变量的值可以是对实现指定接口的类的任何实例的引用。仅仅让类实现接口的所有抽象方法是不够的；类或其超类之一必须实际声明以实现接口，否则该类不被视为实现接口。

9.1 接口声明

接口声明指定接口。

接口声明有两种：普通接口声明和注解接口声明（§9.6）。

InterfaceDeclaration:
  NormalInterfaceDeclaration
  AnnotationInterfaceDeclaration

NormalInterfaceDeclaration:
  {InterfaceModifier} `interface` *TypeIdentifier [TypeParameters] [InterfaceExtends]*

[InterfacePermits] InterfaceBody

接口声明中的 TypeIdentifier 指定接口的名称。

如果接口的简单名称与其任何封闭类或接口相同，则为编译时错误。

§6.3 和§6.4.1 规定了接口声明的作用域和遮蔽。

9.1.1 接口修饰符

接口声明可以包括接口修饰符。

InterfaceModifier:
(one of)
Annotation `public protected private`
`abstract static sealed non-sealed strictfp`

有关接口声明注解修饰符的规则在§9.7.4 和§9.7.5 中有规定。

公共的访问修饰符（§6.6）仅适用于顶级接口（§7.6）和成员接口（§8.5，§9.5），不适用于局部接口（§14.3）。

受保护和私有的访问修饰符仅适用于成员接口。

静态修饰符仅适用于成员接口和局部接口。

如果同一关键字作为接口声明的修饰符出现多次，或者接口声明具有多个访问修饰符 public、protected 和 private，则这是编译时错误。

如果接口声明具有多个密封和非密封修饰符，则这是编译时错误。

如果在接口声明中出现两个或多个（不同的）接口修饰符，则通常（尽管不是必需的）它们的出现顺序与上面在 InterfaceModifier 产品中显示的顺序一致。

9.1.1.1 抽象接口

每个接口都是隐式抽象的。

此修饰符已过时，不应在新代码中使用。

9.1.1.2 `strictfp` 接口

接口声明上的 strictfp 修饰符已过时，不应在新代码中使用。它的存在或不存在在编译时或运行时没有影响。

9.1.1.3 静态接口

嵌套接口是隐式静态的。也就是说，每个成员接口和局部接口都是静态的。允许成员接口的声明冗余指定静态修饰符（§9.5），但对局部接口的声明（§14.3）是不允许的。

因为嵌套接口是静态的，所以它没有直接封闭的实例（§8.1.3）。禁止从嵌套接口引用词法封闭的类、接口或方法声明中的类型参数、实例变量、局部变量、形式参数、异常参数

或实例方法(§6.5.5.1、§6.5.6.1、§15.12.3)。

9.1.1.4 密封和非密封接口

如果在声明接口时知道接口的所有直接子类和直接子接口（§9.1.4），并且不需要其他直接子类或直接子接口，则可以声明该接口为密封接口。

回想一下，类被称为其直接超接口的直接子类（§8.1.5），这是很有用的。

如果接口的所有直接超级接口均未密封（§9.1.3），且接口本身未密封，则接口可自由扩展。

具有密封的直接超接口的接口可以自由扩展，当且仅当它被声明为非密封时。

如果接口具有密封的直接超接口且未声明为密封或非密封，则这是编译时错误。

如果接口声明为非密封的，但没有密封的直接超级接口，则这是编译时错误。

9.1.2 泛型接口和类型参数

如果接口声明了一个或多个类型变量（§4.4），则接口是泛型的。

这些类型变量称为接口的类型参数。类型参数部分位于接口名称之后，由尖括号分隔。

为方便起见，此处显示了§8.1.2 和§4.4 中的以下产品：

TypeParameters:
  < *TypeParameterList* >

TypeParameterList:
  TypeParameter {, *TypeParameter*}

TypeParameter:
  {*TypeParameterModifier*} *TypeIdentifier* [*TypeBound*]

TypeParameterModifier:
  Annotation

TypeBound:
  `extends` *TypeVariable*
  `extends` *ClassOrInterfaceType* {*AdditionalBound*}

AdditionalBound:
  `&` *InterfaceType*

有关类型参数声明的注解修饰符的规则在§9.7.4 和§9.8.5 中有规定。

在接口的类型参数部分中，如果 S 是 T 的边界，则类型变量 T 直接依赖于类型变量 S；如果 T 直接依赖于 S 或 T 直接依赖于依赖于 S 的类型变量 U，则 T 依赖于 S（递归使用此定义）。如果接口的类型参数部分中的类型变量依赖于自身，则这是编译时错误。

§6.3 和§6.4.1 规定了接口类型参数的作用域和遮蔽。

根据§6.5.5.1 的规定，从静态上下文或嵌套类或接口引用接口的类型参数受到限制。

泛型接口声明定义了一组参数化类型(§4.5)，类型参数部分的每个可能参数化都有一个类

型参数。所有这些参数化类型在运行时共享相同的接口。

9.1.3 超接口和子接口

如果提供了 extends 子句，则声明的接口将扩展每个指定的接口类型，从而继承每个接口类型的成员类、成员接口、实例方法和静态字段。

指定的接口类型是所声明接口的直接超级接口类型。

实现所声明接口的任何类也被视为实现该接口扩展的所有接口。

InterfaceExtends:
`extends` *InterfaceTypeList*

为方便起见，此处显示了§8.1.5 中的以下内容:

InterfaceTypeList:
InterfaceType {, InterfaceType}

接口声明的 extends 子句中的每个 InterfaceType 必须命名一个可访问接口（§6.6），否则会发生编译时错误。

如果任何 InterfaceType 命名了一个密封的接口（§9.1.1.4），并且所声明的接口不是命名接口的允许直接子接口（§8.1.3），则这是编译时错误。

如果 InterfaceType 具有类型参数，则它必须表示格式良好的参数化类型（§4.5），并且所有类型参数都不能是通配符类型参数，否则会发生编译时错误。

如果第一接口由第二接口的直接超接口类型之一命名，则一个接口是另一接口的直接超接口。

超接口关系是直接超接口关系的传递闭包。如果以下任一项为真，则接口 I 是接口 K 的超接口:

- I 是 K 的直接超接口。
- J 是 K 的直接超接口，I 是 J 的超接口，递归地应用这个定义。

接口被称为其直接超级接口的直接子接口，以及其每个超接口的子接口。

虽然每个类都是 Object 类的扩展，但没有一个接口的所有接口都是扩展。

接口 I 直接依赖于一个类或接口 A，如果 A 在 I 的 extends 子句中被作为超接口或作为超接口名称的完全限定形式的限定符。

接口 I 依赖于一个类或接口 A，如果下列任何一个为真:

- I 直接依赖于 A。
- I 直接依赖于依赖于 A 的类 C(§8.1.5)。

- I 直接依赖于依赖于 A 的接口 J，递归地应用这个定义。

如果接口依赖于自身，则为编译时错误。

如果循环声明的接口在运行时被检测到，当接口被加载时，就会抛出 ClassCircularityError(§12.2.1)。

9.1.4 允许的直接子类和子接口

在普通接口声明中可选的permits子句指定了被声明接口的直接子类和直接子接口(§9.1.1.4)。

InterfacePermits:
`permits` *TypeName {, TypeName}*

如果接口声明有一个 permits 子句但没有密封的修饰符，则是编译时错误。

每个 TypeName 必须命名一个可访问的类或接口(§6.6)，否则会发生编译时错误。

如果在 permits 子句中多次指定相同的类或接口，则会出现编译时错误。即使类或接口以不同的方式命名也是如此。

类或接口的规范名称不需要在 permits 子句中使用，但是 permits 子句一次只能指定一个类或接口。例如，以下程序编译失败:

```
package p;

sealed interface I permits C, D, p.C {}  // error

non-sealed class C implements I {} non-sealed class D implements I {}
```


如果一个密封的接口 I 与一个命名的模块相关联(§7.3)，那么在 I 声明的 permits 子句中指定的每个类或接口必须与 I 相同的模块相关联，否则将发生编译时错误。

如果密封接口 I 与未命名模块相关联（§7.7.5），则 I 声明的 permits 子句中指定的每个类或接口必须与 I 属于同一个包，否则会发生编译时错误。

密封接口及其直接子类和直接子接口需要分别在 permits、implements 和 extends 子句中以循环方式相互引用。因此，在模块化代码库中，它们必须位于同一模块中，因为不同模块中的类和接口不能以循环方式相互引用。在任何情况下，集中办公都是可取的，因为密封的接口层次结构应始终在单个维护域中声明，其中相同的开发人员或开发人员组负责维护层次结构。命名模块通常表示模块化代码库中的维护域。

如果密封接口 I 的声明具有 permits 子句，则 I 的允许直接子类和子接口是 permits 子句指定的类和接口。

permits 子句指定的每个允许的直接子类和子接口必须是 I 的直接子类（§8.1.5）或 I 的直接子接口（§9.1.3），否则会发生编译时错误。

如果密封接口 I 的声明缺少 permits 子句，则 I 的允许直接子类和子接口是在与 I（§7.3）相同的编译单元中声明的类和接口，这些类和接口具有规范名称（§6.7），其直接超接口包括 I。

也就是说，允许的直接子类和子接口被推断为同一编译单元中的类和接口，这些类和接口将 I 指定为直接超接口。规范名称的要求意味着不会考虑局部类、局部接口或匿名类。

如果密封接口 I 的声明缺少一个 permits 子句，并且没有允许的直接子类或子接口，那么这是一个编译时错误。

9.1.5 接口体和成员声明

接口体可以包含接口成员的声明，即字段(§9.3)、方法(§9.4)、类和接口(§9.5)。

InterfaceBody:
{ *{InterfaceMemberDeclaration}* }

InterfaceMemberDeclaration:
ConstantDeclaration
InterfaceMethodDeclaration
ClassDeclaration
InterfaceDeclaration
;

在接口 I 中声明成员 m 或由接口 I 继承的成员 m 声明的作用域在§6.3 中规定。

9.2 接口成员

接口的成员有:

- 在接口声明体中声明的成员 (§9.1.5)。
- 从任何直接超接口类型继承的成员 (§9.1.3)。
- 如果一个接口没有直接的超接口类型，那么该接口隐式声明了一个公共抽象成员方法 m，带有签名 s，返回类型 r，以及 throws 子句 t 对应于每个公共实例方法 m，带有签名 s，返回类型 r，throws 子句 t 并在 Object 中声明 的方法(§4.3.2)，除非接口显式声明了一个具有相同签名，相同返回类型，并且兼容的 throws 子句的抽象方法。

  如果在 Object 中将 m 声明为 final 的情况下，接口显式声明了这样的方法 m，则会出现编译时错误。

  如果接口显式声明了一个方法，该方法的签名与 Object 的公共方法是重写等价的(§8.4.2)，但返回类型不同，或者有不兼容的 throws 子句，或者不是抽象的，那么这就是编译时错误。

接口从它扩展的接口继承这些接口的所有成员，除了(i)字段、类和它隐藏的接口，(ii)它重写的抽象方法和默认方法(§9.4.1)，(iii)私有方法和(iv)静态方法。

一个接口的字段、方法、成员类和成员接口可能有相同的名称，因为它们被用于不同的上下文中，并且通过不同的查找过程消除了歧义(§6.5)。然而，作为一种风格，这是不鼓励

的。

9.3 字段(常量)声明

ConstantDeclaration:
{ConstantModifier} UnannType VariableDeclaratorList ;

ConstantModifier:
(one of)
Annotation `public`
`static final`

UnannType 的定义见§8.3。为方便起见，以下是§4.3 和§8.3 的产品:

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [Dims]

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

VariableInitializer: Expression ArrayInitializer

关于接口字段声明的注解修饰符的规则见§9.7.4 和§9.7.5。

接口声明体中的每个字段声明隐式地分为 public、static 和 final。允许冗余地为这些字段指定任何或所有这些修饰符。

如果相同的关键字作为字段声明的修饰符出现不止一次，则会出现编译时错误。

如果两个或多个(不同的)字段修饰符出现在字段声明中，习惯上(虽然不是必需的)，它们的出现顺序应该与上述 ConstantModifier 产品中的显示顺序一致。

如果 UnannType 和 VariableDeclaratorId 中没有出现方括号，则字段声明的类型由 UnannType 表示，否则由§10.2 指定。

接口字段声明的作用域和遮蔽在§6.3 和§6.4.1 中有规定。

因为接口字段是静态的，它的声明引入了一个静态上下文(§8.1.3)，该上下文限制了引用当前对象的构造的使用。值得注意的是，关键字 this 和 super 在静态上下文中是禁止的(§15.8.3，§15.11.2)，对实例变量、实例方法和在词法封闭声明的类型参数的非限定引用(§6.5.5.1，§6.5.6.1，§15.12.3)也是禁止的。

接口声明的主体声明两个具有相同名称的字段是编译时错误。

如果接口声明了一个具有特定名称的字段，那么该字段的声明被认为隐藏了该接口的超接口中具有相同名称的任何和所有可访问字段声明。

一个接口可以继承多个具有相同名称的字段。这种情况本身不会导致编译时错误。但是，在接口声明体中，任何通过简单名称引用此类字段的尝试都会导致编译时错误，因为引用是不明确的。

从接口继承相同字段声明可能有多个路径。在这种情况下，字段被认为只继承一次，并且可以通过它的简单名称引用它，而不会产生歧义。

例子 9.3-1. 模棱两可的继承的字段

如果一个接口继承了两个具有相同名称的字段，例如，它的两个直接超接口声明了具有该名称的字段，则会产生一个二义性成员。任何对这个二义性成员的使用都将导致编译时错误。在程序中:

```
interface BaseColors {
    int RED = 1, GREEN = 2, BLUE = 4;
}
interface RainbowColors extends BaseColors {
    int YELLOW = 3, ORANGE = 5, INDIGO = 6, VIOLET = 7;
}
interface PrintColors extends BaseColors {
    int YELLOW = 8, CYAN = 16, MAGENTA = 32;
}
interface LotsOfColors extends RainbowColors, PrintColors {
    int FUCHSIA = 17, VERMILION = 43, CHARTREUSE = RED+90;
}
```

接口 LotsOfColors 继承了两个名为 YELLOW 的字段。只要接口不包含任何通过简单名称对字段 YELLOW 的引用，就没有问题。(这样的引用可以发生在字段的变量初始化器中。)

即使接口 PrintColors 将值 3 给 YELLOW 而不是值 8，在接口 LotsOfColors 中对字段 YELLOW 的引用仍然会被认为是不明确的。

例子 9.3-2. 多继承字段

如果一个字段多次从同一接口继承，例如，由于此接口和此接口的一个直接超级接口扩展了声明该字段的接口，则只会产生一个成员。这种情况本身不会导致编译时错误。

在上一个示例中，RED、GREEN 和 BLUE 字段由接口 LotsOfColors 以多种方式继承，通过接口 RainbowColors 和接口 PrintColors 继承，但在接口 LotsOfColors 中对 RED 字段的引用并不被认为是模糊的，因为只涉及 RED 字段的一个实际声明。

9.3.1 接口中字段的初始化

接口的字段声明中的每个声明器都必须具有变量初始化器，否则会发生编译时错误。

初始化器不必是常量表达式 (§15.29)。

如果接口字段的初始化器使用同一字段的简单名称，或另一字段声明出现在同一接口中初

始化器的右侧（§3.5），则这是编译时错误。

如§15.8.3、§15.11.2 和§15.12.3 所规定的，接口字段的初始化器不能使用关键字 this 或关键字 super 来指向当前对象。

在运行时，初始化器被求值，字段赋值只执行一次，也就是接口被初始化时(§12.4.2)。

请注意，作为常量变量的接口字段(§4.12.4)会在其他接口字段之前初始化。这也适用于类中的常量变量静态字段(§8.3.2)。这样的字段永远不会被观察到有它们的默认初始值(§4.12.5)，即使是通过不可靠的程序。

例子 9.3.1-1. 字段的前向引用

```
interface Test {
   float f = j;
   int j = 1;
   int k = k + 1;
}
```

该程序会导致两个编译时错误，因为在声明 j 之前，f 的初始化中引用了 j，而 k 的初始化则引用了 k 本身。

9.4 方法声明

InterfaceMethodDeclaration:
{InterfaceMethodModifier} MethodHeader MethodBody

InterfaceMethodModifier:
(one of)
Annotation `public private`
`abstract default static strictfp`

为方便起见，此处显示了§8.4、§8.4.5 和§8.5.7 中的以下产品：

MethodHeader:
Result MethodDeclarator [Throws]
TypeParameters {Annotation} Result MethodDeclarator [Throws]

Result:
UnannType `void`

MethodDeclarator:
Identifier (*[ReceiverParameter ,] [FormalParameterList]*) *[Dims]*

MethodBody:
Block
;

有关接口方法声明的注解修饰符的规则在§9.7.4 和§9.7.5 中有规定。

接口声明主体中的方法可以声明为公共的或私有的（§6.6）。如果没有给出访问修饰符，

则该方法是隐式公共的。允许但不鼓励在接口声明中为方法声明冗余指定公共修饰符。

默认方法是在带有默认修饰符的接口中声明的实例方法。其主体始终由块表示，该块为实现接口的任何类提供默认实现，而不重写该方法。默认方法不同于在类中声明的具体方法（§8.4.3.1），也不同于既不继承也不重写的私有接口方法。

接口可以声明静态方法，这些方法在调用时不引用特定对象。静态接口方法不同于默认方法、抽象接口方法和非静态私有接口方法，它们都是实例方法。

静态接口方法的声明引入了静态上下文（§8.1.3），这限制了引用当前对象的构造的使用。值得注意的是，在静态上下文中禁止使用 this 和 super 关键字（§15.8.3，§15.11.2），对实例变量、实例方法和词法封装声明的类型参数的非限定引用也是如此（§6.5.5.1，§6.6.1，§15.12.3）。

从静态上下文或嵌套类或接口引用实例方法受到限制（§15.12.3）。

接口方法声明上的 strictfp 修饰符已过时，不应在新代码中使用。其存在或不存在在运行时没有影响。

缺少私有、默认或静态修饰符的接口方法是隐式抽象的。其主体由分号表示，而不是块。允许但不鼓励为这种方法声明冗余指定抽象修饰符。

> 请注意，接口方法不能使用 protected 声明或包访问，也不能使用 final、synchronized 或 native 修饰符声明。

如果同一个关键字不止一次出现在接口方法声明的修饰符中，或者一个接口方法声明有多个访问修饰符 public 和 private(§6.6)，那么将会出现编译时错误。

如果接口方法声明有多个关键字 abstract、default 或 static，则会出现编译时错误。

如果包含关键字 private 的接口方法声明同时也包含关键字 abstract 或 default，则会出现编译时错误。允许接口方法声明同时包含 private 和 static。

如果包含关键字 abstract 的接口方法声明同时包含关键字 strictfp，则会出现编译时错误。

接口声明体显式或隐式声明两个具有重写等价签名的方法是编译时错误(§8.4.2)。然而，一个接口可以继承几个具有这样签名的抽象方法(§9.4.1)。

在接口中声明的方法可以是泛型的。接口中泛型方法的类型参数规则与类中的泛型方法相同(§8.4.4)。

9.4.1 继承和重写

接口 I 从其直接超级接口类型继承所有抽象和默认方法 m，对于这些方法，以下所有条件均为真：

- m 是 I、J 的直接超接口类型的成员。

• I 中声明的任何方法的签名都不是 m 作为 J 成员的签名的子签名（§8.4.2）。

• 不存在作为 I，J'（m 不同于 m'，J 不同于 J'）的直接超接口成员的方法 m'，因此 m'从 J' 的接口重写方法 m 的声明（§9.4.1.1）。

请注意，方法是在逐个签名的基础上重写的。例如，如果一个接口声明了两个同名的公共方法（§9.4.2），并且一个子接口重写了其中一个，则该子接口仍然继承另一个方法。

上面的第三个子句防止子接口重新继承已被其另一个超接口重写的方法。例如，在该程序中：

```
interface Top {
    default String name() { return "unnamed"; }
}
interface Left extends Top {
    default String name() { return getClass().getName(); }
}
interface Right extends Top {}

interface Bottom extends Left, Right {}
```

Right 从 Top 继承 name(), 但是 Bottom 从 Left 而不是 Right 继承 name()。
这是因为 Left 中的 name()重写了 Top 里的 name()的声明。

接口不会从其超接口继承私有或静态方法。

如果接口 I 声明了私有或静态方法 m，并且 m 的签名是 I 的超接口类型中的公共实例方法 m'的子签名，并且 m'可以被 I 中的代码访问，则会发生编译时错误。

本质上，接口中的静态方法不能隐藏超级接口类型中的实例方法。这类似于§8.4.8.2 中的规则，其中类中的静态方法不能隐藏超类类型或超接口类型中的实例方法。请注意，§8.4.8.2 中的规则提到“声明或继承静态方法”的类，而上述规则只提到“声明静态方法”接口，因为接口不能继承静态方法。还请注意，§8.4.8.2 中的规则允许在超类/超级接口中隐藏实例和静态方法，而上述规则仅考虑超级接口类型中的公共实例方法。

同样，接口中的私有方法不能重写超级接口类型中的实例方法（无论是公共的还是私有的）。这类似于§8.4.8.1 和§8.48.3 中的规则，其中类中的私有方法不能重写超类类型或超接口类型中的任何实例方法，因为§8.418.1 要求被重写方法是非私有的，而§8.4.8.3 要求重写方法提供至少与被重写方法一样多的访问。总之，只能重写接口中的公共方法，并且只能由子接口或实现类中的公共法重写。

9.4.1.1 重写 (通过实例方法)

在接口 I 中声明或由接口 I 继承的实例方法 m_I，从 I 中重写接口 J 中声明的另一个实例方法 m_J，如果以下所有条件均为真：

• I 是 J 的子接口。

• I 不继承 m_J。

• m_I 的签名是 m_J 签名的子签名（§8.4.2），m_J 是命名为 J 的 I 的超类型的成员。

• m_J 是公共的。

strictfp 修饰符的存在或不存在对重写方法的规则绝对没有影响。例如，允许非 strictfp 的

方法重写 strictfp 方法，并且允许 strictfp 方法重写非 strictfp 的方法。

可以使用方法调用表达式（§15.12）访问重写的默认方法，该表达式包含由超级接口名称限定的关键字 super。

9.4.1.2 重写的要求

§8.4.8.3 规定了接口方法的返回类型与任何被重写的接口方法返回类型之间的关系。

§8.4.8.3 规定了接口方法的 throws 子句与任何被重写的接口方法的 throws 子句之间的关系。

§8.4.8.3 规定了接口方法的签名与任何被重写的接口方法签名之间的关系。

§8.4.8.3 中规定了接口方法的可访问性与任何被重写接口方法的可用性之间的关系。

如果默认方法与 Object 类的非私有方法重写等价（§8.4.2），则为编译时错误，因为实现接口的任何类都将继承其自身的方法实现。

禁止将 Object 方法中的任何一个声明为默认方法可能令人惊讶。毕竟，有像 java.util.List 这样的例子，其 toString 和 equals 方法的行为是精确定义的。然而，当理解了一些更广泛的设计决策时，动机变得更加清晰：

- 首先，允许从超类继承的方法重写从超接口继承的方法（§8.4.8.1）。因此，每个实现类都会自动重写接口的 toString 默认值。这是 Java 编程语言中长期存在的行为。我们不希望随着默认方法的设计而改变这一点，因为这将与允许接口不引人注目地发展的目标相冲突，只有在类的层次结构中没有默认行为时才提供默认行为。
- 其次，接口不继承 Object，而是隐式声明了许多与 Object 相同的方法(§9.2)。因此，Object 中声明的 toString 和接口中声明的 toString 没有共同的祖先。在最好的情况下，如果两者都是一个类继承的候选者，它们就会发生冲突。解决这个问题需要笨拙地混合类继承树和接口继承树。
- 第三，在接口中声明对象方法的用例通常采用线性接口层次结构；该特性不能很好地推广到多继承场景。
- 第四，Object 方法是如此的基本，以至于允许任意的超接口以静默方式添加更改其行为的默认方法似乎是危险的。

但是，接口可以自由定义另一个方法，该方法为重写 Object 方法的类提供有用的行为。例如，java.util.List 接口可以声明一个 elementString 方法，该方法产生 toString 契约所描述的字符串;类中的 toString 实现者可以委托给这个方法。

9.4.1.3 继承具有重写等价签名的方法

一个接口可以继承几个具有重写等价签名的方法(§8.4.2)。

如果一个接口 I 继承了一个默认方法，该方法的签名与 I 继承的另一个方法是重写等价的，那么就会发生编译时错误。(无论其他方法是抽象的还是默认的，都是这样。)

否则，所有继承的方法都是抽象的，接口被认为继承了所有的方法。

继承的方法中必须有一个方法对其他继承的方法具有返回类型可替换性，否则将发生编译时错误。(在这种情况下，throws 子句不会导致错误。)

从接口继承相同方法声明可能有多个路径。这一事实不会造成任何困难，而且本身也不会导致编译时错误。

自然，当子接口继承了两个具有匹配签名的不同默认方法时，就会出现行为冲突。我们主动检测这种冲突，并将错误通知程序员，而不是在编译具体类时等待问题出现。可以通过声明重写所有冲突方法的新方法来避免该错误，从而防止继承所有冲突方法。

类似地，当子接口继承了具有匹配签名的抽象方法和默认方法时，就会产生错误。在这种情况下，可能会优先考虑其中一个方法—也许我们会假设默认方法为抽象方法提供了合理的实现。但是这是有风险的，因为除了名称和签名的巧合之外，我们没有理由相信默认方法的行为与抽象方法的契约一致——在最初开发子接口时，默认方法甚至可能还不存在。在这种情况下，更安全的做法是要求用户主动断言默认实现是适当的(通过重写声明)。

相比之下，类中继承的具体方法的长期行为是它们重写接口中声明的抽象方法(参见§8.4.8)。关于潜在契约违反的争论同样适用于这里，但在这种情况下，类和接口之间存在固有的不平衡。为了保持类层次结构的独立性，我们倾向于通过简单地将优先级赋予具体方法来最小化类接口冲突。

9.4.2 重载

如果一个接口的两个方法(无论是在同一个接口中声明的，还是都被一个接口继承的，或者一个声明的和一个继承的)有相同的名称，但不同的签名不是重写等价的(§8.4.2)，那么这个方法名被称为重载的。

这个事实不会造成任何困难，本身也不会导致编译时错误。返回类型之间或具有相同名称但签名不同且重写不等价的两个方法的 throws 子句之间没有必要的关系。

例子 9.4.2-1. 重载抽象方法声明

```
interface PointInterface {
   void move(int dx, int dy);
}
interface RealPointInterface extends PointInterface {
   void move(float dx, float dy);
   void move(double dx, double dy);
}
```

在这里，名为 move 的方法在接口 RealPointInterface 中使用三个不同的签名被重载，其中两个声明，一个继承。任何实现 RealPointInterface 接口的非抽象类都必须提供所有三个方法签名的实现。

9.4.3 接口方法体

默认方法有一个块体。当类实现了接口但没有提供自己的方法实现时，该代码块提供了方法的实现。

私有或静态接口方法也有一个块体，它提供了方法的实现。

如果接口方法声明是抽象的(显式或隐式)，并且有一个块作为它的主体，那么它就是编译时错误。

如果接口方法声明为 default、private 或 static，并且其主体有分号，则会出现编译时错误。

在方法体中的 return 语句的规则见§14.17。

如果一个方法被声明为有返回类型(§8.4.5)，那么如果该方法体可以正常完成(§14.1)，则会发生编译时错误。

9.5 成员类和接口声明

接口体(§9.1.5)可以包含成员类和成员接口的声明(§8.5)。

接口声明体中的每个成员类或接口声明都是隐式 public 和 static 的(§9.1.1.3)。允许冗余指定这两个修饰符中的一个或两个。

如果接口中的成员类或接口声明具有修饰符 protected 或 private，则为编译时错误。

接口声明体中成员类声明的修饰符规则见§8.1.1。

接口声明体中成员接口声明的修饰符规则见§9.1.1。

如果一个接口声明了具有特定名称的成员类或接口，那么该成员类或接口的声明被认为隐藏了该接口的超接口中具有相同名称的成员类和接口的任何和所有可访问声明。

接口从其直接超接口继承所有成员类和直接超接口的接口，这些成员类和接口不会被接口中的声明所隐藏。

一个接口可以继承多个具有相同名称的成员类或接口。这种情况本身不会导致编译时错误。但是，在接口体中试图通过其简单名称引用任何此类成员类或接口将导致编译时错误，因为引用是不明确的。

从接口继承相同成员类或接口声明可能有多个路径。在这种情况下，成员类或接口被认为只继承一次，并且可以通过其简单名称引用它，而不会产生歧义。

9.6 注解接口

注解接口声明指定注解接口，这是一种专门化的接口。为了将注解接口声明与普通接口声明区分开来，关键字 interface 前面要加一个@符号。

AnnotationInterfaceDeclaration:
{InterfaceModifier} `@ interface` *TypeIdentifier AnnotationInterfaceBody*

注意，at 符号(@)和关键字接口是不同的标记。可以用空格分隔它们，但出于风格考虑，不建议这样做。

除非在本节及其子节中显式修改，否则所有适用于普通接口声明(§9.1)的规则都适用于注解接口声明。

例如，注解接口声明具有与普通接口声明相同的作用域规则。

如果注解接口声明的修饰符是密封的或非密封的，这是编译时错误(§9.1.1.4)。

注解接口声明可以指定顶级接口或成员接口，但不能指定局部接口(§14.3)。

由于§14.3 中的 LocalClassOrInterfaceDeclaration 产品，语法上不允许注解接口声明出现在块中。

如果注解接口声明直接或间接出现在局部类、局部接口或匿名类声明的主体中，则为编译时错误（§14.3、§15.9.5）。

该规则，加上上述注解接口声明的语法限制，确保注解接口始终具有规范名称（§6.7）。具有这样的名称很重要，因为注解接口的目的是供其他编译单元中的注解使用。由于局部类或接口没有规范名称，因此在其语法体中任何位置声明的注解接口（如果允许的话）也不会有规范名称。

以下代码显示了此规则的效果和相关的语法限制：

```
class C {
    @interface A1 {}       /* Legal: an annotation interface can be a
                           member interface */

    void m()

    @interface A2 {}       /* Illegal: an annotation interface cannot
                           be a local interface */

    class D {
            @interface A3 {}     /* Illegal: an annotation interface
                                 cannot be specified anywhere within the body of
                                 local class D */

            class E {
                @interface A4 {}
                  /* Illegal: an annotation interface cannot be specified anywhere
                     within the body of local class D, even as a member of a class
                     E nested in D */
            }
        }
    }
}
```

注解接口从来都不是泛型（§9.1.2）。

与普通接口声明不同，注解接口声明不能通过 AnnotationTypeDeclaration 产品声明任何类型变量。

注解接口的直接超接口类型始终为 java.lang.annotation.Annotation (§9.1.3)。

与普通的接口声明不同，注解接口声明不能通过 extends 子句选择直接的超接口类型，这是通过 AnnotationTypeDeclaration 产品实现的。

注解接口声明没有通过 extends 显式指定超接口类型这一事实的一个后果是，注解接口的子接口本身从来不是注解接口，因为子接口的声明必须使用 extends 子句。类似地，java.lang.annotation.Annotation 本身也不是一个注解接口。

注解接口从 java.lang.annotation.Annotation 继承了几个方法，包括对应于 Object 的实例方法的隐式声明的方法(§9.2)，但是这些方法没有定义注解接口的元素(§9.6.1)。

由于这些方法没有定义注解接口的元素，因此在符合注解接口的注解中使用它们是非法的（§9.7）。如果没有这条规则，我们就无法确保元素是可在注解中表示的类型，或者它们的访问器方法是可用的。

9.6.1 注解接口元素

注解接口声明的主体可以包含方法声明，每个方法声明定义注解接口的一个元素。注解接口除了由注解接口声明中显式声明的方法定义的元素外，没有其他元素。

AnnotationInterfaceBody:
{ *{AnnotationInterfaceMemberDeclaration}* }

AnnotationInterfaceMemberDeclaration:
AnnotationInterfaceElementDeclaration
ConstantDeclaration
ClassDeclaration
InterfaceDeclaration
;

AnnotationInterfaceElementDeclaration:
{AnnotationInterfaceElementModifier} UnannType Identifier ( ) *[Dims]*
[DefaultValue] ;

AnnotationInterfaceElementModifier:
(one of)
Annotation `public`
`abstract`

为方便起见，此处显示了§4.3 中的以下内容:

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

根据上述语法，注解接口声明中的方法声明不能具有形式参数、类型参数或 throws 子句；并且不能是 private、default 或 static。因此，注解接口不能具有与普通接口相同的多种方法。请注意，注解接口仍然可以从其隐式超接口 java.lang.annotation.Annotation 继承默认方法，尽管到 Java SE 19 为止还不存在这样的默认方法。

按照惯例，注解接口元素声明中应该出现的唯一修饰符是注解。

注解接口体中声明的方法的返回类型必须是下列类型之一，否则将发生编译时错误:

- 原生类型
- 字符串

- 类或类的调用(§4.5)
- 枚举类类型
- 注解接口类型
- 组件类型是上述类型之一的数组类型 (§10.1)。

该规则排除嵌套数组类型的元素，例如:

```
@interface Verboten {
   String[][] value();
}
```

返回数组的方法的声明允许将表示数组类型的方括号对放在空的形式参数列表之后。支持这种语法是为了与早期版本的 Java 编程语言兼容。强烈建议在新代码中不要使用此语法。

如果在注解接口中声明的任何方法的签名与在类 Object 或接口 java.lang.annotation.Annotation 中声明的任何公共或受保护方法的签名是重写等价的(§8.4.2)，则为编译时错误。

如果注解接口 T 的声明包含类型 T 的元素(直接或间接地)，则会出现编译时错误。

例如，这是非法的:

```
@interface SelfRef { SelfRef value(); }
```

还有这个:

```
@interface Ping { Pong value(); }
@interface Pong { Ping value(); }
```


没有元素的注解接口称为标记注解接口。

带有一个元素的注解接口称为单元素注解接口。

按照约定，单个元素注解接口中唯一元素的名称是 value。单元素注解提供了对该约定的语言支持(§9.7.3)。

例子 9.6.1-1. 注解接口声明

下面的注解接口声明定义了一个包含几个元素的注解接口:

```
/**
 *       Describes the "request-for-enhancement" (RFE)
 *       that led to the presence of the annotated API element.
 */
@interface RequestForEnhancement {
    int id();                //  Unique ID number associatedwith RFE
    String synopsis();       //  Synopsis of RFE
    String engineer();       //  Name of engineer who implemented RFE
    String date();           //  Date RFE was implemented
```

```
}
```

例子 9.6.1-2. 标记注解接口声明

下面的注解接口声明定义了一个标记注解接口:

```
/**
 *       An annotation with this type indicates that the
 *       specification of the annotated API element is
 *       preliminary and subject to change.
 */
@interface Preliminary {}
```

例子 9.6.1-3. 单元素注解接口声明

一个单元素注解接口定义一个名为 value 的元素的约定如下注解接口声明所示:

```
/**
 *       Associates a copyright notice with the annotated API element. */
@interface Copyright {
   String value();
}
```

下面的注解接口声明定义了一个单元素注解接口，它唯一的元素是数组类型:

```
/**
 *       Associates a list of endorsers with the annotated class.
 */
@interface Endorsers {
   String[] value();
}
```

下面的注解接口声明显示了一个 class 类型的元素，其值受有界通配符的约束:

```
interface Formatter {}

// Designates a formatter to pretty-print the annotated class
@interface PrettyPrinter {
    Class<? extends Formatter> value();

}
```

下面的注解接口声明包含一个类型为注解接口类型的元素:

```
/**
 *       Indicates the author of the annotated program element.
 */
@interface Author {
   Name value();
}
/**
 *       A person's name. This annotation interface is not
 *       designed to be used directly to annotate program elements,
 *       but to define elements of other annotation interfaces.
 */
 @interface Name {
    String first();
```

```
    String last();
}
```

注解接口声明的语法允许除方法声明外的其他成员声明。例如，可以选择声明一个嵌套的枚举类供注解接口的元素使用:

```
@interface Quality {
    enum Level { BAD, INDIFFERENT, GOOD }
    Level value();

}
```

9.6.2 注解接口元素的默认值

注解接口元素可以有一个默认值，通过将关键字 default 和一个值附加到定义该元素的方法声明中来指定。

DefaultValue:
`default` *ElementValue*

为了方便起见，以下是§9.7.1 的产品:

ElementValue:
ConditionalExpression
ElementValueArrayInitializer
Annotation

ElementValueArrayInitializer:
`{` *[ElementValueList] [,]* `}`

ElementValueList:
ElementValue {, ElementValue}

注意，被指定有默认值的注解接口元素不是默认方法(§9.4)。注解接口的声明不能声明默认方法(§9.6.1)。

如果元素的类型与指定的默认值不相称(§9.7)，则会出现编译时错误。

默认值不会编译为注解，而是在读取注解时动态应用。因此，即使在进行更改之前编译的类中，更改默认值也会影响注解（假设这些注解缺少默认元素的显式值）。

例子 9.6.2-1. 带默认值的注解接口声明

以下是对§9.6.1 中的 RequestForEnhancement 注解接口的改进:

```
@interface RequestForEnhancement {
    int id();          // No default - must be specified in
                       // each annotation
    String synopsis(); // No default - must be specified in
                       // each annotation
    String engineer() default "[unassigned]";
    String date() default "[unimplemented]";
}
```

9.6.3 可重复的注解接口

如果注解接口 A 的声明使用@Repeatable 注解（§9.6.4.8）进行（元）注解，则注解接口 A 是可重复的，其值 value 表示 A 的包含注解接口。

注解接口 AC 是 A 的包含注解接口，如果以下所有条件都为真:

1. AC 声明了一个返回类型为 a[]的 value()方法。
2. AC 声明的除 value()之外的任何方法都有一个默认值。
3. AC 被保留的时间至少与 A 一样长，其中保留是通过@Retention 注解显式或隐式表达的(§9.6.4.2)。具体地说:
   - 如果 AC 的保留是 java.lang.annotation.RetentionPolicy.SOURCE, 那么 A 的保留是 java.lang.annotation.RetentionPolicy.SOURCE。
   - 如果 AC 的保留是 java.lang.annotation.RetentionPolicy.CLASS, 那么 A 的保留是 java.lang.annotation.RetentionPolicy.CLASS 或 java.lang.annotation.RetentionPolicy.SOURCE。
   - 如果 AC 的保留是 java.lang.annotation.RetentionPolicy.RUNTIME, 那么 A 的保留是 java.lang.annotation.RetentionPolicy.SOURCE, java.lang.annotation.RetentionPolicy.CLASS,或 java.lang.annotation.RetentionPolicy.RUNTIME.
4. A 至少适用于与 AC 相同类型的程序元素（§9.6.4.1）。具体而言，如果适用于 A 的程序元素的种类由集合 m_1 表示，适用于 AC 的程序元素种类由集合 m_2 表示，则 m_2 中的每一种都必须出现在 m_1 中，除非:
   - 如果 m_2 中的类型为 java.lang.annotation.ElementType.ANNOTATION_TYPE, 那么 java.lang.annotation.ElementType.ANNOTATION_TYPE 或 java.lang.annotation.ElementType.TYPE 或 java.lang.annotation.ElementType.TYPE_USE 中的至少一个必须出现在 m_1 中。
   - 如果 m_2 中的类型为 java.lang.annotation.ElementType.TYPE, 那么 java.lang.annotation.ElementType.TYPE 或 java.lang.annotation.ElementType.TYPE_USE 中的至少一个必须出现在 m_1 中。
   - 如果 m_2 中的类型为 java.lang.annotation.ElementType.TYPE_PARAMETER, 那么 java.lang.annotation.ElementType.TYPE_PARAMETER 或 java.lang.annotation.ElementType.TYPE_USE 中的至少一个必须出现在 m_1 中。

     本子句实现了一个策略，即注解接口只能在某些类型的程序元素上重复（如果适用）。
5. 如果 A 的声明具有对应于 java.lang.annotation.Documented 的（元）注解，那么 AC 的声明必须有一个对应于 java.lang.annotation.Documented 的（元）注解。

   注意，允许 AC 被@Documented 注解，而不允许 A 被@Documented 注解。

6. 如果 A 的声明具有对应于 java.lang.annotation.Inherited 的（元）注解，那么 AC 的声明必须有一个对应于 java.lang.annotation.Inherited 的（元）注解。

    注意，允许 AC 被@Inherited 注解，而不允许 A 被@Inherited 注解。

如果注解接口 A 使用@Repeatable 注解进行（元）注释，其 value 元素指示的类型不是 A 的包含注解接口，则这是编译时错误。

例子 9.6.3-1. 不规范的包含注解接口

考虑以下声明:

```
import java.lang.annotation.Repeatable;

@Repeatable(FooContainer.class)
@interface Foo {}

@interface FooContainer { Object[] value(); }
```

编译 Foo 声明产生了一个编译时错误，因为 Foo 使用@Repeatable 注解尝试指定 FooContainer 作为它的包含注解接口，但是 FooContainer 不是 Foo 的包含注解接口。(FooContainer.value()的返回类型不是 Foo[]。)

@Repeatable 注解不能重复，因此一个可重复注解接口只能指定一个包含注解的接口。

当可重复注解接口的多个注解在逻辑上被容器注解替换时，允许指定多个包含注解的接口将在编译时导致不期望的选择（§9.7.5）。

注解接口可以是最多一个注解接口的包含注解接口。

如果注解接口 A 的声明指定了 AC 的包含注解接口，则 AC 的 value（）方法具有涉及 A 的返回类型，特别是 A[]。

注解接口不能将自身指定为它的包含注解接口。

包含注解接口的 value()方法的要求暗示了这一点。具体来说，如果注解接口 A 指定自己(通过@Repeatable)作为它的包含注解接口，那么 A 的 value()方法的返回类型必须是 A[];但这将导致编译时错误，因为注解接口不能在其元素中引用自身(§9.6.1)。更一般的情况是，两个注解接口不能将对方指定为它自己的包含注解接口，因为循环注解接口声明是非法的。

注解接口 AC 可以是某些注解接口 A 的包含注解接口，同时也具有自己的包含注解接口 SC。也就是说，包含注解接口本身可能是可重复的注解接口。

例子 9.6.3-2. 限制注解可以重复的地方

一个注解的接口声明表明了 java.lang.annotation.ElementType.TYPE 的 target 至少可以出现和接口声明表明了 java.lang.annotation.ElementType.ANNOTATION_TYPE 的 target 的注解一样多的位置。例如，给出下面可重复地和包含注解接口的声明:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Target;
```

```
@Target(ElementType.TYPE)
@Repeatable(FooContainer.class)
@interface Foo {}

@Target(ElementType.ANNOTATION_TYPE)
@interface FooContainer {
   Foo[] value();
}
```

@Foo 可以出现在任何类或接口声明中，而@FooContainer 只能出现在注解接口声明中。因此，下面的注解接口声明是合法的:

```
@Foo @Foo
@interface Anno {}
```

下面的接口声明是非法的:

```
@Foo @Foo
interface Intf {}
```

更广泛地说，如果 Foo 是一个可重复的注解接口，而 FooContainer 是它的包含注解接口，那么:

- 如果 Foo 没有@Target 元注解，FooContainer 也没有@Target 元注解，那么@Foo 可以在任何支持注解的程序元素上重复。
- 如果 Foo 没有@Target 元注解，但 FooContainer 有@Target 元注解，那么@Foo 只能在可能出现@FooContainer 的程序元素上重复。
- 如果 Foo 有一个@Target 元注解，那么在 Java 编程语言的设计者的判断中，FooContainer 必须在知道 Foo 的适用性的情况下声明。具体来说，FooContainer 可能出现的程序元素的种类在逻辑上必须与 Foo 的种类或者其子集相同。

  例如，如果 Foo 适用于字段和方法声明，那么如果 FooContainer 仅适用于字段声明（防止@Foo 在方法声明上重复），则 FooContainer 可以合法地充当 Foo 的包含注解接口。但是如果 FooContainer 只适用于形式参数声明，那么 FooContainer 是 Foo 的包含注解接口的一个糟糕的选择，因为@FooContainer 不能在一些重复@Foo 的程序元素上隐式声明。

  类似地，如果 Foo 适用于字段和方法声明，那么如果 FooContainer 适用于字段和参数声明，FooContainer 就不能合法地作为 Foo 的包含注解接口。虽然可以使用程序元素的交集并使 Foo 只在字段声明上可重复，但 FooContainer 的其他程序元素的存在表明 FooContainer 不是为 Foo 设计的包含注解接口。

例子 9.6.3-3. 可重复的包含注解的接口

下列声明是合法的:

```
import java.lang.annotation.Repeatable;

// Foo: Repeatable annotation interface @Repeatable(FooContainer.class)
@interface Foo { int value(); }

// FooContainer: Containing annotation interface of Foo
// Also a repeatable annotation interface itself
@Repeatable(FooContainerContainer.class)
@interface FooContainer { Foo[] value(); }
```

```
// FooContainerContainer: Containing annotation interface
// of FooContainer
@interface FooContainerContainer { FooContainer[] value(); }
```

因此，一个接口是包含注解接口的注解本身可以被重复:

```
@FooContainer({@Foo(1)}) @FooContainer({@Foo(2)})
class Test {}
```

可重复且包含的注解接口遵循可重复注解接口与包含注解接口的注解的混合规则(§9.7.5)。例如，不可能在写多个@FooContainer 注解的同时写多个@Foo 注解，也不可能在写多个@FooContainerContainer 注解的同时写多个@FooContainer 注解。但是，如果注解接口 FooContainerContainer 本身是可重复的，则可以在多个@FooContaender 注解旁边编写多个@Foo 注解。

9.6.4 预定义注解接口

Java SE 平台 API 中预定义了几个注解接口。一些预定义的注解接口在 Java 编程语言中具有特殊的语义，并要求 Java 编译器的特殊行为，如本节中指定的那样。本节没有提供预定义注解接口的完整规范，读者可以参考 Java SE 平台 API 文档(§1.4)。

9.6.4.1 @Target

类型为 java.lang.annotation.Target 的注解用于注解接口 A 的声明，以指定 A 适用的上下文。java.lang.annotation.Target 有一个类型为 java.lang.annotation.ElementType[]的元素 value，用于指定上下文。

注解接口可能适用于声明上下文，其中注解适用于声明，或者适用于类型上下文，其中注解适用于声明和表达式中使用的类型。

有十个声明上下文，每个都对应于 java.lang.annotation.ElementType 的一个枚举常量:

1. 模块声明 (§7.7)

   对应于 java.lang.annotation.ElementType.MODULE

2. 包声明(§7.4.1)

   对应于 java.lang.annotation.ElementType.PACKAGE

3. 类声明(包括枚举声明和记录声明) 和接口声明(包括注解接口声明) (§8.1.1, §8.5, §8.9, §8.10, §9.1.1, §9.5, §9.6)

   对应于 java.lang.annotation.ElementType.TYPE

   另外，注解接口声明对应于 java.lang.annotation.ElementType.ANNOTATION_TYPE

4. 方法声明(包括注解接口的元素) (§8.4.3, §9.4, §9.6.1)

   对应于 java.lang.annotation.ElementType.METHOD

5. 构造函数声明(§8.8.3)

对应于 java.lang.annotation.ElementType.CONSTRUCTOR

6. 泛型类、接口、方法和构造函数的类型参数声明 (§8.1.2, §9.1.2, §8.4.4, §8.8.4)

   对应于 java.lang.annotation.ElementType.TYPE_PARAMETER

7. 字段声明(包括枚举常量) (§8.3.1, §9.3, §8.9.1)

   对应于 java.lang.annotation.ElementType.FIELD

8. 形式和异常参数声明 (§8.4.1, §9.4, §14.20)

   对应于 java.lang.annotation.ElementType.PARAMETER

9. 在语句(§14.4.2，§14.14.1，§14.14.2，§14.20.3)和模式(§14.30.1)中声明局部变量

   对应于 java.lang.annotation.ElementType.LOCAL_VARIABLE

10. 记录组件声明(§8.10.1)

    对应于 java.lang.annotation.ElementType.RECORD_COMPONENT

共有 17 种类型上下文（§4.11），都由 java.lang.annotation.ElementType 的枚举常量 TYPE_USE 表示。

如果同一枚举常量在 java.lang.annotation.Target 类型注解的 value 元素中出现多次，则这是编译时错误。

如果是类型为 java.lang.annotation.Target 的注解不存在于注解接口 A 的声明中，则 A 适用于所有声明上下文，而不适用于任何类型上下文。

9.6.4.2 `@Retention`

注解可能只出现在源代码中，也可能以类或接口的二进制形式出现。通过 Java SE 平台的反射库，以二进制形式出现的注解在运行时可能可用，也可能不可用。注解接口 java.lang.annotation.Retention 被用来在这些可能性之间选择。

如果一个注解 a 对应于一个注解接口 A，A 有一个元注解 m 对应于 java.lang.annotation.Retention, 那么:

- 如果 m 有一个元素值为 java.lang.annotation.RetentionPolicy.SOURCE, 那么 Java 编译器必须确保 a 没有出现在类或接口的二进制表示中。
- 如果 m 有一个元素值为 java.lang.annotation.RetentionPolicy.CLASS 或 java.lang.annotation.RetentionPolicy.RUNTIME, 那么 Java 编译器必须确保 a 以 a 出现的类或接口的二进制表示形式表示，除非 a 注解了局部变量声明或 a 注解了 lambda 表达式的形式参数声明。

  对于局部变量声明或 lambda 表达式的形式参数声明的注解，在二进制表示形式中从不保留。相反，如果注解接口指定了合适的保留策略，则在二进制表示中保留局部变量类型的注解或 lambda 表达式的形式参数类型的注解。

请注意，对注解接口用@Target(java.lang.annotation.ElementType.LOCAL_VARIABLE) 和 @Retention(java.lang.annotation.RetentionPolicy.CLASS) 或 @Retention(java.lang.annotation.RetentionPolicy.RUNTIME)进行元注解并不非法。

如果 m 有一个值为 java.lang.annotation.RetentionPolicy.RUNTIME 的元素， JavaSE 平台的反射库必须在运行时使 a 是可用的。

如果 A 没有对应于 java.lang.annotation.Retention 的元注解, 那么 Java 编译器必须将 A 视为具有对应于 java.lang.annotation.Retention 的（元）注解，该注解有一个元素值为 java.lang.annotation.RetentionPolicy.CLASS。

9.6.4.3 @Inherited

注解接口 java.lang.annotation.Inherited 用于指示与给定注解接口相对应的类 C 上的注解由 C 的子类继承。

9.6.4.4 @Override

当程序员打算重写方法声明时，有时会使用过载，从而导致一些微妙的问题。注解接口 Override 支持此类问题的早期检测。

经典的例子涉及 equals 方法。程序员在类 Foo 中编写以下代码:

```
public boolean equals(Foo that) { ... }
```

当他们打算写作:

```
public boolean equals(Object that) { ... }
```

这是完全合法的，但类 Foo 从 Object 继承了 equals 实现，这可能会导致一些微妙的错误。

如果类或接口 Q 中的方法声明用@Override 注解，则以下三个条件之一必须为真，否则会发生编译时错误:

- 这个方法从 Q 重写了一个在 Q 的超类型中声明的方法(§8.4.8.1，§9.4.1.1)
- 该方法重写等价于 Object 的公共方法(§4.3.2, §8.4.2)
- Q 是一个记录类(§8.10)，方法是 Q 的一个记录组件的访问方法(§8.10.3)

这种行为与 Java SE 5.0 不同，在 Java SE 5.0 中，如果@Override 应用于实现超接口方法的方法，而该方法不存在于超类中，则只会导致编译时错误。

关于重写 Object 的公共方法的子句是由在接口中使用@Override 引起的。考虑以下声明:

```
class Foo { @Override public int hashCode() {..} }
interface Bar { @Override int hashCode(); }
```

根据第一个子句，在类声明中使用@Override 是合法的，因为 Foo.hashCode 从 Foo 重写方法 Object.hashCode。

对于接口声明，考虑接口具有公共抽象成员，这些成员对应于 Object 的公共成员（§9.2）。如果接口选择显式声明它们（即声明重写等价于 Object 的公共方法的成员），则该接口被视为重写它们，并且

允许使用@Override。

但是，考虑一个试图在 clone 方法上使用@Override 的接口：（本例中也可以使用 finalize）

```
interface Quux { @Override Object clone(); }
```

因为 Object.clone 不是公共国的，在 Quux 中没有隐式声明为 clone 的成员。因此，Quux 中 clone 的显式声明不被视为“实现”任何其他方法，使用@Override 是错误的。(事实上，Quux.clone 是公共的是不相关的。)

相反，声明 clone 的类声明只是重写 Object.clone，所以可以使用@Override:

```
class Beep { @Override protected Object clone() {..} }
```

关于记录类的子句是由于在记录声明中@Override 的特殊含义造成的。也就是说，它可以用来指定方法声明是记录组件的访问器方法。考虑以下记录声明:

```
record Roo(int x) {
    @Override
    public int x() {
       return Math.abs(x);
    }
}
```

对访问器方法 int x()使用@Override 确保如果记录组件 x 被修改或删除，那么相应的访问器方法也必须被修改或删除。

9.6.4.5@SuppressWarnings

Java 编译器越来越能够发出有用的“lint-like”警告。为了鼓励使用此类警告，当程序员知道警告不适当时，应该有某种方法禁用程序部分中的警告。

注解接口 SuppressWarnings 支持程序员控制 Java 编译器发出的警告。它定义了一个元素，该元素是一个字符串数组。

如果一个声明用@SuppressWarnings(value = {S_1, ..., S_k})注解，那么如果该警告是由注解声明或其任何部分生成的，则 Java 编译器必须抑制（即，不报告）由 $S_1 \cdots S_k$ 之一指定的任何警告。

Java 编程语言定义了四种可以由@SuppressWarnings 指定的警告:

- 未经检查的警告（§4.8、§5.1.6、§5.1.9、§8.4.1、§8.4.8.3、§15.12.4.2、§15.13.2、§15.27.3）由字符串“unchecked”指定。
- 弃用警告（§9.6.4.6）由字符串“deprecation”指定。
- 移除警告（§9.6.4.6）由字符串“removal”指定。
- 预览警告（§1.5）由字符串“preview”指定。

任何其他字符串都指定非标准警告。Java 编译器必须忽略它无法识别的任何此类字符串。

鼓励编译器供应商记录他们支持的@SuppressWarnings 字符串，并进行合作以确保在多个编译器中识别相同的字符串。

9.6.4.6 @Deprecated

程序员有时不愿意使用某些程序元素（模块、类、接口、字段、方法和构造函数），因为它们被认为是危险的，或者存在更好的替代方案。不推荐使用的注释接口允许编译器警告这些程序元素的使用。

不推荐使用的程序元素是其声明用@Deprecated 注解的模块、类、接口、字段、方法或构造函数。不推荐使用程序元素的方式取决于注解的 forRemoval 元素的值：

- 如果 forRemoval=false（默认值），则通常不推荐使用程序元素。

  通常已弃用的程序元素不打算在未来的版本中删除，但程序员仍然应该迁移，不再使用它。

- 如果 forRemoval=true, 那么程序元素最终将被弃用。

  一个最终被弃用的程序元素将在未来的版本中被删除。当升级到新版本时，程序员应该停止使用它，否则将面临源代码和二进制代码不兼容的风险(§13.2)。

当在程序元素的声明中(无论是显式声明还是隐式声明)使用(按名称重写、调用或引用)通常已弃用的程序元素时，Java 编译器必须产生弃用警告，除非：

- 该用法在一个声明中，该声明本身通常或最终被弃用；或者
- 该用途在声明中，该声明被注解为抑制弃用警告（§9.6.4.5）；或者
- 出现用法的声明和通常不推荐使用的程序元素的声明都在同一个最外层类中；或者
- 该用法在导入通常不推荐使用的类、接口或成员的导入声明中；或者
- 该用法在 exports 或 opens 指令中 (§7.7.2)。

当在程序元素声明（无论是显式声明还是隐式声明）中使用（按名称重写、调用或引用）一个最终弃用的程序元素时，Java 编译器必须生成删除警告，除非：

- 该用途在声明中，声明被注解以抑制移除警告（§9.6.4.5）；或者
- 出现用法的声明和最终不推荐使用的程序元素的声明都在同一个最外层类中；或者
- 使用在导入声明中，该声明导入了最终不推荐使用的类、接口或成员；或者
- 该用法在 exports 或 opens 指令中。

  终端弃用非常紧急，即使使用的元素本身已弃用，使用终端弃用的元素也会导致删除警告，因为无法保证两个元素将同时删除。要消除警告但继续使用元素，程序员必须通过@SuppressWarnings 注解手动确认风险。

在以下情况下，不会产生弃用警告或删除警告：

- 使用局部变量或形式参数（由名称引用），即使局部变量或形式参数的声明用@Deprecated 注解。

- 使用包的名称（由限定类型名称、导入声明或 exports 或 opens 指令引用），即使包的声明用@Deprecated 注解。
- 限定的 exports 或 opens 指令使用模块的名称，即使友元模块的声明用@Deprecated 注解。

导出或打开包的模块声明通常由控制包声明的程序员或团队控制。因此，当包被模块声明导出或打开时，警告包声明带有@Deprecated 注解是没有什么好处的。相比之下，向友方模块导出或打开包的模块声明通常不受控制友方模块的程序员或团队的控制。简单地导出或打开包不会使模块声明依赖于友元模块，所以如果友元模块已弃用，警告几乎没有价值;模块声明的程序员几乎总是希望取消这样的警告。

唯一可能导致弃用警告或移除警告的隐式声明是容器注解(§9.7.5)。也就是说，如果T是一个可重复的注解接口，TC 是它的包含注解接口，并且 TC 已被弃用，那么重复@T 注解将导致警告。该警告是由于隐式的@TC 容器注解造成的。强烈不建议在不废弃相应的可重复注解接口的情况下废弃包含注解接口。

9.6.4.7 `@SafeVarargs`

带有不可具体化元素类型的可变参数(§4.7)可能导致堆污染(§4.12.2)，并产生编译时未检查警告(§5.1.9)。然而，如果可变方法体相对于可变参数表现良好，那么这样的警告是没有信息的。

注解接口 SafeVarargs 用于注解方法或构造函数声明时，会做出程序员断言，防止 Java 编译器对可变方法或构造函数的声明或调用报告未检查的警告，否则编译器会报告未检查的警告，因为可变参数具有不可具体化的元素类型。

注解@SafeVarargs 具有非局部效果，因为除了可变方法本身声明的未检查警告外，它还抑制了方法调用表达式中的未检查警告(§8.4.1)。相反，注解@SuppressWarnings("unchecked")具有局部效果，因为它仅抑制与方法声明相关的未经检查的警告。

@SafeVarargs 的规范目标是类似 java.util.Collections.addAll 的方法，其声明以下面开头：

```
public static <T> boolean
  addAll(Collection<? super T> c, T... elements)
```

可变参数声明了类型 T[]，它是不可具体化的。但是，该方法基本上只是从输入数组中读取数据，并将元素添加到集合中，这两种操作对于数组来说都是安全的。因此，在 java.util.Collections.addAll 的方法调用表达式中任何编译时未检查的警告都是可以论证的虚假和无信息的。将@SafeVarargs 应用于方法声明可以防止在方法调用表达式中产生这些未检查的警告。

如果使用@SafeVargs 注解固定参数的方法或构造函数声明，则会出现编译时错误。

如果使用@SafeVarargs 注解了既不是 static、也不是 final、也不是 private 的可变参数方法声明，则是编译时错误。

由于@SafeVarargs 仅适用于静态方法、final 和/或私有实例方法以及构造函数，因此在发生方法重写的情况下，注解不可用。注解继承只适用于类上的注解(不适用于方法、接口或构造函数)，因此@SafeVargs 样式的注解不能通过类中的实例方法或通过接口传递。

9.6.4.8 `@Repeatable`

在可重复注解接口的声明中使用注解接口 java.lang.Annotation.Repeatable 来指示其包含的注解接口(§9.6.3)。

> 请注意，A 的声明上的@Repeatable 元注解(表示 AC)不足以使 AC 成为 A 的包含注解接口。AC 被认为是 A 的包含注解接口的形式良好性规则有很多。

9.6.4.9 `@FunctionalInterface`

注解接口 FunctionalInterface 用于指示该接口应为函数式接口(§9.8)。它有助于及早发现函数式接口中出现的或函数式接口继承的不适当的方法声明。

如果接口声明使用@FunctionalInterface 注解，但实际上不是函数式接口，则会出现编译时错误。

由于某些接口附带地具有函数性，因此没有必要也不希望所有的函数式接口声明都使用@FunctionalInterface 进行注解。

9.7 注解

注解是将信息与程序元素相关联的标记，但在运行时不起作用。注解表示注解接口的特定实例(§9.6)，通常为该接口的元素提供值。

有三种注解。第一类是最普遍的，而其他类型只是第一类的简写。

Annotation:
  NormalAnnotation
  MarkerAnnotation
  SingleElementAnnotation

普通注解在§9.7.1 中描述，标记注解在§9.7.2 中描述，单元素注解在§9.7.3 中描述。在程序中，注解可以出现在不同的语法位置，如§9.7.4 所述。 可能出现在一个位置的同一接口的注解的数量是由接口声明决定的，如§9.7.5 所述。

9.7.1 普通注解

普通注解指定注解接口的名称以及可选的以逗号分隔的元素-值对列表。每一对都包含一个与注解接口(§9.6.1)的一个元素相关联的元素值。

NormalAnnotation:
  `@` *TypeName* `(` *[ElementValuePairList]* `)`

ElementValuePairList:
  ElementValuePair {`,` *ElementValuePair*}

ElementValuePair:
  Identifier `=` *ElementValue*

ElementValue:
ConditionalExpression
ElementValueArrayInitializer
Annotation

ElementValueArrayInitializer:
{ *[ElementValueList] [,]* }

ElementValueList:
ElementValue {, ElementValue}

注意@符号(@)是自身的标记(§3.11)。可以在它和 TypeName 之间放置空格，但出于样式的考虑，不建议这样做。

TypeName 指定与注解对应的注解接口。该注解被称为该接口的注解。

TypeName 必须命名一个可访问的注解接口(§6.6)，否则会发生编译时错误。

元素-值对中的 Identifier 必须是注解接口的一个元素(即方法)的简单名称，否则将发生编译时错误。

这个方法的返回类型定义了元素-值对的元素类型。

如果元素类型是数组类型，则不需要使用花括号来指定元素-值对的元素值。如果元素值不是 ElementValueArrayInitializer，则唯一元素为元素值的数组值与该元素相关联。如果元素值是 ElementValueArrayInitializer，则由 ElementValueArrayInitializer 表示的数组值与该元素相关联。

如果元素类型与元素值不相称，则为编译时错误。当且仅当下列条件之一成立时，元素类型 T 才与元素值 v 相称：

• T 是数组类型 E[]，并且：

- 如果 v 是 ConditionalExpression 或者 Annotation，则 v 与 E 相称；否则
- 如果 v 是 ElementValueArrayInitializer，则 v 包含的每个元素值与 E 相称。

  ElementValueArrayInitializer 类似于普通数组初始化器(§10.6)，除了 ElementValueArrayInitializer 可以在语法上包含注解以及表达式和嵌套的初始化器。然而，嵌套初始化器在 ElementValueArrayInitializer 中是语义上不合法的，因为它们永远不会与注解接口声明中的数组类型元素相称(不允许嵌套数组类型)。

• T 不是数组类型，并且 v 的类型与 T 是赋值兼容的(§5.2)，并且：

- 如果 T 是一个原生类型或 String，那么 v 是一个常量表达式(§15.29)。
- 如果 T 是 Class 或 Class 的调用(§4.5)，那么 v 是类字面量(§15.8.2)。
- 如果 T 是一个枚举类类型(§8.9)，那么 v 是一个枚举常量(§8.9.1)。
- v 不是 null。

请注意，如果 T 不是数组类型或注解接口，则元素值必须是 ConditionalExpression(§15.25)。使用 ConditionalExpression 而不是更通用的类似 *Expression 的*产品的表达式是一种语法技巧，可以防止赋值表达式作为元素值。由于赋值表达式不是常量表达式，因此它不能是与原生或字符串类型元素的元素值相称的元素值。

普通注解必须包含对应注解接口的每个元素的元素-值对，除非那些元素具有默认值，否则将发生编译时错误。

对于具有默认值的元素，普通注解可以(但不是必需)包含元素-值对。

注解中的元素-值对按照与注解接口声明中相应元素相同的顺序呈现，这是惯例，但不是必需的。

注解接口声明上的注解称为元注解。

接口 A 的注解可以作为接口 A 本身声明的元注解出现。更一般地说，允许在“注解”关系的传递闭包中存在循环。

例如，可以用接口 T 的元注解对注解接口 S 的声明进行注解，也可以用接口 S 的元注解注解 T 自己的声明。预定义的注解接口(§9.6.4)包含几个这样的循环。

例子 9.7.1-1. 普通注解

下面是一个使用§9.6.1 中的注解接口的普通注解的示例:

```
@RequestForEnhancement(
   id  =2868724,
   synopsis = "Provide time-travel functionality",
   engineer = "Mr. Peabody",
   date = "4/1/2004"
)
public static void travelThroughTime(Date destination) { ... }
```

下面是一个利用默认值的普通注解的例子，使用§9.6.2 中的注解接口:

```
@RequestForEnhancement(
   id  = 4561414,
   synopsis = "Balance the federal budget"
)
public static void balanceFederalBudget() {
    throw new UnsupportedOperationException("Not implemented");
}
```

9.7.2 标记注解

标记注解是为与标记注解接口一起使用而设计的速记(§9.6.1)。

MarkerAnnotation:
`@` *TypeName*

它是普通注解的简写:

```
@TypeName()
```

对带有元素的注解接口使用标记注解是合法的，只要所有元素都有缺省值(§9.6.2)。

例子 9.7.2-1. 标记主机

以下是使用§9.6.1 中的 Preliminary 标记注解接口的示例:

```
@Preliminary public class TimeTravel { ... }
```

9.7.3 单元素注解

单元素注解是专为使用单元素注解接口而设计的速记形式(§9.6.1)。

SingleElementAnnotation:
`@` *TypeName* `(` *ElementValue* `)`

它是普通注解的简写:

```
@TypeName(value = ElementValue)
```

对于包含多个元素的注解接口，使用单元素注解是合法的，只要一个元素被命名为 value，而其他所有元素都有默认值(§9.6.2)。

例子 9.7.3-1. 单元素注解

以下注解都使用了§9.6.1 中的单元素注解接口。

下面是一个单元素注解的例子:

```
@Copyright("2002 Yoyodyne Propulsion Systems, Inc.")
public class OscillationOverthruster { ... }
```

下面是一个数组值单元素注解的示例:

```
@Endorsers({"Children", "Unscrupulous dentists"})
public class Lollipop { ... }
```

下面是一个单元素数组值的单元素注解的示例:(注意，大括号被省略了)

```
@Endorsers("Epicurus")
public class Pleasure { ... }
```

下面是一个带有 Class 类型元素的单元素注解示例，该元素的值由边界通配符限制。

```
class GorgeousFormatter implements Formatter { ... }

@PrettyPrinter(GorgeousFormatter.class)
public class Petunia { ... }

// Illegal; String is not a subtype of Formatter @PrettyPrinter(String.class)
public class Begonia { ... }
```

下面是一个包含普通注解的单元素注解示例:

```
@Author(@Name(first = "Joe", last = "Hacker"))
public class BitTwiddle { ... }
```

下面是一个使用在注解接口声明中定义的枚举类的单元素注解示例:

```
@Quality(Quality.Level.GOOD)
public class Karma { ... }
```

9.7.4 注解可能出现在哪

声明注解是一个应用于声明的注解，它的注解接口适用于该声明所代表的声明上下文中(§9.6.4.1);或应用于类、接口或类型参数声明的注解，其注解接口适用于类型上下文(§4.11)。

类型注解是应用于类型(或类型的任何部分)的注解，其注解接口适用于类型上下文中。

例如，给定字段声明:

```
@Foo int f;
```

如果 Foo 被@Target(ElementType.FIELD)元注解，@Foo 是对 f 的声明注解，如果 Foo 被@Target(ElementType.TYPE_USE)元注解，则是对 int 的类型注释。@Foo 可以同时是声明注解和类型注解。

类型注解可以应用于数组类型或其中的任何组件类型(§10.1)。例如，假设 A、B 和 C 是用@Target(ElementType.TYPE_USE)元注解的注解接口，然后给定字段声明:

```
@C int @A [] @B [] f;
```

@A 应用于数组类型 int[][], @B 应用于组件类型 int[],@C 应用于元素类型 int。更多示例见§10.2。

这种语法的一个重要属性是，在两个声明中，只有数组级别的数量不同，类型左侧的注解指向相同的类型。例如, 在以下所有声明中@C 应用于类型 int:

```
@C int f;
@C int[] f;
@C int[][] f;
```

习惯上(虽然不是必需的)，将声明注解写在所有其他修饰符之前，并将类型注解紧接在它们应用的类型之前。

注解可能出现在程序中的语法位置，它可能适用于声明或类型，或两者都适用。这可以发生在六种声明上下文中，其修饰符紧接在被声明实体的类型之前:

- 方法声明 (包括注解接口的元素)
- 构造函数声明

- 字段声明 (包括枚举常量)
- 形式和异常参数声明
- 局部变量声明
- 记录组件声明

Java 编程语言的语法明确地将这些位置的注解视为声明的修饰符(§8.3)，但这纯粹是一种语法问题。注解是应用于声明还是被声明实体的类型-因此，无论该注解是声明注解还是类型注解-取决于注解接口的适用性:

• 如果注解的接口适用于与声明相对应的声明上下文中，而不适用于类型上下文中，则认为注解只适用于声明。

• 如果注解的接口适用于类型上下文中，而不适用于与声明对应的声明上下文中，则认为该注解只适用于最接近该注解的类型。

• 如果注解的接口适用于与声明相对应的声明上下文中和类型上下文中，则认为注解既适用于声明，也适用于最接近注解的类型。

在上述第二种和第三种情况中，最接近注解的类型确定如下:

• 如果注解出现在 void 方法声明或使用 var 的局部变量声明之前(§14.4)，则没有最接近的类型。如果认为注解的接口只适用于最接近注解的类型，则会发生编译时错误。

• 如果注解出现在构造函数声明之前，那么最接近的类型就是新构造对象的类型。新构造的对象的类型是立即包含构造函数声明的类型的完全限定名。在该完全限定名称中，注解应用于构造函数声明所指示的简单类型名。

• 在所有其他情况下，最接近的类型是在源代码中为声明的实体编写的类型;如果该类型是数组类型，则元素类型被认为最接近注解。

> 例如, 字段声明@Foo public static String f;最接近@Foo 的类型是 String。(如果字段声明的类型被写做 java.lang.String, 那么 java.lang.String 会是@Foo 最接近的类型, 以后的规则将禁止将类型注解应用于包名 java。) 在泛型方法声明 @Foo <T> int[] m() {...}中, 为声明的实体编写的类型为 int[], 因此@Foo 使用与元素类型 int。
>
> 不使用 var 的局部变量声明类似于 lambda 表达式的形式参数声明，在源代码中都允许声明注解和类型注解，但只有类型注解可以存储在 class 文件中。

如果接口 A 的注解在语法上是以下语句的修饰符，则会出现编译时错误:

- 模块声明, 但 A 不适用于模块声明。
- 包声明, 但 A 不适用于包声明。
- 类或接口声明, 但 A 不适用于类型声明或类型上下文中; 或者

  注解接口声明, 但 A 不适用于注解接口声明或类型声明或在类型上下文中。

- 方法声明(包含注解接口的元素), 但 A 不适用于方法声明或类型上下文中。
- 构造函数声明, 但 A 不适用于构造函数声明或类型上下文中。
- 泛型类、接口、方法或构造函数的类型参数声明, 但 A 不适用于类型参数声明或类型上下文中。
- 字段声明 (或枚举常量), 但 A 不适用于字段声明或类型上下文中。
- 形式或异常参数声明, 但 A 不适用于形式参数和异常参数声明或在类型上下文中。
- 一个 receiver 参数，但 A 不适用于类型上下文。
- 在语句或模式中声明的局部变量，但 A 不适用于局部变量声明或类型上下文中。
- 一个记录组件, 但是 A 不适用于记录组件声明、字段声明、方法声明、形式参数和异常参数声明，或在类型上下文中。

> 这 11 条条款中有 6 条提到“……或者在类型上下文中”，因为它们描述了本节前面提到的六个语法位置，其中注解可以合理地应用于声明或类型。此外，11 个子句中的两个-用于类和接口声明，以及用于类型参数声明-提到了“……或者在类型上下文中”，因为有时可以方便地将接口用@Target(ElementType.TYPE_USE)元注解的注解应用到类、接口或类型参数的声明中(因此，适用于类型上下文中)。

如果以下两个条件都为真，则类型注解是允许的:

- 与注解最接近的简单名称被分类为 TypeName, 而不是 PackageName。
- 如果与注解最接近的简单名称后面跟着“.”，并且另一个 TypeName-也就是说，注解以 @Foo T.U 的形式出现-那么 U 表示 T 的内部类。

> 第二个子句背后的直觉是如果 Outer.this 在由 Outer 包围的嵌套类中是合法的，那么 Outer 可以被注解，因为它表示在运行时某些对象的类型。另一方面，如果 Outer.this 是不合法的-因为它出现的类在运行时没有 Outer 的封闭实例-那么 Outer 可能不会被注解，因为它在逻辑上只是一个名称，类似于完全限定类型名中的包名组件。
>
> 例如，在下面的程序中，不可能在 B 的主体中编写 A.this，因为 B 没有词法上封闭的实例。因此，不可能将@Foo 应用于类型 A.B 中的 A，因为 A 在逻辑上只是一个名称，而不是一个类型。

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class A {
    static class B {}
}

@Foo A.B x; // Illegal
```

> 另一方面，在接下来的程序中，可以在 D 的主体中编写 C.this。因此，可以将@Foo 应用于类型 C.D 中的 C，因为 C 在运行时表示某些对象的类型。

```
@Target(ElementType.TYPE_USE)
@interface Foo {}
```

```
class Test {
    static class C {
        class D {}
    }

    @Foo C.D x; // Legal
}
```

如果接口 A 的注解应用于类型上下文中类型的最外层，而 A 不适用于类型上下文中或声明上下文中(如果有的话)，它们占据相同的语法位置，则会出现编译时错误。

如果接口 A 的注解在类型上下文中应用于类型的一部分(也就是说，不是最外层)，而 A 在类型上下文中不适用，则会出现编译时错误。

如果接口 A 的注解在类型上下文中应用于类型(或类型的任何部分)，并且 A 在类型上下文中适用，但注解不被允许，那么这是一个编译时错误。

例如，假设一个注解接口 TA 只使用@Target(ElementType.TYPE_USE)进行元注解。术语@TA java.lang.Object 和 java.@TA lang.Object 是非法的，因为@TA 最接近的简单名称被归类为包名。另一方面, java.lang.@TA Object 是合法的。

请注意，这些非法术语"无论到哪"都是非法的。禁止注解包名称的禁令适用范围很广：适用于仅为类型上下文的位置，如 class ... extends @TA java.lang.Object {...}，以及既是声明上下文又是类型上下文的位置，如@TA java.lang.Object f; 。(由于包、类、接口和类型参数声明只引入简单的名称，因此不存在仅作为声明上下文的位置可以注解包名称。)

如果 TA 另外使用@Target(ElementType.FIELD)进行元注解，则术语@TA java.lang.Object 在既是声明上下文又是类型上下文的位置中是合法的，例如字段声明@TA java.lang.Object f;。这里，@TA 被认为适用于 f 的声明(而不是类型 java.lang.Object)，因为 TA 适用于字段声明上下文。

9.7.5 同一个接口的多个注解

如果同一接口 A 的多个注解出现在声明上下文或类型上下文中，则为编译时错误，除非 A 是可重复的(§9.6.3)，并且 A 和 A 的包含注解接口都适用于声明上下文或类型上下文(§9.6.4.1)。

尽管不是必需的，但通常情况下，同一接口的多个注解连续出现。

如果声明上下文或类型上下文具有可重复注解接口 A 的多个注解，则就好像该上下文没有接口 A 的显式声明的注解以及 A 的包含注解接口的一个隐式声明的注解。

隐式声明的注解称为容器注解，出现在上下文中的接口 A 的多个注解称为基本注解。容器注解的(数组类型的)value 元素的元素都是从左到右的基本注解，它们出现在上下文中。

如果在声明上下文或类型上下文中存在可重复注解接口 A 的多个注解以及 A 的包含注解接口的任何注解，则为编译时错误。

换句话说，在与其容器相同的接口的注解也出现的情况下，不可能重复注解。这将禁止诸如以下内容的迟钝代码:

```
@Foo(0) @Foo(1) @FooContainer({@Foo(2)}) class A {}
```

如果这段代码是合法的，那么就需要多层的包含:首先接口 Foo 的基本注解将被接口 FooContainer 的隐式声明的容器注解所包含，然后该注解和接口 FooContainer 的显式声明的注解将被包含在另一个隐式声明的注解中。在 Java 编程语言的设计者的判断中，这种复杂性是不可取的。另一种方法是将接口 Foo 的基本注解视为与@Foo(2)一起出现在显式的@FooContainer 注解中，这是不可取的，因为它可能会改变反射程序解释@FooContainer 注解的方式。

如果在声明上下文中或类型上下文中，存在可重复注解接口 A 的一个注解和包含注解接口 A 的多个注解，则会出现编译时错误。

该规则允许以下代码:

```
@Foo(1) @FooContainer({@Foo(2)})
class A {}
```

由于可重复注解接口 Foo 只有一个基本注解，所以没有隐式声明容器注解，即使 FooContainer 是 Foo 的包含注解接口。但是，重复 FooContainer 接口的注解，如下:

```
@Foo(1) @FooContainer({@Foo(2)}) @FooContainer({@Foo(3)})
class A {}
```

是禁止的, 即使 FooContainer 可以通过自己的包含注解接口重复使用。当底层可重复接口的注解存在时，重复本身就是容器的注解是不明智的。

9.8 函数式接口

函数式接口是没有声明为密封的接口，只有一个抽象方法(Object 的方法除外)，因此表示单个函数契约。这个“单一”方法可以采用多个抽象方法的形式，这些抽象方法具有从超接口继承的重写等价签名;在这种情况下，继承的方法在逻辑上表示单个方法。

对于没有声明为密封的接口 I，让 M 是 I 的抽象方法的集合，这些抽象方法不具有与 Object 类的任何公共实例方法相同的签名(§4.3.2)。然后，I 是一个函数式接口，如果 M 中存在一个方法 m，以下两个都是真的:

- m 的签名是 M 中每个方法签名的子签名(§8.4.2)。
- 对 M 里的每个方法，m 都是返回类型可替换的 (§8.4.5)。

除了通过声明和实例化一个类(§15.9)来创建接口实例的通常过程之外，函数式接口的实例还可以通过方法引用表达式和 lambda 表达式来创建(§15.13 和§15.27)。

函数式接口的定义排除了接口中同时也是 Object 中的公共方法的方法。这是为了允许对像 java.util.Compator<T>这样的接口进行函数处理，该接口声明了多个抽象方法，其中只有一个是真正的“新”方法-int Compare(T，T)。另一个 - boolean equals(Object) - 是抽象方法的显式声明，否则将在接口中隐式声明(§9.2)，并由实现该接口的每个类自动实现。

请注意，如果 Object 的非公共方法(如 clone())在接口中显式声明为公共方法，则它们不会由实现该接口的每个类自动实现。从 Object 继承的实现是受保护的，而接口方法是公共的，因此实现接口的唯一方法是让类用公共方法重写非公共 Object 方法。

例子 9.8-1. 函数式接口

函数式接口的一个简单示例是：

```
interface Runnable {
   void run();
}
```

以下接口不是函数式接口，因为它未声明任何已不是 Object 成员的内容：

```
interface NonFunc {
   boolean equals(Object obj);
}
```

但是，它的子接口可以通过声明一个不是 Object 成员的抽象方法来成为函数式接口：

```
interface Func extends NonFunc {
   int compare(String o1, String o2);
}
```

类似地，众所周知的接口 java.util.Comparator<T>是函数式接口，因为它有一个抽象的非对象方法：

```
interface Comparator<T> {
   boolean equals(Object obj);
   int compare(T o1, T o2);
}
```

以下接口不是函数式接口，因为虽然它只声明了一个不是 Object 成员的抽象方法，但它声明了两个不是 Object 公共成员的抽象方法：

```
interface Foo {
    int m();
    Object clone();
}
```


例子 9.8-2. 函数式接口和擦除

在下面的接口层次结构中，Z 是一个函数式接口，因为虽然它继承了两个不是 Object 成员的抽象方法，但它们具有相同的签名，因此继承的方法在逻辑上表示单个方法：

```
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<String> arg); }
interface Z extends X, Y {}
```

类似地，Z 是以下接口层次结构中的函数式接口，因为 Y.m 是 X.m 的子签名，并且与 X.m 是返回类型可替换的：

```
interface X { Iterable m(Iterable<String> arg); }
interface Y { Iterable<String> m(Iterable arg); }
interface Z extends X, Y {}
```

函数式接口的定义尊重这样一个事实，即一个接口不能有两个彼此不是子签名但具有相同擦除的成员(§9.4.1.2)。因此，在以下三个接口层次结构中，Z 导致编译时错误，Z 不是函数式接口：(因为它的抽象成员都不是所有其他抽象成员的子签名)。

```
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<Integer> arg); }
interface Z extends X, Y {}

interface X { int m(Iterable<String> arg, Class c); }
```

```
interface Y { int m(Iterable arg, Class<?> c); } interface Z
extends X, Y {}

interface X<T> { void m(T arg); }
interface Y<T> { void m(T arg); }
interface Z<A, B> extends X<A>, Y<B> {}
```

类似地，“函数式接口”的定义尊重这样一个事实，即如果一个接口对所有其他接口是返回类型可替换的，则该接口只能有具有重写等价签名的方法。因此，在下面的接口层次结构中，Z 导致编译时错误，Z 不是函数式接口：(因为它的抽象成员对所有其他抽象成员都不是返回类型可替换的)。

```
interface X { long m(); }
interface Y { int m(); }
interface Z extends X, Y {}
```

在下面的示例中，Foo<T,N>和 Bar 的声明是合法的：在每个声明中，名为 m 的方法不是彼此的子签名，但具有不同的擦除。尽管如此，每个方法都不是子签名这一事实意味着 Foo<T，N>和 Bar 不是函数式接口。然而，Baz 是一个函数式接口，因为它从 Foo<Integer，Integer>继承的方法具有相同的签名，因此在逻辑上表示单个方法。

```
interface Foo<T, N extends Number> {
    void m(T arg);
    void m(N arg);
}
interface Bar extends Foo<String, Integer> {}
interface Baz extends Foo<Integer, Integer> {}
```

最后，以下示例演示了与上面相同的规则，但使用的是泛型方法：

```
interface Exec { <T> T execute(Action<T> a); }
  // Functional

interface X { <T> T execute(Action<T> a); }
interface Y { <S> S execute(Action<S> a); }
interface Exec extends X, Y {}
  // Functional: signatures are logically "the same"

interface X { <T> T execute(Action<T> a); }
interface Y { <S,T> S execute(Action<S> a); }
interface Exec extends X, Y {}
  // Error: different signatures, same erasure
```

例子 9.8-3. 泛型函数式接口

函数式接口可以是泛型，例如 java.util.function.Predicate<T>。这样的函数式接口可以以一种产生不同抽象方法的方式进行参数化-即，不能用单个声明合法重写的多个方法。例如：

```
interface I { Object m(Class c); }
interface J<S> { S m(Class<?> c); }
interface K<T> { T m(Class<?> c); }
interface Functional<S,T> extends I, J<S>, K<T> {}
```

Functional<S,T>是一个函数式接口 - I.m 对 J.m 和 K.m 是返回类型可替代的- 但是函数式接口类型 Functional<String,Integer>显然不能用单一方法来实现。然而，作为函数式接口类型的 Functional<S,T>的其他参数化也是可能的。

函数式接口的声明允许在程序中使用函数式接口类型。有四种函数式接口类型:

- 非泛型(§6.1)函数式接口的类型
- 一种参数化类型，它是泛型函数接口的参数化(§4.5)
- 泛型函数式接口的原始类型(§4.8)
- 一个交集类型(§4.9)诱导了一个概念上的函数式接口

  在特殊情况下，将交集类型视为函数式接口类型是很有用的。通常，这看起来像是函数式接口类型与一个或多个标记接口类型的交集，例如 Runnable&java.io.Serializable。这样的交集可以用在强制转换中(§15.16)，强制 lambda 表达式符合某种类型。如果交集中的接口类型之一是 java.io.Serializable，对序列化的特殊运行时支持被触发(§15.27.4)。

9.9 函数类型

函数式接口 I 的函数类型是一种方法类型(§8.2)，可以用来重写(§8.4.8)I 的抽象方法。

设 M 为 I 定义的抽象方法的集合。I 的函数类型包括:

• 类型参数、形式参数类型和返回类型:

设 m 为 M 中的一个方法:

1. M 中每个方法签名的子签名;而且
2. 返回类型 R(可能是 void)，其中 R 与 M 中每个方法的返回类型相同，或者 R 是引用类型，是 M 中每个方法返回类型的子类型(在适应了任何类型参数(§8.4.4)后，如果两个方法有相同的签名)。

如果不存在这样的方法，则设 m 为 M 中的一个方法，具有:

1. M 中每个方法签名的子签名;而且
2. 一个返回类型，这样 m 对 M 中的每个方法都是返回类型可替换的(§8.4.5)。

函数类型的类型参数、形式参数类型和返回类型由 m 给出。

• throws 子句:

函数类型的 throws 子句派生自 M 中方法的 throws 子句，如下所示:

1. 如果函数类型是泛型的，则 throws 子句首先适应函数类型的类型参数(§8.4.4)。

   如果函数类型不是泛型的，但 M 中至少有一个方法是泛型的，则首先擦除 throws 子句。

2. 然后，函数类型的 throws 子句包含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。

- 对于每一个 throws 子句, E 是在该子句中命名的某种类型的子类型。

当 M 中的一些返回类型是原始的，而另一些不是时，函数类型的定义试图选择最具体的类型，如果可能的话。例如，如果返回类型是 LinkedList 和 LinkedList<String>，然后立即选择后者作为函数类型的返回类型。当没有最特定的类型时，定义通过查找最可替换的返回类型来进行补偿。例如，如果有第三个返回类型 List<？>，则其中一个返回类型不是其他类型的子类型(因为原始 LinkedList 不是 List<？>的子类型)；相反，选择 LinkedList<String>作为函数类型的返回类型是，因为它对 LinkedList 和 List<？>都是返回类型可替换的。

驱动函数类型抛出异常类型定义的目标是支持这样一种不变性，即带有所产生的 throws 子句的方法可以重写函数式接口的每个抽象方法。根据§8.4.6，这意味着函数类型抛出的异常不能比集合 M 中的任何单个方法抛出的异常“多”，因此我们寻找尽可能多的异常类型，这些异常类型被每个方法的 throws 子句“覆盖”。

函数式接口类型的函数类型指定如下:

• 如上面定义的，非泛型函数式接口 I 的类型的功能类型就是函数式接口 I 的函数类型。

• 参数化函数式接口类型的函数类型 I<A_1...A_n>, 其中 A_1...A_n 是类型，I 的相应的类型参数是 P_1...P_n, 通过将替换[P_1:=A_1，…，P_n:=A_n]应用于泛型函数式接口 I<P_1…P_n>的函数类型来得到。

• 参数化函数式接口类型 I<A_1...A_n>的函数类型, 其中一个或多个 A_1…A_n 是通配符,是 I 的非通配符参数化的函数类型 I<T_1...T_n>。非通配符参数确定如下。

假设 P_1...P_n 是 I 的类型参数，有对应的边界 B_1...B_n。对于所有 i (1 ⩽ i ⩽ n), T_i 是根据 A_i 的形式推导出来的:

- 如果 A_i 是一个类型, 那么 T = A_i。
- 如果 A_i 是一个通配符，并且对应的类型参数的边界 B_i 提到 P_1…P_n 之一，那么 T_i 没有定义，也没有函数类型。
- 否则:
  - > 如果 A_i 是无边界通配符?, 那么 T_i= B_i。
  - > 如果 A_i 是一个上界通配符 ? extends U_i, 那么 T_i = glb(U_i, B_i) (§5.1.10)。
  - > 如果 A_i 是一个下界通配符 ? super L_i, 那么 T_i = L_i。

• 泛型函数式接口的原始类型的函数类型 I<…>是对泛型函数式接口 I<…>的函数类型的擦除。

• 引起概念函数式接口的交集类型的函数类型就是概念函数式接口的函数类型。

例子 9.9-1. 函数类型

给定以下接口:

```
interface X  {void m()  throws  IOException; }
interface Y  {void m()  throws  EOFException; }
interface Z  {void m()  throws  ClassNotFoundException;}
```

以下的函数类型

```
interface XY extends X, Y {}
```

是:

```
()->void throws EOFException
```

以下的函数类型

```
interface XYZ extends X, Y, Z {}
```

是:

```
()->void (throws nothing)
```

给定以下接口:

```
interface A {
    List<String> foo(List<String> arg)
      throws IOException, SQLTransientException;
}
interface B {
    List foo(List<String> arg)
      throws EOFException, SQLException, TimeoutException;
}
interface C {
    List foo(List arg) throws Exception;
}
```

以下的函数类型

```
interface D extends A, B {}
```

是:

```
(List<String>)->List<String>
  throws EOFException, SQLTransientException
```

以下的函数类型

```
interface E extends A, B, C {}
```

是:

```
(List)->List throws EOFException, SQLTransientException
```

函数式接口的函数类型是不确定定义的：虽然 M 中的签名“相同”，但它们在语法上可能不同(例如，HashMap.Entry 和 Map.Entry)；返回类型可以是所有其他返回类型的子类型，但也可能有其他返回类型也是子类型(例如 List<?>和 List<? extends Object>)；抛出类型的顺序未指定。这些区别很微妙，但有时也很重要。然而，在 Java 编程语言中，函数类型的使用方式不会造成不确定性。注意，当有多个抽象方法时，“最具体的方法”的返回类型和 throws 子句也是不确定地定义的(§15.12.2.5)。

当泛型函数式接口使用通配符参数化时，有许多不同的实例化可以满足通配符并产生不同的函数类型。

例如，每一个 Predicate<Integer> (函数类型 Integer -> boolean)，Predicate<Number> (函数类型 Number -> boolean)，和 Predicate<Object> (函数类型 Object -> boolean) 都是一个 Predicate<? super Integer>。有时，可以从上下文中知道预期的函数类型(§15.27.3)，例如 lambda 表达式的参数类型。其他时候，有必要选择一个;在这些情况下，使用边界。(这个简单的策略不能保证结果类型将满足某些复杂边界，因此不支持所有复杂情况。)

例子 9.9-2. 泛型函数类型

函数类型可以是泛型的，就像函数式接口的抽象方法可以是泛型的一样。例如，如下接口层次结构:

```
interface G1 {
    <E extends Exception> Object m() throws E;
}
interface G2 {
    <F extends Exception> String m() throws Exception;
}
interface G extends G1, G2 {}
```

G 的函数类型是:

```
<F extends Exception> ()->String throws F
```

函数式接口的泛型函数类型可以通过方法引用表达式(§15.13)实现，但不能通过 lambda 表达式(§15.27)实现，因为没有泛型 lambda 表达式的语法。