第十三章

二进制兼容性

只要有源代码，Java 编程语言的开发工具应该在必要时支持自动重新编译。特定的实现还可以在版本数据库中存储类和接口的源和二进制表示，并实现一个 ClassLoader，通过向客户端提供类和接口的二进制兼容版本，使用数据库的完整性机制来防止链接错误。

被广泛分发的包、类和接口的开发人员面临着一系列不同的问题。Internet 是我们最喜欢的广泛分布系统的例子，在 Internet 中，自动重新编译直接或间接依赖于要更改的类或接口的预先存在的二进制文件通常是不切实际或不可能的。相反，该规范定义了一组允许开发人员对包或类或接口进行的更改，同时保持(不破坏)与现有二进制文件的兼容性。

在 SOM(Forman，Conner，Danforth，and Raper，Proceages of OOPSLA'95)中的发行版到发行版的二进制兼容性框架内，Java 编程语言二进制文件在作者确定的所有相关转换下都是二进制兼容的(关于实例变量的添加有一些警告)。使用他们的方案，下面是 Java 编程语言支持的一些重要的二进制兼容更改的列表：

- 重新实现现有方法、构造函数和初始化器以提高性能。
- 更改方法或构造函数以返回输入上的值，这些输入以前要么引发通常不应该发生的异常，要么通过进入无限循环或导致死锁而失败。
- 向现有类或接口添加新的字段、方法或构造函数。
- 删除类的私有字段、方法或构造函数。
- 更新整个包时，删除包中的包访问字段、方法或类和接口的构造函数。
- 对现有类或接口声明中的字段、方法或构造函数进行重新排序。
- 在类层次结构中向上移动方法。
- 对类或接口的直接超接口列表进行重新排序。
- 在类型层次结构中插入新的类或接口类型。

本章规定了所有实现所保证的二进制兼容性的最低标准。当混合了类和接口的二进制文件时，Java 编程语言保证了兼容性，这些类和接口的二进制文件不知道来自兼容的来源，但其源代码已经以此处描述的兼容方式进行了修改。请注意，我们讨论的是应用程序版本之间的兼容性。有关 Java SE 平台各版本之间的兼容性的讨论超出了本章的范围。

我们鼓励开发系统提供工具，提醒开发人员更改对无法重新编译的已有二进制文件的影响。

本章首先规定了 Java 编程语言的任何二进制格式必须具备的一些属性(§13.1)。接下来，它定义了二进制兼容性，解释了它是什么和不是什么(§13.2)。它最后列举了对包(§13.3)、类(§13.4)和接口(§13.5)的大量可能的更改，指定了这些更改中的哪些保证保持二进制兼容性，哪些不保证。

有时，形式：(JVMS§X.Y)的引用被用来表示 Java 虚拟机规范 Java SE 19 版中的概念。

13.1 二进制的形式

必须将程序编译成 Java 虚拟机规范 Java SE 19 版指定的 class 文件格式，或者编译成可以由用 Java 编程语言编写的类加载器映射到该格式的表示形式。

与类或接口声明对应的 class 文件必须具有某些属性。这些属性中有许多是专门为支持源代码转换而选择的，以保持二进制兼容性。需要的属性包括：

1. 类或接口必须使用其二进制名称命名，该名称必须满足以下约束：

   • 顶级类或接口的二进制名称(§7.6)是其规范名称(§6.7)。

   • 成员类或接口的二进制名(§8.5、§9.5)由其直接封闭类或接口的二进制名组成，后跟$，然后是成员的简单名称。

   • 局部类或接口的二进制名(§14.3)由其直接封闭类或接口的二进制名组成，后跟$，后跟非空的数字序列，然后是局部类的简单名称。

   • 匿名类的二进制名(§15.9.5)由其直接封闭类或接口的二进制名组成，后面跟$，后面跟一个非空的数字序列。

   • 泛型类或接口声明的类型变量的二进制名(§8.1.2、§9.1.2)是其直接封闭类或接口的二进制名，后跟$，后跟类型变量的简单名称。

   • 泛型方法声明的类型变量的二进制名称(§8.4.4)是声明该方法的类或接口的二进制名称，后跟$，后跟方法的描述符(JVMS§4.3.3)，后跟$，后跟类型变量的简单名称。

   • 由泛型构造函数声明的类型变量的二进制名(§8.8.4)是声明构造函数的类的二进制名，后跟$，后跟构造函数的描述符(JVMS§4.3.3)，后跟$，然后是类型变量的简单名称。

2. 对另一个类或接口的引用必须是符号的，使用类或接口的二进制名称。

3. 对作为常量变量的字段的引用(§4.12.4)必须在编译时解析为由常量变量的初始化器表示的值 V。

   如果这样的字段是静态的，则在二进制文件的代码中不应该出现对该字段的引用，包括声明该字段的类或接口。这样的字段必须始终显示为已初始化(§12.4.2)；不得观察该字段的默认初始值(如果不同于 V)。

   如果这样的字段是非静态的，则在二进制文件的代码中不应该出现对该字段的引用，

除非在包含该字段的类中。(它将是一个类而不是一个接口，因为接口只有静态字段。)类应该具有在实例创建期间将字段的值设置为 V 的代码(§12.5)。

4. 给定表示类 C 中的字段访问的合法表达式，引用名为 f 的字段，该字段不是常量变量，并且在(可能是不同的)类或接口 D 中声明，我们按如下方式定义该字段引用的限定类或接口：

- 如果该表达式由一个简单的名称引用，那么如果 f 是当前类或接口 C 的成员，那么就让 Q 等于 C。否则，设 Q 是最内部的词法封闭类或接口声明，f 是它的成员。在任何一种情况下，Q 都是引用的限定类或接口。
- 如果引用的格式为 TypeName.f，其中 TypeName 表示类或接口，则由 TypeName 表示的类或接口是引用的限定类或接口。
- 如果表达式的格式为 ExpressionName.f 或 Primary.f，则：
  - 如果 ExpressionName 或 Primary 的编译时类型是交集类型 V_1 & ... & V_n (§4.9),那么引用的限定类或接口是 V_1 的擦除(§4.6)。
  - 否则，ExpressionName 或 Primary 的编译时类型的擦除是引用的限定类或接口。
- 如果表达式的形式为 super.f，则 C 的超类是引用的限定类或接口。
- 如果表达式的格式为 TypeName.super.f，则由 TypeName 表示的类的超类是引用的限定类或接口。

必须将对 f 的引用编译为对引用的限定类或接口的符号引用，外加字段的简单名称 f。

引用还必须包括对字段的声明类型的擦除的符号引用，以便验证器可以检查类型是否与预期一致。

5. 给定类或接口 C 中的方法调用表达式或方法引用表达式，引用在(可能不同的)类或接口 D 中声明(或隐式声明(§9.2))的名为 m 的方法，我们按如下方式定义方法调用的限定类或接口：

- 如果 D 为 Object，则方法调用的限定类或接口为 Object。
- 否则:
  - 如果该方法是由一个简单的名称引用的，那么如果 m 是当前类或接口 C 的成员，则 Q 是 C；否则，让 Q 是 m 是其成员的最内部的词法封闭类或接口声明。在任何一种情况下，Q 都是方法调用的限定类或接口。
  - 如果表达式的形式为 TypeName.super.m 或 ReferenceType::m, 则由 TypeName 或 ReferenceType 的擦除表示的类或接口, 是方法调用的限定类或接口。
  - 如果表达式的形式为 ExpressionName.m 或 Primary.m 或 ExpressionName::m 或 Primary::m, 那么:

- 如果 ExpressionName 或 Primary 的编译时类型是交集类型 V_1 & ... & V_n, 则方法调用的限定类或接口是 V_1 的擦除。
- 否则，ExpressionName 或 Primary 的编译时类型的擦除是方法调用的限定类或接口。

- 如果表达式的形式为 super.m 或 super::m, 那么 C 的超类就是方法调用的限定类或接口。
- 如果表达式的形式为 TypeName.super.m or TypeName.super::m，那么如果 TypeName 表示类 X, X 的超类是方法调用的限定类或接口; 如果 TypeName 表示接口 X, X 是方法调用的限定类或接口。

对方法的引用必须在编译时解析为对方法调用的限定类或接口的符号引用，加上方法的声明签名(§8.4.2)的擦除。方法的签名必须包括由§15.12.3 确定的以下所有内容:

- 方法的简单名称
- 该方法的参数数量
- 对每个参数类型的符号引用

对方法的引用还必须包括对所表示的方法的返回类型的擦除的符号引用，或者所表示的方法被声明为 void 并且没有返回值。

6. 给定类或接口 C 中的类实例创建表达式(§15.9)或显式构造函数调用语句(§8.8.7.1)或形式为 ClassType :: New(§15.13)的方法引用表达式，引用在(可能不同的)类或接口 D 中声明的构造函数 m，我们将构造函数调用的限定类定义如下:

- 如果表达式的形式为 new D(...)或 ExpressionName.new D(...)或 Primary.new D(...)或 D：：new, 则构造函数调用的限定类是 D。
- 如果表达式的形式为 new D(...){...} 或 ExpressionName.new D(...){...}或 Primary.new D(...){...}, 则构造函数调用的限定类是由表达式声明的匿名类。
- 如果表达式的形式为 super(...)或 ExpressionName.super(...) 或 Primary.super(...), 则构造函数调用的限定类是 C 的直接超类。
- 如果表达式的形式为 this(...), 则构造函数调用的限定类是 C。

对构造函数的引用必须在编译时解析为对构造函数调用的限定类的符号引用，加上构造函数的声明签名(§8.8.2)。构造函数的签名必须同时包括以下两项:

- 构造函数参数的数量
- 对每个形参的类型的符号引用

类或接口的二进制表示还必须包含以下所有内容:

1 如果它是一个类，并且不是 Object，则是对这个类的直接超类的符号引用。

2 对每个直接超接口的符号引用(如果有的话)。

3 类或接口中声明的每个字段的规范，作为字段的简单名称和对字段类型擦除的符号引用给出。

4 如果它是一个类，则如上所述，每个构造函数的擦除签名。

5 对于类或接口中声明的每个方法(对于接口，不包括其隐式声明的方法(§9.2))，其擦除的签名和返回类型，如上所述。

6 实现类或接口所需的代码：

- 对于接口，字段初始化器的代码和每个方法的实现都带有块体(§9.4.3)。
- 对于类，字段初始化器、实例初始化器和静态初始化器的代码，每个方法块体的实现(§8.4.7)，以及每个构造函数的实现。

7 每个类或接口必须包含足够的信息以恢复其规范名称(§6 7)。

8 每个成员类或接口必须有足够的信息来恢复其源代码级别的访问修饰符(§6.6)。

9 每个嵌套的类或接口都必须有一个其直接封闭类或接口的符号引用 (§8.1.3)。

10 每个类或接口必须包含对其所有成员类和接口的符号引用(§8.5、§9.5)，以及对其主体内声明的所有其他嵌套类和接口的符号引用。

11 如果 Java 编译器发出的构造与源代码中显式或隐式声明的构造不对应，则必须将其标记为合成的，除非发出的构造是类初始化方法(JVMS§29)。

12 如果 Java 编译器发出的构造对应于在源代码中隐式声明的形参(§8.8.1、§8.8.9、§8.9.3、§15.9.5.1)，则必须将其标记为强制的。

以下形式参数在源代码中隐式声明：

- 非私有内部成员类的构造函数的第一个形参(§8.8.1、§8.8.9)。
- 匿名类的匿名构造函数的第一个形参，其超类是内部类(不是在静态上下文中)(§15.9.5.1)。
- 在枚举类中隐式声明的 valueOf 方法的形参名称(§8.9.3)。
- 记录类的紧凑构造函数的形式参数(§8.10.4)。

作为参考，以下构造在源代码中被隐式声明，但没有被标记为强制的，因为只有形式参数和模块才能在 class 文件中被这样标记(JVMS§4.7.24，JVMS§4.7.25)：

- 普通类和枚举类的默认构造函数(§8.8.9, §8.9.2)
- 记录类的规范构造函数 (§8.10.4)
- 匿名构造函数 (§15.9.5.1)
- 枚举类的 values 和 valueOf 方法(§8.9.3)

- 枚举类的公共字段(§8.9.3)
- 记录类的私有字段和公共方法 (§8.10.3)
- 接口的公共方法 (§9.2)
- 容器注解 (§9.7.5)

对应于模块声明的 class 文件必须具有 class 文件的属性，该类的二进制名称为 module-info，并且没有超类、没有超接口、没有字段和方法。此外，模块的二进制表示形式必须包含以下所有内容:

- 模块名称的规范，作为对 module 之后指示的名称的符号引用。并且，规范必须包括模块是普通的还是开放的 (§7.7)。
- 由 requires 指令表示的每个依赖项的规范，作为对指令指示的模块名称的符号引用(§7.7.1)。此外，规范必须包括依赖项是否可传递以及依赖项是否是静态的。
- 由 exports 或 opens 指令表示的每个包的规范，作为对指令所示包名称的符号引用(§7.7.2)。此外，如果该指令是限定的，则该规范必须提供对该指令的 to 子句所指示的模块名称的符号引用。
- 由 uses 指令表示的每个服务的规范，作为对该指令所指示的类或接口的名称的符号引用(§7.7.3)。
- 由 provides 指令表示的服务提供者的规范，作为对该指令的 with 子句(§7.7.4)所指示的类和接口的名称的符号引用。此外，规范必须提供对指令所指示为服务的类或接口的名称的符号引用。

以下各节讨论在不破坏与现有二进制文件兼容性的情况下可以对类和接口声明进行的更改。根据上面给出的翻译要求，Java 虚拟机及其 class 文件格式支持这些更改。任何其他有效的二进制格式，例如由类加载器根据上述要求映射回 class 文件的压缩或加密表示，也必然支持这些更改。

13.2 二进制兼容性是什么不是什么

如果先前链接而没有错误的先前存在的二进制文件将继续无错误地链接，则对类型的更改与先前存在的二进制文件是二进制兼容的(等价地，不会破坏与之的二进制兼容性)。

二进制文件被编译为依赖于其他类和接口的可访问成员和构造函数。为了保持二进制兼容性，类或接口应该将其可访问的成员和构造函数、它们的存在和行为视为与其用户的约定。

Java 编程语言旨在防止契约的添加和意外的名称冲突破坏二进制兼容性。具体地说，添加更多重载特定方法名称的方法不会破坏与现有二进制文件的兼容性。重载解析算法在编译时选择了预先存在的二进制文件用于方法查找的方法签名(§15.12.2)。

> 如果 Java 编程语言设计为在运行时选择要执行的特定方法，那么可能会在运行时检测到这种歧义。这样的规则意味着，添加额外的重载方法以使调用点可能产生歧义，可能会破坏与未知数量的预先存在的二

进制文件的兼容性。有关更多讨论，请参见第 13.4.23 节。

二进制兼容性与源代码兼容性不同。特别是，§13.4.6 中的示例表明，可以从不会全部编译的源代码中生成一组兼容的二进制文件。这个示例很典型：添加了一个新的声明，更改了源代码未更改部分中名称的含义，而源代码未修改部分的预先存在的二进制文件保留了名称的完全限定的先前含义。生成一组一致的源代码需要提供与前面含义相对应的限定名称或字段访问表达式。

13.3 包和模块的演化

新的顶级类或接口可以添加到包中，而不会破坏与预先存在的二进制文件的兼容性，前提是新类或接口不重用以前赋予不相关类或接口的名称。如果一个新的类或接口重用了以前给不相关类或接口的名称，则可能会导致冲突，因为两个类或接口的二进制文件都不能由同一个类加载器加载。

顶级类和接口中的非公共类和非公共类或接口的超类或超接口的更改仅影响声明它们的包中的类和接口。这些类和接口可能会被删除或以其他方式更改，即使此处描述了不兼容性，前提是该包的受影响二进制文件一起更新。

如果声明导出或打开包的模块更改为不导出或打开该包，或将该包导出或打开给不同的朋友组，则如果链接了需要但不再有权访问该包的公共和受保护类和接口的现有二进制文件，则会引发 IllegaAccessError。对于已广泛分布的模块，不建议进行此类更改。

如果模块未声明为导出或打开给定的包，则将模块更改为导出或打开该包不会破坏与现有二进制文件的兼容性。但是，更改模块以导出包可能会阻止程序启动，因为读取模块的任何模块也可能读取导出同名包的其他模块。

将 requires 指令添加到模块声明中，或将 transitive 修饰符添加到 requires 指令中，不会破坏与现有二进制文件的兼容性。但是，这可能会阻止程序启动，因为该模块现在可以读取多个导出同名包的模块。

删除模块声明中的 requires 指令，或从 requires 命令中删除 transitive 修饰符，可能会破坏与任何已经存在的二进制文件的兼容性，这些二进制文件在引用由该模块导出的类和接口的过程中依赖于指令或修饰符来提高给定模块的可读性。当从已存在的二进制文件链接这样的引用时，可能会引发 IllegalAccessError。

在模块声明中添加或删除 uses 或 provides 指令并不会破坏与现有二进制文件的兼容性。

13.4 类的演化

本节描述更改类及其成员和构造函数的声明对已存在的二进制文件的影响。

13.4.1 抽象类

如果一个未声明为抽象的类被更改为声明为抽象的类，那么试图创建该类新实例的现有二

进制文件将在链接时抛出一个 InstantiationError，或者(如果使用了反射方法)在运行时抛出一个 InstantiationException;因此，不建议对分布广泛的类进行这样的更改。

将一个被声明为抽象的类更改为不再被声明为抽象的类并不会破坏与现有二进制文件的兼容性。

13.4.2 密封类，非密封类和 `final` 类

13.4.2.1 密封类

如果可自由扩展的类(§8.1.1.2)被更改为声明为密封的，则如果加载了此类的先前存在的子类的二进制文件，并且不是此类的允许的直接子类(§8.1.6)，则抛出 IncompatibleClassChangeError；不建议对广泛分布的类进行这样的更改。

将声明为 final 的类更改为声明为 sealed 不会破坏与现有二进制文件的兼容性。

将类添加到密封类的一组允许的直接子类不会破坏与现有二进制文件的兼容性。

如果从密封类的一组允许的直接子类中删除一个类，则如果加载已删除类的预先存在的二进制文件，则会引发不兼容的 IncompatibleClassChangeError。

从没有密封直接超类或密封直接超接口的类中删除密封修饰符不会破坏与现有二进制文件的兼容性。

> 如果密封类 C 确实有一个密封的直接超类或密封的直接超接口，则删除密封修饰符将阻止 C 重新编译，因为每个具有密封的直接超类或密封直接超接口的类必须是 final、密封的或非密封的。

13.4.2.2 非密封类

将声明为密封的类更改为声明为非密封的类不会破坏与现有二进制文件的兼容性。

将声明为 final 的类更改为声明为非密封的类不会破坏与现有二进制文件的兼容性。

> 非密封 C 类必须具有密封直接超类或密封直接超接口（§8.1.1.2）。删除非密封修饰符将阻止 C 重新编译，因为具有密封直接超类或密封直接超接口的每个类必须是 final、sealed 或 non-sealed。

13.4.2.3 `final` 类

如果将未声明为 final 类更改为声明为 final 的类，则如果加载了此类的现有子类的二进制文件，则会引发 IncompatibleClassChangeError，因为 final 类不能有子类；建议不要对广泛分发的类进行这样的更改。

从没有密封的直接超类或密封的直接超接口的类中删除 final 修饰符不会破坏与现有二进制文件的兼容性。

> 如果 final 类 C 确实具有密封的直接超类或密封的直接超接口，则删除 final 修饰符将阻止 C 重新编译，因为每个具有密封的直接超类或密封的直接超接口的类必须是 final 的、密封的或非密封的(§8.1.1.2)。

13.4.3 公共类

将未声明为公共的类更改为声明为公共的类不会破坏与先前存在的二进制文件的兼容性。

如果声明为公共的类被更改为不声明为公共的，则如果链接了需要但不再有权访问该类类型的预先存在的二进制文件，则会引发 IllegalAccessError；对于广泛分发的类，不建议进行此类更改。

13.4.4 超类和超接口

如果类是其自身的超类，则在加载时引发 ClassCircularityError。对于广泛分布的类，不建议对类层次结构进行更改，因为当新编译的二进制文件与预先存在的二进制文件一起加载时，可能会导致这种循环。

更改类的直接超类类型或直接超接口类型集不会破坏与先前存在的二进制文件的兼容性，前提是类的整个超类或超接口集不会丢失任何成员。

例如，用由原始类型命名的类或接口的参数化替换类的原始超类型是二进制兼容的。

如果对直接超类或直接超接口集合的改变导致任何类或接口分别不再是超类或超接口，则如果用修改后的类的二进制文件加载先前存在的二进制文件，则可能导致链接错误。对于广泛分布的类，不建议进行这样的更改。

例子 13.4.4-1. 改变超类

假设下面的测试程序:

```
class Hyper { char h = 'h'; }
class Super extends Hyper { char s = 's'; }
class Test extends Super {
    public static void printH(Hyper h) {
        System.out.println(h.h);
    }
    public static void main(String[] args) {
       printH(new Super());
    }
}
```

被编译并执行，生成输出:

```
h
```

假设随后编译了新版本的类 Super:

```
class Super { char s = 's'; }
```

此版本的类 Super 不是 Hyper 的子类。如果我们随后使用新版本的 Super 运行现有的二进制文件 Hyper 和 Test，则在链接时会抛出 VerifyError。验证器抱怨说，因为 new Super()的结果不能作为参数替换 Hyper 类型的形式参数，因为 Super 不是 Hyper 的子类。

考虑在没有验证步骤的情况下可能会发生什么是有指导意义的：程序可能会运行并打印:

```
s
```

这表明，如果没有验证器，Java 类型系统可能会因为链接不一致的二进制文件而失败，即使每个文件都是由正确的 Java 编译器生成的。

经验教训是，缺少验证器或无法使用验证器的实现将无法维护类型安全，因此不是有效的实现。

例子 13.4.4-2. 引入超类

一般而言，在各种情况下，对客户端二进制兼容的类转换可能与该客户端的源代码不兼容。

例如，多捕获子句中的替代项(§14.20)中的替代项不是彼此的子类或超类的要求只是一个来源限制。以下代码:

```
try {
    failByThrowingAorB();
} catch (A|B e) {
    ...
}
```

是合法的如果在编译代码时 A 和 B 没有子类/超类关系。此后，对于该客户端，A 和 B 被改变为具有这样的关系是二进制兼容的。先前编译的代码将继续执行，但由于更改与此客户端的源代码不兼容，因此无法重新编译代码。

13.4.5 类类型参数

添加或删除类的类型参数本身对二进制兼容性没有任何影响。

如果在字段或方法的类型中使用这样的类型参数，则可能具有更改前述类型的正常含义。

重命名类的类型参数对预先存在的二进制文件没有影响。

更改类的类型参数的第一个边界可能会更改在其自身类型中使用该类型参数的任何成员的擦除（§4.6），这可能会影响二进制兼容性。这种边界的变化类似于方法或构造函数类型参数的第一个边界的变化（§13.4.13）。

更改任何其他边界对二进制兼容性没有影响。

13.4.6 类体和成员声明

添加具有相同名称和可访问性（对于字段）或相同名称、可访问性、签名和返回类型（对于方法）的实例（分别为静态）成员作为超类或子类的实例（各自为静态）的成员，不会导致与已有二进制文件不兼容。即使链接的类集会遇到编译时错误，也不会发生错误。

删除未声明为私有的类成员或构造函数可能会导致链接错误，如果该成员或构造函数由预先存在的二进制文件使用。

例子 13. 4. 6-1. 改变类体

```
class Hyper {
    void hello() { System.out.println("hello from Hyper"); } }
class Super extends Hyper {
    void hello() { System.out.println("hello from Super"); }
```

```
}
class Test {
    public static void main(String[] args) {
       new Super().hello();
    }
}
```

该程序产生以下输出：

```
hello from Super
```

假设生成了类 Super 的新版本：

```
class Super extends Hyper {}
```

然后，重新编译 Super 并使用 Test 和 Hyper 的原始二进制文件执行此新二进制文件，生成输出：

```
hello from Hyper
```

和预期一样。

super 关键字可用于访问超类中声明的方法，绕过当前类中声明任何方法。表达式 super.Identifier 在编译时被解析为超类 S 中的方法 m。如果方法 m 是一个实例方法，那么在运行时调用的方法是具有与 m 相同签名的方法，该方法是包含涉及 super 的表达式的类的直接超类的成员。

例子 13.4.6-2. 改变超类

```
class Hyper {
    void hello() { System.out.println("hello from Hyper"); }
}
class Super extends Hyper { }
class Test extends Super {

    public static void main(String[] args) {
       new Test().hello();
    }
    void hello() {
       super.hello();
    }
}
```

该程序产生以下输出：

```
hello from Hyper
```

假设生成了类 Super 的新版本：

```
class Super extends Hyper {
   void hello() { System.out.println("hello from Super"); }
}
```

然后，如果重新编译了 Super 和 Hyper，但没有编译 Test，则使用现有的 Test 二进制文件运行新的二进制文件将生成输出：

```
hello from Super
```

和你所期望的一样。

13.4.7 访问成员和构造函数

更改成员或构造函数的声明访问权限以允许较少的访问可能会破坏与现有二进制文件的兼容性，导致在解析这些二进制文件时引发链接错误。如果访问修改符从包访问更改为私有访问，从受保护访问更改为包访问或私有访问；或从公共访问更改为受保护访问、包访问或私有访问，则更少的访问被允许。因此，对于分布广泛的类，不建议更改成员或构造函数以允许较少的访问。

也许令人惊讶的是，二进制格式的定义使得当子类（已经）定义了一个具有较少的访问权限的方法时，将成员或构造函数更改为具有更多访问权限不会导致链接错误。

例子 13. 4. 7-1. 改变可访问性

如果包 points 定义了类 Point:

```
package points;
public class Point {
    public int x, y;
    protected void print() {
        System.out.println("(" + x + "," + y + ")");
    }
}
```

被以下程序使用:

```
class Test extends points.Point {
    public static void main(String[] args) {
        Test t = new Test();
        t.print();
    }
    protected void print() {
        System.out.println("Test");
    }
}
```

然后编译这些类并执行 Test 以生成输出：

```
Test
```

如果类 Point 中的方法 print 改变为 public，只有 Point 类被重编，然后和已有的二进制文件 Test 一起被执行，不会发生链接错误。即使在编译时，由受保护的方法重写公共方法是不合适的，也会发生这种情况。（例如，除非将 Test 里的 print 更改为 public，否则无法使用新的 Point 类重新编译 Test 类）

允许超类将受保护的方法更改为公共方法，而不破坏预先存在的子类的二进制文件，有助

于降低二进制文件的脆弱性。另一种方法是，如果这种更改会导致链接错误，则会产生额外的二进制不兼容。

13.4.8 字段声明

广泛分布的程序不应向其客户端公开任何字段。除了下面讨论的二进制兼容性问题外，这通常是良好的软件工程实践。向类中添加字段可能会破坏与未重新编译的现有二进制文件的兼容性。

假设对具有限定类 C 的字段 f 的引用。进一步假设 f 实际上是在 C、S 的超类中声明的实例（分别为静态）字段，并且 f 的类型是 X。

如果将与 f 同名的 X 类型的新字段添加到作为 C 本身或 C 的超类的 S 的子类中，则可能会发生链接错误。除上述情况外，只有在以下任一情况为真时，才会发生此类 l 链接错误：

- 新字段比老字段具有更少的访问控制权限
- 新字段是静态（分别为实例）字段。

特别是，当一个类因为字段访问以前引用了具有不兼容类型的超类的字段而不能重新编译时，不会发生链接错误。带有此类引用的先前编译的类将继续引用超类中声明的字段。

例子 13.4.8-1. 添加字段声明

```
class Hyper { String h = "hyper"; }
class Super extends Hyper { String s = "super"; }
class Test {
    public static void main(String[] args) {
       System.out.println(new Super().h);
    }
}
```

这个程序产生输出：

```
hyper
```

假设产生了一个新版本的类 Super：

```
class Super extends Hyper {
    String s = "super";
    int h = 0;
}
```

然后，重编 Hyper 和 Super，使用 Test 的旧二进制文件执行得到的新二进制文件并产生输出：

```
hyper
```

Hyper 的字段 h 由 Test 的原始二进制文件输出。虽然这乍一看可能令人惊讶，但它有助于减少运行时发生的不兼容的数量。（在理想情况下，所有需要重新编译的源文件只要其中任何一个发生更改，都会被重新编译，从而消除了这种意外。但这样的大规模重新编译往往是不切实际的，甚至是不可能的，特别是在互联网上。而且，如前所述，这种重新编译有时需要对源代码进行进一步的更改。）

作为另一个示例，如果程序：

```
class Hyper { String h = "Hyper"; }
class Super extends Hyper { }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```

编译并执行时，它会生成以下输出：

```
Hyper
```

假设随后编译了新版本的类 Super：

```
class Super extends Hyper { char h = 'h'; }
```

如果生成的二进制文件与 Hyper 和 Test 的现有二进制文件一起使用，则输出仍为：

```
Hyper
```

即使编译这些二进制文件的源代码：

```
class Hyper { String h = "Hyper"; }
class Super extends Hyper { char h = 'h'; }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }

}
```

将导致编译时错误，因为 main 的源代码中的 h 现在将被解释为引用在 Super 中声明的 char 字段，并且 char 值不能被赋给 String。

从类中删除字段将破坏与引用该字段的任何现有二进制文件的兼容性，并且当链接来自现有二进制文件中的此类引用时，将引发 NoSuchFieldError。只有私有字段可以从广泛分布的类中安全删除。

出于二进制兼容性的目的，添加或删除类型涉及类型变量（§4.4）或参数化类型（§4.5）的字段 f 相当于添加（分别删除）类型为 f 的类型的擦除（§4.6）的同名字段。

13.4.9 final 字段和静态常量变量

如果未声明为 final 的字段被更改为声明为 final，则可能会破坏与试图为该字段分配新值的现有二进制文件的兼容性。

例子 13.4.9-1. 将变量更改为 final

```
class Super { char s; }
class Test extends Super {
```

```
        public static void main(String[] args) {
            Super x = new Super();
            x.s = 'a';
            System.out.println(x.s);
        }
    }
```

该程序产生以下输出：

```
a
```

假设生成了类 Super 的新版本：

```
class Super { final char s = 'b'; }
```

如果 Super 被重新编译了，但是 Test 没有，然后，将新的二进制文件与现有的 Test 的二进制文件一起运行，会导致 IllegalAccessError。

删除关键字 final 或更改字段的初始化值不会破坏与现有二进制文件的兼容性。

如果一个字段是一个常量变量(§4.12.4)，而且是静态的，那么删除关键字 final 或更改它的值不会因为导致它们不运行而破坏与先前存在的二进制文件的兼容性，但是除非重新编译它们，否则它们不会看到该字段的使用的任何新值。这一结果是决定支持条件编译的副作用(§14.22)。(人们可能会认为，如果在常量表达式（§15.29）中使用，则不会看到新值，但在其他情况下会看到新值。事实并非如此；预先存在的二进制文件根本看不到新值。)

避免在广泛分布的代码中出现“多变的常数”问题的最佳方法是只对真正不太可能改变的值使用静态常数变量。除了真正的数学常量，我们建议源代码尽量少使用静态常量变量。

如果需要 final 的只读特性，更好的选择是声明一个私有静态变量和一个合适的访问器方法来获取它的值。因此我们推荐：

```
private static int N;
public static int getN() { return N; }
```

而不是:

```
public static final int N = ...;
```

以下各项没有问题：

```
public static int N = ...;
```

如果 N 不需要是只读的。

13.4.10 静态字段

如果未声明为私有的字段未声明为静态，并且更改为声明为静态，或者反之亦然，则如果该字段由预期其他类型的字段的预先存在的二进制文件使用，则将导致链接错误，特别是 IncompatibleClassChangeError。不建议在已广泛分发的代码中进行此类更改。

13.4.11 transient 字段

添加或删除字段的临时修饰符不会破坏与先前存在的二进制文件的兼容性。

13.4.12 方法和构造函数声明

向类添加方法或构造函数不会破坏与任何先前存在的二进制文件的兼容性，即使在类无法再重新编译的情况下也是如此，因为调用以前引用了具有不兼容类型的超类的方法或构造函数。具有这种引用的先前编译的类将继续引用在超类中声明的方法或构造函数。

假设引用具有限定类 C 的方法 m。进一步假设 m 实际上是在 C，S 的超类中声明的实例(分别是静态的)方法。

如果将与 m 具有相同签名和返回类型的 X 类型的新方法添加到 S 的子类中，而该子类是 C 或 C 本身的超类，则可能会发生链接错误。除上述情况外，只有在下列情况之一成立的情况下，才会发生这种链接错误:

- 新方法比旧方法更不容易访问。
- 新方法是一个静态(分别是实例)方法。

从类中删除方法或构造函数可能会破坏与引用此方法或构造函数的现有二进制文件的兼容性;当从已存在的二进制文件链接这样的引用时，可能会引发 NoSuchMethodError。只有在超类中没有声明具有匹配签名和返回类型的方法时，才会发生这样的错误。

如果非内部类的源代码不包含声明的构造函数，则隐式声明了不带参数的默认构造函数(§8.8.9)。在此类的源代码中添加一个或多个构造函数声明将防止隐式声明此默认构造函数，从而有效地删除一个构造函数，除非其中一个新构造函数也没有参数，从而替换默认构造函数。没有参数的默认构造函数被赋予与其声明的类相同的访问修饰符，因此，如果要保留与现有二进制文件的兼容性，任何替换都应该具有相同或更多的访问权限。

13.4.13 方法和构造函数类型参数

添加或删除方法或构造函数的类型参数本身对二进制兼容性没有任何影响。

如果在方法或构造函数的类型中使用了这样的类型参数，则这可能具有更改上述类型的正常含义。

重命名方法或构造函数的类型参数对预先存在的二进制文件没有影响。

更改方法或构造函数的类型参数的第一个边界可能会更改在其自身类型中使用该类型参数的任何成员的擦除（§4.6），这可能会影响二进制兼容性。具体地说:

- 如果使用类型参数作为字段的类型，效果就好像字段被删除了，而添加了一个具有相同名称的字段，其类型是类型变量的新擦除。
- 如果将类型参数用作方法的任何方法的形式参数的类型，而不是返回类型，那么效果就好像该方法被删除了，并被替换为一个新方法，该方法除了前面提到的形式参数的

类型之外，其他类型都相同，它们现在有类型参数的新擦除作为它们的类型。

- 如果类型参数用作方法的返回类型，而不是该方法的任何方法的形式参数的类型，那么效果就好像该方法被删除了，并替换为除返回类型（现在是类型参数的新擦除）之外相同的新方法。
- 如果将类型参数用作方法的返回类型和方法的一个或多个形式参数的类型，则效果就好像该方法被删除，并被替换为除了返回类型之外相同的新方法，返回类型现在是类型参数的新擦除，并且除了前述形式参数的类型之外，前述形参的类型现在具有类型参数的新擦除作为其类型。

更改任何其他界限不会影响二进制兼容性。

13.4.14 方法和构造函数形式参数

更改方法或构造函数的形参的名称不会影响先前存在的二进制文件。

将方法的名称或形参的类型更改为方法或构造函数，或者在方法或构造函数声明中添加参数或删除参数，都会创建具有新签名的方法或构造函数，并且具有删除具有旧签名的方法或构造函数和添加具有新签名的方法或构造函数的综合效果(§13.4.12)。

将方法的最后一个形式参数的类型从 T[]改变成类型为 T 的可变参数，也就是，改变成 T...(§8.4.1), 反之亦然, 不会影响先前存在的二进制文件。

为了二进制兼容性的目的，添加或删除其签名涉及类型变量(§4.4)或参数化类型(§4.5)的方法或构造函数 m 等效于添加(分别删除)签名为 m 的签名的擦除(§4.6)的其他等价方法。

13.4.15 方法返回类型

更改方法的结果类型，或用 void 替换结果类型，或用结果类型替换 void，具有删除旧方法并添加具有新结果类型或新的 void 结果的新方法的综合效果(见§13.4.12)。

为了二进制兼容的目的，添加或删除其返回类型涉及类型变量(§4.4)或参数化类型(§4.5)的方法或构造函数 m 等同于添加(分别删除)返回类型为 m 的返回类型的擦除(§4.6)的其他等价方法。

13.4.16 抽象方法

将声明为抽象的方法更改为不再声明为抽象的方法不会破坏与现有二进制文件的兼容性。

将未声明为抽象的方法更改为声明为抽象的方法将破坏与先前调用该方法的现有二进制文件的兼容性，从而导致 AbstractMethodError。

例子 13. 4. 16-1. 将方法改变成抽象方法

```
class Super { void out() { System.out.println("Out"); } }
class Test extends Super {
    public static void main(String[] args) {
       Test t = new Test();
       System.out.println("Way ");
```

```
        t.out();
    }
}
```

此程序生成以下输出：

```
Way Out
```

假设生成了新版本的 Super 类：

```
abstract class Super {
  abstract void out();
}
```

如果 Super 被重新编译但是 Test 没有被重新编译，然后用现存的 Test 的二进制文件运行新的二进制文件会产生一个 AbstractMethodError, 因为类 Test 没有实现方法 out，因此是（或应该是）抽象的。

13.4.17 final 方法

将声明为 final 的方法更改为不再声明为 final 的方法并不会破坏与现有二进制文件的兼容性。

将未声明为 final 的实例方法更改为 final 可能会破坏与现有二进制文件的兼容性，这些二进制文件依赖于重写该方法的能力。

例子 13.4.17-1. 将方法改变成 final 方法

```
class Super { void out() { System.out.println("out"); } }
class Test extends Super {
    public static void main(String[] args) {
       Test t = new Test();
       t.out();
    }
    void out() { super.out(); }
}
```

这个程序产生输出:

```
out
```

假设产生了一个 Super 类的新版本:

```
class Super { final void out() { System.out.println("!"); } }
```

如果 Super 被重新编译但是 Test 没有被重新编译，然后用现有的 Test 的二进制文件运行新的二进制文件会导致 IncompatibleClassChangeError 因为类 Test 试图不正确地重写实例方法 out。

将未声明为 final 的类(静态)方法更改为 final 并不会破坏与现有二进制文件的兼容性，因为无法重写该方法。

13.4.18 native 方法

添加或删除方法的 native 修饰符不会破坏与预先存在的二进制文件的兼容性。

对类型的更改对未重新编译的现有 native 方法的影响超出了本规范的范围，应该在实现的描述中提供。我们鼓励(但不是必需)以限制这种影响的方式实现 native 方法。

13.4.19 静态方法

如果未声明为私有的方法也被声明为静态的(即类方法)，并且被更改为不声明为静态的(即实例方法)，或者反过来，那么与预先存在的二进制文件的兼容性可能会被破坏，导致链接时间错误，即 IncompatibleClassChangeError，如果这些方法是由预先存在的二进制文件使用的。不建议在已广泛分发的代码中进行这样的更改。

13.4.20 同步方法

添加或删除方法的同步修饰符并不会破坏与现有二进制文件的兼容性。

13.4.21 方法和构造函数抛出

对方法或构造函数的 throws 子句的更改不会破坏与预先存在的二进制文件的兼容性;这些子句仅在编译时检查。

13.4.22 方法和构造函数体

对方法或构造函数体的更改不会破坏与现有二进制文件的兼容性。

方法上的关键字 final 并不意味着该方法可以安全内联;它只意味着该方法不能被重写。仍然有可能在链接时提供该方法的新版本。此外，为了进行反射，必须保留原始程序的结构。

因此，我们注意到 Java 编译器不能在编译时扩展内联方法。通常，我们建议实现使用后期绑定(运行时)代码生成和优化。

13.4.23 方法和构造函数重载

添加重载现有方法或构造函数的新方法或构造函数不会破坏与现有二进制文件的兼容性。每次调用要使用的签名是在编译这些现有二进制文件时确定的；因此，将不会使用新添加的方法或构造函数，即使它们的签名既适用又比最初选择的签名更具体。

虽然添加新的重载方法或构造函数可能会在下次编译类或接口时导致编译时错误，因为没有最具体的方法或构造函数(§15.12.2.5)，但在执行程序时不会发生这样的错误，因为在执行时不会进行重载解析。

例子 13.4.23-1. 添加重载方法

```
class Super {
   static void out(float f) {
      System.out.println("float");
   }
}
class Test {
```

```
        public static void main(String[] args) {
           Super.out(2);
        }
    }
```

此程序生成以下输出：

```
    float
```

假设生成了新版本的 Super 类：

```
    class Super {
        static void out(float f) { System.out.println("float"); }
        static void out(int i)   { System.out.println("int"); }
    }
```

如果 Super 被重新编译但是 Test 没有被重新编译，然后用现有的 Test 的二进制文件运行新的二进制文件仍然产生输出：

```
    float
```

然而，如果 Test 被重新编译了，使用新版本的 Super，输出就是：

```
    int
```

正如在前面的案例中可能天真地预期的那样。

13.4.24 方法重写

如果一个实例方法被添加到子类中，并且它重写了超类中的一个方法，那么子类方法将通过预先存在的二进制文件中的方法调用找到，并且这些二进制文件不会受到影响。

如果将类方法添加到类中，则将找不到该方法，除非方法调用的限定类是子类。

13.4.25 静态初始化器

添加、删除或更改类的静态初始化器（§8.7）不会影响预先存在的二进制文件。

13.4.26 枚举类的演化

在枚举类中添加或重新排序枚举常量不会破坏与预先存在的二进制文件的兼容性。

从枚举类中删除枚举常量将删除对应于枚举常量的公共字段（§8.9.3）。后果在§13.4.8 中规定。对于分布广泛的枚举类，不建议进行这种更改。

在所有其他方面，枚举类的二进制兼容性规则与普通类的二进制兼容规则相同。

13.4.27 记录类的演化

在记录类中添加、删除、更改或重新排序记录组件可能会破坏与未重新编译的现有二进制文件的兼容性；对于分布广泛的记录类，不建议进行这种更改。

更准确地说，添加、删除、更改或重新排序记录组件可能会更改组件字段和访问器方法的相应隐式声明，以及更改规范构造函数和其他支持方法的签名和实现，其后果如§13.4.8 和

§13.2.12 所述。

在所有其他方面，记录类的二进制兼容性规则与普通类的二进制兼容规则相同。

13.5 接口的演化

本节描述对接口及其成员声明的更改对预先存在的二进制文件的影响。

13.5.1 公共接口

将未声明为公共的接口更改为声明为公共的接口不会破坏与现有二进制文件的兼容性。

如果声明为公共的接口更改为未声明为公共，则如果链接了需要但不再具有对接口类型访问权限的现有二进制文件，则会引发 IllegaAccessError，因此不建议对广泛分布的接口进行此类更改。

13.5.2 密封和非密封接口

如果可自由扩展的接口(§9.1.1.4)被更改为声明为密封的，则如果加载了该接口的现有子类或子接口的二进制文件，并且不是该接口的允许的直接子类或子接口(§9.1.4)，则抛出 IncompatibleClassChangeError；不建议对广泛分布的类进行此类更改。

将类或接口分别添加到密封接口的允许的直接子类或子接口集中，不会破坏与现有二进制文件的兼容性。

如果从密封接口的允许直接子类或子接口集中移除某个类或接口，则如果加载了移除的类或接口的先前存在的二进制文件，则会引发 IncompatibleClassChangeError。

将声明为密封的接口更改为非密封不会破坏与现有二进制文件的兼容性。

> 非密封接口 I 必须有一个密封的直接超接口。删除非密封修饰符将阻止重新编译 I，因为每个具有密封的直接超接口的接口都必须是密封的或非密封的。

从没有密封的直接超接口的接口中删除 sealed 修饰符不会破坏与先前存在的二进制文件的兼容性。

> 如果密封的接口 I 确实具有密封的直接超接口，则删除 sealed 修饰符将阻止重新编译 I，因为每个具有密封的直接超接口的接口都必须是密封的或非密封的。

13.5.3 超接口

接口层次结构的更改导致错误的方式与类层次结构的更改相同，如§13.4.4 中所述。特别是，导致类以前的任何超接口不再是超接口的更改可能会破坏与先前存在的二进制文件的兼容性，从而导致 VerifyError。

13.5.4 接口成员

向接口添加抽象、私有或静态方法不会破坏与现有二进制文件的兼容性。

向 C 的超接口添加字段可能会隐藏从 C 的超类继承的字段。如果原始引用是对实例字段的引用，则会产生 IncompatibleClassChangeError。如果原始引用是赋值，则会产生 IllegalAccessError。

从接口中删除成员可能会导致预先存在的二进制文件中出现链接错误。

例子 13.5.4-1. 删除接口成员

```
interface I { void hello(); }
class Test implements I {
  public static void main(String[] args) {
    I anI = new Test();
    anI.hello();
  }
  public void hello() { System.out.println("hello"); }
}
```

此程序生成以下输出：

```
hello
```

假设编译了新版本的接口 I：

```
interface I {}
```

如果 I 已重新编译但 Test 没有重编，则使用现有的 Test 的二进制文件运行新的二进制文件将导致 NoSuchMethodError。

13.5.5 接口类型参数

对接口的类型参数进行更改的效果与对类的类型参数进行类似更改的效果相同。

13.5.6 字段声明

更改接口中的字段声明的注意事项与更改类中的静态 final 字段的注意事项相同，如§13.4.8 和§13.4.9 中所述。

13.5.7 接口方法声明

更改接口中的方法声明的注意事项包括更改类中的方法的注意事项，如§13.4.7、§13.4.14、§13.4.15、§13.4.19、§13.4.21、§13.4.22 和§13.4.23 中所述。

添加默认方法或将方法从抽象更改为默认不会破坏与已有二进制文件的兼容性，但如果已有二进制文件试图调用该方法，则可能会导致 IncompatibleClassChangeError。如果方法调用的限定接口 K 是两个接口 I 和 J 的子接口，其中 I 和 J 都声明了具有相同签名和结果的默认方法，并且 I 和 J 都不是对方的子接口，则会发生此错误。

换句话说，添加默认方法是二进制兼容的更改，因为它不会在链接时引入错误，即使它在编译时或调用时引入错误。在实践中，通过引入默认方法而发生意外冲突的风险与向非 final 类添加新方法相关联的风险相似。在发生冲突的情况下，向类中添加方法不太可能触

发 LinkageError，但是在子类中意外重写该方法可能导致不可预测的方法行为。这两种更改都可能在编译时导致错误。这两种更改都可能在编译时导致错误。

例子 13.5.7-1. 添加默认方法

```
interface Painter {
    default void draw() {
       System.out.println("Here's a picture...");
    }
}

interface Cowboy {}

public class CowboyArtist implements Cowboy, Painter {
   public static void main(String... args) {
      new CowboyArtist().draw();
   }
}
```

这个程序产生输出:

```
Here's a picture...
```

假设在 Cowboy 中添加了一个默认方法:

```
interface Cowboy {
    default void draw() {
        System.out.println("Bang!");
    }
}
```

如果 Cowboy 被重新编译但是 CowboyArtist 没有被重新编译, 然后，使用 CowboyArtist 的现有二进制文件运行新的二进制文件将链接而不出错，但在 main 尝试调用 draw（）时会导致 IncompatibleClassChangeError。

13.5.8 注解接口

注解接口的行为与任何其他接口完全相同。从注解接口添加或删除元素类似于添加或删除方法。对于注解接口的其他更改，有一些重要的考虑因素，例如使注解接口可重复（§9.6.3），但这些因素对 Java 虚拟机的二进制文件链接没有影响。相反，这样的更改会影响 Java SE 平台中显示程序中注解存在的反射 api 的行为。API 规范描述了当底层注解接口发生各种变化时的行为(§1.4)。

添加或删除注解不会影响 Java 编程语言中程序二进制表示的正确链接。