第四章

class 文件格式

本章描述了 Java 虚拟机的 class 文件格式。每个 class 文件包含单个类、接口或模块的定义。尽管类、接口或模块不需要具有字面上包含在文件中的外部表示（例如，因为类由类加载器生成），我们将把类、接口或模块的任何有效表示通俗地称为 class 文件格式。

class 文件由 8 位字节流组成。通过分别读取两个和四个连续的 8 位字节来构造 16 位和 32 位的数。多字节数据项总是以大端顺序存储，其中高字节优先。本章定义了数据类型 u1、u2 和 u4，分别表示无符号的 1 字节、2 字节或 4 字节数。

> 在 Java SE 平台 API 中，class 文件格式由接口 java.io.DataInput 和 java.io.DataOutput 以及类 java.io.DataInputStream 和 java.io.DataOutputStream 支持。例如，u1、u2 和 u4 类型的值可以通过 java.io.DataInput 接口的 readUnsignedByte、readUnsignedShort 和 readInt 等方法读取。

本章介绍了使用伪结构的 class 文件格式，伪结构以类似 C 的结构符号编写。为了避免与类和类实例等字段混淆，描述 class 文件格式的结构的内容称为项。连续项按顺序存储在 class 文件中，无需填充或对齐。

表由零个或多个可变大小的项组成，用于多个 class 文件结构中。尽管我们使用类似 C 的数组语法来引用表项，但表是大小不同结构的流这一事实意味着不可能将表索引直接转换为表中的字节偏移量。

当我们将数据结构称为数组时，它由零个或多个连续的固定大小项组成，可以像数组一样索引。

本章中对 ASCII 字符的引用应解释为对应于 ASCII 字符的 Unicode 代码点。

4.1 `ClassFile` 结构

class 文件由单个 ClassFile 结构组成:

```
ClassFile {
    u4             magic;
    u2             minor version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1]
    u2             access_flags;
    u2             this class;
    u2             super_class;
    u2             interfaces count;
    u2             interfaces[interfaces_count];
    u2             fields count;
    field_info     fields[fields_count];
```

```
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

ClassFile 结构中的项如下:

`magic`

magic 项提供标识 class 文件格式的魔术数字; 它的值为 0xCAFEBABE。

`minor_version, major_version`

minor_version 和 major_version 项的值是这个 class 文件的次版本号和主版本号。主版本号和次版本号共同决定 class 文件格式的版本。如果一个 class 文件有主版本号 M 和次版本号 m，我们将其 class 文件格式的版本表示为 M.m。

符合 Java SE N 的 Java 虚拟机实现必须完全支持表 4.1-A 第四列“支持的主版本号”中指定的 class 文件格式的主版本。A ..B 是指主版本 A 到 B，包括 A 和 B。第三列“Major”显示每个 Java SE 发行版引入的主要版本，也就是第一个可以接受包含 major_version 项的 class 文件的发行版。对于非常早期的版本，显示的是 JDK 版本而不是 Java SE 版本。

表 4.1-A. class 文件格式主版本号

Java SE	Released	Major	Supported majors
1.0.2	May 1996	45	45
1.1	February 1997	45	45
1.2	December 1998	46	45 .. 46
1.3	May 2000	47	45 .. 47
1.4	February 2002	48	45 .. 48
5.0	September 2004	49	45 .. 49
6	December 2006	50	45 .. 50
7	July 2011	51	45 .. 51
8	March 2014	52	45 .. 52
9	September 2017	53	45 .. 53
10	March 2018	54	45 .. 54
11	September 2018	55	45 .. 55
12	March 2019	56	45 .. 56
13	September 2019	57	45 .. 57
14	March 2020	58	45 .. 58
15	September 2020	59	45 .. 59
16	March 2021	60	45 .. 60

17	September 2021	61	45 .. 61
18	March 2022	62	45 .. 62
19	September 2022	63	45 .. 63

对于 major_version 大于等于 56 的 class 文件, minor_version 必须是 0 或 65535。

对于 major_version 大于等于 45，小于等于 55 的 class 文件，minor_version 可以是任何值。

对于 JDK 对 class 文件格式版本的支持，有必要从历史的角度来看。JDK1.1 支持版本 45.0 到 45.65535(含)。当 JDK 1.2 引入对主版本 46 的支持时，该主版本下唯一支持的次版本是 0。后来，JDK 继续引入对新的主版本(47、48 等)的支持，但在新的主版本下只支持 0 的次版本。最后，Java SE 12(见下文)中预览特性的引入激发了 class 文件格式的次版本的标准角色，因此 JDK 12 在主版本 56 下支持次版本 0 和 65535。后续的 JDK 引入了对 N.0 和 N.65535 的支持，其中 N 是实现的 Java SE 平台的相应主版本。例如，JDK 13 支持 57.0 和 57.65535。

Java SE 平台可以定义预览功能。符合 Java SE N(N>12)的 Java 虚拟机实现必须支持 Java SE N 的所有预览功能，而不支持任何其他 Java SE 版本的预览功能。默认情况下，实现必须禁用受支持的预览功能，并且必须提供启用所有预览功能的方法，而不能提供仅启用部分预览功能的方法。

如果 class 文件具有对应于 Java SE N（根据表 4.1-A）的 major_version 和 65535 的 minor_version，则称其依赖于 Java SE N 的预览特性（N>12）。

符合 Java SE N（N > 12）的 Java 虚拟机实现必须表现如下:

- 仅当启用 Java SE N 的预览功能时，才能加载依赖于 Java SE N 预览功能的 class 文件。
- 决不能加载依赖于另一个 Java SE 版本的预览特性的 class 文件。
- 无论是否启用了 Java SE N 的预览功能，都可以加载不依赖于任何 Java SE 版本的预览功能的 class 文件。

`constant_pool_count`

constant_pool_count 项的值等于 constant_pool 表项数加 1。如果一个 constant_pool 的索引大于零且小于 constant_pool_count，则它被认为是有效的，§4.4.5 中提到的 long 和 double 类型的常量除外。

`constant_pool[]`

constant_pool 是一个结构表(§4.4)，表示各种字符串常量、类和接口名、字段名，以及 ClassFile 结构及其子结构中引用的其他常量。每个 constant_pool 表项的格式由其第一个“标签”字节表示。

constant_pool 表索引从 1 到 constant_pool_count – 1。

`access_flags`

access_flags 项的值是一个标志掩码，用于表示该类或接口的访问权限和属性。当设置每个标志时，其说明在表 4.1-B 中指定。

表 4.1-B. 类访问和属性修饰符

标志名	值	说明
ACC_PUBLIC	0x0001	声明为 public; 可以从包外访问。
ACC_FINAL	0x0010	声明为 final; 不允许有子类。
ACC_SUPER	0x0020	当使用 invokespecial 指令调用超类方法时，要特别处理超类方法。
ACC_INTERFACE	0x0200	是一个接口，不是一个类。
ACC_ABSTRACT	0x0400	声明为 abstract; 不能被实例化。
ACC_SYNTHETIC	0x1000	声明为 synthetic; 没有出现在源代码中。
ACC_ANNOTATION	0x2000	声明为注解接口。
ACC_ENUM	0x4000	声明为枚举类。
ACC_MODULE	0x8000	是一个模块，不是类或接口。

ACC_MODULE 标志表示这个 class 文件定义的是模块，而不是类或接口。如果设置了 ACC_MODULE 标志，则特殊规则适用于本节末尾给出的 class 文件。如果未设置 ACC_MODULE 标志，则当前段落正下方的规则适用于 class 文件。

通过设置 ACC_INTERFACE 标志来区分接口。如果未设置 ACC_INTERFACE 标志，则该 class 文件定义一个类，而不是接口或模块。

如果设置了 ACC_INTERFACE 标志，则还必须设置 ACC_ABSTRACT 标志，并且不得设置 ACC_FINAL、ACC_SUPER、ACC_ENUM 和 ACC_MODULE 标志。

如果没有设置 ACC_INTERFACE 标志，则可以设置表 4.1-B 中除 ACC_ANNOTATION 和 ACC_MODULE 之外的任何其他标志。但是，此 class 文件不能同时设置其 ACC_FINAL 和 ACC_ABSTRACT 标志(JLS§8.1.1.2)。

ACC_SUPER 标志指示两个可选语义中的哪一个将由 invokespecial (§invokespecial)表示，如果它出现在这个类或接口中。Java 虚拟机指令集的编译器应设置 ACC_SUPER 标志。在 Java SE 8 和更高版本中，Java 虚拟机认为在每个 class 文件中都设置了 ACC_SUPER 标志，而不管 class 文件中标志的实际值和 class 文件的版本。

ACC_SUPER 标志的存在是为了向后兼容 Java 编程语言的旧版编译器编译的代码。在 JDK 1.0.2 之前，编译器生成 access_flags，其中现在表示 ACC_SUPER 的标志没有指定的意义，如果设置了标志，Oracle 的 Java 虚拟机实现将忽略该标志。

ACC_SYNTHETIC 标志指示此类或接口是由编译器生成的，不会出现在源代码中。

注解接口 (JLS§9.6) 必须设置其 ACC_ANNOTATION 标志。如果设置了

ACC_ANNOTATION 标志，则还必须设置 ACC_INTERFACE 标志。

ACC_ENUM 标志指示该类或其超类被声明为枚举类(JLS§8.9)。

表 4.1-B 中未分配的 access_flags 项的所有位都保留以备将来使用。它们应该在生成的 class 文件中设置为零，并且应该被 Java 虚拟机实现忽略。

`this_class`

this_class 项的值必须是 constant_pool 表的有效索引。该索引处的 constant_pool 条目必须是代表此 class 文件定义的类或接口的 CONSTANT_Class_info 结构(§4.4.1)。

`super_class`

对于类，super_class 项的值必须为零，或者必须是 constant_pool 表的有效索引。如果 super_class 项的值非零，则该索引处的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构，该结构表示由该 class 文件定义的类的直接超类。无论是直接超类还是它的任何超类，都不能在其 ClassFile 结构的 access_flags 项中设置 ACC_FINAL 标志。

如果 super_class 项的值为零，则这个 class 文件必须表示类 Object，即唯一没有直接超类的类或接口。

对于接口，super_class 项的值必须始终是 constant_pool 表的有效索引。该索引处的 constant_pool 条目必须是表示类 Object 的 CONSTANT_Class_info 结构。

`interfaces_count`

interfaces_count 项的值给出了此类或接口类型的直接超接口的数量。

`interfaces[]`

interfaces 数组中的每个值必须是 constant_pool 表的有效索引。interfaces[i]的每个值的 constant_pool 条目，其中 0≤i<interfaces_count，必须是一个 CONSTANT_Class_info 结构，表示一个接口，该接口是这个类或接口类型的直接超接口，按照从左到右的顺序在类型的源代码中给出。

`fields_count`

fields_count 项的值提供 fields 表中的 field_info 结构的数量。field_info 结构表示该类或接口类型声明的所有字段，包括类变量和实例变量。

`fields[]`

fields 表中的每个值都必须是一个 field_info 结构(§4.5)，给出该类或接口中一个字段的完整描述。fields 表只包括由该类或接口声明的字段。它不包括表示从超类或超接口继承的字段的项。

`methods_count`

methods_count 项的值给出了 methods 表中 method_info 结构的数量。

`methods[]`

methods 表中的每个值都必须是一个 method_info 结构(§4.6)，给出这个类或接口中一

个方法的完整描述。如果在 method_info 结构的 access_flags 项中没有设置 ACC_NATIVE 和 ACC_ABSTRACT 标志，那么也会提供实现该方法的 Java 虚拟机指令。

method_info 结构体代表该类或接口类型声明的所有方法，包括实例方法、类方法、实例初始化方法(§2.9.1)，以及任何类或接口初始化方法(§2.9.2)。方法表不包括表示从超类或超接口继承的方法的项。

`attributes_count`

attributes_count 项的值给出了该类 attributes 表中属性的数量。

`attributes[]`

attributes 表的每个值都必须是 attribute_info 结构(§4.7)。

本规范定义的 ClassFile 结构的 attributes 表如表 4.7-C 所示。

关于定义在 ClassFile 结构的 attributes 表中的属性的规则见§4.7。

ClassFile 结构的 attributes 表中关于非预定义属性的规则见§4.7.1。

如果 ACC_MODULE 标志在 access_flags 项中被设置，那么 access_flags 项中的其他标志都不能被设置，以下规则适用于 ClassFile 结构的其余部分:

- major_version, minor_version：*N*53.0 (即 Java SE 9 及以上版本)
- this_class: module-info
- super_class, interfaces_count, fields_count, methods_count: 零
- attributes: 必须存在一个 Module 属性。除了 Module,ModulePackages,ModuleMainClass,InnerClasses,SourceFile,SourceDebugExtension, RuntimeVisibleAnnotations, 和 RuntimeInvisibleAnnotations 外, 任何预定义的属性(§4.7)都不会出现。

4.2 名字

4.2.1 二进制类和接口名称

class 文件结构中出现的类和接口名称总是以完全限定的形式表示，称为二进制名称(JLS§13.1)。这样的名称总是表示为 CONSTANT_Utf8_info 结构(§4.4.7)，因此可以从整个 Unicode 代码空间中提取，如果没有进一步的限制。类和接口名称引用自那些 CONSTANT_NameAndType_info 结构（§4.4.6），这些结构的名称是其描述符的一部分（§4.3），并且来自所有 CONSTANT_Class_info 结构(§4.4.1)。

由于历史原因，class 文件结构中出现的二进制名称的语法与 JLS§13.1 中记录的二进制名称语法不同。在这种内部形式中，通常分隔组成二进制名称的标识符的 ASCII 句点（.）被 ASCII 正斜杠（/）替换。标识符本身必须是非限定名称（§4.2.2）。

例如，类 Thread 的普通二进制名称是 java.lang.Thread。在 class 文件格式的描述符中使用的内部形式中，

对类 Thread 名称的引用是使用表示字符串 java/lang/Thread 的 CONSTANT_Utf8_info 结构实现的。

4.2.2 非限定名

方法、字段、局部变量和形式参数的名称存储为非限定名称。非限定名称必须至少包含一个 Unicode 代码点，并且不得包含任何 ASCII 字符. ; [/（即，句号、分号、左方括号或正斜杠）。

方法名被进一步限制，除了特殊方法名 < init > 和 < clinit > （§2.9），它们不能包含 ASCII 字符 < 或 > （即左尖括号或右尖括号）。

请注意，没有方法调用指令可以引用 < clinit > ，只有 invokespecial 指令(§invokespecial)可以引用 < init > 。

4.2.3 模块和包名

Module 属性引用的模块名存储在常量池的 CONSTANT_Module_info 结构中(§4.4.11)。CONSTANT_Module_info 结构封装了一个 CONSTANT_Utf8_info 结构，该结构表示模块名。模块名不像类名和接口名那样以“内部形式”编码，也就是说，模块名中分隔标识符的 ASCII 句号(.)不会被 ASCII 正斜杠(/)替换。

模块名可以从整个 Unicode 代码空间中提取，受以下约束:

- 模块名不能包含'\u0000'到'\u001F'范围内的任何代码点。
- ASCII 反斜杠(\)保留下来用作模块名称中的转义字符。除非后跟 ASCII 反斜杠、ASCII 冒号(:)或 ASCII at 符号(@)，否则它不能出现在模块名中。ASCII 字符序列\\可用于在模块名称中编码反斜杠。

• ASCII 冒号(:)和 at 符号(@)被保留以备将来在模块名称中使用。除非它们被转义，否则它们不能出现在模块名称中。ASCII 字符序列\:和\@可以用来编码模块名称中的冒号和@符号。

Module 属性引用的包名存储在常量池的 CONSTANT_Package_info 结构中(§4.4.12)。CONSTANT_Package_info 结构封装了一个 CONSTANT_Utf8_info 结构，该结构表示以内部形式编码的包名。

4.3 描述符

描述符是表示字段或方法类型的字符串。描述符在 class 文件格式中使用修改过的 UTF-8 字符串(§4.4.7)表示，因此可以从整个 Unicode 代码空间中提取。

4.3.1 语法符号

描述符使用语法指定。语法是一组结果，描述字符序列如何形成各种语法正确的描述符。语法的终止符以固定宽度字体显示。非终止符以斜体显示。非终止符的定义由定义的非终止符的名称和冒号引入。非终止符的一个或多个可选定义紧跟在后续行之后。

产品右侧的语法{x}表示 x 出现的次数为零或多次。

产品右侧的短语(之一)表示以下行中的每个终止符都是一个替代定义。

4.3.2 字段描述符

字段描述符表示类、实例或局部变量的类型。

FieldDescriptor:
  FieldType

FieldType:
  BaseType
  ObjectType
  ArrayType

BaseType:
  (one of)
  `B C D F I J S Z`

ObjectType:
  `L` *ClassName* `;`

ArrayType:
  `[` *ComponentType*

ComponentType:
  FieldType

BaseType 的字符 L,ObjectType 的;, ArrayType 的 [都是 ASCII 字符。

ClassName 表示以内部形式编码的二进制类或接口名称(§4.2.1)。

将字段描述符解释为类型如表 4.3-A 所示。

表示数组类型的字段描述符仅在表示 255 或更少维度的类型时才有效。

表 4.3-A. 字段描述符的说明

FieldType 术语	类型	说明
B	byte	有符号字节
C	char	基本多语种平面中的 Unicode 字符码点，用 UTF-16 编码
D	double	双精度浮点值
F	float	单精度浮点值
I	int	整型
J	long	长整型
L *ClassName* ;	reference	类 ClassName 的实例
S	short	有符号短整型
Z	boolean	true 或 false
[	reference	一维数组

Int 类型的实例变量的字段描述符就是 I。

Object 类型的实例变量的字段描述符是 LJava/lang/Object; 。请注意，使用的是类 Object 的二进制名称的内部形式。

多维数组类型的实例变量的字段描述符 double[] [][]是[[[D。

4.3.3 方法描述符

方法描述符包含零个或多个参数描述符，表示该方法接受的参数类型，以及一个返回描述符，表示该方法返回的值的类型(如果有的话)。

MethodDescriptor:
(*{ParameterDescriptor}*) *ReturnDescriptor*

ParameterDescriptor:
FieldType

ReturnDescriptor:
FieldType
VoidDescriptor

VoidDescriptor:
V

字符 V 表示该方法不返回值(其结果为 void)。

以下方法的方法描述符:

```
Object m(int i, double d, Thread t) {...}
```

是

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```

请注意，使用的是 Thread 和 Object 二进制名称的内部形式。

方法描述符仅在表示总长度为 255 或更小的方法参数时有效，其中该长度包括在实例或接口方法调用的情况下对 this 的贡献。通过对单个参数的贡献求和来计算总长度，其中 long 或 double 类型的参数对长度贡献两个单位，而任何其他类型的参数贡献一个单位。

无论描述的方法是类方法还是实例方法，方法描述符都是相同的。尽管传递了实例方法 this，但除了预期的参数之外，对调用该方法的对象的引用也没有反映在方法描述符中。对 this 的引用是通过调用实例方法的 Java 虚拟机指令隐式传递的(§2.6.1，§4.11)。

4.4 常量池

Java 虚拟机指令不依赖于类、接口、类实例或数组的运行时布局。相反，指令引用 constant_pool 表中的符号信息。

所有 constant_pool 表条目具有以下通用格式:

```
cp_info {
  u1 tag;
  u1 info[];
}
```

constant_pool 表中的每个条目必须以一个 1 字节标签开始，该标签指示条目所表示的常量类型。表 4.4-A 中列出了 17 种常量及其相应的标签，并在本章中按章节编号排序。每个标签字节后面必须紧跟两个或多个字节，提供有关特定常量的信息。附加信息的格式取决于标签字节，即 info 数组的内容随 tag 值而变化。

表 4.4-A. 常量池标签(按节)

常量类型	标签	小节
CONSTANT_Class	7	§4.4.1
CONSTANT_Fieldref	9	§4.4.2
CONSTANT_Methodref	10	§4.4.2
CONSTANT_InterfaceMethodref	11	§4.4.2
CONSTANT_String	8	§4.4.3
CONSTANT_Integer	3	§4.4.4
CONSTANT_Float	4	§4.4.4

CONSTANT_Long	5	§4.4.5
CONSTANT_Double	6	§4.4.5
CONSTANT_NameAndType	12	§4.4.6
CONSTANT_Utf8	1	§4.4.7
CONSTANT_MethodHandle	15	§4.4.8
CONSTANT_MethodType	16	§4.4.9
CONSTANT_Dynamic	17	§4.4.10
CONSTANT_InvokeDynamic	18	§4.4.10
CONSTANT_Module	19	§4.4.11
CONSTANT_Package	20	§4.4.12

在版本号为 v 的 class 文件中，constant_pool 表中的每个条目必须有一个标签，该标签首先在 class 文件格式的版本 v 或更早版本中定义（§4.1）。也就是说，每个条目必须表示一种被批准在 class 文件中使用的常量。表 4.4-B 列出了每个标签及其定义的 class 文件格式的第一个版本。还显示了引入该版本的 class 文件格式的 Java SE 平台的版本。

表 4.4-B. 常量池标签 (按标签)

常量类型	标签	class 文件格式	Java SE
CONSTANT_Utf8	1	45.3	1.0.2
CONSTANT_Integer	3	45.3	1.0.2
CONSTANT_Float	4	45.3	1.0.2
CONSTANT_Long	5	45.3	1.0.2
CONSTANT_Double	6	45.3	1.0.2
CONSTANT_Class	7	45.3	1.0.2
CONSTANT_String	8	45.3	1.0.2
CONSTANT_Fieldref	9	45.3	1.0.2
CONSTANT_Methodref	10	45.3	1.0.2
CONSTANT_InterfaceMethodref	11	45.3	1.0.2
CONSTANT_NameAndType	12	45.3	1.0.2
CONSTANT_MethodHandle	15	51.0	7
CONSTANT_MethodType	16	51.0	7
CONSTANT_Dynamic	17	55.0	11
CONSTANT_InvokeDynamic	18	51.0	7
CONSTANT_Module	19	53.0	9

CONSTANT_Package	20	53.0	9

constant_pool 表中的一些条目是可加载的，因为它们表示可以在运行时推送到栈上以实现进一步计算的实体。在版本号为 v 的 class 文件中，如果 constant_pool 表中的条目具有第一次被视为在 class 文件格式的版本 v 或更早版本中可加载的标记，则该条目是可加载的。表 4.4-C 列出了每一个标签，以及被认为可加载的 class 文件格式的第一个版本。还显示了引入该版本的 class 文件格式的 JavaSE 平台的版本。

在除 CONSTANT_Class 之外的所有情况下，标签首先被视为可加载的，在与第一次定义标签的 class 文件格式版本相同的版本中。

表 4.4-C. 可加载常量池标签

常量类型	标签	class 文件格式	Java SE
CONSTANT_Integer	3	45.3	1.0.2
CONSTANT_Float	4	45.3	1.0.2
CONSTANT_Long	5	45.3	1.0.2
CONSTANT_Double	6	45.3	1.0.2
CONSTANT_Class	7	49.0	5.0
CONSTANT_String	8	45.3	1.0.2
CONSTANT_MethodHandle	15	51.0	7
CONSTANT_MethodType	16	51.0	7
CONSTANT_Dynamic	17	55.0	11

4.4.1 CONSTANT_Class_info 结构

CONSTANT_Class_info 结构用来表示一个类或接口：

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

CONSTANT_Class_info 结构的项如下所示：

`tag`

tag 项的值为 CONSTANT_Class (7)。

`name_index`

name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体(§4.4.7)，表示以内部形式编码的有效二进制类或接口名称(§4.2.1)。

因为数组是对象，操作码 anewarray 和 multianewarray-而不是操作码 new——可以通过

constant_pool 表中的 CONSTANT_Class_info 结构引用数组“类”。 对于这样的数组类，类名是数组类型的描述符(§4.3.2)。

例如, 表示二维数组类型 int[][]的类名是[[I, 而表示类 Thread[]的类名是 [Ljava/lang/Thread;.

数组类型描述符只有在表示 255 或更少维时才有效。

4.4.2 CONSTANT_Fieldref_info, CONSTANT_Methodref_info 和 CONSTANT_InterfaceMethodref_info 结构

字段、方法和接口方法用类似的结构表示:

```
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_InterfaceMethodref_info { u1 tag; u2 class_index;
    u2 name_and_type_index;
}
```

这些结构的项如下:

`tag`

CONSTANT_Fieldref_info 结构的 tag 项的值为 CONSTANT_Fieldref (9)。

CONSTANT_Methodref_info 结构的 tag 项的值为 CONSTANT_Methodref (10)。

CONSTANT_InterfaceMethodref_info 结构的 tag 项的值为 CONSTANT_InterfaceMethodref (11)。

`class_index`

class_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构体(§4.4.1)，该结构体表示具有该字段或方法成员的类或接口类型。

在 CONSTANT_Fieldref_info 结构中，class_index 项可以是类类型也可以是接口类型。

在 CONSTANT_Methodref_info 结构中，class_index 项应该是类类型，而不是接口类型。

在 CONSTANT_InterfaceMethodref_info 结构中，class_index 项应该是接口类型，而不是类类型。

`name_and_type_index`

name_and_type_index 项的值必须是 constant_pool 表的有效索引。该索引的

constant_pool 条目必须是一个 CONSTANT_NameAndType_info 结构体(§4.4.6)。这个 constant_pool 条目表示字段或方法的名称和描述符。

在 CONSTANT_Fieldref_info 结构中，指定的描述符必须是字段描述符(§4.3.2)。否则，指定的描述符必须是一个方法描述符(§4.3.3)。

如果在 CONSTANT_Methodref_info 结构中的方法名以'<' ('\u003c')开始，那么该名称必须是特殊名称<init>，代表一个实例初始化方法(§2.9.1)。这个方法的返回类型必须是 void。

4.4.3 CONSTANT_String_info 结构

CONSTANT_String_info 结构用来表示 String 类型的常量对象：

```
CONSTANT_String_info {
    u1 tag;
    u2 string_index;
}
```

CONSTANT_String_info 结构的项如下所示：

`tag`

tag 项的值为 CONSTANT_String (8)。

`string_index`

string_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示 String 对象要初始化的 Unicode 代码点序列。

4.4.4 CONSTANT_Integer_info 和 CONSTANT_Float_info 结构

CONSTANT_Integer_info 和 CONSTANT_Float_info 结构表示 4 字节数字常量 (int 和 float):

```
CONSTANT_Integer_info {
    u1 tag;
    u4 bytes;
}

CONSTANT_Float_info {
    u1 tag;
    u4 bytes;
}
```

这些结构的项如下所示：

`tag`

CONSTANT_Integer_info 结构的 tag 项的值为 CONSTANT_Integer (3)。

CONSTANT_Float_info 结构的 tag 项的值为 CONSTANT_Float (4)。

`bytes`

CONSTANT_Integer_info 结构的 bytes 项表示 int 常量的值。值的字节按大端顺序存储

(高字节优先)。

CONSTANT_Float_info 结构的 bytes 项表示 IEEE 754 binary32 浮点格式(§2.3.2)中 float 常量的值。条目的字节以大端序(高字节优先)存储。

CONSTANT_Float_info 结构表示的值如下所示。该值的字节首先转换为 int 常量 bits。则:

- 如果 bits 是 0x7f800000，float 值将是正无穷大。
- 如果 bits 是 0xff800000，float 值将是负无穷大。
- 如果 bits 在 0x7f800001 到 0x7fffffff 范围内或在 0xff800001 到 0xffffffff 范围内，float 值将为 NaN。
- 在所有其他情况下，让 s, e 和 m 是三个值，可以从 bits 计算:

```
int s = ((bits >> 31) == 0) ? 1 : -1;
int e = ((bits >> 23) & 0xff);
int m = (e == 0) ?
          (bits & 0x7fffff) << 1 :
                 (bits & 0x7fffff) | 0x800000;
```

则 float 的值等于数学表达式 $s \cdot m \cdot 2^{e-150}$ 的结果。

4.4.5 CONSTANT_Long_info 和 CONSTANT_Double_info 结构

CONSTANT_Long_info 和 CONSTANT_Double_info 表示 8 字节数字常量(long 和 double):

```
CONSTANT_Long_info {
  u1 tag;
  u4 high_bytes;
  u4 low_bytes;
}

CONSTANT_Double_info {
  u1 tag;
  u4 high_bytes;
  u4 low_bytes;
}
```

所有 8 字节常量在 class 文件的 constant_pool 表中占据两个条目。如果 CONSTANT_Long_info 或 CONSTANT_Double_info 结构是 constant_pool 表中第 n 个索引的表项，那么表中的下一个可用表项位于第 n+2 个索引。constant_pool 索引 n+1 必须是有效的，但被认为是不可用的。

回想起来，让 8 字节常量接受两个常量池条目是一个糟糕的选择。

这些结构的项如下所示:

`tag`

CONSTANT_Long_info 结构的 tag 项的值为 CONSTANT_Long (5)。

CONSTANT_Double_info 结构的 tag 项的值为 CONSTANT_Double (6)。

`high_bytes, low_bytes`

CONSTANT_Long_info 结构的无符号 high_bytes 和 low_bytes 项一起表示 long 常量的值

```
((long) high_bytes << 32) + low_bytes
```

其中 high_bytes 和 low_bytes 的每个字节都以大端序(高字节优先)存储。

CONSTANT_Double_info 结构的 high_bytes 和 low_bytes 项一起表示 IEEE 754 binary64 浮点格式(§2.3.2)中的 double 值。每个项的字节都以大端序(高字节优先)存储。

CONSTANT_Double_info 结构表示的值如下所示。high_bytes 和 low_bytes 项被转换为 long 常量 bits，它等于

```
((long) high_bytes << 32) + low_bytes
```

则:

- 如果 bits 为 0x7ff0000000000000L，则 double 值为正无穷大。
- 如果 bits 为 0xfff0000000000000L，则 double 值为负无穷大。
- 如果 bits 在 0x7ff0000000000001L 到 0x7fffffffffffffl 之间，或者在 0xfff0000000000001L 到 0xffffffffffffffffL 之间，则 double 值为 NaN。
- 在所有其他情况下，让 s, e 和 m 是三个值，可以从 bits 计算:

```
int s = ((bits >> 63) == 0) ? 1 : -1;
int e = (int)((bits >> 52) & 0x7ffL);
long m = (e == 0) ?
             (bits & 0xfffffffffffffL) << 1 :
         (bits & 0xfffffffffffffL) | 0x10000000000000L;
```

然后浮点值等于数学表达式 $s \cdot m \cdot 2^{e-1075}$ 的 double 值。

4.4.6 CONSTANT_NameAndType_info 结构

CONSTANT_NameAndType_info 结构用于表示一个字段或方法，不表明它属于哪个类或接口类型:

```
CONSTANT_NameAndType_info {
    u1 tag;
    u2 name_index;
    u2 descriptor_index;
}
```

CONSTANT_NameAndType_info 结构的项如下所示:

`tag`

tag 项的值为 CONSTANT_NameAndType (12)。

`name_index`

name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体(§4.4.7)，该结构体要么表示一个有效的非限定名，用来表示一个字段或方法(§4.2.2)，要么表示特殊的方法名<init>(§2.9.1)。

`descriptor_index`

descriptor_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体(§4.4.7)，表示一个有效的字段描述符或方法描述符(§4.3.2，§4.3.3)。

4.4.7 `CONSTANT_Utf8_info` 结构

CONSTANT_Utf8_info 结构用于表示常量字符串值:

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

CONSTANT_Utf8_info 结构的项如下所示:

`tag`

tag 项的值是 CONSTANT_Utf8(1)。

`length`

length 项的值给出了 bytes 数组中的字节数(而不是结果字符串的长度)。

`bytes[]`

bytes 数组包含字符串的字节。

没有字节的值是(byte)0。

没有字节可以在(byte)0xf0 到(byte)0xff 的范围内。

字符串内容用修改后的 UTF-8 编码。对经过修改的 UTF-8 字符串进行编码，使只包含非空 ASCII 字符的代码点序列可以使用每个代码点只使用一个字节表示，但可以表示 Unicode 代码空间中的所有代码点。修改后的 UTF-8 字符串不再以空结束。编码方式如下:

- “\u0001”到“\u007F”范围内的代码点由单个字节表示:

0	*bits 6-0*

  字节中的 7 位数据表示所表示的代码点的值。

- 空代码点('\u0000')和在'\u0080'到'\u07FF'范围内的代码点由一对字节 x 和 y 表示:

x:	1	1	0	bits 10-6
y:	1	0	bits 5-0	

这两个字节表示带有值的代码点:

```
((x & 0x1f) << 6) + (y & 0x3f)
```

- “\u0800”到“\uFFFF”范围内的代码点由 3 个字节的 x, y 和 z 表示:

x:	1	1	1	0	bits 15-12
y:	1	0	bits 11-6		
z:	1	0	bits 5-0		

这三个字节表示带有值的代码点:

```
((x & 0xf) << 12) + ((y & 0x3f) << 6) + (z & 0x3f)
```

- 代码点在 U+FFFF 以上的字符(所谓的补充字符)通过分别编码其 UTF-16 表示的两个代理代码单元来表示。每个代理代码单元由三个字节表示。这意味着补充字符由六个字节表示，u, v, w, x, y 和 z:

u:	1	1	1	0	1	1	0	1
v:	1	0	1	0	(bits 20-16)-1			
w:	1	0	bits 15-10					
x:	1	1	1	0	1	1	0	1
y:	1	0	1	1	bits 9-6			
z:	1	0	bits 5-0					

这 6 个字节表示带有值的代码点:

```
0x10000 + ((v & 0x0f) << 16) + ((w & 0x3f) << 10) + ((y & 0x0f) << 6) + (z
& 0x3f)
```

多字节字符的字节以大端（高字节优先）顺序存储在 class 文件中。

此格式与“标准”UTF-8 格式之间有两个区别。首先，空字符（char）0 使用 2 字节格式而不是 1 字节格式进行编码，这样修改后的 UTF-8 字符串就不会嵌入空字符。其次，仅使用标准 UTF-8 的 1 字节、2 字节和 3 字节格式。Java 虚拟机不能识别标准 UTF-8 的四字节格式;它使用自己的两倍三字节格式。

有关标准 UTF-8 格式的更多信息，请参阅 Unicode 标准版本 13.0 的第 3.9 节 Unicode 编码形式。

4.4.8 CONSTANT_MethodHandle_info 结构

CONSTANT_MethodHandle_info 结构用来表示方法句柄:

```
CONSTANT_MethodHandle_info {
    u1 tag;
    u1 reference_kind;
    u2 reference_index;
}
```

CONSTANT_MethodHandle_info 结构的项如下所示:

`tag`

tag 项的值为 CONSTANT_MethodHandle (15)。

`reference_kind`

reference_kind 的取值范围是 1 ~ 9。该值表示该方法句柄的类型，它表征了其字节码行为(§5.4.3.5)。

`reference_index`

reference_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须如下所示:

- 如果 reference_kind 项的值是 1 (REF_getField), 2 (REF_getStatic), 3 (REF_putField), 或 4 (REF_putStatic), 那么该索引的 constant_pool 条目必须是一个 CONSTANT_Fieldref_info 结构体(§4.4.2)，表示要为其创建方法句柄的字段。

- 如果 reference_kind 项的值是 5 (REF_invokeVirtual)或 8 (REF_newInvokeSpecial), 那么该索引的 constant_pool 条目必须是一个 CONSTANT_Methodref_info 结构体(§4.4.2)，表示要为其创建方法句柄的类的方法或构造函数(§2.9.1)。

- 如果 reference_kind 项的值是 6 (REF_invokeStatic)或 7 (REF_invokeSpecial), 如果 class 文件版本号小于 52.0，该索引的 constant_pool 条目必须是一个 CONSTANT_Methodref_info 结构体，表示要为其创建方法句柄的类的方法; 如 class 文件版本号为 52.0 或以上，该索引的 constant_pool 条目必须是一个 CONSTANT_Methodref_info 结构体或一个 CONSTANT_InterfaceMethodref_info 结构体(§4.4.2)，该结构体表示要为其创建方法句柄的类或接口的方法。

- 如果 reference_kind 项的值是 9(REF_invokeInterface), 那么该索引的 constant_pool 条目必须是一个 CONSTANT_InterfaceMethodref_info 结构体，表示要为其创建方法句柄的接口方法。

如果 reference_kind 项的值是 5(REF_invokeVirtual), 6 (REF_invokeStatic), 7 (REF_invokeSpecial), 或 9 (REF_invokeInterface), 由 CONSTANT_Methodref_info 结构体或 CONSTANT_InterfaceMethodref_info 结构体表示的方法名不能为<init>或<clinit>。

如果值为 8(REF_newInvokeSpecial), CONSTANT_Methodref_info 结构表示的方法名必须

是<init>。

4.4.9 CONSTANT_MethodType_info 结构

CONSTANT_MethodType_info 结构用来表示方法类型:

```
CONSTANT_MethodType_info {
    u1 tag;
    u2 descriptor_index;
}
```

CONSTANT_MethodType_info 结构的项如下所示:

tag

tag 项的值为 CONSTANT_MethodType (16)。

descriptor_index

descriptor_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体(§4.4.7)，表示一个方法描述符(§4.3.3)。

4.4.10 CONSTANT_Dynamic_info 和 CONSTANT_InvokeDynamic_info 结构

constant_pool 表中的大多数结构通过组合表中静态记录的名称、描述符和值直接表示实体。相反，CONSTANT_Dynamic_info 和 CONSTANT_InvokeDynamic_info 结构通过指向动态计算实体的代码间接表示实体。该代码被称为 bootstrap 方法，由 Java 虚拟机在解析来自这些结构的符号引用时调用(§5.1，§5.4.3.6)。每个结构指定一个引导方法以及一个辅助名称和类型，这些名称和类型表示要计算的实体。更详细地:

• CONSTANT_Dynamic_info 结构用来表示一个动态计算的常数，它是在 ldc 指令(§ldc)过程中通过调用 bootstrap 方法产生的任意值。由结构指定的辅助类型限制动态计算常数的类型。

• CONSTANT_InvokeDynamic_info 结构用来表示一个动态计算的调用站点，一个 java.lang.invoke.CallSite 的实例，它是在 invokedynamic 指令过程中通过调用 bootstrap 方法产生的(§invokedynamic)。该结构指定的辅助类型限制动态计算调用站点的方法类型。

```
CONSTANT_Dynamic_info {
    u1 tag;
    u2 bootstrap_method_attr_index;
    u2 name_and_type_index;
}

CONSTANT_InvokeDynamic_info {
    u1 tag;
    u2 bootstrap_method_attr_index;
    u2 name_and_type_index;
}
```

这些结构的项如下所示:

tag

CONSTANT_Dynamic_info 结构 tag 项的值为 CONSTANT_Dynamic (17)。

CONSTANT_InvokeDynamic_info 结构的 tag 项的值为 CONSTANT_InvokeDynamic (18)。

`bootstrap_method_attr_index`

bootstrap_method_attr_index 项的值必须是该 class 文件 bootstrap 方法表 bootstrap_methods 数组的有效索引(§4.7.23)。

CONSTANT_Dynamic_info 结构是唯一的，因为它们在语法上允许通过 bootstrap 方法表引用自己。与其强制在类加载时检测这些循环(这是一个潜在的昂贵检查)，我们最初允许循环，但强制在解析时失败(§5.4.3.6)。

`name_and_type_index`

name_and_type_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_NameAndType_info 结构体(§4.4.6)。这个 constant_pool 条目表示名称和描述符。

在 CONSTANT_Dynamic_info 结构中，指定的描述符必须是字段描述符(§4.3.2)。

在 CONSTANT_InvokeDynamic_info 结构中，指定的描述符必须是一个方法描述符(§4.3.3)。

4.4.11 `CONSTANT_Module_info` 结构

CONSTANT_Module_info 结构用来表示一个模块:

```
CONSTANT_Module_info {
  u1 tag;
  u2 name_index;
}
```

CONSTANT_Module_info 结构的项如下:

`tag`

tag 项的值是 CONSTANT_Module(19)。

`name_index`

name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，代表一个有效的模块名(§4.2.3)。

CONSTANT_Module_info 结构只允许在声明模块的 class 文件的常量池中使用，即 access_flags 项具有 ACC_MODULE 标志集的 ClassFile 结构。在所有其他 class 文件中，CONSTANT_Module_info 结构是不合法的。

4.4.12 `CONSTANT_Package_info` 结构

CONSTANT_Package_info 结构用于表示一个被模块导出或打开的包:

```
CONSTANT_Package_info {
  u1 tag;
  u2 name_index;
```

```
}
```

CONSTANT_Package_info 结构的内容如下:

`tag`

tag 项的值是 CONSTANT_Package(20)。

`name_index`

name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示以内部形式编码的有效包名(§4.2.3)。

CONSTANT_Package_info 结构只允许在声明模块的 class 文件的常量池中使用，即 access_flags 项具有 ACC_MODULE 标志集的 ClassFile 结构。在所有其他 class 文件中，CONSTANT_Package_info 结构是非法的。

4.5 字段

每个字段都由一个 field_info 结构描述。

一个 class 文件中不能有两个字段具有相同的名称和描述符(§4.3.2)。

该结构的格式如下:

```
field_info {
    u2             access_flags;
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

field_info 结构的项如下:

`access_flags`

access_flags 项的值是用于表示该字段的访问权限和属性的标签的掩码。当设置时，每个标签的说明，在表 4.5-A 中指定。

表 4.5-A. 字段访问和属性标志

标签名	值	说明
ACC_PUBLIC	0x0001	声明为 public; 可以从包外访问。
ACC_PRIVATE	0x0002	声明为 private; 只能在定义类和属于同一嵌套的其他类中访问（§5.4.4）。
ACC_PROTECTED	0x0004	声明为 protected; 可以在子类中访问。

ACC_STATIC	0x0008	声明为 static。
ACC_FINAL	0x0010	声明为 final;从未在对象构造后直接被赋值(JLS §17.5).
ACC_VOLATILE	0x0040	声明为 volatile; 不能被缓存。
ACC_TRANSIENT	0x0080	声明为 transient; 不由持久对象管理器写入或读取。
ACC_SYNTHETIC	0x1000	声明为 synthetic; 源代码中不存在。
ACC_ENUM	0x4000	声明为枚举类的元素。

类字段可以设置表 4.5-A 中的任何标志。然而，类的每个字段最多可以设置一个 ACC_PUBLIC、ACC_ PRIVATE 和 ACC_PROTECTED 标志（JLS§8.3.1），并且不能同时设置 ACC_FINAL 和 ACC_VOLATILE 标志(JLS §8.3.1.4)。

接口字段必须设置其 ACC_PUBLIC、ACC_STATIC 和 ACC_FINAL 标志；它们可以设置 ACC_SYNTHETIC 标志，并且不得设置表 4.5-A 中的任何其他标志（JLS§9.3）。

ACC_SYNTHETIC 标志表示该字段是由编译器生成的，不出现在源代码中。

ACC_ENUM 标志表示该字段用于保存枚举类的元素（JLS§8.9）。

表 4.5-A 中未分配的 access_flags 项的所有位保留供将来使用。在生成的 class 文件中，它们应设置为零，Java 虚拟机实现应忽略它们。

`name_index`

name_index 项的值必须是 constant_pool 表中的有效索引。该索引处的 constant_pool 条目必须是 CONSTANT_Utf8_info structure 结构（§4.4.7），表示字段的有效非限定名称（§4.2.2）。

`descriptor_index`

descriptor_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，它代表一个有效的字段描述符(§4.3.2)。

`attributes_count`

attributes_count 项的值表示该字段的附加属性的数量。

`attributes[]`

attributes 表的每个值都必须是 attribute_info 结构(§4.7)。

一个字段可以有任意数量的与之关联的可选属性。

本规范定义的属性出现在 field_info 结构的 attributes 表中，如表 4.7-C 所示。

关于定义在 field_info 结构的 attributes 表中的属性的规则见§4.7。

关于 field_info 结构的 attributes 表中非预定义属性的规则见§4.7.1。

4.6 方法

每个方法，包括每个实例初始化方法(§2.9.1)和类或接口初始化方法(§2.9.2)，都用 method_info 结构描述。

一个 class 文件中的两个方法不能有相同的名称和描述符(§4.3.3)。

该结构的格式如下:

```
method_info {
    u2             access_flags;
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

method_info 结构的项如下:

`access_flags`

access_flags 项的值是一个标志掩码，用于表示该方法的访问权限和属性。当设置每个标志时，其说明如表 4.6-A 所示。

表 4.6-A. 方法访问和属性标志

标志名	值	说明
ACC_PUBLIC	0x0001	声明为 public; 可以从包外访问。
ACC_PRIVATE	0x0002	声明为 private; 只能在定义类和属于同一嵌套的其他类中访问（§5.4.4）。
ACC_PROTECTED	0x0004	声明为 protected; 可以在子类中访问。
ACC_STATIC	0x0008	声明为 static。
ACC_FINAL	0x0010	声明为 final; 不能被重写 (§5.4.5)。
ACC_SYNCHRONIZED	0x0020	声明为 synchronized; 调用由监视器使用包装。
ACC_BRIDGE	0x0040	由编译器生成的桥接方法。
ACC_VARARGS	0x0080	使用可变数量的参数声明。
ACC_NATIVE	0x0100	声明为 native; 使用 Java 编程语言以外的语言实现。
ACC_ABSTRACT	0x0400	声明 abstract; 不提供实现。

ACC_STRICT	0x0800	主版本号至少为 46，最多为 60 的 class 文件中: 声明为 strictfp。
ACC_SYNTHETIC	0x1000	声明为 synthetic; 没有出现在源代码中。

值 0x0800 仅在主版本号至少为 46 且不超过 60 的 class 文件中被解释为 ACC_STRICT 标志。对于此 class 文件中的方法，下面的规则确定 ACC_STRICT 标志是否可以与其他标志一起设置。(在 Java SE 1.2 到 16(§2.8)中，设置 ACC_STRICT 标志限制了方法的浮点指令。) 对于 class 文件中主版本号小于 46 或大于 60 的方法，值 0x0800 不解释为 ACC_STRICT 标志，而是没有赋值;在这样的 class 文件中设置 ACC_STRICT 标志是没有意义的。

类的方法可以设置表 4.6-A 中的任何标志。然而，类的每个方法最多可以设置 ACC_PUBLIC、ACC_PRIVATE 和 ACC_PROTECTED 标志其中一个（JLS§8.4.3）。

接口方法可设置表 4.6-A 中的任何标志，但 ACC_PROTECTED、ACC_FINAL、ACC_SYNCHRONIZED 和 ACC_NATIVE（JLS§9.4）除外。在版本号小于 52.0 的 class 文件中，接口的每个方法必须设置其 ACC_PUBLIC 和 ACC_ ABSTRACT 标志；在版本号为 52.0 或更高的 class 文件中，接口的每个方法必须设置其 ACC_PUBLIC 和 ACC_PRIVATE 标志中的一个。

如果类或接口的方法设置了其 ACC_ABSTRACT 标志，则它不能设置任何 ACC_ PRIVATE、ACC_STATIC、ACC_FINAL、ACC_SYNCHRONIZED 或 ACC_NATIVE 标志，也不能（在主版本号至少为 46 且最多为 60 的 class 文件中）设置其 ACC_STRICT 标志。

实例初始化方法（§2.9.1）可以设置最多一个 ACC_PUBLIC、ACC_PRIVATE 和 ACC_PROTECTED 标志，也可以设置 ACC_VARARGS 和 ACC_SYNTHETIC 标志，并且可以（在主版本号至少为 46 且最多为 60 的 class 文件中）设置 ACC_STRICT 标志，但不得设置表 4.6-A 中的任何其他标志。

在版本号为 51.0 或更高版本的 class 文件中，名为 < clinit > 的方法必须设置 ACC_STATIC 标志。

Java 虚拟机隐式调用类或接口初始化方法（§2.9.2）。除了 ACC_STATIC 标志和（在主版本号至少为 46 且最多为 60 的 class 文件中）ACC_STRICT 标志的设置之外，其 access_flags 的值被忽略，并且该方法不受前述关于标志合法组合的规则的约束。

ACC_BRIDGE 标志用于指示由 Java 编程语言的编译器生成的桥接方法。

ACC_VARARGS 标志表示该方法在源代码级别接受可变数量的参数。声明为采用可变数量参数的方法必须在 ACC_VARARGS 标志设置为 1 的情况下编译。所有其他方法必须在 ACC_VARARGS 标志设置为 0 的情况下编译。

ACC_SYNTHETIC 标志表示该方法是由编译器生成的，不会出现在源代码中，除非它是 §4.7.8 中指定的方法之一。

表 4.6-A 中未赋值的 access_flags 项的所有位保留供将来使用。(这包括主版本号小于 46 或大于 60 的 class 文件中对应于 0x0800 的位。) 在生成的 class 文件中，它们应设置为零，Java 虚拟机实现应忽略它们。

`name_index`

name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示一个有效的非限定名称(§4.2.2)，或者(如果该方法在类而不是接口中)特殊方法名称<init>，或者特殊方法名称<clinit>。

`descriptor_index`

descriptor_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体，表示一个有效的方法描述符(§4.3.3)。并且:

- 如果这个方法在一个类中而不是在一个接口中，并且方法的名称是<init>，那么描述符必须表示一个 void 方法。
- 如果方法的名称是<clinit>，那么描述符必须表示一个 void 方法，并且在版本号为 51.0 或更高的 class 文件中，表示一个不带参数的方法。

如果在 access_flags 项中设置了 ACC_VARARGS 标志，该规范的未来版本可能要求方法描述符的最后一个参数描述符是数组类型。

`attributes_count`

attributes_count 项的值表示此方法的附加属性的数量。

`attributes[]`

attributes 表的每个值都必须是 attribute_info 结构(§4.7)。

方法可以有任意数量的可选属性与其关联。

本规范定义的属性出现在 method_info 结构的 attributes 表中，如表 4.7-C 所示。

关于定义在 method_info 结构的 attributes 表中出现的属性的规则见§4.7。

method_info 结构的 attributes 表中关于非预定义属性的规则见§4.7.1。

4.7 属性

属性在 class 文件格式的 ClassFile、field_info、method_info、Code_attribute 和 record_component_info 结构中使用(§4.1、§4.5、§4.6、§4.7.3、§4.7.30)。

所有属性的一般格式如下:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
```

```
}
```

对于所有属性，attribute_name_index 项必须是类常量池中的有效的无符号 16 位索引。attribute_name_index 的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示属性的名称。attribute_length 项的值以字节为单位表示后续信息的长度。长度不包括包含 attribute_name_index 和 attribute_length 项的初始 6 个字节。

该规范预定义了 30 个属性。为了便于导航，它们被列出了三次:

- 表 4.7-A 按属性在本章中的章节号排序。每个属性都显示了定义它的 class 文件格式的第一个版本。同样显示的是 Java SE 平台的版本，它引入了 class 文件格式的版本(§4.1)。
- 表 4.7-B 按定义每个属性的 class 文件格式的第一个版本排序。
- 表 4.7-C 按 class 文件中每个属性定义出现的位置排序。

在本规范中它们使用的上下文中，也就是说，在它们出现的 class 文件结构的 attributes 表中，这些预定义属性的名称是保留的。

attributes 表中预定义属性存在的任何条件都将在描述该属性的部分显式指定。如果没有指定条件，则该属性可以在 attributes 表中出现任意次数。

预定义属性根据其用途分为三组:

1. 7 个属性对于 Java 虚拟机正确解释 class 文件至关重要:

   - `ConstantValue`
   - `Code`
   - `StackMapTable`
   - `BootstrapMethods`
   - `NestHost`
   - `NestMembers`
   - `PermittedSubclasses`

   在版本号为 v 的 class 文件中，如果 Java 虚拟机的实现支持 class 文件格式的 v 版本，并且该属性是在 v 版本或更早版本中首次定义的，则必须识别并正确读取其中的每个属性，并且该属性出现在定义它出现的位置。

2. 以下十个属性对于 Java 虚拟机正确解释 class 文件不是关键的，但是对于 Java SE 平台的类库正确解释 class 文件是关键的，或者对于工具是有用的(在这种情况下，指定属性的部分将其描述为“可选”):

   - `Exceptions`

- InnerClasses
- EnclosingMethod
- Synthetic
- Signature
- Record
- SourceFile
- LineNumberTable
- LocalVariableTable
- LocalVariableTypeTable

在版本号为 v 的 class 文件中，如果 Java 虚拟机的实现支持 class 文件格式的 v 版本，并且该属性是在 v 版本或更早版本中首次定义的，则必须识别并正确读取其中的每个属性，并且该属性出现在定义它出现的位置。

3. 13 个属性对于 Java 虚拟机对 class 文件的正确解释不是至关重要的，但是包含了关于 class 文件的元数据，这些元数据要么是由 Java SE 平台的类库公开的，要么是由工具提供的(在这种情况下，指定属性的部分将其描述为“可选”):

- SourceDebugExtension
- Deprecated
- RuntimeVisibleAnnotations
- RuntimeInvisibleAnnotations
- RuntimeVisibleParameterAnnotations
- RuntimeInvisibleParameterAnnotations
- RuntimeVisibleTypeAnnotations
- RuntimeInvisibleTypeAnnotations
- AnnotationDefault
- MethodParameters
- Module
- ModulePackages
- ModuleMainClass

Java 虚拟机的实现可以使用这些属性包含的信息，否则必须静默地忽略这些属性。

表 4.7-A. 预定义的 `class` 文件属性 (按章节)

属性	章节	`class` 文件	Java SE
`ConstantValue`	§4.7.2	45.3	1.0.2
`Code`	§4.7.3	45.3	1.0.2
`StackMapTable`	§4.7.4	50.0	6
`Exceptions`	§4.7.5	45.3	1.0.2
`InnerClasses`	§4.7.6	45.3	1.1
`EnclosingMethod`	§4.7.7	49.0	5.0
`Synthetic`	§4.7.8	45.3	1.1
`Signature`	§4.7.9	49.0	5.0
`SourceFile`	§4.7.10	45.3	1.0.2
`SourceDebugExtension`	§4.7.11	49.0	5.0
`LineNumberTable`	§4.7.12	45.3	1.0.2
`LocalVariableTable`	§4.7.13	45.3	1.0.2
`LocalVariableTypeTable`	§4.7.14	49.0	5.0
`Deprecated`	§4.7.15	45.3	1.1
`RuntimeVisibleAnnotations`	§4.7.16	49.0	5.0
`RuntimeInvisibleAnnotations`	§4.7.17	49.0	5.0
`RuntimeVisibleParameterAnnotations`	§4.7.18	49.0	5.0
`RuntimeInvisibleParameterAnnotations`	§4.7.19	49.0	5.0
`RuntimeVisibleTypeAnnotations`	§4.7.20	52.0	8
`RuntimeInvisibleTypeAnnotations`	§4.7.21	52.0	8
`AnnotationDefault`	§4.7.22	49.0	5.0
`BootstrapMethods`	§4.7.23	51.0	7
`MethodParameters`	§4.7.24	52.0	8
`Module`	§4.7.25	53.0	9
`ModulePackages`	§4.7.26	53.0	9
`ModuleMainClass`	§4.7.27	53.0	9
`NestHost`	§4.7.28	55.0	11
`NestMembers`	§4.7.29	55.0	11
`Record`	§4.7.30	60.0	16
`PermittedSubclasses`	§4.7.31	61.0	17

表 4.7-B. 预定义的 `class` 文件属性 (按 `class` 文件格式)

属性	class 文件	Java SE	章节
ConstantValue	45.3	1.0.2	§4.7.2
Code	45.3	1.0.2	§4.7.3
Exceptions	45.3	1.0.2	§4.7.5
SourceFile	45.3	1.0.2	§4.7.10
LineNumberTable	45.3	1.0.2	§4.7.12
LocalVariableTable	45.3	1.0.2	§4.7.13
InnerClasses	45.3	1.1	§4.7.6
Synthetic	45.3	1.1	§4.7.8
Deprecated	45.3	1.1	§4.7.15
EnclosingMethod	49.0	5.0	§4.7.7
Signature	49.0	5.0	§4.7.9
SourceDebugExtension	49.0	5.0	§4.7.11
LocalVariableTypeTable	49.0	5.0	§4.7.14
RuntimeVisibleAnnotations	49.0	5.0	§4.7.16
RuntimeInvisibleAnnotations	49.0	5.0	§4.7.17
RuntimeVisibleParameterAnnotations	49.0	5.0	§4.7.18
RuntimeInvisibleParameterAnnotations	49.0	5.0	§4.7.19
AnnotationDefault	49.0	5.0	§4.7.22
StackMapTable	50.0	6	§4.7.4
BootstrapMethods	51.0	7	§4.7.23
RuntimeVisibleTypeAnnotations	52.0	8	§4.7.20
RuntimeInvisibleTypeAnnotations	52.0	8	§4.7.21
MethodParameters	52.0	8	§4.7.24
Module	53.0	9	§4.7.25
ModulePackages	53.0	9	§4.7.26
ModuleMainClass	53.0	9	§4.7.27
NestHost	55.0	11	§4.7.28
NestMembers	55.0	11	§4.7.29
Record	60.0	16	§4.7.30
PermittedSubclasses	61.0	17	§4.7.31

表 4.7-C. 预定义的 class 文件属性(按位置)

属性	位置	class 文件
SourceFile	ClassFile	45.3
InnerClasses	ClassFile	45.3
EnclosingMethod	ClassFile	49.0
SourceDebugExtension	ClassFile	49.0
BootstrapMethods	ClassFile	51.0
Module, ModulePackages, ModuleMainClass	ClassFile	53.0
NestHost, NestMembers	ClassFile	55.0
Record	ClassFile	60.0
PermittedSubclasses	ClassFile	61.0
ConstantValue	field_info	45.3
Code	method_info	45.3
Exceptions	method_info	45.3
RuntimeVisibleParameterAnnotations, RuntimeInvisibleParameterAnnotations	method_info	49.0
AnnotationDefault	method_info	49.0
MethodParameters	method_info	52.0

表 4.7-C (续). 预定义的 class 文件属性 (按位置)

属性	位置	class 文件
Synthetic	ClassFile, field_info, method_info	45.3
Deprecated	ClassFile, field_info, method_info	45.3
Signature	ClassFile, field_info, method_info, record_component_info	49.0
RuntimeVisibleAnnotations, RuntimeInvisibleAnnotations	ClassFile, field_info, method_info, record_component_info	49.0
LineNumberTable	Code	45.3
LocalVariableTable	Code	45.3
LocalVariableTypeTable	Code	49.0
StackMapTable	Code	50.0
RuntimeVisibleTypeAnnotations, RuntimeInvisibleTypeAnnotations	ClassFile, field_info, method_info, Code, record_component_info	52.0

4.7.1 定义和命名新属性

允许编译器在 class 文件结构、field_info 结构、method_info 结构和 Code 属性的 attributes 表中定义和发出包含新属性的 class 文件(§4.7.3)。允许 Java 虚拟机实现识别和使用这些 attributes 表中的新属性。但是，任何未定义为本规范一部分的属性不得影响 class 文件的语义。Java 虚拟机实现需要静默地忽略它们无法识别的属性。

例如，允许定义新属性以支持特定于供应商的调试。因为 Java 虚拟机实现需要忽略它们无法识别的属性，所以用于该特定 Java 虚拟机实现的 class 文件将被其他实现使用，即使这些实现不能利用 class 文件包含的额外调试信息。

Java 虚拟机实现被特别禁止抛出异常或仅仅因为某些新属性的存在而拒绝使用 class 文件。当然，如果给定的 class 文件不包含它们所需的所有属性，操作 class 文件的工具可能无法正确运行。

两个本来是不同的属性，但恰巧使用了相同的属性名和相同的长度，在识别这两个属性的

实现上会发生冲突。在本规范之外定义的属性应该根据 Java 语言规范，Java SE 19 版本(JLS§6.1)中描述的包命名约定来选择名称。

该规范的未来版本可能会定义额外的属性。

4.7.2 ConstantValue 属性

ConstantValue 属性是 field_info 结构的 attributes 表中的固定长度属性(§4.5)。ConstantValue 属性表示常量表达式的值(JLS§15.28)，用法如下:

- 如果设置了 field_info 结构的 access_flags 项中的 ACC_STATIC 标志，那么由 field_info 结构表示的字段在声明该字段的类或接口的初始化时被赋值，该值由其 ConstantValue 属性表示(§5.5)。这发生在调用该类或接口的类或接口初始化方法之前(§2.9.2)。
- 否则，Java 虚拟机必须静默地忽略该属性。

field_info 结构的 attributes 表中最多只能有一个 ConstantValue 属性。

ConstantValue 属性的格式如下:

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}
```

ConstantValue_attribute 结构的内容如下:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "ConstantValue"。

`attribute_length`

attribute_length 项的值必须为 2。

`constantvalue_index`

constantvalue_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目给出了该属性表示的值。constant_pool 表项必须是适合该字段的类型，如表 4.7.2-A 所示。

表 4.7.2-A. 常量值属性类型

字段类型	条目类型
int, short, char, byte, boolean	CONSTANT_Integer
float	CONSTANT_Float
long	CONSTANT_Long
double	CONSTANT_Double

String	CONSTANT_String

4.7.3 Code 属性

Code 属性是 method_info 结构的 attributes 表中的一个变长属性(§4.6)。Code 属性包含 Java 虚拟机指令和方法的辅助信息，包括实例初始化方法和类或接口初始化方法(§2.9.1，§2.9.2)。

如果方法是 native 或 abstract 的，并且不是类或接口初始化方法，那么它的 method_info 结构在其 attributes 表中不能有 Code 属性。否则，它的 method_info 结构必须在其 attributes 表中只有一个 Code 属性。

Code 属性的格式如下:

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {   u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count]; }
```

Code_attribute 结构的项如下:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串"Code"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`max_stack`

max_stack 项的值给出了该方法(§2.6.2)在执行过程中任意点的操作数栈的最大深度。

`max_locals`

max_locals 项的值表示调用该方法时(§2.6.1)分配的局部变量数组中局部变量的数量，包括在调用时传递参数给方法的局部变量。

对于 long 或 double 类型的值，最大的局部变量索引是 max_locals - 2。任何其他类型的值的最大局部变量索引是 max_locals - 1。

`code_length`

code_length 项的值给出了该方法的代码数组中的字节数。

code_length 的值必须大于零(因为代码数组不能为空)并且小于 65536。

`code[]`

代码数组给出了实现该方法的 Java 虚拟机代码的实际字节。

当将代码数组读入按字节寻址机器的内存时，如果数组的第一个字节按 4 字节边界对齐，则 tableswitch 和 lookupswitch 的 32 位偏移将按 4 字节对齐。(有关代码数组对齐结果的更多信息，请参阅这些指令的描述。)

对代码数组内容的详细限制非常广泛，并在单独章节（§4.9）中给出。

`exception_table_length`

exception_table_length 项的值给出 exception_table 数组中的条目数。

`exception_table[]`

exception_table 数组中的每个条目描述代码数组中的一个异常处理程序。exception_table 数组中处理程序的顺序很重要（§2.10）。

每个 exception_table 条目包含以下四项:

`start_pc, end_pc`

两项 start_pc 和 end_pc 的值指示异常处理程序处于活动状态的代码数组中的范围。start_pc 的值必须是指令操作码的代码数组中的有效索引。end_pc 的值必须是指令操作码的代码数组的有效索引，或者必须等于代码数组的长度 code_length。start_pc 的值必须小于 end_pc。

start_ pc 是包含的而 end_pc 是不包含的；也就是说，当程序计数器在区间[start_pc, end_pc)内时，异常处理程序必须处于活动状态。

end_pc 是不包含的这一事实是 Java 虚拟机设计中的一个历史错误：如果一个方法的 Java 虚拟机代码正好是 65535 字节长，并且以一条 1 字节长的指令结束，那么该指令就不能受到异常处理程序的保护。编译器编写者可以通过将任何方法、实例初始化方法或静态初始化器（任何代码数组的大小）生成的 Java 虚拟机代码的最大大小限制为 65534 字节来解决此错误。

`handler_pc`

handler_pc 项的值表示异常处理程序的开始。项的值必须是代码数组的有效索引，并且必须是指令的操作码的索引。

`catch_type`

如果 catch_type 项的值非零，它必须是 constant_pool 表的有效索引。该索引上的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构(§4.4.1)，表示异常处理程序指定捕获的一异常类。只有当抛出的异常是给定类或其子类的实例时，才会调用异常处理程序。

验证者检查该类是 Throwable 或 Throwable 的子类（§4.9.2）。

如果 catch_type 项的值为零，则对所有异常调用此异常处理程序。

这是用来实现 finally (§3.13)。

`attributes_count`

attributes_count 项的值指示 Code 属性的属性数。

`attributes[]`

attributes 表的每个值都必须是 attribute_info 结构(§4.7)。

一个 Code 属性可以有任意数量的可选属性与其关联。

表 4.7-C 列出了本规范定义的 Code 属性的 attributes 表中的属性。

关于定义在 Code 属性的 attributes 表中的属性的规则见§4.7。

Code 属性的 attributes 表中关于非预定义属性的规则见§4.7.1。

4.7.4 `StackMapTable` 属性

StackMapTable 属性是 Code 属性的 attributes 表中的一个可变长度属性(§4.7.3)。在类型检查的验证过程中使用 StackMapTable 属性(§4.10.1)。

一个 Code 属性的 attributes 表中最多只能有一个 StackMapTable 属性。

在一个版本号为 50.0 或更高的 class 文件中，如果一个方法的 Code 属性没有 StackMapTable 属性，它就有一个隐式的栈映射属性(§4.10.1)。这个隐含的栈映射属性等价于 number_of_entries 为零的 StackMapTable 属性。

StackMapTable 属性的格式如下:

```
StackMapTable_attribute {
    u2              attribute_name_index;
    u4              attribute_length;
    u2              number_of_entries;
    stack_map_frame entries[number_of_entries];
}
```

StackMapTable_attribute 结构的项如下:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "StackMapTable"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`number_of_entries`

number_of_entries 项的值表示 entries 表中 stack_map_frame 项的数目。

`entries[]`

entries 表中的每个条目描述了该方法的一个栈映射帧。entries 表中栈映射帧的顺序很

重要。

栈映射帧（显式或隐式）指定其应用的字节码偏移量，以及该偏移量的局部变量和操作数栈项的验证类型。

entries 表中描述的每个栈映射帧在某些语义上依赖于前一帧。方法的第一个栈映射帧是隐式的，由类型检查器根据方法描述符计算（§4.10.1.6）。因此，entries[0]处的 stack_map_frame 结构描述了该方法的第二个栈映射帧。

栈映射帧应用的字节码偏移量是通过获取帧中指定的 offset_delta 值(显式或隐式)，并在前一帧的字节码偏移量上加上 offset_delta + 1 来计算的，除非前一帧是该方法的初始帧。在这种情况下，栈映射帧应用的字节码偏移量是帧中指定的 offset_delta 值。

> 通过使用偏移量增量而不是存储实际的字节码偏移量，根据定义，我们确保栈映射帧按正确的顺序排序。此外，通过始终如一地对所有显式帧(而不是隐式的第一帧)使用 offset_delta + 1 公式，我们保证了没有重复项。

如果指令在 Code 属性的代码数组中从偏移量 i 开始，并且 Code 属性有一个 StackMapTable 属性，该属性的 entries 数组包含一个应用于字节码偏移量 i 的栈映射帧，则字节码中的指令有一个对应的栈映射帧。

验证类型指定一个或两个位置的类型，其中位置要么是单个局部变量，要么是单个操作数栈条目。验证类型由区分并集 verification_type_info 表示，它由一个单字节标签组成，表示使用的是联合中的哪个项，后面跟着 0 个或多个字节，提供关于该标签的更多信息。

```
union verification_type_info {
    Top_variable_info;
    Integer_variable_info;
    Float_variable_info;
    Long_variable_info;
    Double_variable_info;
    Null_variable_info;
    UninitializedThis_variable_info;
    Object_variable_info;
    Uninitialized_variable_info;
}
```

在局部变量数组或操作数栈中指定一个位置的验证类型由 verification_type_info 联合中的下列项表示:

- Top_variable_info 项表示局部变量的验证类型为 top。

```
Top_variable_info {
    u1 tag = ITEM_Top; /* 0 */
}
```

- Integer_variable_info 项表示该位置的验证类型为 int。

```
Integer_variable_info {
    u1 tag = ITEM_Integer; /* 1 */ }
```

- Float_variable_info 项表示该位置的验证类型为 float。

```
Float_variable_info {
    u1 tag = ITEM_Float; /* 2 */ }
```

- Null_variable_info 类型表示该位置的验证类型为 null。

```
Null_variable_info {
    u1 tag = ITEM_Null; /* 5 */
}
```

- UninitializedThis_variable_info 项表明该位置的验证类型为 uninitializedThis。

```
UninitializedThis_variable_info {
    u1 tag = ITEM_UninitializedThis; /* 6 */
}
```

- Object_variable_info 项表示该位置具有验证类型，该验证类型是由 CONSTANT_class_info 结构（§4.4.1）表示的类，该结构位于 constant_pool 表中 cpool_index 给出的索引处。

```
Object_variable_info {
    u1 tag = ITEM_Object; /* 7 */
    u2 cpool_index;
}
```

- Uninitialized_variable_info 项表示该位置具有验证类型 uninitialized(Offset)。Offset 项表示在包含此 StackMapTable 属性的 Code 属性的 code 数组中，创建存储在该位置的对象的 new 指令（§new）的偏移量。

```
Uninitialized_variable_info {
    u1 tag = ITEM_Uninitialized; /* 8 */ u2 offset;
}
```


指定局部变量数组或操作数栈中两个位置的验证类型由以下 verification_type_info 联合项表示：

- Long_variable_info 项指示两个位置中的第一个位置具有验证类型 long。

```
Long_variable_info {
    u1 tag = ITEM_Long; /* 4 */
}
```

- Double_variable_info 项指示两个位置中的第一个具有验证类型 double。

```
Double_variable_info {
    u1 tag = ITEM_Double; /* 3 */
}
```

- Long_variable_info 和 Double_VALIABLE_ info 项表示两个位置中第二个位置的验证类型，如下所示：
  - 如果两个位置中的第一个是局部变量，则：
    > 它不能是具有最高索引的局部变量。

> 下一个编号较高的局部变量具有验证类型 top。

- 如果两个位置中的第一个是操作数栈项，则：

> 它不能是操作数栈的最高位置。

> 靠近操作数栈顶部的下一个位置具有验证类型 top。

栈映射帧由一个区分的并集 stack_map_frame 表示，该并集由一个单字节标签组成，该标记指示正在使用该并集的哪个项，后跟零个或多个字节，提供有关该标签的更多信息。

```
union stack_map_frame {
    same_frame;
    same_locals_1_stack_item_frame;
    same_locals_1_stack_item_frame_extended;
    chop_frame;
    same_frame_extended;
    append_frame;
    full_frame;
}
```

标记指示栈映射帧的帧类型：

- 帧类型 same_ frame 由范围[0-63]中的标签表示。此帧类型表示帧具有与前一帧完全相同的局部变量，并且操作数栈为空。帧的 offset_delta 值是标记项 frame_type 的值。

```
same_frame {
    u1 frame_type = SAME; /* 0-63 */
}
```

- 帧类型 same_locals_1_stack_item_frame 由范围[64, 127]中的标签表示。此帧类型表示帧具有与前一帧完全相同的局部变量，并且操作数栈具有一个条目。帧的 offset_delta 值由公式 frame_type-64 给出。一个栈条目的验证类型出现在帧类型之后。

```
same_locals_1_stack_item_frame {
    u1 frame_type = SAME_LOCALS_1_STACK_ITEM; /* 64-127 */
    verification_type_info stack[1];
}
```

- 范围[128-246]内的标签保留供将来使用。

- 帧类型 same_locals_1_stack_item_frame_extended 由标签 247 表示。此帧类型表示帧具有与前一帧完全相同的局部变量，并且操作数栈具有一个条目。与帧类型 same_locals_1_stack_item_frame 不同，帧的 offset_delta 值是显式给出的。一个栈条目的验证类型出现在 offset_delta 之后。

```
same_locals_1_stack_item_frame_extended {
    u1 frame_type = SAME_LOCALS_1_STACK_ITEM_EXTENDED; /* 247 */
    u2 offset_delta;
    verification_type_info stack[1];
}
```

- 帧类型 chop_frame 由[248-250]范围内的标签表示。这种帧类型表明该帧与前一帧具有相同的局部变量，除了最后 k 个局部变量不存在，并且操作数栈为空。k 的值由公式 251

- frame_type 给出。帧的 offset_delta 值是显式给出的。

```
chop_frame {
    u1 frame_type = CHOP; /* 248-250 */ u2 offset_delta;
}
```

假设上一帧中的局部变量的验证类型是由 locals 给出的，这是一个结构与 full_frame 帧类型相同的数组。如果前一帧中的 locals[M-1]表示局部变量 X, locals[M]表示局部变量 Y, 那么去掉一个局部变量的效果就是新一帧中的 locals[M-1]表示局部变量 X，而 locals[M]没有定义。

如果 k 大于前一个帧中 locals 中局部变量的数量，即新帧中局部变量的数量小于零，则为错误。

- 帧类型 same_frame_extended 由标签 251 表示。此帧类型指示此帧具有与前一帧完全相同的局部变量，且操作数栈为空。与帧类型 same_frame 不同，该帧的 offset_delta 值是显式给出的。

```
same_frame_extended {
    u1 frame_type = SAME_FRAME_EXTENDED; /* 251 */
    u2 offset_delta;
}
```

- append_frame 的帧类型由[252-254]范围内的标签表示。此帧类型表明该帧具有与前一帧相同的局部变量，除了定义了 k 个额外的局部变量，并且操作数栈为空。k 的值由公式 frame_type - 251 给出。帧的 offset_delta 值是显式给出的。

```
append_frame {
    u1 frame_type = APPEND; /* 252-254 */
    u2 offset_delta;
    verification_type_info locals[frame_type - 251];
}
```

locals 中的第 0 项表示第一个附加局部变量的验证类型。如果 locals[M]表示局部变量 N, 则:

  - locals[M+1] 表示局部变量 N+1 如果 locals[M] 是 Top_variable_info,Integer_variable_info,Float_variable_info, Null_variable_info,UninitializedThis_variable_info,Object_variable_info, 或 Uninitialized_variable_info 其中之一; 并且
  - locals[M+1] 表示局部变量 N+2 如果 locals[M] 是 Long_variable_info 或 Double_variable_info。

如果对于任何索引 i, locals[i]表示一个索引大于该方法最大局部变量数量的局部变量，那么将会出错。

- 帧类型 full_frame 由标签 255 表示。帧的 offset_delta 值是显式给出的。

```
full_frame {
    u1 frame_type = FULL_FRAME; /* 255 */
    u2 offset_delta;
    u2 number_of_locals;
    verification_type_info locals[number_of_locals];
    u2 number_of_stack_items;
    verification_type_info stack[number_of_stack_items];
}
```

locals 中的第 0 项表示局部变量 0 的验证类型。如果 locals[M]表示局部变量 N, 则:

- locals[M+1] 表示局部变量 N+1 如果 locals[M] 是 Top_variable_info, Integer_variable_info,Float_variable_info,Null_variable_info,UninitializedThis_variable_info,Object_variable_info,或 Uninitialized_variable_info 之一;并且
- locals[M+1] 表示局部变量 N+2 如果 locals[M] 是 Long_variable_info 或 Double_variable_info。

如果对于任何索引 i, locals[i]表示一个索引大于该方法最大局部变量数量的局部变量，那么将会出错。

栈中的第 0 个条目表示操作数栈底部的验证类型，栈中的后续条目表示靠近操作数栈顶部的栈条目的验证类型。我们将操作数栈的底部称为栈条目 0，将操作数栈的后续条目称为栈条目 1、2 等。如果 stack[M]表示栈条目 N, 则:

- stack[M+1] 表示栈条目 N+1 如果 stack[M] 是 Top_variable_info, Integer_variable_info,Float_variable_info,Null_variable_info,UninitializedThis_variable_info,Object_variable_info,或 Uninitialized_variable_info 之一;并且
- stack[M+1] 表示栈条目 N+2 如果 stack[M] 是 Long_variable_info 或 Double_variable_info。

如果对于任何索引 i, stack[i]表示一个索引大于该方法的最大操作数栈大小的栈条目，则会出错。

4.7.5 Exceptions 属性

Exceptions 属性是 method_info 结构的 attributes 表中的一个变长属性(§4.6)。Exceptions 属性指示方法可能抛出哪些已检查的异常。

在 method_info 结构的 attributes 表中最多可以有一个 Exceptions 属性。

Exceptions 属性有以下格式:

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
```

```
        u2 number_of_exceptions;
        u2 exception_index_table[number_of_exceptions];
    }
```

Exceptions_attribute 结构的项如下:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串"Exceptions"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`number_of_exceptions`

number_of_exceptions 项的值表示 exception_index_table 中条目的个数。

`exception_index_table[]`

exception_index_table 数组中的每个值都必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构体(§4.4.1)，表示该方法声明要抛出的类类型。

只有满足以下三个条件中的至少一个时，方法才应该抛出异常:

- 异常是 RuntimeException 或其子类之一的实例。
- 异常是 Error 或其子类之一的实例。
- 异常是刚刚描述的 exception_index_table 中指定的异常类之一或其子类之一的实例。

这些要求在 Java 虚拟机中不强制执行；它们仅在编译时强制执行。

4.7.6 `InnerClasses` 属性

InnerClasses 属性是 ClassFile 结构的 attributes 表中的可变长度属性（§4.1）。

如果类或接口 C 的常量池包含至少一个 CONSTANT_Class_info 条目（§4.4.1），该条目表示不是包成员的类或接口，则 C 的 ClassFile 结构的 attributes 表中必须正好有一个 InnerClasses 属性。

InnerClasses 属性具有以下格式:

```
    InnerClasses_attribute {
        u2 attribute_name_index;
        u4 attribute_length;
        u2 number_of_classes;
        {   u2 inner_class_info_index;
            u2 outer_class_info_index;
            u2 inner_name_index;
            u2 inner_class_access_flags;
        } classes[number_of_classes];
    }
```

InnerClasses_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表中的有效索引。该索引处的 constant_pool 条目必须是表示字符串"InnerClasses"的 CONSTANT_Utf8_info 结构（§4.4.7）。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的六个字节。

`number_of_classes`

number_of_classes 项的值指示 classes 数组中的条目数。

`classes[]`

constant_pool 表中表示非包成员的类或接口 C 的每个 CONSTANT_Class_info 条目在 classes 数组中必须正好有一个对应条目。

如果类或接口的成员是类或接口，则其 constant_pool 表（因此其 InnerClasses 属性）必须引用每个此类成员（JLS§13.1），即使该类未另行提及该成员。

此外，每个嵌套类和嵌套接口的 constant_pool 表必须引用其封闭类，因此，每个嵌套类和嵌套接口都将具有每个封闭类以及其自身的嵌套类和接口的 InnerClasses 信息。

classes 数组中的每个条目包含以下四项：

`inner_class_info_index`

inner_class_info_index 项的值必须是 constant_pool 表中的有效索引。该索引处的 constant_pool 条目必须是表示 C 的 CONSTANT_Class_info 结构。

`outer_class_info_index`

如果 C 不是类或接口的成员-也就是说，如果 C 是顶级类或接口（JLS§7.6）或局部类（JLS§14.3）或匿名类（JLS§15.9.5）-则 outer_class_info_index 项的值必须为零。

否则，outer_class_info_index 项的值必须是 constant_pool 表中的有效索引，并且该索引处的条目必须是表示 C 是其成员的类或接口的 CONSTANT_Class_info 结构。outer_class_info_index 项的值不能等于 inner_class_info_index 项的值。

`inner_name_index`

如果 C 是匿名的(JLS§15.9.5)，inner_name_index 项的值必须为零。

否则，inner_name_index 项的值必须是 constant_pool 表的有效索引，并且该索引的条目必须是一个 CONSTANT_Utf8_info 结构，该结构表示 C 的原始简单名称，就像编译这个 class 文件的源代码中给出的那样。

`inner_class_access_flags`

inner_class_access_flags 项的值是一个标志掩码，用于表示类或接口 C 的访问权限和属性，如编译该 class 文件的源代码中声明的。当源代码不可用时，编译器使用它来恢复原始信息。标志如表 4.7.6-A 所示。

表 4.7.6-A. 嵌套的类访问和属性标志

标志名	值	说明
ACC_PUBLIC	0x0001	在源代码中标记或隐式 public。
ACC_PRIVATE	0x0002	在源代码中标记为 private。
ACC_PROTECTED	0x0004	在源代码中标记为 protected。
ACC_STATIC	0x0008	在源代码中标记或隐式 static。
ACC_FINAL	0x0010	在源代码中标记或隐式 final。
ACC_INTERFACE	0x0200	是源代码中的 interface。
ACC_ABSTRACT	0x0400	在源代码中标记或隐式 abstract。
ACC_SYNTHETIC	0x1000	声明为 synthetic; 源代码中不存在。
ACC_ANNOTATION	0x2000	声明为注解接口。
ACC_ENUM	0x4000	声明为枚举类。

表 4.7.6-A 中未赋值的 inner_class_access_flags 项的所有位保留供将来使用。在生成的 class 文件中，它们应设置为零，Java 虚拟机实现应忽略它们。

如果 class 文件的版本号为 51.0 或更高，并且在其 attributes 表中具有 InnerClasses 属性，则对于 InnerClass 属性的 classes 数组中的所有条目，如果 inner_name_index 项的值为零，则 outer_class_info_index 项的值必须为零。

Oracle 的 Java 虚拟机实现不检查 InnerClasses 属性与表示该属性引用的类或接口的 class 文件的一致性。

4.7.7 EnclosingMethod 属性

EnclosingMethod 属性是 ClassFile 结构 attributes 表中的固定长度属性(§4.1)。一个类必须具有 EnclosingMethod 属性，当且仅当它代表一个局部类或匿名类时(JLS§14.3,JLS§15.9.5)。

ClassFile 结构的 attributes 表中最多只能有一个 EnclosingMethod 属性。

EnclosingMethod 属性有以下格式：

```
EnclosingMethod_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 class_index;
    u2 method_index;
}
```

EnclosingMethod_attribute 结构的项如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "EnclosingMethod"。

`attribute_length`

attribute_length 项的值必须为 4。

`class_index`

class_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构(§4.4.1)，表示包含当前类声明的最内层类。

`method_index`

如果当前类没有立即包含在方法或构造函数中，那么 method_index 项的值必须为零。

特别是，如果当前类立即被实例初始化器、静态初始化器、实例变量初始化器或类变量初始化器封装在源代码中，那么 method_index 必须为零。(前两个同时涉及局部类和匿名类，而后两个涉及在字段赋值的右侧声明的匿名类。)

否则，method_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_NameAndType_info 结构(§4.4.6)，表示上面 class_index 属性所引用的类中方法的名称和类型。

Java 编译器的责任是确保通过 method_index 标识的方法确实是包含该 EnclosingMethod 属性的类中最接近的词法封闭方法。

4.7.8 `Synthetic` 属性

Synthetic 属性是 ClassFile、field_info 或 method_info 结构的 attributes 表中的固定长度属性（§4.1、§4.5 和§4.6）。未出现在源代码中的类成员必须使用 Synthetic 属性进行标记，否则必须设置其 ACC_SYNTHETIC 标志。此要求的唯一例外是编译器生成的成员，这些成员不被视为实现工件，即：

- 实例初始化方法，表示 Java 编程语言的默认构造函数(§2.9.1)
- 类或接口初始化方法(§2.9.2)
- 枚举和记录类的隐式声明成员(JLS §8.9.3, JLS §8.10.3)

JDK 1.1 中引入了 Synthetic 属性来支持嵌套类和接口。

这是 class 文件格式的一个限制，只有形式参数和模块可以标记为 ACC_MANDATED(§4.7.24, §4.7.25)，以表明，尽管是编译器生成的，但它们不被认为是实现工件。没有办法标记其他编译器生成的构造，这样它们也不会被认为是实现工件(JLS§13.1)。这一限制意味着 Java SE 平台的反射 api 可能不能准确地指示此类构造的“强制”状态。

Synthetic 属性有以下格式：

```
Synthetic_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

Synthetic_attribute 项的结构如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表中的有效索引。该索引处的 constant_pool 条目必须是表示字符串"Synthetic"的 CONSTANT_Utf8_info 结构（§4.4.7）。

`attribute_length`

attribute_length 项的值必须为 0。

4.7.9 `Signature` 属性

Signature 属性是 ClassFile、field_info、method_info 或 record_component_info 结构的 attributes 表中的固定长度属性（§4.1、§4.5、§4.6、§4.7.30）。Signature 属性存储类、接口、构造函数、方法、字段或记录组件的签名（§4.7.9.1），这些组件在 Java 编程语言中的声明使用类型变量或参数化类型。有关这些构造的详细信息，请参阅 Java 语言规范 Java SE 19 版。

在 ClassFile、field_info、method_info 或 record_component_info 结构的 attributes 表中最多可以有一个 Signature 属性。

Signature 属性有以下格式：

```
Signature_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 signature_index;
}
```


Signature_attribute 结构的项如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串"Signature"。

`attribute_length`

attribute_length 项的值必须为 2。

`signature_index`

signature_index 项的值必须是 constant_pool 表的有效索引。如果 Signature 属性是 ClassFile 结构的属性，那么该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示类签名;如果 Signature 属性是 method_info 结构的属性，则为方法签名;或者字段签名。

Oracle 的 Java 虚拟机实现在类加载或链接期间不会检查 Signature 属性的格式良好性。相反，Signature 属性由 Java SE Platform 类库的方法检查，这些方法公开类、接口、构造函数、方法和字段的泛型签名。例如 Class 中的 getGenericSuperclass 和 java.lang.reflect.Executable 中的 toGenericString。

4.7.9.1 签名

签名对用 Java 编程语言编写的声明进行编码，这些声明使用 Java 虚拟机类型系统之外的类型。它们支持反射和调试，以及在只有 class 文件可用时进行编译。

Java 编译器必须为其声明使用类型变量或参数化类型的任何类、接口、构造函数、方法、字段或记录组件发出签名。具体来说，Java 编译器必须发出:

- 任何类或接口声明的类签名，它要么是泛型的，要么具有作为超类或超接口的参数化类型，或者两者兼有。
- 任何泛型的方法或构造函数声明的方法签名，或类型变量或参数化类型作为返回类型或形式参数类型，或在 throws 子句中有类型变量，或它们的任何组合。

  如果方法或构造函数声明的 throws 子句不涉及类型变量，那么编译器可以将该声明视为没有 throws 子句，以发出方法签名。

- 类型使用类型变量或参数化类型的任何字段、形式参数、局部变量或记录组件声明的字段签名。

签名使用遵循§4.3.1 符号的语法来指定。除了这些符号之外:

- 结果右边的语法[x]表示 x 出现了零次或一次。也就是说，x 是一个可选的符号。包含可选符号的选项实际上定义了两个选项:一个省略可选符号，另一个包含可选符号。
- 很长的右边可以通过明显的缩进在第二行继续。

语法包括终止符 Identifier，用于表示 Java 编译器生成的类型、字段、方法、形式参数、局部变量或类型变量的名称。这样的名称不能包含任何 ASCII 字符. ; [/ < > : (即方法名中禁止的字符(§4.2.2)和冒号)，但可能包含 Java 编程语言中不能出现的标识符(JLS§3.8)。

签名依赖于称为类型签名的非终止符的层次结构:

- Java 类型签名表示 Java 编程语言的引用类型或原生类型。

  JavaTypeSignature:
  ReferenceTypeSignature
  BaseType

  为了方便起见，这里重复§4.3.2 的以下内容:

  BaseType:
  (one of)
  B C D F I J S Z

- 引用类型签名表示 Java 编程语言的引用类型，即类或接口类型、类型变量或数组类型。

  类的类型签名表示(可能是参数化的)类或接口类型。必须对类的类型签名进行公式化，以便通过擦除任何类型参数并将每个.字符转换为$字符，将其可靠地映射到所表示的类的二进制名称。

类型变量签名表示类型变量。

数组类型签名表示数组类型的一个维度。

ReferenceTypeSignature:
ClassTypeSignature
TypeVariableSignature
ArrayTypeSignature

ClassTypeSignature:
`L` *[PackageSpecifier]*
SimpleClassTypeSignature {ClassTypeSignatureSuffix} `;`

PackageSpecifier:
Identifier `/` *{PackageSpecifier}*

SimpleClassTypeSignature:
Identifier [TypeArguments]

TypeArguments:
`<` *TypeArgument {TypeArgument}* `>`

TypeArgument:
[WildcardIndicator] ReferenceTypeSignature
`*`

WildcardIndicator:
`+`
`-`

ClassTypeSignatureSuffix:
`.` *SimpleClassTypeSignature*

TypeVariableSignature:
`T` *Identifier* `;`

ArrayTypeSignature:
`[` *JavaTypeSignature*

类签名对有关（可能是泛型的）类或接口声明的类型信息进行编码。它描述类或接口的任何类型参数，并列出其（可能参数化的）直接超类和直接超接口（如果有）。类型参数由其名称描述，后跟任何类边界和接口边界。

ClassSignature:
[TypeParameters] SuperclassSignature {SuperinterfaceSignature}

TypeParameters:

< *TypeParameter {TypeParameter}* >

TypeParameter:
Identifier ClassBound {InterfaceBound}

ClassBound:
: *[ReferenceTypeSignature]*

InterfaceBound:
: *ReferenceTypeSignature*

SuperclassSignature:
ClassTypeSignature

SuperinterfaceSignature:
ClassTypeSignature

方法签名对有关（可能是泛型）方法声明的类型信息进行编码。它描述了该方法的任何类型参数；任何形式参数的（可能参数化的）类型；（可能参数化的）返回类型（如果有）；以及方法的 throws 子句中声明的任何异常的类型。

MethodSignature:
[TypeParameters] (*{JavaTypeSignature}*) *Result {ThrowsSignature}*

Result:
JavaTypeSignature
VoidDescriptor

ThrowsSignature:
^ *ClassTypeSignature*
^ *TypeVariableSignature*

为方便起见，此处重复§4.3.3 中的以下内容:

VoidDescriptor:
V

由 Signature 属性编码的方法签名可能与 method_info 结构中的方法描述符不完全对应（§4.3.3）。特别是，无法保证方法签名中形式参数类型的数量与方法描述符中参数描述符的数量相同。大多数方法的数字是相同的，但 Java 编程语言中的某些构造函数有一个隐式声明的参数，编译器用参数描述符表示该参数，但可以从方法签名中省略。有关参数注解的类似情况，请参见§4.7.18 中的注释。

字段签名对字段、形式参数、局部变量或记录组件声明的(可能是参数化的)类型进行编码。

FieldSignature:
ReferenceTypeSignature

4.7.10 SourceFile 属性

SourceFile 属性是 ClassFile 结构 attributes 表中的一个可选的固定长度属性(§4.1)。

ClassFile 结构的 attributes 表中最多只能有一个 SourceFile 属性。

SourceFile 属性有以下格式:

```
SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

SourceFile_attribute 结构的项如下所示:

attribute_name_index

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7),表示字符串 "SourceFile"。

attribute_length

attribute_length 项的值必须是 2。

sourcefile_index

sourcefile_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个表示字符串的 CONSTANT_Utf8_info 结构。

sourcefile_index 项引用的字符串将被解释为表示编译该 class 文件的源文件的名称。它不会被解释为指示包含文件的目录的名称或文件的绝对路径名;这种特定于平台的附加信息必须在实际使用文件名时由运行时解释器或开发工具提供。

4.7.11 SourceDebugExtension 属性

SourceDebugExtension 属性是 ClassFile 结构的 attributes 表中的可选属性(§4.1)。

ClassFile 结构的 attributes 表中最多可以有一个 SourceDebugExtension 属性。

SourceDebugExtension 属性有以下格式:

```
SourceDebugExtension_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 debug_extension[attribute_length];
}
```

SourceDebugExtension_attribute 结构的项如下所示:

attribute_name_index

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7),表示字符串 "SourceDebugExtension"。

attribute_length

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

debug_extension[]

debug_extension 数组保存了对 Java 虚拟机没有语义影响的扩展调试信息。该信息使用修改后的 UTF-8 字符串(§4.4.7)表示，没有终止零字节。

注意，debug_extension 数组可能表示比可以用 String 类实例表示的字符串更长的字符串。

4.7.12 LineNumberTable 属性

LineNumberTable 属性是 Code 属性 attributes 表中的一个可选变长度属性(§4.7.3)。调试器可以使用它来确定 code 数组的哪一部分对应于原始源文件中的给定行号。

如果一个 Code 属性的 attributes 表中有多个 LineNumberTable 属性，那么它们可以以任何顺序出现。

在 Code 属性的 attributes 表中，源文件的每行可能有多个 LineNumberTable 属性。也就是说，LineNumberTable 属性可以一起表示源文件的给定行，并且不需要与源行一一对应。

LineNumberTable 属性有以下格式：

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    {   u2 start_pc;
        u2 line_number;
    } line_number_table[line_number_table_length];
}
```


LineNumberTable_attribute 结构的项如下所示：

attribute_name_index

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "LineNumberTable"。

attribute_length

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

line_number_table_length

line_number_table_length 项的值表示 line_number_table 数组的条目数。

line_number_table[]

line_number_table 数组中的每个条目表示原始源文件中的行号在 code 数组中的给定点发生更改。每个 line_number_table 条目必须包含以下两项：

start_pc

start_pc 项的值必须是这个 Code 属性的 code 数组的有效索引。该项指示 code 数

组中的索引，原始源文件中新行的代码从该索引开始。

`line_number`

line_number 项的值给出原始源文件中相应的行号。

4.7.13 `LocalVariableTable` 属性

LocalVariableTable 属性是 Code 属性的 attributes 表中的一个可选变长属性(§4.7.3)。调试器可以在方法执行期间使用它来确定给定局部变量的值。

如果一个 Code 属性的 attributes 表中有多个 LocalVariableTable 属性，那么它们可以以任何顺序出现。

在一个 Code 属性的 attributes 表中，每个局部变量的 LocalVariableTable 属性不能超过一个。

LocalVariableTable 属性有以下格式:

```
LocalVariableTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_table_length;
    {   u2 start_pc;
        u2 length;
        u2 name_index;
        u2 descriptor_index;
        u2 index;
    } local_variable_table[local_variable_table_length];
}
```

LocalVariableTable_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "LocalVariableTable"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`local_variable_table_length`

local_variable_table_length 项的值表示 local_variable_table 数组的条目数。

`local_variable_table[]`

local_variable_table 数组中的每一项都表示局部变量在其中具有值的 code 数组偏移范围，并表示可以在当前帧的局部变量数组中找到该局部变量的索引。每一个条目必须包含以下五项:

`start_pc, length`

start_pc 项的值必须是 Code 属性的 code 数组的有效索引，并且必须是指令的操作

码的索引。

start_pc + length 的值必须是 Code 属性的 code 数组的有效索引，并且是指令操作码的索引，或者它必须是该 code 数组末尾以外的第一个索引。

start_pc 和 length 项表示给定的局部变量在区间[start_pc, start_pc + length)中 code 数组的下标处有一个值，即在包含 start_pc 和不包含 start_pc + length 之间。

`name_index`

name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须包含一个 CONSTANT_Utf8_info 结构体，表示一个有效的非限定名，这个名字表示一个局部变量(§4.2.2)。

`descriptor_index`

descriptor_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须包含一个 CONSTANT_Utf8_info 结构体，该结构体表示一个字段描述符，用于编码源程序中局部变量的类型(§4.3.2)。

`index`

index 项的值必须是当前帧的局部变量数组的有效索引。给定的局部变量位于当前帧的局部变量数组的 index 位置。

如果给定的局部变量是 double 或 long 类型，则它同时占用 index 和 index + 1。

4.7.14 `LocalVariableTypeTable` 属性

LocalVariableTypeTable 属性是 Code 属性的 attributes 表中的一个可选变长属性(§4.7.3)。调试器可以在方法执行期间使用它来确定给定局部变量的值。

如果给定 Code 属性的 attributes 表中有多个 LocalVariableTypeTable 属性，那么它们可以以任何顺序出现。

在一个 Code 属性的 attributes 表中，每个局部变量的 LocalVariableTypeTable 属性不能超过一个。

LocalVariableTypeTable 属性与 LocalVariableTable 属性(§4.7.13)的区别在于，它提供的是签名信息而不是描述符信息。这种差异只对类型使用类型变量或参数化类型的变量有意义。这样的变量将出现在两个表中，而其他类型的变量将只出现在 LocalVariableTable 中。

LocalVariableTypeTable 属性有以下格式:

```
LocalVariableTypeTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_type_table_length;
    {   u2 start_pc;
        u2 length;
        u2 name_index;
        u2 signature_index;
        u2 index;
    } local_variable_type_table[local_variable_type_table_length];
```

```
}
```

LocalVariableTypeTable_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "LocalVariableTypeTable"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`local_variable_type_table_length`

local_variable_type_table_length 项的值表示 local_variable_type_table 数组的条目数。

`local_variable_type_table[]`

local_variable_type_table 数组中的每一项都表示局部变量在其中具有值的 code 数组偏移范围，并表示可以在当前帧的局部变量数组中找到该局部变量的索引。每个条目必须包含以下五项:

`start_pc, length`

start_pc 项的值必须是 Code 属性的 code 数组的有效索引，并且必须是指令的操作码的索引。

start_pc + length 的值必须是 Code 属性的 code 数组的有效索引，并且是指令操作码的索引，或者它必须是该 code 数组末尾以外的第一个索引。

start_pc 和 length 项表示给定的局部变量在区间[start_pc, start_pc + length)中 code 数组的下标处有一个值，即在包含 start_pc 和不包含 start_pc + length 之间。

`name_index`

name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须包含一个 CONSTANT_Utf8_info 结构体，表示一个有效的非限定名，该名称表示一个局部变量(§4.2.2)。

`signature_index`

signature_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须包含一个 CONSTANT_Utf8_info 结构体，该结构体表示一个字段签名，用于编码源程序中某个局部变量的类型(§4.7.9.1)。

`index`

index 项的值必须是当前帧局部变量数组的有效索引。给定的局部变量位于当前帧的局部变量数组的 index 位置。

如果给定的局部变量是 double 或 long 类型，则它同时占用 index 和 index + 1。

4.7.15 Deprecated 属性

Deprecated 属性是 ClassFile, field_info，或 method_info 结构(§4.1，§4.5，§4.6)的 attributes 表中一个可选的固定长度属性。可以使用 Deprecated 属性来标记类、接口、方法或字段，以表明类、接口、方法或字段已被取代。

读取 class 文件格式的运行时解释器或工具(如编译器)可以使用此标记来通知用户正在引用已被替代的类、接口、方法或字段。Deprecated 属性的存在不会改变类或接口的语义。

Deprecated 属性有以下格式:

```
Deprecated_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

Deprecated_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "Deprecated"。

`attribute_length`

attribute_length 项的值必须为零。

4.7.16 RuntimeVisibleAnnotations 属性

RuntimeVisibleAnnotations 属性是 ClassFile、field_info、method_info 或 record_component_info 结构的 attributes 表中的一个变长属性(§4.1、§4.5、§4.6、§4.7.30)。RuntimeVisibleAnnotations 属性将运行时可见注释存储在相应的类、字段、方法或记录组件的声明上。

ClassFile、field_info、method_info 或 record_component_info 结构的 attributes 表中最多可以有一个 RuntimeVisibleAnnotations 属性。

RuntimeVisibleAnnotations 属性有以下格式:

```
RuntimeVisibleAnnotations_attribute {
    u2         attribute_name_index;
    u4         attribute_length;
    u2         num_annotations;
    annotation annotations[num_annotations];
}
```

RuntimeVisibleAnnotations_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串

"RuntimeVisibleAnnotations"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`num_annotations`

num_annotations 项的值给出了由该结构表示的运行时可见注解的数量。

`annotations[]`

annotations 表中的每个条目表示声明上的单个运行时可见注解。annotation 结构有以下格式：

```
annotation {
    u2 type_index;
    u2 num_element_value_pairs;
    {   u2            element_name_index;
        element_value value;
    } element_value_pairs[num_element_value_pairs];
}
```

annotation 结构的项如下所示：

`type_index`

type_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字段描述符(§4.3.2)。字段描述符表示这个 annotation 结构所表示的注解的类型。

`num_element_value_pairs`

num_element_value_pairs 项的值给出了这个 annotation 结构所表示的注解的元素-值对的数量。

`element_value_pairs[]`

element_value_pairs 表的每个值表示这个 annotation 结构所表示的注解中的单个元素-值对。每个 element_value_pairs 条目包含以下两项：

`element_name_index`

element_name_index 项的值必须是到 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体(§4.4.7)。constant_pool 条目表示这个 element_value_pairs 条目所表示的元素-值对的元素名称。

换句话说，条目表示 type_index 指定的注解接口的一个元素。

`value`

value 项的值表示这个 element_value_pairs 条目所表示的元素-值对的值。

4.7.16.1 `element_value` 结构

element_value 结构是一个区分并集，表示元素-值对的值。它有以下格式：

```
element_value {
    u1 tag;
    union {
        u2 const_value_index;
        {   u2 type_name_index;
            u2 const_name_index;
        } enum_const_value;

        u2 class_info_index;

        annotation annotation_value;
        {   u2            num_values;
            element_value values[num_values];
        } array_value;
    } value;
}
```

tag 项使用一个 ASCII 字符来表示元素-值对的值的类型。这将确定使用 value 联合的哪个项。表 4.7.16.1-A 显示了 tag 项的有效字符，每个字符表示的类型，以及每个字符的 value 联合中使用的项。表的第四列用于下面对 value 联合的一项的描述。

表 4.7.16.1-A. 将 tag 值解释为类型

tag 项	类型	value 项	常量类型
B	byte	const_value_index	CONSTANT_Integer
C	char	const_value_index	CONSTANT_Integer
D	double	const_value_index	CONSTANT_Double
F	float	const_value_index	CONSTANT_Float
I	int	const_value_index	CONSTANT_Integer
J	long	const_value_index	CONSTANT_Long
S	short	const_value_index	CONSTANT_Integer
Z	boolean	const_value_index	CONSTANT_Integer
s	String	const_value_index	CONSTANT_Utf8
e	枚举类	enum_const_value	不适用
c	Class	class_info_index	不适用
@	注解接口	annotation_value	不适用
[	数组类型	array_value	不适用

value 项表示元素-值对的值。该项是一个并集，它自己的项如下:

`const_value_index`

const_value_index 项表示元素值对的值是一个原生类型或 String 类型的常量。

const_value_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是与 tag 项相适应的类型，如表 4.7.16.1-A 的第四列所指定。

enum_const_value

enum_const_value 项表示一个枚举常量作为该元素-值对的值。

enum_const_value 项由以下两个项组成:

type_name_index

type_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字段描述符(§4.3.2)。constant_pool 条目给出了由 element_value 结构(§4.2.1)表示的枚举常量类型的二进制名称的内部形式。

const_name_index

const_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体(§4.4.7)。constant_pool 条目给出了由 element_value 结构表示的枚举常量的简单名称。

class_info_index

class_info_index 项表示一个类字面量作为这个元素值对的值。

class_info_index 项必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，代表一个返回描述符(§4.3.3)。返回描述符给出了与 element_value 结构所表示的类字面量对应的类型。类型对应于类字面量如下:

- 对于类字面量 C.class，其中 C 是类、接口或数组类型的名称，相应的类型是 C。constant_pool 中的返回描述符将是 ObjectType 或 ArrayType。
- 对于类字面量 p.class，其中 p 是原生类型的名称，对应的类型是 p。constant_pool 中的返回描述符将是一个 BaseType 字符。
- 对于类字面量 void.class, 对应的类型是 void.。constant_pool 中的返回描述符将是 V。

例如, 类字面量 Object.class 对应于类型 Object, 因此 constant_pool 条目是 Ljava/lang/Object;, 而类字面量 int.class 对应于类型 int, 因此 constant_pool 条目是 I。

类字面量 void.class 对应于 void, 因此 constant_pool 条目是 V, 而类字面量 Void.class 对应于类型 Void, 因此 constant_pool 条目是 Ljava/lang/Void;.

annotation_value

annotation_value 项表示一个“嵌套”注解作为这个元素-值对的值。

annotation_value 项的值是一个 annotation 结构(§4.7.16)，它给出了由 element_value 结构表示的注解。

array_value

array_value 项表示一个数组作为这个元素-值对的值。

array_value 项由以下两项组成:

num_values

num_values 项的值给出了 element_value 结构所表示的数组中的元素数量。

`values[]`

values 表中的每个值都给出了这个 element_value 结构所表示的对应数组元素。

4.7.17 `RuntimeInvisibleAnnotations` 属性

RuntimeInvisibleAnnotations 属性是 ClassFile、field_info、method_info 或 record_component_info 结构(§4.1、§4.5、§4.6、§4.7.30) attributes 表中的一个变长属性。RuntimeInvisibleAnnotations 属性将运行时不可见的注解存储在相应的类、方法、字段或记录组件的声明上。

ClassFile、field_info、method_info 或 record_component_info 结构的 attributes 表中最多可以有一个 RuntimeInvisibleAnnotations 属性。

RuntimeInvisibleAnnotations 属性类似于 RuntimeVisibleAnnotations 属性(§4.7.16)，不同的是，由 RuntimeInvisibleAnnotations 属性表示的注解不能被反射 api 返回，除非 Java 虚拟机已经被指示通过一些特定于实现的机制(如命令行标志)保留这些注解。如果没有这样的指令，Java 虚拟机就会忽略此属性。

RuntimeInvisibleAnnotations 属性有以下格式:

```
RuntimeInvisibleAnnotations_attribute {
    u2         attribute_name_index;
    u4         attribute_length;
    u2         num_annotations;
    annotation annotations[num_annotations];
}
```

RuntimeInvisibleAnnotations_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "RuntimeInvisibleAnnotations"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`num_annotations`

num_annotations 项的值给出了由该结构表示的运行时不可见注解的数量。

`annotations[]`

annotations 表中的每个条目表示声明上的单个运行时不可见注解。§4.7.16 规定了 annotation 结构。

4.7.18 `RuntimeVisibleParameterAnnotations` 属性

RuntimeVisibleParameterAnnotations 属性是 method_info 结构的 attributes 表中的一个变长属性(§4.6)。RuntimeVisibleParameterAnnotations 属性将运行时可见注解存储在相应方法的形参声明上。

在 method_info 结构的 attributes 表中最多只能有一个 RuntimeVisibleParameterAnnotations 属性。

RuntimeVisibleParameterAnnotations 属性有以下格式:

```
RuntimeVisibleParameterAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 num_parameters;
    {   u2         num_annotations;
        annotation annotations[num_annotations];
    } parameter_annotations[num_parameters];
}
```

RuntimeVisibleParameterAnnotations_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "RuntimeVisibleParameterAnnotations"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`num_parameters`

num_parameters 项的值给出了由该结构表示的运行时可见参数注解的数量。

不能保证这个数字与方法描述符中的参数描述符的数量相同。

`parameter_annotations[]`

parameter_annotations 表中的每一个条目都表示对单个形参声明的所有运行时可见注解。每个 parameter_annotations 条目包含以下两项:

`num_annotations`

num_annotations 项的值表示与 parameter_annotations 项对应的形式参数声明上的运行时可见注解的数量。

`annotations[]`

annotations 表中的每个条目都表示与 parameter_annotations 条目对应的形式参数声明上的单个运行时可见注解。

parameter_annotations 表中的第 i 个条目可以(但不是必须)对应方法描述符中的第 i 个参数描述符(§4.3.3)。

例如，编译器可以选择在表中创建只对应于那些表示源代码中显式声明参数的参数描述符的条目。在 Java 编程语言中，内部类的构造函数被指定在显式声明的参数之前有一个隐式声明的参数(JLS§8.8.1)，因此 class 文件中对应的<init>方法在任何表示显式声明参数的参数描述符之前有一个表示隐式声明的参数的参数描述符。如果在源代码中对第一个显式声明的参数进行了注解，那么编译器可以创建 parameter_annotations[0]来存储与第二个参数描述符对应的注解。

4.7.19 RuntimeInvisibleParameterAnnotations 属性

RuntimeInvisibleParameterAnnotations 属性是 method_info 结构的 attributes 表中的一个变长属性(§4.6)。RuntimeInvisibleParameterAnnotations 属性将运行时不可见的注解存储在相应方法的形参声明上。

在 method_info 结构的 attributes 表中最多只能有一个 RuntimeInvisibleParameterAnnotations 属性。

RuntimeInvisibleParameterAnnotations 属性类似于 RuntimeVisibleParameterAnnotations 属性(§4.7.18)，不同的是，RuntimeInvisibleParameterAnnotations 属性所代表的注解不能被反射 api 返回，除非 Java 虚拟机已经被特别指示通过一些实现特定的机制(如命令行标志)保留这些注解。如果没有这样的指令，Java 虚拟机就会忽略此属性。

RuntimeInvisibleParameterAnnotations 属性有以下格式:

```
RuntimeInvisibleParameterAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 num_parameters;
    {   u2         num_annotations;
        annotation annotations[num_annotations];
    } parameter_annotations[num_parameters];
}
```

RuntimeInvisibleParameterAnnotations_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "RuntimeInvisibleParameterAnnotations"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`num_parameters`

num_parameters 项的值给出了由该结构表示的运行时不可见参数注解的数量。

不能保证这个数字与方法描述符中的参数描述符的数量相同。

`parameter_annotations[]`

parameter_annotations 表中的每个条目都表示对单个形式参数声明的所有运行时不可见注解。每个 parameter_annotations 条目包含以下两项:

`num_annotations`

num_annotations 项的值表示与 parameter_annotations 项对应的形式参数声明上的运行时不可见注解的数量。

`annotations[]`

annotations 表中的每个条目都表示与 parameter_annotations 条目对应的形式参数声明上的单个运行时不可见注解。§4.7.16 规定了 annotation 结构。

parameter_annotations 表中的第 i 个条目可以(但不是必须)对应方法描述符中的第 i 个参数描述符(§4.3.3)。

关于 parameter_annotations[0]不与方法描述符中的第一个参数描述符对应的例子，请参阅§4.7.18。

4.7.20 RuntimeVisibleTypeAnnotations 属性

RuntimeVisibleTypeAnnotations 属性是 ClassFile 、 field_info 、 method_info 或 record_component_info 结构的 attributes 表中的一个变长属性，或者 Code 属性(§4.1、§4.5、§4.6、§4.7.30、§4.7.3)。RuntimeVisibleTypeAnnotations 属性将运行时可见注解存储在相应类、字段、方法或记录组件的声明中使用的类型上，或者存储在相应方法体中的表达式中。RuntimeVisibleTypeAnnotations 属性还将运行时可见注解存储在泛型类、接口、方法和构造函数的类型参数声明上。

在 ClassFile、field_info、method_info 或 record_component_info 结构或 Code 属性的 attributes 表中最多可以有一个 RuntimeVisibleTypeAnnotations 属性。

只有当类型在与 attributes 表的父结构或属性对应的各种声明或表达式中被注解时，attributes 表才包含 RuntimeVisibleTypeAnnotations 属性。

例如，类声明的 implements 子句中关于类型的所有注解都记录在类的 ClassFile 结构的 RuntimeVisibleTypeAnnotations 属性中。同时，字段声明中关于类型的所有注解都记录在字段的 field_info 结构的 RuntimeVisibleTypeAnnotations 属性中。

RuntimeVisibleTypeAnnotations 属性有以下格式：

```
RuntimeVisibleTypeAnnotations_attribute {
    u2              attribute_name_index;
    u4              attribute_length;
    u2              num_annotations;
    type_annotation annotations[num_annotations];
}
```

RuntimeVisibleTypeAnnotations_attribute 结构的项如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体，表示字符串 "RuntimeVisibleTypeAnnotations"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`num_annotations`

num_annotations 项的值给出了由该结构表示的运行时可见类型注解的数量。

`annotations[]`

annotations 表中的每个条目表示声明或表达式中使用的类型上的单个运行时可见注解。type_annotation 结构有如下格式：

```
type_annotation {
  u1 target_type;
  union {
        type_parameter_target;
        supertype_target;
        type_parameter_bound_target;
        empty_target;
        formal_parameter_target;
        throws_target;
        localvar_target;
        catch_target;
        offset_target;
        type_argument_target;
  } target_info;
  type_path target_path;
  u2  type_index;
  u2  num_element_value_pairs;
  {  u2  element_name_index;
     element_value value;
  } element_value_pairs[num_element_value_pairs];
}
```

前三项——target_type、target_info 和 target_path——指定注解的类型的精确位置。最后三项—type_index、num_element_value_pairs 和 element_value_pairs[]—指定注解自己的类型和元素-值对。

type_annotation 结构的项如下所示：

`target_type`

target_type 项的值表示注解出现的目标类型。各种类型的目标对应于 Java 编程语言的类型上下文，其中类型用于声明和表达式(JLS§4.11)。

target_type 的合法值在表 4.7.20-A 和表 4.7.20-B 中指定。每个值都是一个单字节标记，指示 target_info 联合项的哪个项跟在 target_type 项之后，以提供关于目标的更多信息。

表 4.7.20-A 和表 4.7.20-B 中的目标类型对应于 JLS§4.11 中的类型上下文。也就是说，target_type 值 0x10-0x17 和 0x40-0x42 对应类型上下文 1-11，而 target_type 值 0x43-0x4B 对应类型上下文 12-17。

target_type 项的值决定 type_annotation 结构是否出现在 ClassFile 结构、field_info 结构、method_info 结构或 Code 属性中的 RuntimeVisibleTypeAnnotations 属性中。表 4.7.20-C 给出了 type_annotation 结构的 RuntimeVisibleTypeAnnotations 属性和每个合法的 target_type 值的位置。

`target_info`

target_info 项的值精确地表示声明或表达式中的注解类型。

target_info 联合的项在§4.7.20.1 中指定。

`target_path`

target_path 项的值精确地表示 target_info 所指示的类型的哪一部分被注解了。

type_path 结构的格式见§4.7.20.2。

```
type_index, num_element_value_pairs, element_value_pairs[]
```

这些项在 type_annotation 结构中的含义与它们在 annotation 结构中的含义相同(§4.7.16)。

表 4.7.20-A. target_type 值的解释(第 1 部分)

值	目标类型	target_info 项
0x00	泛型类或接口的类型参数声明	type_parameter_target
0x01	泛型方法或构造函数的类型参数声明	type_parameter_target
0x10	类型在类声明的 extends 或 implements 子句中(包括匿名类声明的直接超类或直接超接口)，或在接口声明的 extends 子句中	supertype_target
0x11	泛型类或接口的类型参数声明边界中的类型	type_parameter_bound_target
0x12	泛型类或接口的类型参数声明边界中的类型	type_parameter_bound_target
0x13	字段或记录组件声明中的类型	empty_target
0x14	方法的返回类型，或新构造对象的类型	empty_target
0x15	方法或构造函数的接收器类型	empty_target
0x16	在方法、构造函数或 lambda 表达式的形参声明中的类型	formal_parameter_target
0x17	方法或构造函数中的 throws 子句中的类型	throws_target

表 4.7.20-B. target_type 值的解释(第 2 部分)

值	目标类型	target_info 项
0x40	局部变量声明中的类型	localvar_target
0x41	资源变量声明中的类型	localvar_target
0x42	异常参数声明中的类型	catch_target
0x43	instanceof 表达式中的类型	offset_target
0x44	new 表达式中的类型	offset_target
0x45	使用::new 的方法引用表达式中的类型	offset_target
0x46	使用:: Identifier 的方法引用表达式中的类型	offset_target
0x47	强制转换表达式中的类型	type_argument_target
0x48	new 表达式或显式构造函数调用语句中泛型构造函数的类型参数	type_argument_target
0x49	方法调用表达式中泛型方法的类型参数	type_argument_target
0x4A	使用::new 的方法引用表达式中泛型构造函数的类型参数	type_argument_target
0x4B	使用::Identifier 的方法引用表达式中泛型方法的类型参数	type_argument_target

表 4.7.20-C. target_type 值的封闭属性的位置

值	目标类型	位置
0x00	泛型类或接口的类型参数声明	ClassFile
0x01	泛型方法或构造函数的类型参数声明	method_info
0x10	在类或接口声明的 extends 子句中的类型，或在接口声明的 implements 子句中的类型	ClassFile

0x11	泛型类或接口的类型参数声明边界中的类型	ClassFile
0x12	在泛型方法或构造函数的类型参数声明边界中的类型	method_info
0x13	字段或记录组件声明中的类型	field_info, record_component_info
0x14	方法或构造函数的返回类型	method_info
0x15	方法或构造函数的接收器类型	method_info
0x16	在方法、构造函数或 lambda 表达式的形参声明中的类型	method_info
0x17	方法或构造函数的 throws 子句中的类型	method_info
0x40-0x4B	局部变量声明、资源变量声明、异常参数声明、表达式中的类型	Code

4.7.20.1 target_info 联合

target_info 联合的项(第一个除外)精确地指定了声明或表达式中的注解类型。第一项不是指定哪种类型，而是指定注解类型参数的哪个声明。项目如下:

- type_parameter_target 项指示注解出现在泛型类、泛型接口、泛型方法或泛型构造函数的第 i 个类型参数的声明上。

```
type_parameter_target {
  u1 type_parameter_index;
}
```

  type_parameter_index 项的值指定要注解的类型参数声明。type_parameter_index 值为 0 指定第一个类型参数声明。

- supertype_target 项指示注解出现在类或接口声明的 extends 或 implements 子句中的类型上。

```
supertype_target {
  u2 supertype_index;
}
```

  supertype_index 值 65535 指定注解出现在类声明的 extends 子句中的超类上。

  任何其他 supertype_index 值都是封闭 ClassFile 结构的 interfaces 数组的索引，并指定注解出现在类声明的 implements 子句或接口声明的 extends 子句中的超接口上。

- type_parameter_bound_target 项表示注解出现在泛型类、接口、方法或构造函数的第 j 个类型参数声明的第 i 个边界上。

```
type_parameter_bound_target {
  u1 type_parameter_index;
  u1 bound_index;
}
```

  type_parameter_index 项的值指定哪个类型参数声明具有带注解的边界。type_parameter_index 值 0 指定第一个类型参数声明。

bind_index 项的值指定由 type_parameter_index 指示的类型参数声明的哪个边界被注解。bind_index 值 0 指定类型参数声明的第一个边界。

type_parameter_bound_target 项记录边界已被注解，但不记录构成边界的类型。可以通过检查存储在适当 Signature 属性中的类签名或方法签名来找到类型。

- empty_target 项表示注解出现在字段声明中的类型、记录组件声明中的类型、方法的返回类型、新构造对象的类型或方法或构造函数的接收器类型上。

```
empty_target {
}
```

每个位置中只显示一个类型，因此在 target_info 联合中没有表示每个类型的信息。

- formal_parameter_target 项表示注解出现在方法、构造函数或 lambda 表达式的形式参数声明中的类型上。

```
formal_parameter_target {
    u1 formal_parameter_index;
}
```

formal_parameter_index 项的值指定哪个形式参数声明具有注解类型。i 的 formal_parameter_index 值可以但不需要对应于方法描述符中的第 i 个参数描述符（§4.3.3）。

formal_parameter_target 项记录形式参数的类型被注解，但不记录类型本身。虽然 formal_parameter_index 值 0 并不总是指示方法描述符中的第一个参数描述符，但是可以通过检查方法描述符来找到类型；有关涉及 parameter_annotations 表的类似情况，请参见§4.7.18 中的注释。

- throws_target 项表示注解出现在方法或构造函数声明的 throws 子句中的第 i 个类型上。

```
throws_target {
    u2 throws_type_index;
}
```

throws_type_index 项的值是包含 RuntimeVisibleTypeAnnotations 属性的 method_info 结构的 Exceptions 属性的 exception_index_table 数组的索引。

- localvar_target 项表示注解出现在局部变量声明中的类型上，包括在 try-with-resources 语句中声明为资源的变量。

```
localvar_target {
  u2 table_length; {
    u2 start_pc;
    u2 length;
    u2 index;
  } table[table_length];
}
```

table_length 项的值给出了 table 数组中的条目数。每个条目指示一个 code 数组偏移范围，在该范围内局部变量具有一个值。它还指示当前帧的局部变量数组的索引，在该索引处可以找到该局部变量。每个条目包含以下三项:

`start_pc, length`

给定局部变量在区间[start_pc，start_pc+length]中的 code 数组索引处具有一个值，即在 start_pc(包含)和 start_pc+length(不包含)之间。

`index`

给定的局部变量必须位于当前帧的局部变量数组的 index 处。

如果 index 处的局部变量的类型为 double 或 long，则它同时占用 index 和 index+1。

需要一个表来完全指定其类型被注解的局部变量，因为单个局部变量可以在多个活动范围内用不同的局部变量索引表示。每个表项中的 start_pc、length 和 index 项指定与 LocalVariableTable 属性相同的信息。

localvar_target 项记录局部变量的类型被注解，但不记录类型本身。可以通过检查适当的 LocalVariableTable 属性来找到类型。

- catch_target 项表示注解出现在异常参数声明中的第 i 个类型上。

```
catch_target {
    u2 exception_table_index;
}
```

exception_table_index 项的值是包含 RuntimeVisibleTypeAnnotations 属性的 Code 属性的 exception-table 数组的索引。

异常参数声明中存在多个类型的可能性源于 try 语句的多 catch 子句，其中异常参数的类型是类型的联合（JLS§14.20）。编译器通常为联合中的每个类型创建一个 exception_table 条目，这允许 catch_target 项区分它们。这保留了类型与其注解之间的对应关系。

- offset_target 项表示注解出现在 instanceof 表达式或 new 表达式中的类型上，或方法引用表达式中的::之前的类型上。

```
offset_target {
  u2 offset;
}
```

offset 项的值指定对应于 instanceof 表达式的字节码指令、对应于 new 表达式的 new 字节码指令或对应于方法引用表达式的字节代码指令的 code 数组偏移。

- type_argument_target 项表示注解出现在强制转换表达式中的第 i 个类型上，或者出现在显式类型参数列表中的第 i 个类型参数上，用于以下任一项：new 表达式、显式构造函数调用语句、方法调用表达式或方法引用表达式。

```
type_argument_target {
    u2 offset;
    u1 type_argument_index;
}
```

offset 项的值指定与强制转换表达式相对应的字节码指令、与 new 表达式相对应的 new 字节码指令、与显式构造函数调用语句相对应的字节码指令、与方法调用表达式相对应的字节码指令或与方法引用表达式相对应的字节码指令的 code 数组偏移量。

对于强制转换表达式，type_argument_index 项的值指定了强制转换操作符中注解的类型。type_argument_index 值为 0 指定强制转换操作符中的第一个(或唯一的)类型。

强制转换表达式中可能有多个类型，因为强制转换为交集类型。

对于显式类型参数列表，type_argument_index 项的值指定要注解的类型参数。type_argument_index 值为 0 指定第一个类型参数。

4.7.20.2 type_path 结构

在声明或表达式中使用类型的地方，type_path 结构标识该类型的哪一部分被注解。注解可能出现在类型本身上，但如果类型是引用类型，则可能在其他位置出现注解:

- 如果在声明或表达式中使用了数组类型 T[]，则注解可以出现在该数组类型的任何组件类型上，包括元素类型。
- 如果是嵌套类型 T1.T2 用于声明或表达式，则注解可以出现在最里面的成员类型和任何允许类型注解的封闭类型的名称上(JLS§9.7.4)。
- 如果参数化类型 T<A>或 T<? extends A>或 T<? super A>用在声明或表达式中，则注解可以出现在任何类型参数或任何通配符类型参数的边界上。

例如，考虑 String[][]中注解的不同部分:

```
@Foo String[][]    //  Annotates the class  type String
String @Foo [][]   //  Annotates the array  type String[][]
String[] @Foo []   //  Annotates the array  type String[]
```

或者嵌套类型 Outer.Middle.Inner 的不同部分注解在:

```
@Foo Outer.Middle.Inner
Outer.@Foo Middle.Inner
Outer.Middle.@Foo Inner
```

或者参数化类型 Map<String,Object>和 List<...>的不同部分注解在:

```
@Foo Map<String,Object>
Map<@Foo String,Object>
Map<String,@Foo Object>

List<@Foo ? extends String>
List<? extends @Foo String>
```

type_path 结构有以下格式:

```
type_path {
    u1 path_length;
    {   u1 type_path_kind;
        u1 type_argument_index;
    } path[path_length];
}
```

path_length 项的值给出 path 数组条目的个数:

- 如果 path_length 的值为 0，并且被注解的类型是嵌套类型，那么该注解将应用于允许类

型注解的类型的最外层。

- 如果 path_length 的值为 0，并且被注解的类型不是嵌套类型，那么注解将直接出现在类型本身上。
- 如果 path_length 的值是非零的，那么 path 数组中的每个条目都表示从左到右的迭代步骤，指向数组类型、嵌套类型或参数化类型中注解的精确位置。(在数组类型中，迭代访问数组类型本身，然后是它的组件类型，然后是该组件类型的组件类型，以此类推，直到到达元素类型。) 每一个条目包含以下两项:

`type_path_kind`

type_path_kind 项的合法值列在表 4.7.20.2-A 中。

表 4.7.20.2-A. type_path_kind 值的解释

值	解释
0	注解在数组类型中更深入
1	注解在嵌套类型中更深入
2	注解位于参数化类型的通配符类型参数的边界上
3	注解是在参数化类型的类型参数上的

`type_argument_index`

如果 type_path_kind 项的值为 0、1 或 2，那么 type_argument_index 项的值为 0。

如果 type_path_kind 项的值是 3，那么 type_argument_index 项的值指定注解参数化类型的哪个类型参数，其中 0 表示参数化类型的第一个类型参数。

表 4.7.20.2-B.对于 `@A Map<@B ? extends @C String, @D List<@E Object>>`的 `type_path` 结构

注解	path_length	path
@A	0	[]
@B	1	[{type_path_kind: 3; type_argument_index: 0}]
@C	2	[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 2; type_argument_index: 0}]
@D	1	[{type_path_kind: 3; type_argument_index: 1}]
@E	2	[{type_path_kind: 3; type_argument_index: 1}, {type_path_kind: 3; type_argument_index: 0}]

表 4.7.20.2-C.对于 `@I String @F [] @G [] @H []`的 type_path 结构

注解	path_length	path
@F	0	[]
@G	1	[{type_path_kind: 0; type_argument_index: 0}]
@H	2	[{type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]
@I	3	[{type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]

表 4.7.20.2-D.对于`@A List<@B Comparable<@F Object @C [] @D [] @E []>>`的 type_path 结构

注解	path_length	path
@A	0	[]
@B	1	[{type_path_kind: 3; type_argument_index: 0}]
@C	2	[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}]
@D	3	[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]
@E	4	[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]
@F	5	[{type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]

表 4.7.20.2-E. `@A Outer . @B Middle . @C Inner` 的 `type_path` **结构**

假设:	`class Outer {`
	`  class Middle {`
	`  class Inner {}`
	`  }`
	`}`	

注解	path_length	path
@A	0	[]
@B	1	[{type_path_kind: 1; type_argument_index: 0}]
@C	2	[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 1; type_argument_index: 0}]

表 4.7.20.2-F. `type_path` **structures for** `Outer . @A MiddleStatic . @B Inner`

假设:	`class Outer {`
	`  static class MiddleStatic {`
	`  class Inner {}`
	`  }`
	`}`	

注解	path_length	path
@A	0	[]
@B	1	[{type_path_kind: 1; type_argument_index: 0}]
在类型 Outer . MiddleStatic . Inner 中，简单名称 Outer 上的类型注解是不允许的，因为它右边的类型名称 MiddleStatic 不引用 Outer 的内部类。		

表 4.7.20.2-G. `Outer . MiddleStatic . @A InnerStatic` 的 `type_path` 结构

假设:	`class Outer {`
	`  static class MiddleStatic {`
	`  static class InnerStatic {}`
	`  }`
	`}`	

注解	path_length	path
@A	0	[]
在类型 Outer . MiddleStatic . InnerStatic 中，简单名称 Outer 上的类型注解是不允许的，因为它右边的类型名称 MiddleStatic 不引用 Outer 的内部类。类似地，简单名称 MiddleStatic 上的类型注解也是不允许的，因为它右边的类型名称 InnerStatic 不引用 MiddleStatic 的内部类。		

表 4.7.20.2-H. `Outer . Middle<@A Foo . @B Bar> . Inner<@D String @C []>` 的 `type_path` 结构

假设:	class Outer {
	  class Middle<T> {
	  class Inner<U> {}
	  }
	}	

注解	path_length	path
@A	2	[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}]
@B	3	[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 1; type_argument_index: 0}]
@C	3	[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}]
@D	4	[{type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 1; type_argument_index: 0}, {type_path_kind: 3; type_argument_index: 0}, {type_path_kind: 0; type_argument_index: 0}]

4.7.21 RuntimeInvisibleTypeAnnotations 属性

RuntimeInvisibleTypeAnnotations 属性是 ClassFile、field_info、method_info 或 record_component_info 结构或 Code 属性(§4.1、§4.5、§4.6、§4.7.30、§4.7.3) attributes 表中的一个变长属性。RuntimeInvisibleTypeAnnotations 属性将运行时不可见的注解存储在类、字段、方法或记录组件的相应声明中使用的类型上，或者存储在相应方法体中的表达式中。RuntimeInvisibleTypeAnnotations 属性还将注解存储在泛型类、接口、方法和构造函数的类型参数声明上。

ClassFile、field_info、method_info 或 record_component_info 结构或 Code 属性的 attributes 表中最多可以有一个 RuntimeInvisibleTypeAnnotations 属性。

只有当类型在与 attributes 表的父结构或属性对应的各种声明或表达式中被注解时，attributes 表才包含 RuntimeInvisibleTypeAnnotations 属性。

RuntimeInvisibleTypeAnnotations 属性有以下格式:

```
RuntimeInvisibleTypeAnnotations_attribute {
    u2              attribute_name_index;
    u4              attribute_length;
    u2              num_annotations;
    type_annotation annotations[num_annotations];
}
```

RuntimeInvisibleTypeAnnotations_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的

constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构，表示字符串 "RuntimeInvisibleTypeAnnotations"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`num_annotations`

num_annotations 项的值给出了由该结构表示的运行时不可见类型注解的数量。

`annotations[]`

annotations 表中的每个条目表示声明或表达式中使用的类型上的单个运行时不可见注解。type_annotation 结构在§4.7.20 中有详细说明。

4.7.22 `AnnotationDefault` 属性

AnnotationDefault 属性是某个 method_info 结构(§4.6)的 attributes 表中的一个变长属性，也就是那些表示注解接口元素的结构(JLS§9.6.1)。AnnotationDefault 属性记录了 method_info 结构所表示元素的默认值(JLS§9.6.2)。

在 method_info 结构的 attributes 表中最多可以有一个 AnnotationDefault 属性，它代表一个注解接口的元素。

AnnotationDefault 属性有以下格式：

```
AnnotationDefault_attribute {
    u2            attribute_name_index;
    u4            attribute_length;
    element_value default_value;
}
```

AnnotationDefault_attribute 结构的项如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "AnnotationDefault"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`default_value`

default_value 项表示注解接口元素的默认值，该元素由包含这个 AnnotationDefault 属性的 method_info 结构表示。

4.7.23 `BootstrapMethods` 属性

BootstrapMethods 属性是 ClassFile 结构的 attributes 表中的一个变长属性(§4.1)。BootstrapMethods 属性记录了用于生成动态计算的常量和动态计算的调用站点的 bootstrap 方法(§4.4.10)。

如果 ClassFile 结构的 constant_pool 表至少有一个 CONSTANT_Dynamic_info 或 CONSTANT_InvokeDynamic_info 条目，那么 ClassFile 结构的 attributes 表中必须有一个 BootstrapMethods 属性。

ClassFile 结构的 attributes 表中最多只能有一个 BootstrapMethods 属性。

BootstrapMethods 属性有以下格式:

```
BootstrapMethods_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 num_bootstrap_methods;
    {   u2 bootstrap_method_ref;
        u2 num_bootstrap_arguments;
        u2 bootstrap_arguments[num_bootstrap_arguments];
    } bootstrap_methods[num_bootstrap_methods];
}
```

BootstrapMethods_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "BootstrapMethods"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`num_bootstrap_methods`

num_bootstrap_methods 项的值决定了 bootstrap_methods 数组中 bootstrap 方法说明符的数量。

`bootstrap_methods[]`

bootstrap_methods 表中的每个条目都包含一个指向 CONSTANT_MethodHandle_info 结构的索引，该结构指定了一个 bootstrap 方法，以及一个指向 bootstrap 方法静态参数的索引序列(可能为空)。

每个 bootstrap_methods 条目必须包含以下三项:

`bootstrap_method_ref`

bootstrap_method_ref 项的值必须是到 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_MethodHandle_info 结构(§4.4.8)。

该方法句柄将在解析动态计算的常量或调用站点时被解析(§5.4.3.6)，然后被调用，就像通过调用 java.lang.invoke.MethodHandle 中的 invokeWithArguments 一样。方法句柄必须能够接受§5.4.3.6 中所述的参数数组，否则解析将失败。

`num_bootstrap_arguments`

num_bootstrap_arguments 项的值给出了 bootstrap_arguments 数组中项目的数量。

`bootstrap_arguments[]`

bootstrap_arguments 数组中的每个条目都必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是可加载的(§4.4)。

4.7.24 `MethodParameters` 属性

MethodParameters 属性是 method_info 结构的 attributes 表中的一个变长属性(§4.6)。MethodParameters 属性记录关于方法的形式参数的信息，比如它们的名称。

在 method_info 结构的 attributes 表中最多可以有一个 MethodParameters 属性。

MethodParameters 属性有以下格式：

```
MethodParameters_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 parameters_count;
    {   u2 name_index;
        u2 access_flags;
    } parameters[parameters_count];
}
```

MethodParameters_attribute 结构的项如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "MethodParameters"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`parameters_count`

parameters_count 项的值表示方法描述符(§4.3.3)中参数描述符的数量，该描述符由属性的封闭 method_info 结构的 descriptor_index 引用。

这不是 Java 虚拟机实现在格式检查时必须执行的约束(§4.8)。将方法描述符中的参数描述符与下面 parameter 数组中的项匹配的任务是由 Java SE 平台的反射库完成的。

`parameters[]`

parameters 数组中的每一个条目都包含以下两项：

`name_index`

name_index 项的值必须为零或为 constant_pool 表的有效索引。

如果 name_index 项的值为零，则此 parameters 元素表示一个没有名称的形式参数。

如果 name_index 项的值非零，该索引的 constant_pool 项必须是一个 CONSTANT_Utf8_info 结构，表示一个有效的非限定名称，该名称表示一个形式参数(§4.2.2)。

`access_flags`

access_flags 项的值如下所示：

`0x0010 (ACC_FINAL)`

指示形参声明为 final。

`0x1000 (ACC_SYNTHETIC)`

根据编写源代码的语言规范(JLS§13.1)，表示形式参数在源代码中没有显式或隐式声明。(形参是生成该 class 文件的编译器的实现构件。)

`0x8000 (ACC_MANDATED)`

表示形式参数在源代码中隐式声明，根据编写源代码的语言规范(JLS§13.1)。(形式参数是由语言规范指定的，因此该语言的所有编译器都必须发出它。)

parameters 数组中的第 i 项对应于封闭方法描述符中的第 i 个参数描述符。(parameters_count 项是一个字节，因为一个方法描述符被限制为 255 个参数。) 实际上，这意味着 parameters 数组存储方法所有参数的信息。可以想象其他的方案，其中 parameters 数组中的条目指定它们相应的参数描述符，但这将过分复杂化 MethodParameters 属性。

parameters 数组中的第 i 项可能对应也可能不对应封闭方法的 Signature 属性(如果存在)中的第 i 个类型，或者对应封闭方法的参数注解中的第 i 个注解。

4.7.25 `Module` 属性

Module 属性是 ClassFile 结构的 attributes 表中的一个变长属性(§4.1)。Module 属性指出模块所需的模块;模块导出和打开的包;以及模块使用和提供的服务。

ClassFile 结构的 attributes 表中最多只能有一个 Module 属性。

Module 属性有以下格式：

```
Module_attribute {
    u2 attribute_name_index;
    u4 attribute_length;

    u2 module_name_index;
    u2 module_flags;
    u2 module_version_index;

    u2 requires_count;
    {   u2 requires_index;
        u2 requires_flags;
        u2 requires_version_index;
    } requires[requires_count];

    u2 exports_count;
    {   u2 exports_index;
        u2 exports_flags;
        u2 exports_to_count;
        u2 exports_to_index[exports_to_count];
    } exports[exports_count];

    u2 opens_count;
    {   u2 opens_index;
```

```
            u2 opens_flags;
            u2 opens_to_count;
            u2 opens_to_index[opens_to_count];
        } opens[opens_count];

        u2 uses_count;
        u2 uses_index[uses_count];

        u2 provides_count;
        {   u2 provides_index;
            u2 provides_with_count;
            u2 provides_with_index[provides_with_count];
        } provides[provides_count];
    }
```

Module_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "Module"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`module_name_index`

module_name_index 项的值必须是到 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Module_info 结构(§4.4.11)，表示当前模块。

`module_flags`

module_flags 项的值如下所示:

`0x0020 (ACC_OPEN)`

表示该模块处于打开状态。

`0x1000 (ACC_SYNTHETIC)`

指示此模块未显式或隐式声明。

`0x8000 (ACC_MANDATED)`

指示此模块已隐式声明。

`module_version_index`

module_version_index 项的值必须为零或为 constant_pool 表的有效索引。如果该项的值为零，则不存在当前模块的版本信息。如果该项的值非零，那么该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体，表示当前模块的版本。

`requires_count`

requires_count 项的值表示 require 表中的条目数。

如果当前模块是 java.base, 则 requires_count 必须为 0。

如果当前模块不是 java.base, 则 requires_count 必须至少为 1。

`requires[]`

require 表中的每一项都指定了当前模块的依赖关系。每一个条目的项如下:

`requires_index`

requires_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Module_info 结构，表示当前模块所依赖的模块。

在 require 表中最多只能有一个条目可以用它的 requires_index 项指定给定名称的模块。

`requires_flags`

requires_flags 项的值如下所示:

`0x0020 (ACC_TRANSITIVE)`

指示依赖于当前模块的任何模块，隐式声明依赖于此条目所指示的模块。

`0x0040 (ACC_STATIC_PHASE)`

指示此依赖在静态阶段(即编译时)是强制性的，但在动态阶段(即运行时)是可选的。

`0x1000 (ACC_SYNTHETIC)`

指示没有在模块声明的源中显式或隐式声明此依赖项。

`0x8000 (ACC_MANDATED)`

指示此依赖项在模块声明的源中隐式声明。

`requires_version_index`

requires_version_index 项的值必须是零或一个到 constant_pool 表的有效索引。如果项的值为 0，则不存在关于依赖关系的版本信息。如果该项的值非零，那么该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构体，表示 requires_index 指定的模块版本。

除非当前模块是 java.base, 在 require 表中，只有一个条目必须具备以下所有条件:

- 指示 java.base 的 requires_index 项。
- 未设置 ACC_SYNTHETIC 标志的 requires_flags 项。(可以设置 ACC_MANDATED 标志。)
- 如果 class 文件版本号为 54.0 或以上，则不设置同时具有 ACC_TRANSITIVE 和 ACC_STATIC_PHASE 标志的 requires_flags 项。

`exports_count`

exports_count 项的值指示 exports 表中的条目数。

`exports[]`

exports 表中的每个条目都指定了当前模块导出的一个包，因此包中的 public 和 protected 类型，以及它们的 public 和 protected 成员，可以从当前模块外部访问，也可能从一组有限的“朋友”模块访问。

每一个条目的项如下:

`exports_index`

exports_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Package_info 结构体(§4.4.12)，表示当前模块导出的包。

exports 表中最多有一个条目可以用它的 exports_index 项指定给定名称的包。

`exports_flags`

exports_flags 项的值如下所示:

`0x1000 (ACC_SYNTHETIC)`

指示此导出未在模块声明的源中显式或隐式声明。

`0x8000 (ACC_MANDATED)`

指示此导出在模块声明的源中隐式声明。

`exports_to_count`

exports_to_count 的值表示 exports_to_index 表中的条目数。

如果 exports_to_count 为零，则此包由当前模块以未限定的方式导出;任何其他模块中的代码都可以访问包中的类型和成员。

如果 exports_to_count 为非零，则当前模块以限定方式导出该包；只有 exports_to_index 表中列出的模块中的代码可以访问包中的类型和成员。

`exports_to_index[]`

exports_to_index 表中每个条目的值必须是 constant_pool 表中的有效索引。该索引处的 constant_pool 条目必须是 CONSTANT_Module_info 结构，表示其代码可以访问该导出包中的类型和成员的模块。

对于 exports 表中的每个条目，其 exports_to_index 表中最多有一个条目可以指定给定名称的模块。

`opens_count`

opens_count 项的值指示 opens 表中的条目数。

如果当前模块是开放的，则 opens_count 必须为零。

`opens[]`

opens 表中的每个条目都指定了当前模块打开的包，这样，包中的所有类型及其所有成员都可以通过 Java SE 平台的反射库从当前模块外部访问，可能是从有限的一组“朋友”模块访问。

每个条目中的项如下:

`opens_index`

opens_index 项的值必须是 constant_pool 表中的有效索引。该索引处的

constant_pool 条目必须是表示当前模块打开的包的 CONSTANT_Package_info 结构。

opens 表中最多有一个条目可以指定具有给定名称的包及其 opens_index 项。

`opens_flags`

opens_flags 项的值如下所示:

`0x1000 (ACC_SYNTHETIC)`

指示此开放未在模块声明的源中显式或隐式声明。

`0x8000 (ACC_MANDATED)`

指示此开放是在模块声明的源中隐式声明的。

`opens_to_count`

opens_to_count 的值表示 opens_to_index 表中的条目数。

如果 opens_to_count 为零，则此包将由当前模块以未限定的方式打开;任何其他模块中的代码都可以反射地访问包中的类型和成员。

如果 opens_to_count 非零，则此包由当前模块以限定的方式打开;只有在 opens_to_index 表中列出的模块中的代码才能反射地访问包中的类型和成员。

`opens_to_index[]`

opens_to_index 表中的每个条目的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Module_info 结构，表示一个模块，该模块的代码可以访问这个打开的包中的类型和成员。

对于 opens 表中的每个条目，在其 opens_to_index 表中最多有一个条目可以指定给定名称的模块。

`uses_count`

uses_count 项的值表示 uses_index 表中的条目数。

`uses_index[]`

uses_index 表中的每个条目的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构(§4.4.1)，表示当前模块可能通过 java.util.ServiceLoader 发现的服务接口。

uses_index 表中最多只有一个条目可以指定给定名称的服务接口。

`provides_count`

provides_count 项的值表示 provides 表中的条目数量。

`provides[]`

provides 表中的每个条目表示给定服务接口的服务实现。

每一个条目的项如下:

`provides_index`

provides_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构，表示当前模块为其提供服务实现的服务接口。

在 provides 表中最多有一个条目可以用它的 provides_index 项指定给定名称的服务接口。

`provides_with_count`

provides_with_count 的值表示 provides_with_index 表中的条目数。

provides_with_count 必须不能为 0。

`provides_with_index[]`

provides_with_index 表中的每个条目的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构，表示由 provides_index 指定的服务接口的服务实现。

对于 provides 表中的每个条目，其 provides_with_index 表中最多有一个条目可以指定给定名称的服务实现。

4.7.26 `ModulePackages` 属性

ModulePackages 属性是 ClassFile 结构的 attributes 表中的一个变长属性(§4.1)。ModulePackages 属性表示模块中由 Module 属性导出或打开的所有包，以及 Module 属性中记录的服务实现的所有包。ModulePackages 属性还可以指示模块中既没有导出、也没有开放、也不包含服务实现的包。

在 ClassFile 结构的 attributes 表中，最多只能有一个 ModulePackages 属性。

ModulePackages 属性有以下格式:

```
ModulePackages_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 package_count;
    u2 package_index[package_count];
}
```

ModulePackages_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "ModulePackages"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`package_count`

package_count 项的值表示 package_index 表的表项数。

`package_index[]`

package_index 表中的每个条目的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Package_info 结构体(§4.4.12)，表示当前模块中的一个包。

在 package_index 表中最多只有一个条目可以指定给定名称的包。

4.7.27 `ModuleMainClass` 属性

ModuleMainClass 属性是 ClassFile 结构 attributes 表中的一个固定长度的属性(§4.1)。ModuleMainClass 属性表示模块的主类。

在 ClassFile 结构的 attributes 表中，最多只能有一个 ModuleMainClass 属性。

ModuleMainClass 属性有以下格式：

```
ModuleMainClass_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 main_class_index;
}
```

ModuleMainClass_attribute 结构的项如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "ModuleMainClass"。

`attribute_length`

attribute_length 项的值必须为 2。

`main_class_index`

main_class_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构体(§4.4.1)，表示当前模块的主类。

4.7.28 `NestHost` 属性

NestHost 属性是 ClassFile 结构的 attributes 表中的固定长度属性。NestHost 属性记录当前类或接口声称属于的嵌套的嵌套主机（§5.4.4）。

ClassFile 结构的 attributes 表中最多只能有一个 NestHost 属性。

NestHost 属性有以下格式：

```
NestHost_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 host_class_index;
}
```

NestHost_attribute 结构的项如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "NestHost"。

`attribute_length`

attribute_length 项的值必须是 2。

`host_class_index`

host_class_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构体(§4.4.1)，它表示一个类或接口，是当前类或接口的嵌套主机。

如果嵌套主机不能被加载，或者与当前类或接口不在同一个运行时包中，或者没有为当前类或接口授权嵌套成员，那么在访问控制过程中可能会发生错误(§5.4.4)。

4.7.29 NestMembers 属性

NestMembers 属性是 ClassFile 结构的 attributes 表中的一个变长属性(§4.1)。NestMembers 属性记录了被授权在当前类或接口托管的嵌套中声明成员关系的类和接口(§5.4.4)。

ClassFile 结构的 attributes 表中最多只能有一个 NestMembers 属性。

ClassFile 结构的 attributes 表不能同时包含 NestMembers 属性和 NestHost 属性。

该规则防止嵌套主机在另一个嵌套中申请成员资格。它隐式地是它所托管的嵌套的成员。它隐式地是它所托管的嵌套的成员。

NestMembers 属性有以下格式：

```
NestMembers_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    u2 classes[number_of_classes];
}
```

NestMembers_attribute 结构的项如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "NestMembers"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`number_of_classes`

number_of_classes 项的值表示 classes 数组中条目的数量。

`classes[]`

classes 数组中的每个值都必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构体(§4.4.1)，表示一个类或接口，该类或接口是当前类或接口所托管的嵌套的成员。

classes 数组由访问控制查询(§5.4.4)。它应该包含对其他类和接口的引用，这些类和接口位于同一个运行时包中，并且具有引用当前类或接口的 NestHost 属性。不满足这些条件的数组项将被访问控制忽略。

4.7.30 Record 属性

Record 属性是 ClassFile 结构的 attributes 表中的一个变长属性(§4.1)。Record 属性表明当前类是一个记录类(JLS§8.10)，并存储关于记录类(JLS§8.10.1)的记录组件的信息。

ClassFile 结构的 attributes 表中最多只能有一个 Record 属性。

Record 属性有以下格式：

```
Record_attribute {
    u2                    attribute_name_index;
    u4                    attribute_length;
    u2                    components_count;
    record_component_info components[components_count];
}
```

Record_attribute 结构的项如下所示：

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "Record"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`components_count`

components_count 项的值表示 components 表中的条目数。

`components[]`

按照记录组件声明的顺序，components 表中的每个条目指定当前类的一个记录组件。record_component_info 结构有以下格式：

```
record_component_info {
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

record_component_info 结构的项如下所示：

`name_index`

name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示一个有效的非限定名称，该名称表示记录组件(§4.2.2)。

`descriptor_index`

descriptor_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示一个字段描述符，它编码记录组件的类型(§4.3.2)。

`attributes_count`

attributes_count 项的值表示这个记录组件的附加属性的数量。

`attributes[]`

attributes 表的每个值都必须是 attribute_info 结构(§4.7)。

一个记录组件可以有任意数量的可选属性与其关联。

本规范定义的属性出现在 record_component_info 结构的 attributes 表中，如表 4.7-C 所示。

关于在 record_component_info 结构的 attributes 表中定义的属性的规则见§4.7。

在 record_component_info 结构的 attributes 表中，关于非预定义属性的规则见§4.7.1。

4.7.31 PermittedSubclasses 属性

PermittedSubclasses 属性是 ClassFile 结构 attributes 表中的一个变长属性(§4.1)。PermittedSubclasses 属性记录被授权直接扩展或实现当前类或接口的类和接口(§5.3.5)。

> Java 编程语言使用修饰符 sealed 来表示限制其直接子类或直接子接口的类或接口。有人可能会认为这个修饰符对应于 class 文件中的 ACC_SEALED 标志，因为相关的修饰符 final 对应于 ACC_FINAL 标志。事实上，密封的类或接口是通过 class 文件中 PermittedSubclasses 属性的存在来指示的。

在 access_flags 项没有 ACC_FINAL 标志集的 ClassFile 结构的 attributes 表中，最多可能有一个 PermittedSubclasses 属性。

在 access_flags 项设置了 ACC_FINAL 标志的 ClassFile 结构的 attributes 表中，必须没有 PermittedSubclasses 属性。

> sealed 不同于 final: sealed 类有一个授权子类的列表, 而 final 类没有子类。因此, ClassFile 结构可以有一个 PermittedSubclasses 属性，或者设置 ACC_FINAL 标志，但不能两者都有。

PermittedSubclasses 属性有以下格式:

```
PermittedSubclasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    u2 classes[number_of_classes];
```

```
}
```

PermittedSubclasses_attribute 结构的项如下所示:

`attribute_name_index`

attribute_name_index 项的值必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Utf8_info 结构(§4.4.7)，表示字符串 "PermittedSubclasses"。

`attribute_length`

attribute_length 项的值表示属性的长度，不包括最初的 6 个字节。

`number_of_classes`

number_of_classes 项的值表示 classes 数组中条目的数量。

`classes[]`

classes 数组中的每个值都必须是 constant_pool 表的有效索引。该索引的 constant_pool 条目必须是一个 CONSTANT_Class_info 结构体(§4.4.1)，表示一个被授权直接扩展或实现当前类或接口的类或接口。

当创建的类或接口试图直接扩展或实现当前的类或接口时，会参考 classes 数组(§5.3.5)。表示不试图直接扩展或实现当前类或接口的类或接口的数组项将被忽略。表示不试图直接扩展或实现当前类或接口的类或接口的数组项将被忽略。

4.8 格式检查

当一个 class 文件被 Java 虚拟机(§5.3)加载时，Java 虚拟机首先确保该文件具有 class 文件的基本格式(§4.1)。这个过程被称为格式检查。检查如下:

- 前四个字节必须包含正确的魔法数字。
- 所有预定义的属性(§4.7)都必须有合适的长度，除了 StackMapTable, RuntimeVisibleAnnotations,RuntimeInvisibleAnnotations,RuntimeVisibleParameterAnnotations,RuntimeInvisibleParameterAnnotations,RuntimeVisibleTypeAnnotations,RuntimeInvisibleTypeAnnotations, 和 AnnotationDefault。
- class 文件不能被截断或在末尾有额外的字节。
- 常量池必须满足§4.4 中记录的约束。

  例如，常量池中的每个 CONSTANT_Class_info 结构必须在其 name_index 项中包含一个针对 CONSTANT_Utf8_info 结构的有效常量池索引。

- 常量池中的所有字段引用和方法引用都必须有有效的名称、有效的类和有效的描述符(§4.3)。

  格式检查不能确保给定的字段或方法实际上存在于给定的类中，也不能确保给定的描述符引用真实的类。格式检查只确保这些项是格式良好的。在验证字节码本身和解析

期间执行更详细的检查。

这些基本 class 文件完整性检查对于任何 class 文件内容的解释都是必要的。格式检查不同于字节码验证，尽管它们在历史上一直被混淆，因为它们都是完整性检查的一种形式。

4.9 Java 虚拟机代码约束

方法、实例初始化方法(§2.9.1)或类或接口初始化方法(§2.9.2)的代码存储在 class 文件的 method_info 结构(§4.7.3)的 Code 属性的 code 数组中。本节描述与 Code_attribute 结构的内容相关的约束。

4.9.1 静态约束

class 文件上的静态约束是那些定义文件格式良好性的约束。除了 class 文件中代码的静态约束外，这些约束在前面的小节中已经给出了。class 文件中代码的静态约束指定了 Java 虚拟机指令必须如何在 code 数组中布局，以及各个指令的操作数必须是什么。

code 数组中指令的静态约束如下:

- 只有§6.5 中记录的指令实例才能出现在 code 数组中。使用保留操作码(§6.2)或本规范中没有记录的操作码的指令实例不能出现在 code 数组中。

  如果 class 文件的版本号是 51.0 或更高，那么使用 jsr、jsr_w 或 ret 操作码的指令实例不能出现在 code 数组中。

- code 数组中第一个指令的操作码从索引 0 开始。

- 对于 code 数组中除最后一条指令外的每条指令，下一条指令的操作码的索引等于当前指令的操作码的索引加上该指令的长度，包括它的所有操作数。

  出于这些目的，wide 指令被视为与任何其他指令一样；指定 wide 指令要修改的操作的操作码被视为该 wide 指令的操作数之一。该操作码绝对不能被计算直接访问。

- code 数组中最后一条指令的最后一个字节必须是索引 code_length - 1 处的字节。

code 数组中指令操作数的静态约束如下:

- 每个跳转和分支指令的目标 (jsr, jsr_w, goto, goto_w, ifeq, ifne, ifle, iflt, ifge, ifgt, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmple, if_icmplt, if_icmpge, if_icmpgt, if_acmpeq, if_acmpne) 必须是该方法中指令的操作码。

  跳转或分支指令的目标绝不能是用于指定要由 wide 指令修改的操作的操作码;跳转或分支目标可以是 wide 指令本身。

- 每个 tableswitch 指令的每个目标(包括默认值)都必须是该方法中某个指令的操作码。

  每个 tableswitch 指令在其跳转表中必须有与其 low 跳转表和 high 跳转表操作数的值一致的多个条目，且其 low 值必须小于或等于其 high 值。

tableswitch 指令的目标不能是用来指定要被 wide 指令修改的操作的操作码;tableswitch 目标本身可能是一个 wide 指令。

- 每个 lookupswitch 指令的每个目标(包括默认值)都必须是该方法中某个指令的操作码。

  每个 lookupswitch 指令必须有一定数量的匹配偏移量对，与它的 npairs 操作数的值一致。匹配偏移对必须按带符号的匹配值递增的数字顺序排序。

  lookupswitch 指令的目标不能是用来指定由 wide 指令修改的操作的操作码;lookupswitch 目标本身可能是一个 wide 指令。

- 每个 ldc 指令和每个 ldc_w 指令的操作数必须表示到 constant_pool 表的有效索引。该索引所引用的常量池条目必须是可加载的(§4.4)，并且不能是以下任何一个:

  - 条目的种类为 CONSTANT_Long 或 CONSTANT_Double。
  - 一种 CONSTANT_Dynamic 类型的条目，它引用了一个 CONSTANT_NameAndType_info 结构，该结构表示 J(表示 long)或 D(表示 double)的描述符。

- 每个 ldc2_w 指令的操作数必须表示到 constant_pool 表的有效索引。该索引引用的常量池条目必须是可加载的，特别是以下其中之一:

  - 条目的种类为 CONSTANT_Long 或 CONSTANT_Double。
  - 类型为 CONSTANT_Dynamic 的条目，它引用了一个 CONSTANT_NameAndType_info 结构，该结构表示 J(表示 long)或 D(表示 double)的描述符。

    随后的常量池索引也必须是常量池的有效索引，并且不能使用该索引处的常量池条目。

- 每个 getfield、putfield、getstatic 和 putstatic 指令的操作数必须表示一个到 constant_pool 表的有效索引。该索引引用的常量池条目必须是 CONSTANT_Fieldref 类型的。

- 每个 invokevirtual 指令的 indexbyte 操作数必须表示到 constant_pool 表的有效索引。该索引引用的常量池条目必须是 CONSTANT_Methodref 类型的。

- 每个 invokespecial 和 invokestatic 指令的 indexbyte 操作数必须表示到 constant_pool 表的有效索引。如果 class 文件的版本号小于 52.0，索引引用的常量池条目的类型必须是 CONSTANT_Methodref;如果 class 文件的版本号是 52.0 或更高，那么索引引用的常量池条目必须是 CONSTANT_Methodref 或 CONSTANT_InterfaceMethodref 类型的。

- 每个 invokeinterface 指令的 indexbyte 操作数必须表示到 constant_pool 表的有效索引。该索引引用的常量池条目必须是 CONSTANT_InterfaceMethodref 类型的。

  每个 invokeinterface 指令的 count 操作数的值必须反映存储传递给接口方法的参数所

需的局部变量的数量，正如 CONSTANT_InterfaceMethodref 常量池条目引用的 CONSTANT_NameAndType_info 结构的描述符所暗示的那样。

每个 invokeinterface 指令的第四个操作数字节的值必须为零。

- 每个invokedynamic指令的indexbyte操作数必须表示到constant_pool表的有效索引。该索引引用的常量池条目必须是 CONSTANT_InvokeDynamic 类型的。

  每个 invokedynamic 指令的第三和第四个操作数字节的值必须为零。

- 只有 invokespecial 指令才允许调用实例初始化方法(§2.9.1)。

  方法调用指令不能调用名称以字符'<' ('\u003c')开头的其他方法。特别是，特别命名为<clinit>的类或接口初始化方法从来不会从 Java 虚拟机指令中显式地调用，而只能由 Java 虚拟机本身隐式地调用。

- 每个instanceof、checkcast、new和anwarray指令的操作数，以及每个multianewarray指令的 indexbyte 操作数，必须表示到 constant_pool 表的有效索引。该索引引用的常量池条目必须是 CONSTANT_Class 类型的。

- 任何 new 指令都不能引用类型为 CONSTANT_Class 的表示数组类型的常量池条目(§4.3.2)。new 指令不能用于创建数组。

- 不能使用任何 anewarray 指令来创建超过 255 维的数组。

- multianewarray 指令必须仅用于创建至少与其维数的值相同维数的数组类型。也就是说，虽然 multianewarray 指令不需要创建 indexbyte 操作数所引用的数组类型的所有维数，但它不能尝试创建比数组类型更多的维数。

  每个 multianewarray 指令的维数不能为零。

- 每个 newarray 指令的 atype 操作数必须为以下值之一:T_BOOLEAN(4)、T_CHAR(5)、T_FLOAT(6)、T_DOUBLE(7)、T_BYTE(8)、T_SHORT(9)、T_INT(10)或 T_LONG(11)。

- 每个 iload, fload, aload, istore, fstore, astore, iinc,和 ret 指令的索引操作数必须是一个不大于 max_locals - 1 的非负整数。

  每个 iload_<n>, fload_<n>, aload_<n>, istore_<n>, fstore_<n>和 astore_<n> 指令的隐含索引必须不大于 max_locals – 1。

- 每个 lload, dload, lstore, 和 dstore 指令的索引操作数必须不大于 max_locals – 2。

  每个 lload_<n>, dload_<n>, lstore_<n>, 和 dstore_<n>指令的隐含索引必须不大于 max_locals – 2。

- 修改iload, fload, aload, istore, fstore, astore, iinc或ret指令的每个wide指令的indexbyte操作数必须表示一个不大于 max_locals - 1 的非负整数。

  修改 lload, dload, lstore 或 dstore 指令的每个 wide 指令的 indexbyte 操作数必须表示

一个不大于 max_locals - 2 的非负整数。

4.9.2 结构约束

code 数组上的结构约束指定了 Java 虚拟机指令之间关系的约束。结构约束如下:

- 每个指令只能在操作数栈和局部变量数组中使用适当类型和数量的参数来执行，而不考虑导致其调用的执行路径。

  对 int 类型的值进行操作的指令也允许对 boolean、byte、char 和 short 类型的值执行操作。

  > 如§2.3.4 和§2.11.1 所述，Java 虚拟机在内部将 boolean、byte、short 和 char 类型的值转换为 int 类型。

- 如果一条指令可以沿多条不同的执行路径执行，则在执行该指令之前，操作数栈必须具有相同的深度（§2.6.2），而不管所采用的路径如何。
- 在执行的任何时候，操作数栈都不能增长到大于 max_stack 项所暗示的深度。
- 在执行过程中，从操作数堆栈中弹出的值的个数不能超过其包含的值的个数。
- 在执行过程中，任何时候都不能颠倒持有 long 或 double 类型值的局部变量对的顺序，也不能拆分该对。在任何情况下，都不能单独操作这样一对的局部变量。
- 在分配值之前，无法访问任何局部变量（或局部变量对，如果是 long 或 double 类型的值）。
- 每个 invokespecial 指令必须命名以下其中一个:
  - 实例初始化方法 (§2.9.1)
  - 当前类或接口中的方法
  - 当前类的超类中的方法
  - 当前类或接口的直接超接口中的方法
  - Object 中的方法

  如果 invokespecial 指令命名实例初始化方法，则操作数栈上的目标引用必须是未初始化的类实例。

决不能在初始化的类实例上调用实例初始化方法。另外:

- 如果操作数栈上的目标引用是当前类的未初始化类实例，则 invokespecial 必须从当前类或其直接超类中命名实例初始化方法。
- 如果 invokespecial 指令命名实例初始化方法，并且操作数栈上的目标引用是由早期 new 指令创建的类实例，则 invokespecial 必须从该类实例的类中命名实例初始化法。

如果 invokespecial 指令命名的方法不是实例初始化方法，则操作数栈上的目标引用必须是

类型与当前类赋值兼容的类实例（JLS§5.2）。

> invokespecial 的一般规则是，由 invokespecial 命名的类或接口必须在调用方类或接口的“之上”，而 invokespecial 所针对的的接收方对象必须在调用方类或接口的位置或“之下”。 后一个子句特别重要：一个类或接口只能对它自己的对象执行 invokespecial。有关后一个子句如何在 Prolog 中实现的说明，请参阅§invokespecial。

- 每个实例初始化方法，除了从类 Object 的构造函数派生的实例初始化方法外，必须在访问其实例成员之前调用 this 的另一个实例初始化方法或其直接超类 super 的实例初始化方法。

  然而，在当前类中声明的 this 的实例字段可以在调用任何实例初始化方法之前由 putfield 赋值。

- 当调用任何实例方法或访问任何实例变量时，包含实例方法或实例变量的类实例必须已经初始化。

- 如果在受异常处理程序保护的代码中的局部变量中存在未初始化的类实例，则(i) 如果处理程序在<init>方法中，处理程序必须抛出一个异常或永远循环;并且(ii) 如果处理程序不在<init>方法中，未初始化的类实例必须保持未初始化状态。

- 当执行 jsr 或 jsr_w 指令时，操作数栈或局部变量中绝对不能有未初始化的类实例。

- 作为方法调用指令目标的每个类实例的类型(即操作数栈上目标引用的类型)必须与指令中指定的类或接口类型兼容。

- 每个方法调用的参数类型必须是与方法描述符兼容的方法调用（JLS§5.3，§4.3.3）。

- 每个返回指令必须与其方法的返回类型匹配:

  - 如果方法返回 boolean, byte, char, short 或 int, 只能使用 ireturn 指令。
  - 如果方法返回 float, long 或 double,只能分别使用 freturn, lreturn,或 dreturn 指令。
  - 如果方法返回 reference 类型,只能使用 areturn 指令，并且返回值的类型必须与方法的返回描述符赋值兼容(§4.3.3)。
  - 所有实例初始化方法, 类或接口初始化方法, 以及声明返回 void 的方法必须只使用 return 指令。

- 由 getfield 指令访问或由 putfield 指令修改的每个类实例的类型(即，操作数栈上的目标引用的类型)必须与指令中指定的类类型赋值兼容。

- putfield 或 putstatic 指令存储的每个值的类型必须与存储到（下面）的类实例或类的字段描述符(§4.3.2)兼容:

  - 如果描述符类型是 boolean, byte, char, short, 或 int, 则值必须是 int。
  - 如果描述符类型是 float, long 或 double, 则值必须分别是 float, long 或 double。

- 如果描述符类型是 reference 类型, 则值的类型必须与描述符类型赋值兼容。

- 每个用 aastore 指令存储到数组的值的类型必须是 reference 类型。

  用 aastore 指令存储到数组的组件类型必须是一个 reference 类型。

- 每个 athrow 指令必须只抛出 Throwable 类的实例或 Throwable 子类的值。

  方法的 Code_attribute 结构的 exception_table 数组的 catch_type 项中提到的每个类都必须是 Throwable 或 Throwable 的子类。

- 如果 getfield 或 putfield 用于访问超类中声明的受保护字段，该超类是与当前类不同的运行时包成员，那么被访问的类实例的类型(即操作数栈上的目标引用的类型)必须与当前类赋值兼容。

  如果使用 invokevirtual 或 invokespecial 来访问超类中声明的受保护方法，该超类是与当前类不同的运行时包的成员，那么被访问的类实例类型(即操作数栈上的目标引用类型)必须与当前类赋值兼容。

- 执行永远不会脱离 code 数组的底部。

- 不能从局部变量加载返回地址(类型为 returnAddress 的值)。

- 每个 jsr 或 jsr_w 指令后面的指令只能由单个 ret 指令返回。

- 如果子例程调用链中已经存在该子例程，则返回的 jsr 或 jsr_w 指令不能用于递归调用该子例程。(当在 finally 子句中使用 try-finally 构造时，子例程可以嵌套。)

- 每个 returnAddress 类型的实例最多只能返回一次。

  如果 ret 指令返回到 ret 指令上面的子例程调用链中的一个点，该点对应于 returnAddress 类型的给定实例，那么该实例永远不能用作返回地址。

4.10 class 文件的验证

尽管 Java 编程语言的编译器必须只生成满足前几节中所有静态和结构约束的 class 文件，但 Java 虚拟机不能保证它被要求加载的任何文件都是由该编译器生成的或格式正确。网络浏览器等应用程序不下载源代码，然后编译;这些应用程序下载已经编译好的 class 文件。浏览器需要确定 class 文件是由可信的编译器生成的，还是由试图利用 Java 虚拟机的对手生成的。

> 编译时检查的另一个问题是版本倾斜。用户可能已经成功地将类(例如 PurchaseStockOptions)编译为 TradingClass 的子类。但是 TradingClass 的定义可能已经改变了，因为这个类的编译方式与已经存在的二进制文件不兼容。方法可能已被删除，或者其返回类型或修饰符已更改。字段可能更改了类型，或者从实例变量更改为类变量。方法或变量的访问修饰符可能已从公共的更改为私有的。有关这些问题的讨论，请参阅《Java 语言规范，Java SE 19 版》第 13 章“二进制兼容性”。

由于这些潜在的问题，Java 虚拟机需要自己验证它试图合并的 class 文件是否满足了所需

的约束。Java 虚拟机实现在链接时验证每个 class 文件是否满足必要的约束(§5.4)。

链接时验证增强了运行时解释器的性能。可以消除在运行时为每个解释指令验证约束而必须执行的昂贵检查。Java 虚拟机可以假设已经执行了这些检查。例如，Java 虚拟机已经知道以下信息:

- 没有操作数栈上溢或下溢。
- 所有局部变量的使用和存储都有效。
- 所有 Java 虚拟机指令的参数都是有效类型。

Java 虚拟机实现可以使用两种策略进行验证:

- 对于版本号大于等于 50.0 的 class 文件，必须使用类型检查验证。
- 所有的 Java 虚拟机实现都必须支持类型推断验证，除了那些符合 Java ME CLDC 和 Java Card 配置文件的，以便验证版本号小于 50.0 的 class 文件。

  对支持 Java ME CLDC 和 Java Card 配置文件的 Java 虚拟机实现的验证由它们各自的规范管理。

在这两种策略中，验证主要涉及到对 Code 属性(§4.7.3)的 code 数组强制执行§4.9 中的静态和结构约束。但是，在验证过程中，Code 属性之外还有三个额外的检查必须执行:

- 确保 final 类没有子类化。
- 确保 final 方法不被重写(§5.4.5)。
- 检查每个类(Object 除外)都有一个直接超类。

4.10.1 通过类型检查验证

对于版本号为 50.0 或更高(§4.1)的 class 文件，必须使用本节给出的类型检查规则进行验证。

当且仅当 class 文件的版本号等于 50.0，那么如果类型检查失败，Java 虚拟机实现可以选择尝试通过类型推断来执行验证(§4.10.2)。

> 这是一种务实的调整，旨在缓解向新的核查纪律的过渡。许多操作 class 文件的工具可能会以需要调整方法栈映射帧的方式更改方法的字节码。如果工具没有对栈映射帧进行必要的调整，那么即使字节码在原则上是有效的，类型检查也可能会失败 (并因此在旧的类型推断方案下进行验证)。为了让实现者有时间调整他们的工具，Java 虚拟机实现可能会退回到旧的验证规程，但只是在有限的时间内。
>
> 在类型检查失败但调用类型推断并成功的情况下，预计会有一定的性能损失。这样的惩罚是不可避免的。它还应该作为工具供应商的一个信号，说明他们的输出需要调整，并为供应商提供额外的激励来进行这些调整。
>
> 总之，通过类型推断进行故障转移到验证，既支持向 Java SE 平台逐步添加栈映射帧(如果它们没有出现在 50.0 版本的 class 文件中，则允许故障转移)，也支持从 Java SE 平台逐步删除 jsr 和 jsr_w 指令(如果它们出现在 50.0 版本的 class 文件中，则允许故障转移)。

如果 Java 虚拟机实现试图通过类型推断对 50.0 版本的 class 文件执行验证，那么在所有通

过类型检查验证失败的情况下，它都必须这样做。

> 这意味着 Java 虚拟机实现不能选择在一种情况下使用类型推断，而在另一种情况下不使用。它必须要么拒绝不通过类型检查进行验证的 class 文件，要么在类型检查失败时始终故障转移到类型推断验证器。

类型检查器强制执行通过 Prolog 子句指定的类型规则。英语文本用于以非正式的方式描述类型规则，而 Prolog 子句提供正式的规范。

类型检查器需要为每个带有 Code 属性的方法提供一个栈映射帧的列表(§4.7.3)。栈映射帧的列表由 Code 属性的 StackMapTable 属性(§4.7.4)给出。目的是一个栈映射帧必须出现在一个方法的每个基本块的开始。栈映射帧指定每个操作数栈条目的验证类型，以及每个基本块开始处的每个局部变量的验证类型。类型检查器读取带有 Code 属性的每个方法的栈映射帧，并使用这些映射生成 Code 属性中指令的类型安全的证明。

如果一个类的所有方法都是类型安全的，则该类是类型安全，并且它不是 final 类的子类。

```
classIsTypeSafe(Class) :-
    classClassName(Class, Name),
    classDefiningLoader(Class, L),
    superclassChain(Name, L, Chain),
    Chain \= [],
    classSuperClassName(Class, SuperclassName),
    loadedClass(SuperclassName, L, Superclass),
    classIsNotFinal(Superclass),
    classMethods(Class, Methods),
    checklist(methodIsTypeSafe(Class), Methods).

classIsTypeSafe(Class) :-
    classClassName(Class, 'java/lang/Object'),
    classDefiningLoader(Class, L),
    isBootstrapLoader(L),
    classMethods(Class, Methods),
    checklist(methodIsTypeSafe(Class), Methods).
```

Prolog 谓词 classIsTypeSafe 假设 Class 是一个 Prolog 术语，表示已成功解析和加载的二进制类。该规范没有强制要求这个术语的精确结构，但确实要求在其上定义某些谓词。

> 例如，我们假设有一个谓词 classMethods (Class, Methods)，给定一个如上所述的表示类的术语作为它的第一个参数，它将第二个参数绑定到一个包含类的所有方法的列表上，这个列表稍后将以一种方便的形式描述。

当且仅当谓词 classIsTypeSafe 不是 true，类型检查器必须抛出 VerifyError 异常来指示 class 文件格式不正确。否则，class 文件已成功检查类型，字节码验证已成功完成。

本节的其余部分详细解释了类型检查的过程:

- 首先，我们将 Prolog 谓词用于核心的 Java 虚拟机构件，如类和方法(§4.10.1.1)。
- 第二，我们指定类型检查器所知道的类型系统(§4.10.1.2)。
- 第三，我们指定指令和栈映射帧的 Prolog 表示(§4.10.1.3，§4.10.1.4)。
- 第四，我们指定如何对方法进行类型检查，包括无代码的方法(§4.10.1.5)和有代码的方

法(§4.10.1.6)。

- 第五，我们讨论了所有加载和存储指令共同的类型检查问题(§4.10.1.7)，以及访问受保护成员的问题(§4.10.1.8)。
- 最后，我们指定对每个指令进行类型检查的规则(§4.10.1.9)。、

4.10.1.1 用于 Java 虚拟机构件的访问器

我们规定存在 28 个 Prolog 谓词("访问器")，它们具有特定的预期行为，但本规范中没有给出它们的正式定义。

`classClassName(Class, ClassName)`
提取类 Class 的名字 ClassName。

`classIsInterface(Class)`
当且仅当 Class 类是一个接口时，返回 true。

`classIsNotFinal(Class)`
当且仅当 Class 类不是 final 类时，返回 true。

`classSuperClassName(Class, SuperClassName)`
提取类 Class 的超类的名字，SuperClassName。

`classInterfaces(Class, Interfaces)`
提取类 Class 的直接超接口的列表 Interfaces。

`classMethods(Class, Methods)`
提取类 Class 中声明的方法的列表 Methods。

`classAttributes(Class, Attributes)`
提取类 Class 的属性列表 Attributes。

每个属性都表示为形式为 attribute(AttributeName, AttributeContents)的仿函数应用程序, 其中 AttributeName 是属性名。属性内容的格式未指定。

`classDefiningLoader(Class, Loader)`
提取类 Class 的定义类加载器 Loader。

`isBootstrapLoader(Loader)`
当且仅当类加载器 Loader 是引导类加载器，则为 true。

`loadedClass(Name, InitiatingLoader, ClassDefinition)`
当且仅当存在一个名为 Name 的类，当由类加载器 InitiatingLoader 加载时，其表示(按照本规范)为 ClassDefinition，则为 true。

`methodName(Method, Name)`
提取方法 Method 的名称 Name。

`methodAccessFlags(Method, AccessFlags)`

提取方法 Method 的访问标志 AccessFlags。

`methodDescriptor(Method, Descriptor)`

提取方法 Method 的描述符 Descriptor。

`methodAttributes(Method, Attributes)`

提取方法 Method 的属性列表 Attributes。

`isInit(Method)`

当且仅当 Method (不管类) 是<init>，返回 true。

`isNotInit(Method)`

当且仅当 Method (不管类)不是<init>，返回 true。

`isNotFinal(Method, Class)`

当且仅当类 Class 中的 Method 不是 final，返回 true。

`isStatic(Method, Class)`

当且仅当类 Class 中的 Method 是 static，返回 true。

`isNotStatic(Method, Class)`

当且仅当类 Class 中的 Method 不是 static，返回 true。

`isPrivate(Method, Class)`

当且仅当类 Class 中的 Method 是 private，返回 true。

`isNotPrivate(Method, Class)`

当且仅当类 Class 中的 Method 不是 private，返回 true。

`isProtected(MemberClass, MemberName, MemberDescriptor)`

当且仅当类 MemberClass 中有一个成员名为 MemberName，且其描述符为 MemberDescriptor 且该成员是受保护的，返回 true。

`isNotProtected(MemberClass, MemberName, MemberDescriptor)`

当且仅当类 MemberClass 中有一个成员名为 MemberName 且其描述符为 MemberDescriptor 且不是受保护的，返回 true。

`parseFieldDescriptor(Descriptor, Type)`

将字段描述符 Descriptor 转换为对应的验证类型 Type (§4.10.1.2)。

`parseMethodDescriptor(Descriptor, ArgTypeList, ReturnType)`

将方法描述符 Descriptor 转换为验证类型的列表 ArgTypeList，对应于方法参数类型，以及验证类型 ReturnType，对应于返回类型。

`parseCodeAttribute(Class, Method, FrameSize, MaxStack, ParsedCode, Handlers, StackMap)`

提取 Class 中的 Method 方法的指令流 ParsedCode，以及操作数栈的最大大小 MaxStack、局部变量的最大数量、FrameSize、异常处理程序 Handlers 和栈映射 StackMap。

指令流和栈映射属性的表示必须如§4.10.1.3 和§4.10.1.4 中规定的那样。

`samePackageName(Class1, Class2)`

当且仅当 Class1 和 Class2 的包名相同，返回 true。

`differentPackageName(Class1, Class2)`

当且仅当 Class1 和 Class2 的包名不同，返回 true。

当对方法体进行类型检查时，可以方便地访问有关该方法的信息。为此，我们定义了一个环境，一个六元组，包括:

- 一个类
- 一个方法
- 方法的声明返回类型
- 方法中的指令
- 操作数栈的最大大小
- 异常处理程序的列表

我们指定访问器来从环境中提取信息。

```
allInstructions(Environment, Instructions) :-
   Environment = environment(_Class, _Method, _ReturnType,
                             Instructions, _, _).

exceptionHandlers(Environment, Handlers) :-
   Environment = environment(_Class, _Method, _ReturnType,
                             _Instructions, _, Handlers).

maxOperandStackLength(Environment, MaxStack) :-
   Environment = environment(_Class, _Method, _ReturnType,
                             _Instructions, MaxStack, _Handlers).

thisClass(Environment, class(ClassName, L)) :-
   Environment = environment(Class, _Method, _ReturnType,
                             _Instructions, _, _),
   classDefiningLoader(Class, L),
   classClassName(Class, ClassName).

thisMethodReturnType(Environment, ReturnType) :-
   Environment = environment(_Class, _Method, ReturnType,
                             _Instructions, _, _).
```

我们指定额外的谓词来从环境中提取高级信息。

```
offsetStackFrame(Environment, Offset, StackFrame) :-
   allInstructions(Environment, Instructions),
   member(stackMap(Offset, StackFrame), Instructions).


currentClassLoader(Environment, Loader) :-
   thisClass(Environment, class(_, Loader)).
```

最后，我们指定了一个在整个类型规则中使用的通用谓词:

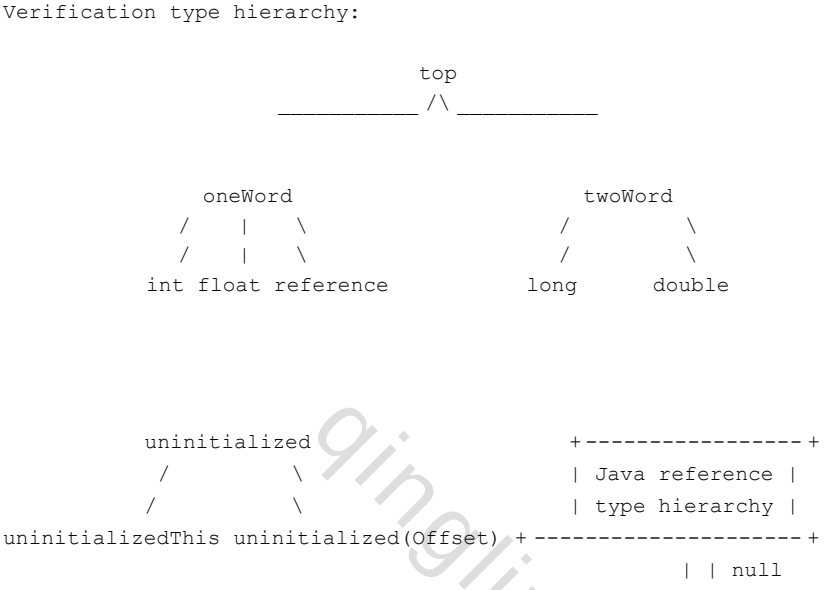
```
notMember(_, []).
```

```
notMember(X, [A | More]) :- X \= A, notMember(X, More).
```

指导确定哪些访问器被规定，哪些被完全指定的原则是，我们不想过度指定 class 文件的表示。为 Class 或 Method 项提供特定的访问器将迫使我们完全指定表示 class 文件的 Prolog 项的格式。

4.10.1.2 验证类型系统

类型检查器基于验证类型的层次结构来实施类型系统，如下所示。



大多数验证类型与表 4.3-A 中字段描述符所表示的原生和引用类型有直接的对应关系:

- 原生类型 double, float, int 和 long(字段描述符 D、F、I、J)分别对应同名的验证类型。
- 原生类型 byte, char, short 和 boolean(字段描述符 B、C、S、Z)都对应于验证类型 int。
- 类和接口类型(字段描述符以 L 开头)对应于使用仿函数类的验证类型。验证类型类(N, L)表示由加载器 L 加载的二进制名称为 N 的类。注意 L 是由 class(N, L)表示的类的初始化加载器(§5.3)，可能是，也可能不是类的定义加载器。

  例如，类类型 Object 将表示为 class('java/lang/ Object', BL)，其中 BL 是引导加载程序。

- 数组类型(字段描述符以[开头)对应于使用仿函数 arrayOf 的验证类型。注意，原生类型 byte, char, short 和 boolean 不对应于验证类型，但是元素类型为 byte, char, short 或 boolean 的数组类型对应于验证类型;这种验证类型支持 baload, bastore, caload, castore, saload, sastore 和 newarray 指令。
  - 验证类型 arrayOf(T)表示组件类型为验证类型 T 的数组类型。
  - 验证类型 arrayOf(byte)表示元素类型为 byte 的数组类型。
  - 验证类型 arrayOf(char)表示元素类型为 char 的数组类型。

- 验证类型 arrayOf(short)表示元素类型为 short 的数组类型。
- 验证类型 arrayOf(boolean)表示元素类型为 boolean 的数组类型。

例如，数组类型 int[]和 Object[]将分别由验证类型 arrayOf(int)和 arrayOf(class('java/lang/Object', BL))表示。数组类型 byte[]和 boolean[][]将分别由验证类型 arrayOf(byte)和 arrayOf(arrayOf(boolean))表示。

其余验证类型说明如下:

- 验证类型 top, oneWord, twoWord 和 reference 在 Prolog 中表示为原子，它们的名称表示所讨论的验证类型。
- 验证类型 uninitialized(Offset)是通过对表示 Offset 数值的参数应用 uninitialized 仿函数来表示的。

验证类型的子类型规则如下。

子类型化是自反的。

```
isAssignable(X, X).
```

Java 编程语言中不是引用类型的验证类型有如下形式的子类型规则:

```
isAssignable(v, X) :- isAssignable(the_direct_supertype_of_v, X).
```

即如果 v 的直接超类型是 X 的子类型，则 v 是 X 的子类型。规则如下:

```
isAssignable(oneWord, top).
isAssignable(twoWord, top).

isAssignable(int, X) :- isAssignable(oneWord, X).
isAssignable(float, X) :- isAssignable(oneWord, X).
isAssignable(long, X) :- isAssignable(twoWord, X).
isAssignable(double, X) :- isAssignable(twoWord, X).

isAssignable(reference, X) :- isAssignable(oneWord, X).
isAssignable(class(_, _), X) :- isAssignable(reference, X).
isAssignable(arrayOf(_), X) :- isAssignable(reference, X).

isAssignable(uninitialized, X) :- isAssignable(reference, X).
isAssignable(uninitializedThis, X) :- isAssignable(uninitialized, X).
isAssignable(uninitialized(_), X) :- isAssignable(uninitialized, X).

isAssignable(null, class(_, _)).
isAssignable(null, arrayOf(_)).
isAssignable(null, X) :- isAssignable(class('java/lang/Object', BL), X),
                         isBootstrapLoader(BL).
```

这些子类型规则不一定是子类型最明显的表述。在 Java 编程语言中，用于引用类型的子类型规则与用于其他验证类型的规则之间有明显的区别。区分允许我们陈述 Java 编程语言引用类型和其他验证类型之间的一般子类型关系。这些关系独立于 Java 引用类型在类型层次结构中的位置，并有助于防止 Java 虚拟机实现加载过多的类。例如，我们不希望开始攀登 Java 超类层次结构来响应形式为 class(foo, L) <: twoWord 的查询。

我们还有一个规则说子类型是自反的，所以这些规则加在一起涵盖了 Java 编程语言中大多数不是引用类型的验证类型。

Java 编程语言中引用类型的子类型规则是用 isJavaAssignable 递归指定的。

```
isAssignable(class(X, Lx), class(Y, Ly)) :-
    isJavaAssignable(class(X, Lx), class(Y, Ly)).

isAssignable(arrayOf(X), class(Y, L)) :-
    isJavaAssignable(arrayOf(X), class(Y, L)).

isAssignable(arrayOf(X), arrayOf(Y)) :-
    isJavaAssignable(arrayOf(X), arrayOf(Y)).
```

对于赋值，接口被视为 Object。

```
isJavaAssignable(class(_, _), class(To, L)) :-
    loadedClass(To, L, ToClass),
    classIsInterface(ToClass).

isJavaAssignable(From, To) :- isJavaSubclassOf(From, To).
```

数组类型是 Object 的子类型。目的还在于，数组类型是 Cloneable 和 java.io.Serializable 的子类型。

```
isJavaAssignable(arrayOf(_), class('java/lang/Object', BL)) :-
    isBootstrapLoader(BL).

isJavaAssignable(arrayOf(_), X) :-
    isArrayInterface(X).

isArrayInterface(class('java/lang/Cloneable', BL)) :-
    isBootstrapLoader(BL).

isArrayInterface(class('java/io/Serializable', BL)) :-
    isBootstrapLoader(BL).
```


原生类型数组之间的子类型是相等关系。

```
isJavaAssignable(arrayOf(X), arrayOf(Y)) :-
    atom(X),
    atom(Y),
    X = Y.
```

引用类型数组之间的子类型是协变的。

```
isJavaAssignable(arrayOf(X), arrayOf(Y)) :-
    compound(X), compound(Y), isJavaAssignable(X, Y).
```

子类化是自反的。

```
isJavaSubclassOf(class(SubclassName, L), class(SubclassName, L)).

isJavaSubclassOf(class(SubclassName, LSub), class(SuperclassName, LSuper)) :-
    superclassChain(SubclassName, LSub, Chain),
    member(class(SuperclassName, L), Chain),
    loadedClass(SuperclassName, L, Sup),
    loadedClass(SuperclassName, LSuper, Sup).
```

```
superclassChain(ClassName, L, [class(SuperclassName, Ls) | Rest]) :-
    loadedClass(ClassName, L, Class),
    classSuperClassName(Class, SuperclassName),
    classDefiningLoader(Class, Ls),
    superclassChain(SuperclassName, Ls, Rest).

superclassChain('java/lang/Object', L, []) :-
    loadedClass('java/lang/Object', L, Class),
    classDefiningLoader(Class, BL),
    isBootstrapLoader(BL).
```

4.10.1.3 指令表示

单独的字节码指令在 Prolog 中表示为术语，其仿函数是指令的名称，其参数是其解析的操作数。

例如，aload 指令表示为术语 aload（N），它包括作为指令操作数的索引 N。

指令作为一个整体表示为以下形式的术语列表：

```
instruction(Offset, AnInstruction)
```

例如, instruction(21, aload(1))。

此列表中的指令顺序必须与 class 文件中的相同。

一些指令具有操作数，这些操作数引用 constant_pool 表中表示字段、方法和动态计算的调用站点的条目。这些条目表示为如下形式的仿函数应用程序：

• 常量池条目的 field(FieldClassName, FieldName, FieldDescriptor) 是一个 CONSTANT_Fieldref_info 结构(§4.4.2)。

FieldClassName 是结构中 class_index 项引用的类的名称。FieldName 和 FieldDescriptor 对应于结构的 name_and_type_index 项引用的名称和字段描述符。

• 常量池条目的 method(MethodClassName, MethodName, MethodDescriptor) 是一个 CONSTANT_Methodref_info 结构(§4.4.2)。

MethodClassName 是结构的 class_index 项引用的类的名称。MethodName 和 MethodDescriptor 对应于结构的 name_and_type_index 项引用的名称和方法描述符。

• 常量池条目的 imethod(MethodIntfName, MethodName, MethodDescriptor) 是一个 CONSTANT_InterfaceMethodref_info 结构(§4.4.2)。

MethodIntfName 是结构的 class_index 项引用的接口的名称。MethodName 和 MethodDescriptor 对应于结构的 name_and_type_index 项引用的名称和方法描述符。

• 常量池条目的 dmethod(CallSiteName, MethodDescriptor) 是一个 CONSTANT_InvokeDynamic_info 结构(§4.4.10)。

CallSiteName 和 MethodDescriptor 对应于结构的 name_and_type_index 项引用的名称和方法描述符。 (bootstrap_method_attr_index 项与验证无关。)

为清晰起见，我们假设字段和方法描述符（§4.3.2，§4.2.3）映射为更可读的名称：前导 L 和尾随；从类名中删除，用于原生类型的 BaseType 字符映射到这些类型的名称。

例如，一条 getfield 指令，其操作数引用了一个常量池条目，该条目表示类 Bar 中 F 类型的字段 foo，将表示为 getfield(field('Bar', 'foo', 'F'))。

其中，ldc 指令的操作数指向 constant_pool 表中的一个可加载条目。有九种可加载条目（见表 4.4-C），由以下形式的仿函数应用程序表示：

- 常量池条目的 int(Value) 是一个 CONSTANT_Integer_info 结构(§4.4.4)。

  Value 是由结构的 bytes 项表示的 int 常量。

  例如，加载 int 常量 91 的 ldc 指令将表示为 ldc(int(91))。

- 常量池条目的 float(Value)是一个 CONSTANT_Float_info 结构(§4.4.4)。

  Value 是由结构的 bytes 项表示的 float 常量。

- 常量池条目的 long(Value)是一个 CONSTANT_Long_info 结构(§4.4.5)。

  Value 是由结构的 high_bytes 项和 low_bytes 项表示的 long 常量。

- 常量池条目的 double(Value)是一个 CONSTANT_Double_info 结构(§4.4.5)。

  Value 是由结构的 high_bytes 项和 low_bytes 项表示的 double 常量。

- 常量池条目的 class(ClassName)是一个 CONSTANT_Class_info 结构(§4.4.1)。

  ClassName 是结构中 name_index 项引用的类或接口的名称。

- 常量池条目的 string(Value)是一个 CONSTANT_String_info 结构(§4.4.3)。

  Value 是由结构的 string_index 项引用的字符串。

- 常量池条目的 methodHandle(Kind, Reference)是一个 CONSTANT_MethodHandle_info 结构(§4.4.8)。

  Kind 是结构的 reference_kind 项的值。Reference 是结构的 reference_index 项的值。

- 常量池条目的 methodType(MethodDescriptor)是一个 CONSTANT_MethodType_info 结构(§4.4.9)。

  MethodDescriptor 是由结构的 descriptor_index 项引用的方法描述符。

- 常量池条目的 dconstant(ConstantName, FieldDescriptor) 是一个 CONSTANT_Dynamic_info 结构(§4.4.10)。

  ConstantName 和 FieldDescriptor 对应于结构的 name_and_type_index 项引用的名称和字段描述符。(bootstrap_method_attr_index 项与验证无关。)

4.10.1.4 栈映射帧和类型转换

栈映射帧在 Prolog 中表示为如下形式的术语列表:

```
stackMap(Offset, TypeState)
```

其中:

- Offset 是一个整数，表示栈映射帧应用的字节码偏移(§4.7.4)。

  此列表中的字节码偏移顺序必须与 class 文件中的相同。

- TypeState 是指令在 Offset 处的预期传入类型状态。

类型状态是从操作数栈中的位置和方法的局部变量到验证类型的映射。它有以下形式:

```
frame(Locals, OperandStack, Flags)
```

where:

- Locals 是一个验证类型的列表，因此列表的第 i 个元素(基于 0 的索引)表示局部变量 i 的类型。

  大小为 2 (long 和 double)的类型由两个局部变量表示(§2.6.1)，第一个局部变量是类型本身，第二个局部变量是 top(§4.10.1.7)。

- OperandStack 是一个验证类型的列表，列表的第一个元素表示操作数栈顶部的类型，顶部以下的栈条目的类型以适当的顺序在列表中跟随。

  大小为 2 (long 和 double)的类型由两个栈条目表示，第一个条目是 top，第二个条目是类型本身。

  > 例如，具有 double 值、int 值和 long 值的栈在类型状态中表示为具有 5 个条目的栈: double 值的 top 和 double 条目，int 值的 int 条目，long 值的 top 和 long 条目。因此，OperandStack 是列表[top，double，int，top，long]。

- Flags 是一个列表，可以是空的，也可以只有一个元素 flagThisUninit。

  如果 Locals 中任何局部变量的类型为 uninitializedThis，则 Flags 具有单个元素 flagThisUninit，否则 Flags 为空列表。

  > flagThisUninit 在构造函数中用于标记 this 尚未完成初始化的类型状态。在这种类型状态下，从方法返回是非法的。

验证类型的子类型被逐点扩展到类型状态。方法的局部变量数组通过构造具有固定的长度(参见§4.10.1.6 中的 methodInitialStackFrame)，但操作数栈会增长和收缩，因此我们需要显式检查操作数栈的长度，这些操作数栈的可赋值性需要用于子类型化。

```
frameIsAssignable(frame(Locals1, StackMap1, Flags1),
                  frame(Locals2, StackMap2, Flags2)) :-
     length(StackMap1, StackMapLength),
     length(StackMap2, StackMapLength),
     maplist(isAssignable, Locals1, Locals2),
     maplist(isAssignable, StackMap1, StackMap2),
```

```
        subset(Flags1, Flags2).
```

大多数单独指令的类型规则(§4.10.1.9)依赖于有效类型转换的概念。如果可以从传入类型状态的操作数栈中弹出预期类型列表，并将其替换为预期结果类型，从而产生新的类型状态，其中操作数栈的长度不超过其声明的最大大小，则类型转换是有效的。

```
validTypeTransition(Environment, ExpectedTypesOnStack, ResultType,
                    frame(Locals, InputOperandStack, Flags),
                    frame(Locals, NextOperandStack, Flags)) :-
     popMatchingList(InputOperandStack, ExpectedTypesOnStack,
                    InterimOperandStack),
     pushOperandStack(InterimOperandStack, ResultType, NextOperandStack),
     operandStackHasLegalLength(Environment, NextOperandStack).
```

从栈中弹出一个类型列表。

```
popMatchingList(OperandStack, [], OperandStack).
popMatchingList(OperandStack, [P | Rest], NewOperandStack) :-
    popMatchingType(OperandStack, P, TempOperandStack, _ActualType),
    popMatchingList(TempOperandStack, Rest, NewOperandStack).
```

从栈中弹出一个单独的类型。具体的行为取决于栈的内容。如果栈的逻辑顶部是指定类型 Type 的某些子类型，则弹出它。如果一个类型占用了两个栈条目，那么栈的逻辑顶部实际上是位于顶部下方的类型，而栈的顶部是不可用的类型 top。

```
popMatchingType([ActualType | OperandStack],
                Type, OperandStack, ActualType) :-
    sizeOf(Type, 1),
    isAssignable(ActualType, Type).

popMatchingType([top, ActualType | OperandStack],
                Type, OperandStack, ActualType) :-
    sizeOf(Type, 2),
    isAssignable(ActualType, Type).

sizeOf(X, 2) :- isAssignable(X, twoWord).
sizeOf(X, 1) :- isAssignable(X, oneWord).
sizeOf(top, 1).
```

将逻辑类型压入栈。具体的行为随类型的大小而变化。如果压入类型的大小为 1，我们只需将其压入栈。如果压入类型的大小是 2，我们就压入它，然后压入 top。

```
pushOperandStack(OperandStack, 'void', OperandStack).
pushOperandStack(OperandStack, Type, [Type | OperandStack]) :-
     sizeOf(Type, 1).
pushOperandStack(OperandStack, Type, [top, Type | OperandStack]) :-
     izeOf(Type, 2).
```

操作数栈的长度不能超过声明的最大大小。

```
operandStackHasLegalLength(Environment, OperandStack) :-
     length(OperandStack, Length),
     maxOperandStackLength(Environment, MaxStack),
```

```
    Length =< MaxStack.
```

dup 指令从传入的类型状态的操作数栈中弹出预期的类型，并将它们替换为预定义的结果类型，从而产生新的类型状态。但是，这些指令没有按照类型转换定义，因为不需要通过子类型关系来匹配类型。相反，dup 指令完全根据栈上的类型种类来操作操作数栈(§2.11.1)。

种类 1 类型占用一个栈条目。如果栈的顶部是 Type 而 Type 不是 top，则可以从栈中弹出种类 1 的逻辑类型 Type (否则它可以表示种类 2 类型的上半部分)。结果是传入的栈，顶部条目被弹出。

```
popCategory1([Type | Rest], Type, Rest) :-
    Type \= top,
    sizeOf(Type, 1).
```

种类 2 类型占用两个栈条目。如果栈的顶部是类型 top，并且它正下方的条目是 Type，则可以从栈中弹出一个种类 2 的逻辑类型 Type。结果是传入的栈，上面的两个条目被弹出。

```
popCategory2([top, Type | Rest], Type, Rest) :-
    sizeOf(Type, 2).
```

dup 指令将类型列表推入栈的方式与为进行有效的类型转换而推入类型的方式基本相同。

```
canSafelyPush(Environment, InputOperandStack, Type, OutputOperandStack) :-
     pushOperandStack(InputOperandStack,Type,OutputOperandStack),
     operandStackHasLegalLength(Environment, OutputOperandStack).


canSafelyPushList(Environment, InputOperandStack, Types,
                  OutputOperandStack) :-
     canPushList(InputOperandStack,Types,OutputOperandStack),
     operandStackHasLegalLength(Environment, OutputOperandStack).


canPushList(InputOperandStack, [], InputOperandStack).
canPushList(InputOperandStack, [Type | Rest], OutputOperandStack) :-

     pushOperandStack(InputOperandStack, Type, InterimOperandStack),
     canPushList(InterimOperandStack, Rest, OutputOperandStack).
```

许多单独指令的类型规则使用下面的子句轻松地从栈中弹出类型列表。

```
canPop(frame(Locals, OperandStack, Flags), Types,
     frame(Locals, PoppedOperandStack, Flags)) :-
     popMatchingList(OperandStack, Types, PoppedOperandStack).
```

最后，某些数组指令(§aaload、§arraylength、§baload、§bastore)查看操作数栈上的类型，以检查它们是否是数组类型。下面的子句从类型状态访问操作数栈的第 i 个元素。

```
nth1OperandStackIs(i, frame(_Locals, OperandStack, _Flags), Element) :-
     nth1(i, OperandStack, Element).
```

4.10.1.5 类型检查抽象方法和本地方法

如果抽象方法和本地方法没有重写 final 方法，则被认为是类型安全的。

```
methodIsTypeSafe(Class, Method) :-
     doesNotOverrideFinalMethod(Class, Method),
     methodAccessFlags(Method, AccessFlags),
     member(abstract, AccessFlags).

methodIsTypeSafe(Class, Method) :-
     doesNotOverrideFinalMethod(Class, Method),
     methodAccessFlags(Method, AccessFlags),
     member(native, AccessFlags).
```

私有方法和静态方法与动态方法调度是正交的，所以它们从不重写其他方法(§5.4.5)。

```
doesNotOverrideFinalMethod(class('java/lang/Object', L), Method) :-
     isBootstrapLoader(L).


doesNotOverrideFinalMethod(Class, Method) :-
     isPrivate(Method, Class).


doesNotOverrideFinalMethod(Class, Method) :-
     isStatic(Method, Class).


doesNotOverrideFinalMethod(Class, Method) :-
     isNotPrivate(Method, Class),
     isNotStatic(Method, Class),
     doesNotOverrideFinalMethodOfSuperclass(Class, Method).


doesNotOverrideFinalMethodOfSuperclass(Class, Method) :-
     classSuperClassName(Class, SuperclassName),
     classDefiningLoader(Class, L),
     loadedClass(SuperclassName, L, Superclass),
     classMethods(Superclass, SuperMethodList),
     finalMethodNotOverridden(Method, Superclass, SuperMethodList).
```


私有和/或静态的 final 方法是不常见的，因为私有方法和静态方法本身不能被重写。因此，如果找到了 final private 方法或 final static 方法，那么它在逻辑上不会被另一个方法重写。

```
finalMethodNotOverridden(Method, Superclass, SuperMethodList) :-
     methodName(Method, Name),
     methodDescriptor(Method, Descriptor),
     member(method(_, Name, Descriptor), SuperMethodList),
     isFinal(Method, Superclass),
     isPrivate(Method, Superclass).

finalMethodNotOverridden(Method, Superclass, SuperMethodList) :-
     methodName(Method, Name),
     methodDescriptor(Method, Descriptor),
     member(method(_, Name, Descriptor),
     SuperMethodList),
     isFinal(Method, Superclass),
     isStatic(Method, Superclass).
```

如果找到非 final 私有方法或非 final 静态方法，请跳过它，因为它与重写正交。

```
finalMethodNotOverridden(Method, Superclass, SuperMethodList) :-
     methodName(Method, Name),
     methodDescriptor(Method, Descriptor),
     member(method(_, Name, Descriptor), SuperMethodList),
     isNotFinal(Method, Superclass),
     isPrivate(Method, Superclass),
```

```
    doesNotOverrideFinalMethodOfSuperclass(Superclass, Method).

finalMethodNotOverridden(Method, Superclass, SuperMethodList) :-
    methodName(Method, Name),
    methodDescriptor(Method, Descriptor),
    member(method(_, Name, Descriptor), SuperMethodList),
    isNotFinal(Method, Superclass), isStatic(Method, Superclass),
    doesNotOverrideFinalMethodOfSuperclass(Superclass, Method).
```

如果找到一个非 final、非私有、非静态的方法，那么 final 方法确实没有被重写。否则,向上递归。

```
finalMethodNotOverridden(Method, Superclass, SuperMethodList) :-
    methodName(Method, Name),
    methodDescriptor(Method, Descriptor),
    member(method(_, Name, Descriptor), SuperMethodList),
    isNotFinal(Method, Superclass),
    isNotStatic(Method, Superclass),
    isNotPrivate(Method, Superclass).

finalMethodNotOverridden(Method, Superclass, SuperMethodList) :-
    methodName(Method, Name),
    methodDescriptor(Method, Descriptor),
    notMember(method(_, Name, Descriptor),
    SuperMethodList),
    doesNotOverrideFinalMethodOfSuperclass(Superclass, Method).
```

4.10.1.6 带有代码的类型检查方法

如果非抽象、非本地方法有代码且代码类型正确，则它们是类型正确的。

```
methodIsTypeSafe(Class, Method) :-
    doesNotOverrideFinalMethod(Class, Method),
    methodAccessFlags(Method, AccessFlags),
    methodAttributes(Method, Attributes),
    notMember(native, AccessFlags),
    notMember(abstract, AccessFlags),
    member(attribute('Code', _), Attributes),
    methodWithCodeIsTypeSafe(Class, Method).
```

如果可以将代码和栈映射帧合并到单个流中，使得每个栈映射帧位于其对应的指令之前，并且合并的流是类型正确的，则拥有代码的方法是类型安全的。方法的异常处理程序（如果有）也必须合法。

```
methodWithCodeIsTypeSafe(Class, Method) :-
    parseCodeAttribute(Class, Method, FrameSize, MaxStack,
                       ParsedCode, Handlers, StackMap),
    mergeStackMapAndCode(StackMap, ParsedCode, MergedCode),
    methodInitialStackFrame(Class, Method, FrameSize, StackFrame, ReturnType),
    Environment = environment(Class, Method, ReturnType, MergedCode,
                              MaxStack, Handlers),
    handlersAreLegal(Environment),
    mergedCodeIsTypeSafe(Environment, MergedCode, StackFrame).
```

让我们首先考虑异常处理程序。

异常处理程序由如下形式的仿函数应用程序表示:

```
handler(Start, End, Target, ClassName)
```

它们的参数分别是处理程序覆盖的指令范围的开始和结束，处理程序代码的第一个指令，以及这个处理程序设计来处理的异常类的名称。

如果异常处理程序的开始(Start)小于结束(End)，有一条指令的偏移量等于 Start，有一条指令的偏移量等于 End，并且该处理程序的异常类可赋值给 Throwable 类，那么该异常处理程序就是合法的。如果处理程序的类条目为 0，则处理程序的异常类为 Throwable，否则为处理程序中命名的类。

如果处理程序覆盖的指令之一是<init>方法的 invokespecial，则在<init>方法内部的处理程序存在额外的需求。在这种情况下，处理程序正在运行的事实意味着正在构造的对象很可能被破坏，因此，处理程序不吞下异常并允许封闭的<init>方法正常返回到调用者是很重要的。相应地，处理程序需要通过向封闭的<init>方法的调用者抛出异常来突然完成，或者永远循环。

```
handlersAreLegal(Environment) :-
    exceptionHandlers(Environment, Handlers),
    checklist(handlerIsLegal(Environment), Handlers).


handlerIsLegal(Environment, Handler) :-
    Handler = handler(Start, End, Target, _),
    Start < End,
    allInstructions(Environment, Instructions),
    member(instruction(Start, _), Instructions),
    offsetStackFrame(Environment, Target, _),
    instructionsIncludeEnd(Instructions, End),
    currentClassLoader(Environment, CurrentLoader),
    handlerExceptionClass(Handler, ExceptionClass, CurrentLoader),
    isBootstrapLoader(BL),
    isAssignable(ExceptionClass, class('java/lang/Throwable', BL)),
    initHandlerIsLegal(Environment, Handler).

instructionsIncludeEnd(Instructions, End) :-
    member(instruction(End, _), Instructions).
instructionsIncludeEnd(Instructions, End) :-
    member(endOfCode(End), Instructions).

handlerExceptionClass(handler(_, _, _, 0),
                      class('java/lang/Throwable', BL), _) :-
    isBootstrapLoader(BL).

handlerExceptionClass(handler(_, _, _, Name),
                      class(Name, L), L) :-
    Name \= 0.

initHandlerIsLegal(Environment, Handler) :-
    notInitHandler(Environment, Handler).

notInitHandler(Environment, Handler) :-
    Environment = environment(_Class, Method, _, Instructions, _, _),
```

```
    isNotInit(Method).

notInitHandler(Environment, Handler) :-
    Environment = environment(_Class, Method, _, Instructions, _, _),
    isInit(Method),
    member(instruction(_, invokespecial(CP)), Instructions),
    CP = method(MethodClassName, MethodName, Descriptor),
    MethodName \= '<init>'.

initHandlerIsLegal(Environment, Handler) :-
    isInitHandler(Environment, Handler),
    sublist(isApplicableInstruction(Target), Instructions,
    HandlerInstructions),
    noAttemptToReturnNormally(HandlerInstructions).

isInitHandler(Environment, Handler) :-
    Environment = environment(_Class, Method, _, Instructions, _, _),
    isInit(Method).
    member(instruction(_, invokespecial(CP)), Instructions),
    CP = method(MethodClassName, '<init>', Descriptor).

isApplicableInstruction(HandlerStart, instruction(Offset, _)) :-
    Offset >= HandlerStart.

noAttemptToReturnNormally(Instructions) :-
    notMember(instruction(_, return), Instructions).

noAttemptToReturnNormally(Instructions) :-
    member(instruction(_, athrow), Instructions).
```


现在让我们转向指令流和栈映射帧。

合并指令和栈映射帧到单个流涉及四种情况:

- 将一个空的 StackMap 和一个指令列表合并会产生原始指令列表。

```
mergeStackMapAndCode([], CodeList, CodeList).
```

- 给定一个以 Offset 位置的指令的类型状态开始的栈映射帧列表，以及一个从 Offset 开始的指令列表，合并后的列表是栈映射帧列表的头部，然后是指令列表的头部，然后是两个列表的尾部的合并。

```
mergeStackMapAndCode([stackMap(Offset, Map) | RestMap],
                     [instruction(Offset, Parse) | RestCode],
                     [stackMap(Offset, Map),
                     instruction(Offset, Parse) | RestMerge]) :-
    mergeStackMapAndCode(RestMap, RestCode, RestMerge).
```

- 否则，给定一个以 OffsetM 位置的指令的类型状态开始的栈映射帧列表，以及一个以 OffsetP 位置开始的指令列表，那么，如果 OffsetP < OffsetM，合并后的列表包括指令列表的头部，随后是栈映射帧列表和指令列表的尾部的合并。

```
mergeStackMapAndCode([stackMap(OffsetM, Map) | RestMap],
                     [instruction(OffsetP, Parse) | RestCode],
                     [instruction(OffsetP, Parse) | RestMerge]) :-
    OffsetP < OffsetM,
    mergeStackMapAndCode([stackMap(OffsetM, Map) | RestMap],
                         RestCode, RestMerge).
```

• 否则，两个列表的合并是未定义的。由于指令列表具有单调递增的偏移量，除非每个栈映射帧的偏移量都有一个对应的指令偏移量，并且栈映射帧的顺序是单调递增的，否则不会定义两个列表的合并。

为了确定方法的合并流是否类型正确，我们首先推断方法的初始类型状态。

方法的初始类型状态由空操作数栈、从 this 类型派生的局部变量类型和参数，以及适当的标志组成，这取决于这是否是一个<init>方法。

```
methodInitialStackFrame(Class,Method,FrameSize,frame(Locals,[],Flags),
                        ReturnType):-
    methodDescriptor(Method,Descriptor),
    parseMethodDescriptor(Descriptor, RawArgs, ReturnType),
    expandTypeList(RawArgs, Args),
    methodInitialThisType(Class, Method, ThisList),
    flags(ThisList, Flags), append(ThisList, Args, ThisArgs),
    expandToLength(ThisArgs, FrameSize, top, Locals).
```

给定一个类型列表，下面的子句生成一个列表，其中大小为 2 的每个类型都被两个条目替换:一个用于自身，一个用于 top 条目。然后，结果对应于 Java 虚拟机中以 32 位单词表示的列表。

```
expandTypeList([], []).
expandTypeList([Item | List], [Item | Result]) :-
    sizeOf(Item, 1),
    expandTypeList(List, Result).
expandTypeList([Item | List], [Item, top | Result]) :-
    sizeOf(Item, 2),
    expandTypeList(List, Result).

flags([uninitializedThis], [flagThisUninit]).
flags(X, []) :- X \= [uninitializedThis].

expandToLength(List, Size, _Filler, List) :-
    length(List, Size).
expandToLength(List, Size, Filler, Result) :-
    length(List, ListLength),
    ListLength < Size,
    Delta is Size - ListLength,
    length(Extra, Delta),
    checklist(=(Filler), Extra),
    append(List, Extra, Result).
```

对于实例方法的初始类型状态，我们计算 this 的类型并将其放入一个列表中。Object 的

<init>方法中的 this 的类型是 Object;在其他<init>方法中，this 的类型为 uninitializedThis;否则，实例方法中的 this 的类型是 class(N, L)，其中 N 是包含该方法的类的名称，L 是它定义的类加载器。

对于静态方法的初始类型状态，this 是不相关的，因此列表为空。

```
methodInitialThisType(_Class, Method, []) :-
     methodAccessFlags(Method, AccessFlags),
     member(static, AccessFlags),
     methodName(Method, MethodName),
     MethodName \= '<init>'.

methodInitialThisType(Class, Method, [This]) :-
     methodAccessFlags(Method, AccessFlags),
     notMember(static, AccessFlags),
     instanceMethodInitialThisType(Class,Method,This).

instanceMethodInitialThisType(Class, Method, class('java/lang/Object', L)) :-
     methodName(Method, '<init>'),
     classDefiningLoader(Class, L),
     isBootstrapLoader(L),
     classClassName(Class, 'java/lang/Object').

instanceMethodInitialThisType(Class, Method, uninitializedThis) :-
     methodName(Method, '<init>'),
     classClassName(Class, ClassName),
     classDefiningLoader(Class, CurrentLoader),
     superclassChain(ClassName, CurrentLoader, Chain),
     Chain \= [].

instanceMethodInitialThisType(Class, Method, class(ClassName, L)) :-
     methodName(Method, MethodName),
     MethodName \= '<init>',
     classDefiningLoader(Class, L),
     classClassName(Class, ClassName).
```

现在我们计算方法的合并流是否类型正确，使用方法的初始类型状态:

• 如果我们有一个栈映射帧和一个传入的类型状态，类型状态必须是可赋值给栈映射帧中的类型状态的。然后，我们可以使用栈映射帧中给出的类型状态对流的其余部分进行类型检查。

```
mergedCodeIsTypeSafe(Environment, [stackMap(Offset, MapFrame) | MoreCode],
                     frame(Locals, OperandStack, Flags)) :-
    frameIsAssignable(frame(Locals, OperandStack, Flags), MapFrame),
    mergedCodeIsTypeSafe(Environment, MoreCode, MapFrame).
```

• 如果合并的代码流以相对于 T 的类型安全的指令 I 开始，并且 I 满足它的异常处理程序(见下文)，并且给定 I 执行后的类型状态，流的尾部是类型安全的，那么它相对于传入的类型状态 T 是类型安全的。

NextStackFrame 表示执行以下指令的内容。对于无条件分支指令，它将具有特殊值 afterGoto. ExceptionStackFrame，表示传递给异常处理程序的内容。

```
mergedCodeIsTypeSafe(Environment, [instruction(Offset, Parse) | MoreCode],
```

```
                        frame(Locals, OperandStack, Flags)) :-
    instructionIsTypeSafe(Parse, Environment, Offset,
                          frame(Locals, OperandStack, Flags),
                          NextStackFrame, ExceptionStackFrame),
    instructionSatisfiesHandlers(Environment, Offset, ExceptionStackFrame),
    mergedCodeIsTypeSafe(Environment, MoreCode, NextStackFrame).
```

- 在无条件分支之后(由传入的 afterGoto 类型状态指示)，如果有一个栈映射帧为下面的指令提供类型状态，我们可以使用栈映射帧提供的类型状态继续进行类型检查。

```
mergedCodeIsTypeSafe(Environment, [stackMap(Offset, MapFrame) | MoreCode],
                     afterGoto) :-
    mergedCodeIsTypeSafe(Environment, MoreCode, MapFrame).
```

- 在没有提供栈映射帧的无条件分支之后编写代码是非法的。

```
mergedCodeIsTypeSafe(_Environment, [instruction(_, _) | _MoreCode],
                     afterGoto) :-
    write_ln('No stack frame after unconditional branch'),

    fail.
```

- 如果代码末尾有一个无条件的分支，那么停止。

```
mergedCodeIsTypeSafe(_Environment, [endOfCode(Offset)], afterGoto).
```

如果目标具有关联的栈帧 Frame，则分支到目标是类型安全的，而当前栈帧 StackFrame，是可赋值给 Frame 的。

```
targetIsTypeSafe(Environment, StackFrame, Target) :-
    offsetStackFrame(Environment, Target, Frame),
    frameIsAssignable(StackFrame, Frame).
```

如果指令满足适用于该指令的每个异常处理程序，则该指令满足其异常处理程序。

```
instructionSatisfiesHandlers(Environment, Offset, ExceptionStackFrame) :-
    exceptionHandlers(Environment, Handlers),
    sublist(isApplicableHandler(Offset), Handlers, ApplicableHandlers),
    checklist(instructionSatisfiesHandler(Environment, ExceptionStackFrame),
              ApplicableHandlers).
```

如果指令的偏移量大于或等于处理程序范围的开始且小于处理程序范围的结束，则异常处理程序适用于该指令。

```
isApplicableHandler(Offset, handler(Start, End, _Target, _ClassName)) :-
    Offset >= Start,
    Offset < End.
```

如果指令的输出类型状态是 ExcStackFrame，并且处理程序的目标(处理程序代码的初始指令)是类型安全的(假设传入类型状态为 T)，则指令满足异常处理程序。通过将操作数栈替换为其唯一元素是处理程序的异常类的栈，类型状态 T 派生自 ExcStackFrame。

```
instructionSatisfiesHandler(Environment, ExcStackFrame, Handler) :-
    Handler = handler(_, _, Target, _),
    currentClassLoader(Environment, CurrentLoader),
    handlerExceptionClass(Handler, ExceptionClass, CurrentLoader),
```

```
    /* The stack consists of just the exception. */
    ExcStackFrame = frame(Locals, _, Flags),
    TrueExcStackFrame = frame(Locals, [ ExceptionClass ], Flags),
    operandStackHasLegalLength(Environment, TrueExcStackFrame),
    targetIsTypeSafe(Environment, TrueExcStackFrame, Target).
```

4.10.1.7 类型检查加载和存储指令

所有加载指令都是一种常见模式的变体，改变指令加载的值的类型。

从局部变量 Index 中加载 Type 类型的值是类型安全的，如果局部变量的类型是 ActualType，那么 ActualType 是可赋值给 Type 的，而将 ActualType 压入传入的操作数栈是一个有效的类型转换(§4.10.1.4)，它会产生一个新的类型状态 NextStackFrame。执行 load 指令后，类型状态为 NextStackFrame。

```
loadIsTypeSafe(Environment, Index, Type, StackFrame, NextStackFrame) :-
    StackFrame = frame(Locals, _OperandStack, _Flags),
    nth0(Index, Locals, ActualType),
    isAssignable(ActualType, Type),
    validTypeTransition(Environment, [], ActualType, StackFrame,
                        NextStackFrame).
```

所有存储指令都是通用模式的变体，改变指令存储的值的类型。

一般来说，如果存储指令引用的局部变量的类型是 Type 的超类型，并且操作数栈的顶部是 Type 的子类型，其中 Type 是指令设计用来存储的类型，那么存储指令就是类型安全的。

更准确地说，如果能够从操作数栈(§4.10.1.4)中取出一个与 Type (即 Type 的子类型)“匹配”的 ActualType 类型，然后合法地将该类型赋给局部变量 L_{Index}，那么存储就是类型安全的。

```
storeIsTypeSafe(_Environment, Index, Type,
                frame(Locals, OperandStack, Flags),
                frame(NextLocals, NextOperandStack, Flags)) :-
  popMatchingType(OperandStack, Type, NextOperandStack, ActualType),
  modifyLocalVariable(Index, ActualType, Locals, NextLocals).
```

给定局部变量 Locals，修改 Index 使其具有 Type 类型将在局部变量列表 NewLocals 中产生。这涉及到一些修改，因为一些值(及其对应的类型)占用两个局部变量。因此，修改 L_N 可能需要修改 L_{N+1}(因为该类型将同时占用 N 和 N+1 槽位)或 L_{N-1}(因为 N 过去是从 N-1 开始的两个字值/类型的上半部分，因此 N-1 必须失效)，或两者都需要修改。这将在下面进一步说明。从 L0 开始数。

```
modifyLocalVariable(Index, Type, Locals, NewLocals) :-
    modifyLocalVariable(0, Index, Type, Locals, NewLocals).
```

给定 LocalsRest，从索引 I 开始的局部变量列表的后缀，修改局部变量 Index 使其具有类型 Type，将导致局部变量列表后缀 NextLocalsRest。

如果 I < Index-1, 只需将输入复制到输出，然后向前递归。如果 I = Index-1, 则可以更改 I 的类型。如果 L_I 的类型大小为 2，就会出现这种情况。一旦我们将 L_{I+1} 设置为新的类型(以及相应的值)，L_I 的类型/值将失效，因为它的上半部分将被丢弃。然后我们向前递归。

```
modifyLocalVariable(I, Index, Type,
                    [Locals1 | LocalsRest],
                    [Locals1 | NextLocalsRest] ) :-
    I < Index - 1,
    I1 is I + 1,
    modifyLocalVariable(I1, Index, Type, LocalsRest, NextLocalsRest).

modifyLocalVariable(I, Index, Type,
                    [Locals1 | LocalsRest],
                    [NextLocals1 | NextLocalsRest] ) :-
    I =:= Index - 1,
    modifyPreIndexVariable(Locals1, NextLocals1),
    modifyLocalVariable(Index, Index, Type, LocalsRest, NextLocalsRest).
```

当我们找到变量时，它只占用一个单词，我们将它改为 Type，就完成了。当我们找到变量时，它占用了两个单词，我们将它的类型更改为 Type，下一个单词更改为 top。

```
modifyLocalVariable(Index, Index, Type,
                    [_ | LocalsRest], [Type | LocalsRest]) :-
      sizeOf(Type, 1).

modifyLocalVariable(Index, Index, Type,
                    [_, _ | LocalsRest], [Type, top | LocalsRest]) :-
      sizeOf(Type, 2).
```

我们引用一个局部变量，它的索引紧接在一个局部变量之前，该局部变量的类型将被修改为预索引变量。如果预索引变量的类型 Type 大小为 1，则它不会改变。如果本地预索引类型 Type 为 2，则需要通过将其类型设置为 top，将其两个字值的下半部分标记为不可用。

```
modifyPreIndexVariable(Type, Type) :- sizeOf(Type, 1).
modifyPreIndexVariable(Type, top) :- sizeOf(Type, 2).
```

4.10.1.8 受保护成员的类型检查

访问成员的所有指令必须遵守有关受保护成员的规则。本节描述了与 JLS§6.6.2.1 相对应的受保护检查。

受保护检查仅适用于当前类的超类的受保护成员。其他类中的受保护成员将通过在解析时进行的访问检查被捕获（§5.4.4）。有四种情况：

- 如果类的名称不是任何超类的名称，则它不能是超类，因此可以安全地忽略它。

```
passesProtectedCheck(Environment, MemberClassName, MemberName,
                     MemberDescriptor, StackFrame) :-
    thisClass(Environment, class(CurrentClassName, CurrentLoader)),
    superclassChain(CurrentClassName, CurrentLoader, Chain),
    notMember(class(MemberClassName, _), Chain).
```

- 如果 MemberClassName 与超类的名称相同，则被解析的类实际上可能是超类。在这种情况下，如果在不同的运行时包中没有名为 MemberClassName 的超类具有名为 MemberName 的受保护成员和描述符 MemberDescriptor，则受保护检查不适用。

这是因为被解析的实际类要么是这些超类中的一个，在这种情况下，我们知道它要么在相同的运行时包中，并且访问是合法的;或有关成员不是受保护的，并且检查不适用;或者它将是一个子类，在这种情况下，检查无论如何都会成功;或者它将是同一个运行时包中的其他类，在这种情况下，访问是合法的，不需要进行检查;或者验证者不需要将此标记为问题，因为它无论如何都会被捕获，因为解析将强制失败。

```
passesProtectedCheck(Environment, MemberClassName, MemberName,
                     MemberDescriptor, StackFrame) :-
    thisClass(Environment, class(CurrentClassName,CurrentLoader)),
    superclassChain(CurrentClassName, CurrentLoader, Chain),
    member(class(MemberClassName, _), Chain),
    classesInOtherPkgWithProtectedMember(
      class(CurrentClassName, CurrentLoader),
      MemberName, MemberDescriptor, MemberClassName, Chain, []).
```

- 如果在不同的运行时包中存在一个受保护的超类成员，则加载 MemberClassName;如果该成员不是受保护的，则该检查不适用。(使用不是受保护的超类成员通常是正确的。)

```
passesProtectedCheck(Environment, MemberClassName, MemberName,
                     MemberDescriptor,
                     frame(_Locals, [Target | Rest], _Flags)) :-
     thisClass(Environment, class(CurrentClassName, CurrentLoader)),
     superclassChain(CurrentClassName, CurrentLoader, Chain),
     member(class(MemberClassName, _), Chain),
     classesInOtherPkgWithProtectedMember(
      class(CurrentClassName, CurrentLoader),
      MemberName, MemberDescriptor, MemberClassName, Chain, List),
     List \= [],
     loadedClass(MemberClassName, CurrentLoader, ReferencedClass),
     isNotProtected(ReferencedClass, MemberName, MemberDescriptor).
```

- 否则，使用 Target 类型对象的成员要求 Target 可赋值给当前类的类型。

```
passesProtectedCheck(Environment, MemberClassName, MemberName,
                     MemberDescriptor,
                     frame(_Locals, [Target | Rest], _Flags)) :-
     thisClass(Environment, class(CurrentClassName, CurrentLoader)),
     superclassChain(CurrentClassName, CurrentLoader, Chain),
     member(class(MemberClassName, _), Chain),
     classesInOtherPkgWithProtectedMember(
      class(CurrentClassName, CurrentLoader),
      MemberName, MemberDescriptor, MemberClassName, Chain, List),
     List \= [],
     loadedClass(MemberClassName,CurrentLoader,ReferencedClass),
     isProtected(ReferencedClass, MemberName, MemberDescriptor),
     isAssignable(Target, class(CurrentClassName, CurrentLoader)).
```

如果 List 是 Chain 中名为 MemberClassName 的类的集合，这些类与 Class 在不同的运行时包中，它们有一个受保护的成员名为 MemberName，其描述符为 MemberDescriptorList，则谓词 classesInOtherPkgWithProtectedMember(Class, MemberName, MemberDescriptor, MemberClassName, Chain, List)为 true。

```
classesInOtherPkgWithProtectedMember(_, _, _, _, [], []).

classesInOtherPkgWithProtectedMember(Class, MemberName,
                                     MemberDescriptor,MemberClassName,
                                     [class(MemberClassName,L)|Tail],
                                     [class(MemberClassName,L)|T]):-
    differentRuntimePackage(Class,class(MemberClassName, L)),
    loadedClass(MemberClassName, L, Super),
    isProtected(Super, MemberName, MemberDescriptor),
    classesInOtherPkgWithProtectedMember(
      Class, MemberName, MemberDescriptor, MemberClassName, Tail, T).

classesInOtherPkgWithProtectedMember(Class, MemberName,
                                     MemberDescriptor,MemberClassName,
                                     [class(MemberClassName,L)|Tail],
                                     T) :-
    differentRuntimePackage(Class, class(MemberClassName, L)),
    loadedClass(MemberClassName,L,Super),
    isNotProtected(Super,MemberName,MemberDescriptor),
    classesInOtherPkgWithProtectedMember(
      Class, MemberName, MemberDescriptor, MemberClassName, Tail, T).

classesInOtherPkgWithProtectedMember(Class, MemberName,
                                     MemberDescriptor,MemberClassName,
                                     [class(MemberClassName, L) | Tail],
                                     T]:-
    sameRuntimePackage(Class,class(MemberClassName,L)),
    classesInOtherPkgWithProtectedMember(
      Class, MemberName, MemberDescriptor, MemberClassName, Tail, T).

sameRuntimePackage(Class1, Class2) :-
    classDefiningLoader(Class1, L),
    classDefiningLoader(Class2, L),
    samePackageName(Class1, Class2).

differentRuntimePackage(Class1, Class2) :-
    classDefiningLoader(Class1, L1),
    classDefiningLoader(Class2, L2),
    L1 \= L2.

differentRuntimePackage(Class1, Class2) :-
    differentPackageName(Class1, Class2).
```


4.10.1.9 类型检查指令

一般来说，指令的类型规则是相对于一个环境 Environment 给出的，该环境定义了指令发生的类和方法(§4.10.1.1)，以及指令发生的方法内的偏移量 offset。该规则规定，如果传入的类型状态 StackFrame 满足某些要求，那么:

- 指令是类型安全的。
- 可以证明，指令完成后的类型状态通常有 NextStackFrame 给出的特定形式，指令突然完

成后的类型状态由 ExceptionStackFrame 给出。

指令突然完成后的类型状态与传入的类型状态相同，只是操作数栈为空。

```
exceptionStackFrame(StackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, _OperandStack, Flags),
    ExceptionStackFrame = frame(Locals, [], Flags).
```

许多指令的类型规则与其他指令的规则完全同构。如果指令 b1 与另一条指令 b2 同构，则指令 b1 的类型规则与指令 b2 的类型规则相同。

```
instructionIsTypeSafe(Instruction, Environment, Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    instructionHasEquivalentTypeRule(Instruction, IsomorphicInstruction),
    instructionIsTypeSafe(IsomorphicInstruction, Environment, Offset,
                          StackFrame, NextStackFrame,
                          ExceptionStackFrame).
```

每个规则的英文描述都力求可读性强、直观和简洁。因此，该描述避免重复上述所有的上下文假设。特别是:

- 描述中没有明确提及环境。
- 当下面的描述提到操作数栈或局部变量时，它指的是类型状态的操作数栈和局部变量组件:传入类型状态或传出类型状态。
- 指令突然完成后的类型状态几乎总是与传入的类型状态相同。描述只讨论突然完成指令后的类型状态，如果不是这样的话。
- 该描述谈到了将类型弹出和压入操作数栈，并没有明确讨论栈下溢或上溢的问题。描述假设这些操作可以成功完成，但是操作数栈操作的 Prolog 子句确保进行了必要的检查。
- 该描述只讨论逻辑类型的操作。在实践中，有些类型使用多个单词。描述从这些表示细节中抽象出来，但操作数据的 Prolog 子句却不是这样。

任何歧义都可以通过引用正式的 Prolog 子句来解决。

aaload

aaload 指令是类型安全的，前提是可以有效地将匹配 int 和数组类型的类型替换为组件类型 ComponentType，其中 ComponentType 是 Object 的子类型，而 ComponentType 生成传出类型状态。

```
instructionIsTypeSafe(aaload, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    nth1OperandStackIs(2, StackFrame, ArrayType),
    arrayComponentType(ArrayType, ComponentType),
    isBootstrapLoader(BL),
    validTypeTransition(Environment,
                        [int, arrayOf(class('java/lang/Object', BL))],
                        ComponentType, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

数组 X 的组件类型为 X。我们将 null 的组件类型定义为 null。

```
arrayComponentType(arrayOf(X), X).
arrayComponentType(null, null).
```

aastore

aastore 指令是类型安全的，当且仅当可以有效地从传入操作数栈中弹出匹配 Object、int 和 Object 数组的类型，从而生成传出类型状态。

```
instructionIsTypeSafe(aastore,_Environment,_Offset,StackFrame,
                      NextStackFrame,ExceptionStackFrame) :-
     isBootstrapLoader(BL),
     canPop(StackFrame,
           [class('java/lang/Object', BL),
           int,
           arrayOf(class('java/lang/Object', BL))],
           NextStackFrame),
           exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

aconst_null

如果可以有效地将类型 null 推送到传入操作数栈上，从而产生传出类型状态，则 aconst_null 指令是类型安全的。

```
instructionIsTypeSafe(aconst_null, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], null, StackFrame, NextStackFrame),
```

```
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

aload, aload_<n>

如果具有操作数 Index 和类型 reference 的加载指令是类型安全的，并且生成输出类型状态 NextStackFrame，则具有操作数 Index 的 aload 指令是类型安全的，并且产生输出类型状态 NextStackFrame。

```
instructionIsTypeSafe(aload(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, reference, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 aload_<n>, 对 0 < n < 3, 是类型安全的当且仅当等效的 aload 指令是类型安全的。

```
instructionHasEquivalentTypeRule(aload_0, aload(0)).
instructionHasEquivalentTypeRule(aload_1, aload(1)).
instructionHasEquivalentTypeRule(aload_2, aload(2)).
instructionHasEquivalentTypeRule(aload_3, aload(3)).
```

anewarray

具有操作数 CP 的 anewarray 指令是类型安全的，当且仅当 CP 指的是表示类、接口或数组类型的常量池条目，则可以合法地将传入操作数栈上的类型匹配 int 替换为具有组件类型 CP 的数组，从而生成传出类型状态。

```
instructionIsTypeSafe(anewarray(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
      (CP = class(_, _) ; CP = arrayOf(_)),
      validTypeTransition(Environment, [int], arrayOf(CP),
                          StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

areturn

areturn 指令是类型安全的，当且仅当封闭方法具有声明的返回类型 ReturnType，即 reference 类型，并且可以有效地从传入操作数栈中弹出与 ReturnType 匹配的类型。

```
instructionIsTypeSafe(areturn, Environment, _Offset, StackFrame,
                            afterGoto, ExceptionStackFrame)
    thisMethodReturnType(Environment, ReturnType),
    isAssignable(ReturnType, reference),
    canPop(StackFrame, [ReturnType], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

arraylength

arraylength 指令是类型安全的，当且仅当可以有效地将传入操作数栈上的数组类型替换为 int 类型，从而生成传出类型状态。

```
instructionIsTypeSafe(arraylength, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     nth1OperandStackIs(1, StackFrame, ArrayType),
     arrayComponentType(ArrayType, _),
     validTypeTransition(Environment,[top],int,StackFrame,NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

astore, astore_<n>

如果具有操作数 Index 和类型 reference 的存储指令是类型安全的，并且生成传出类型状态 NextStackFrame，则具有操作数 Index 的 astore 指令是类型安全的，并且产生传出类型状态 NextStackFrame。

```
instructionIsTypeSafe(astore(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, reference, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 astore_<n>, 对 0 < n < 3, 是类型安全的当且仅当等效的 astore 指令是类型安全的。

```
instructionHasEquivalentTypeRule(astore_0, astore(0)).
instructionHasEquivalentTypeRule(astore_1, astore(1)).
instructionHasEquivalentTypeRule(astore_2, astore(2)).
instructionHasEquivalentTypeRule(astore_3, astore(3)).
```

athrow

athrow 指令是类型安全的当且仅当操作数栈的顶部与 Throwable 匹配。

```
instructionIsTypeSafe(athrow,_Environment,_Offset,StackFrame,
                      afterGoto, ExceptionStackFrame) :-
     isBootstrapLoader(BL),
     canPop(StackFrame, [class('java/lang/Throwable', BL)], _PoppedStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

baload

baload 指令是类型安全的，当且仅当可以有效地将输入操作数栈上匹配 int 和小数组类型的类型替换为 int，从而生成输出类型状态。

```
instructionIsTypeSafe(baload, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :
     nth1OperandStackIs(2, StackFrame, ArrayType),
```

```
    isSmallArray(ArrayType),
    validTypeTransition(Environment, [int, top], int,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

如果数组类型是 byte 数组、boolean 数组或其子类型（null），则数组类型是小数组类型。

```
isSmallArray(arrayOf(byte)).
isSmallArray(arrayOf(boolean)).
isSmallArray(null).
```

bastore

bastore 指令是类型安全的当且仅当可以有效地将匹配 int,int 和小数组类型的类型从传入的操作数栈中取出，从而产生传出的类型状态。

```
instructionIsTypeSafe(bastore, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    nth1OperandStackIs(3, StackFrame, ArrayType),
    isSmallArray(ArrayType),
    canPop(StackFrame, [int, int, top], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```


bipush

bipush 指令是类型安全的,当且仅当等效的 sipush 指令是类型安全的。

```
instructionHasEquivalentTypeRule(bipush(Value), sipush(Value)).
```

caload

caload 指令是类型安全的当且仅当可以有效地将传入操作数栈上匹配 int 和 char 数组的类型替换为 int，从而产生传出的类型状态。

```
instructionIsTypeSafe(caload, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int,arrayOf(char)], int,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

castore

castore 指令是类型安全的当且仅当可以有效地从传入的操作数栈中取出匹配 int,int 和 char 数组的类型，从而产生传出的类型状态。

```
instructionIsTypeSafe(castore, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame)
    canPop(StackFrame, [int, int, arrayOf(char)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

checkcast

带有操作数 CP 的 checkcast 指令是类型安全的当且仅当 CP 指的是一个常量池条目，表示类或数组，可以有效地将传入操作数栈上的 Object 类型替换为 CP 所表示的产生传出类型状态的类型。

```
instructionIsTypeSafe(checkcast(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    (CP = class(_, _) ; CP = arrayOf(_)),
    isBootstrapLoader(BL),
    validTypeTransition(Environment, [class('java/lang/Object', BL)], CP,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

d2f, d2i, d2l

d2f 指令是类型安全的，如果可以从传入的操作数栈中有效地弹出 double，并将其替换为 float，从而得到传出的类型状态。

```
instructionIsTypeSafe(d2f, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [double], float,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

d2i 指令是类型安全的，如果可以有效地从传入的操作数栈中弹出 double，并将其替换为 int，从而得到传出的类型状态。

```
instructionIsTypeSafe(d2i, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [double], int,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

d2l 指令是类型安全的，如果可以有效地从传入的操作数堆栈中取出 double，并将其替换为 long，从而得到传出的类型状态。

```
instructionIsTypeSafe(d2l, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
```

```
        validTypeTransition(Environment, [double], long,
                            StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dadd

dadd 指令是类型安全的，当且仅当可以将传入操作数栈上匹配 double 和 double 的类型替换为 double 产生传出类型状态的类型。

```
instructionIsTypeSafe(dadd, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
        validTypeTransition(Environment, [double, double], double,
                      StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

daload

daload 指令是类型安全的，当且仅当可以将传入操作数栈上匹配 int 和 double 数组的类型替换为产生输出类型状态的 double。

```
instructionIsTypeSafe(daload, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
        validTypeTransition(Environment, [int, arrayOf(double)], double,
                      StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dastore

dastore 指令是类型安全的，当且仅当可以从传入的操作数栈中有效地弹出匹配 double,int 和 double 数组的类型，从而产生传出的类型状态。

```
instructionIsTypeSafe(dastore, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [double, int, arrayOf(double)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dcmp<op>

dcmpg 指令是类型安全的，当且仅当可以有效地将传入操作数栈上匹配 double 和 double 的类型替换为 int，从而产生传出的类型状态。

```
instructionIsTypeSafe(dcmpg, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
        validTypeTransition(Environment, [double, double], int,
                      StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dcmpl 指令是类型安全的，当且仅当等价的 dcmpg 指令是类型安全的。

```
instructionHasEquivalentTypeRule(dcmpl, dcmpg).
```

dconst_<d>

dconst_0 指令是类型安全的，如果可以有效地将 double 类型压入传入的操作数栈，从而产生传出的类型状态。

```
instructionIsTypeSafe(dconst_0, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
        validTypeTransition(Environment, [], double, StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dconst_1 指令是类型安全的，当且仅当等价的 dconst_0 指令是类型安全的。

```
instructionHasEquivalentTypeRule(dconst_1, dconst_0).
```

ddiv

ddiv 指令是类型安全的，当且仅当等价的 dadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(ddiv, dadd).
```


dload, dload_<n>

带操作数 Index 的 dload 指令是类型安全的，并产生出 NextStackFrame 类型状态，如果带操作数 Index 和类型 double 的加载指令是类型安全的，并产生出 NextStackFrame 类型状态。

```
instructionIsTypeSafe(dload(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 dload_<n>,对 0 < n < 3,是类型安全的当且仅当等效的 dload 指令是类型安全的。

```
instructionHasEquivalentTypeRule(dload_0, dload(0)).
instructionHasEquivalentTypeRule(dload_1, dload(1)).
instructionHasEquivalentTypeRule(dload_2, dload(2)).
instructionHasEquivalentTypeRule(dload_3, dload(3)).
```

dmul

dmul 指令是类型安全的当且仅当等价的 dadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(dmul, dadd).
```

dneg

dneg 指令是类型安全的当且仅当传入操作数栈上有一个匹配 double 的类型。dneg 指令不会改变类型状态。

```
instructionIsTypeSafe(dneg, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
        validTypeTransition(Environment, [double], double,
                            StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

drem

drem 指令是类型安全的当且仅当等价的 dadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(drem, dadd).
```


dreturn

dreturn 指令是类型安全的如果封闭方法声明返回类型为 double，可以从传入的操作数栈中有效地弹出匹配 double 的类型。

```
instructionIsTypeSafe(dreturn, Environment, _Offset, StackFrame,
                      afterGoto, ExceptionStackFrame)
    thisMethodReturnType(Environment, double),
    canPop(StackFrame, [double], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dstore, dstore_<n>

具有操作数 Index 的 dstore 指令是类型安全的，并产生出 NextStackFrame 类型状态，如果具有操作数 Index 和类型 double 的存储指令是类型安全的，并产生出 NextStackFrame 类型状态。

```
instructionIsTypeSafe(dstore(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
```

```
    storeIsTypeSafe(Environment, Index, double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 dstore_<n>,对 0 < n < 3,是类型安全的当且仅当等价的 dstore 指令是类型安全的。

```
instructionHasEquivalentTypeRule(dstore_0, dstore(0)).
instructionHasEquivalentTypeRule(dstore_1, dstore(1)).
instructionHasEquivalentTypeRule(dstore_2, dstore(2)).
instructionHasEquivalentTypeRule(dstore_3, dstore(3)).
```

dsub

dsub 指令是类型安全的当且仅当等价的 dadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(dsub, dadd).
```

dup

dup 指令是类型安全的，当且仅当可以有效地将类别 1 类型 Type 替换为类型 Type, Type，从而产生输出的类型状态。

```
instructionIsTypeSafe(dup, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     StackFrame = frame(Locals, InputOperandStack, Flags),
     popCategory1(InputOperandStack, Type, _),
     canSafelyPush(Environment, InputOperandStack, Type, OutputOperandStack),
     NextStackFrame = frame(Locals, OutputOperandStack, Flags),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dup_x1

dup_x1 指令是类型安全的，当且仅当可以将传入操作数栈上的两个类别 1 类型 Type1 和 Type2 替换为类型 Type1、Type2、Type1，从而产生传出的类型状态。

```
instructionIsTypeSafe(dup_x1, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type2, Type1],
                      OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dup_x2

dup_x2 指令是类型安全的，当且仅当它是 dup_x2 指令的类型安全形式。

```
instructionIsTypeSafe(dup_x2, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     StackFrame = frame(Locals, InputOperandStack, Flags),
     dup_x2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack),
     NextStackFrame = frame(Locals, OutputOperandStack, Flags),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dup_x2 指令是 dup_x2 指令的类型安全形式，当且仅当它是类型安全形式 1 dup_x2 指令或类型安全形式 2 dup_x2 指令。

```
dup_x2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).

dup_x2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
```

dup_x2 指令是类型安全形式 1 的 dup_x2 指令，当且仅当可以将传入操作数栈上的三个类别 1 类型 Type1, Type2, Type3 替换为 Type1, Type2, Type3, Type1，从而产生传出的类型状态。

```
dup_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
     popCategory1(InputOperandStack, Type1, Stack1),
     popCategory1(Stack1, Type2, Stack2),
     popCategory1(Stack2, Type3, Rest),
     canSafelyPushList(Environment, Rest, [Type1, Type3, Type2, Type1],
                       OutputOperandStack).
```

dup_x2 指令是类型安全形式 2 dup_x2 指令，当且仅当可以将传入操作数栈上的类别 1 类型 Type1 和类别 2 类型 Type2 替换为类型 Type1, Type2, Type1，从而产生传出的类型状态。

```
dup_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
     popCategory1(InputOperandStack, Type1, Stack1),
     popCategory2(Stack1, Type2, Rest),
     canSafelyPushList(Environment, Rest, [Type1, Type2, Type1],
                       OutputOperandStack).
```

dup2

dup2 指令是类型安全的，当且仅当它是 dup2 指令的类型安全形式。

```
instructionIsTypeSafe(dup2, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     StackFrame = frame(Locals, InputOperandStack, Flags),
     dup2FormIsTypeSafe(Environment,InputOperandStack, OutputOperandStack),
     NextStackFrame = frame(Locals, OutputOperandStack, Flags),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dup2 指令是 dup2 指令的类型安全形式，当且仅当它是类型安全形式 1 dup2 指令或类型安全形式 2 dup2 指令。

```
dup2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2Form1IsTypeSafe(Environment,InputOperandStack, OutputOperandStack).

dup2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2Form2IsTypeSafe(Environment,InputOperandStack, OutputOperandStack).
```

dup2 指令是类型安全形式 1 dup2 指令，当且仅当可以有效地将传入操作数栈上的两个类别 1 类型，Type1 和 Type2 替换为类型 Type1, Type2, Type1, Type2，产生传出的类型状态。

```
dup2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack):-
     popCategory1(InputOperandStack, Type1, TempStack),
     popCategory1(TempStack, Type2, _),
     canSafelyPushList(Environment, InputOperandStack, [Type2, Type1],
                       OutputOperandStack).
```

dup2 指令是类型安全形式 2 dup2 指令，当且仅当可以有效地将传入操作数栈上的种类 2 类型 Type 替换为 Type, Type，从而产生传出的类型状态。

```
dup2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack):-
     popCategory2(InputOperandStack, Type, _),
     canSafelyPush(Environment, InputOperandStack, Type, OutputOperandStack).
```

dup2_x1

dup2_x1 指令是类型安全的，当且仅当它是 dup2_x1 指令的类型安全形式。

```
instructionIsTypeSafe(dup2_x1, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     StackFrame = frame(Locals, InputOperandStack, Flags),
     dup2_x1FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack),
     NextStackFrame = frame(Locals, OutputOperandStack, Flags),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dup2_x1 指令是 dup2_x1 指令的类型安全形式，当且仅当它是类型安全形式 1 dup2_x1 指令或类型安全形式 2 dup_x2 指令。

```
dup2_x1FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x1Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).

dup2_x1FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x1Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
```

dup2_x1 指令是类型安全形式 1 dup2_x1 指令，当且仅当可以将传入操作数栈上的三种类别 1 类型 Type1, Type2, Type3 替换为 Type1, Type2, Type3, Type1, Type2，从而产生传出的类型状态。

```
dup2_x1Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
```

```
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Stack2),
    popCategory1(Stack2, Type3, Rest),
    canSafelyPushList(Environment, Rest, [Type2, Type1, Type3, Type2, Type1],
                      OutputOperandStack).
```

dup2_x1 指令是类型安全形式 2 dup2_x1 指令，当且仅当可以将传入操作数栈上的类别 2 类型 Type1 和类别 1 类型 Type2 替换为类型 Type1, Type2, Type1，从而产生传出的类型状态。

```
dup2_x1Form2IsTypeSafe(Environment,    InputOperandStack,    OutputOperandStack)    :-
    popCategory2(InputOperandStack, Type1, Stack1), popCategory1(Stack1, Type2, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type2, Type1],
                      OutputOperandStack).
```

dup2_x2

dup2_x2 指令是类型安全的，当且仅当它是 dup2_x2 指令的类型安全形式。

```
instructionIsTypeSafe(dup2_x2, Environment, _Offset, StackFrame,

                      NextStackFrame, ExceptionStackFrame) :-

    StackFrame = frame(Locals, InputOperandStack, Flags),
    dup2_x2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dup2_x2 指令是 dup2_x2 指令的类型安全形式，当且仅当下列条件之一成立:

- 它是一个类型安全形式 1 dup2_x2 指令。
- 它是一个类型安全形式 2 dup2_x2 指令。
- 它是一个类型安全形式 3 dup2_x2 指令。
- 它是一个类型安全形式 4 dup2_x2 指令。

```
dup2_x2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).

dup2_x2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).

dup2_x2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x2Form3IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).

dup2_x2FormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x2Form4IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
```

dup2_x2 指令是类型安全形式 1 dup2_x2 指令，当且仅当可以将传入操作数栈上的四种类别 1 类型 Type1, Type2, Type3, Type4 替换为 Type1, Type2, Type3, Type4, Type1, Type2,

从而产生传出的类型状态。

```
dup2_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
     popCategory1(InputOperandStack, Type1, Stack1),
     popCategory1(Stack1, Type2, Stack2),
     popCategory1(Stack2, Type3, Stack3),
     popCategory1(Stack3, Type4, Rest),
     canSafelyPushList(Environment, Rest,
                       [Type2, Type1, Type4, Type3, Type2, Type1],
                       OutputOperandStack).
```

dup2_x2 指令是类型安全形式 2 dup2_x2 指令，当且仅当可以将传入操作数栈上的一个类别 2 类型 Type1 和两个类别 1 类型 Type2, Type3 替换为类型 Type1, Type2, Type3, Type1，从而产生传出的类型状态。

```
dup2_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
     popCategory2(InputOperandStack, Type1, Stack1),
     popCategory1(Stack1, Type2, Stack2),
     popCategory1(Stack2, Type3, Rest),
     canSafelyPushList(Environment, Rest,
                       [Type1, Type3, Type2, Type1],
                       OutputOperandStack).
```

dup2_x2 指令是类型安全形式 3 dup2_x2 指令，当且仅当可以使用类型 Type1, Type2, Type3, Type1, Type2 替换传入操作数栈上的两个类别 1 类型(Type1,Type2)和一个类别 2 类型(Type3)，产生传出的类型状态。

```
dup2_x2Form3IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
     popCategory1(InputOperandStack, Type1, Stack1),
     popCategory1(Stack1, Type2, Stack2),
     popCategory2(Stack2, Type3, Rest),
     canSafelyPushList(Environment, Rest, [Type2, Type1, Type3, Type2, Type1],
                       OutputOperandStack).
```

dup2_x2 指令是类型安全形式 4 dup2_x2 指令，当且仅当可以有效地将传入操作数栈上的两个类别 2 类型 `Type1`, `Type2` 替换为类型 `Type1`, `Type2`, `Type1`，从而产生传出的类型状态。one can validly replace two category 2 types, `Type1`, `Type2`, on the incoming operand stack with the types `Type1`, `Type2`, `Type1`, yielding the outgoing type state.

```
dup2_x2Form4IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
     popCategory2(InputOperandStack, Type1, Stack1),
     popCategory2(Stack1, Type2, Rest),
     canSafelyPushList(Environment, Rest, [Type1, Type2, Type1],
                       OutputOperandStack).
```

f2d, f2i, f2l

如果可以有效地从传入的操作数栈中弹出 float 并使用 double 替换它，从而产生传出的类型状态，那么 f2d 指令就是类型安全的。

```
instructionIsTypeSafe(f2d, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [float], double,
```

```
                        StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

如果可以有效地从传入的操作数栈中弹出 float 并将其替换为 int，从而产生传出的类型状态，那么 f2i 指令是类型安全的。

```
instructionIsTypeSafe(f2i, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
      validTypeTransition(Environment, [float], int,
                        StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

如果可以有效地将 float 从传入操作数栈中弹出，并将其替换为 long，从而产生传出的类型状态，那么 f2l 指令是类型安全的。

```
instructionIsTypeSafe(f2l, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
      validTypeTransition(Environment, [float], long,
                        StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fadd

fadd 指令是类型安全的，当且仅当可以有效地将传入操作数栈上与 float 和 float 匹配的类型替换为产生传出类型状态的 float。

```
instructionIsTypeSafe(fadd, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
      validTypeTransition(Environment, [float, float], float,
                        StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

faload

faload 指令是类型安全的，当且仅当可以有效地将传入操作数栈上匹配 int 和 float 数组的类型替换为产生传出类型状态的 float。

```
instructionIsTypeSafe(faload, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
      validTypeTransition(Environment, [int, arrayOf(float)], float,
                        StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fastore

fastore 指令是类型安全的，当且仅当可以有效地将匹配 float、int 和 float 数组的类型从传入的操作数栈中弹出，从而产生传出的类型状态。

```
instructionIsTypeSafe(fastore, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
      canPop(StackFrame, [float, int, arrayOf(float)], NextStackFrame),
```

```
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fcmp<op>

fcmpg 指令是类型安全的，当且仅当可以有效地将传入操作数堆栈上匹配 float 和 float 的类型替换为 int，从而产生传出的类型状态。

```
instructionIsTypeSafe(fcmpg, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [float, float], int,
                      StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fcmpl 指令是类型安全的，当且仅当等效的 fcmpg 指令是类型安全的。

```
instructionHasEquivalentTypeRule(fcmpl, fcmpg).
```

fconst_<f>

如果可以有效地将类型 float 推送到传入操作数栈上，从而产生传出类型状态，则 fconst_0 指令是类型安全的。

```
instructionIsTypeSafe(fconst_0, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fconst 的其他变体的规则是等效的。

```
instructionHasEquivalentTypeRule(fconst_1, fconst_0).
instructionHasEquivalentTypeRule(fconst_2, fconst_0).
```

fdiv

fdiv 指令是类型安全的，当且仅当等效的 fadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(fdiv, fadd).
```

fload, fload_<n>

如果具有操作数 Index 和类型 float 的加载指令是类型安全的，并且生成输出类型状态 NextStackFrame，则具有操作数 Index 的 fload 指令是类型安全的，并生成输出类型的状态 NextStackFrame。

```
instructionIsTypeSafe(fload(Index), Environment, _Offset, StackFrame, NextStackFrame,
```

```
                        ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 fload_<n>, 对 0 < n < 3,是类型安全的，当且仅当等效 fload 指令是类型安全的。

```
instructionHasEquivalentTypeRule(fload_0, fload(0)).
instructionHasEquivalentTypeRule(fload_1, fload(1)).
instructionHasEquivalentTypeRule(fload_2, fload(2)).
instructionHasEquivalentTypeRule(fload_3, fload(3)).
```

fmul

fmul 指令是类型安全的，当且仅当等效的 fadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(fmul, fadd).
```

fneg

fneg 指令是类型安全的，当且仅当传入操作数栈上有一个类型匹配 float。fneg 指令不会改变类型状态。

```
instructionIsTypeSafe(fneg, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame)
   validTypeTransition(Environment, [float], float,
                       StackFrame, NextStackFrame),
   exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

frem

frem 指令是类型安全的，当且仅当等效的 fadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(frem, fadd).
```

freturn

如果封闭方法声明的返回类型为 float，那么 freturn 指令就是类型安全的，并且可以从传入的操作数栈中有效地弹出与 float 匹配的类型。

```
instructionIsTypeSafe(freturn, Environment, _Offset, StackFrame,
                        afterGoto, ExceptionStackFrame)
    thisMethodReturnType(Environment, float),
    canPop(StackFrame, [float], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fstore, fstore_<n>

如果具有操作数 Index 和类型 float 的存储指令是类型安全的，并产生输出类型状态 NextStackFrame，那么具有操作数 Index 的 fstore 指令是类型安全的，并产生出 NextStackFrame 类型状态，。

```
instructionIsTypeSafe(fstore(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 fstore_<n>, 对 0 < n < 3,是类型安全的，当且仅当等效的 fstore 指令是类型安全的。

```
instructionHasEquivalentTypeRule(fstore_0, fstore(0)).
instructionHasEquivalentTypeRule(fstore_1, fstore(1)).
instructionHasEquivalentTypeRule(fstore_2, fstore(2)).
instructionHasEquivalentTypeRule(fstore_3, fstore(3)).
```

fsub

fsub 指令是类型安全的，当且仅当等价的 fadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(fsub, fadd).
```


getfield

具有操作数 CP 的 getfield 指令是类型安全的，当且仅当 CP 引用一个常量池条目，该条目表示声明类型为 FieldType 的字段，声明在 FieldClassName 类中，并且可以有效地将传入操作数栈上与 FieldClassName 匹配的类型替换为 FieldType 类型，从而产生传出的类型状态。FieldClassName 不能是数组类型。受保护的字段需要额外检查(§4.10.1.8)。

```
instructionIsTypeSafe(getfield(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    CP = field(FieldClassName, FieldName, FieldDescriptor),
    parseFieldDescriptor(FieldDescriptor, FieldType),
    passesProtectedCheck(Environment, FieldClassName, FieldName,
                         FieldDescriptor, StackFrame),
    currentClassLoader(Environment,CurrentLoader),
    validTypeTransition(Environment,
                        [class(FieldClassName, CurrentLoader)], FieldType,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

getstatic

带有操作数 CP 的 getstatic 指令是类型安全的，当且仅当 CP 引用一个常量池条目，它表示一个声明类型为 FieldType 的字段，可以在传入操作数栈上有效地压入 FieldType，从而产生传出的类型状态。

```
instructionIsTypeSafe(getstatic(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    CP = field(_FieldClassName, _FieldName, FieldDescriptor),
    parseFieldDescriptor(FieldDescriptor, FieldType),
    validTypeTransition(Environment, [], FieldType,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

goto, goto_w

goto 指令是类型安全的，当且仅当它的目标操作数是一个有效的分支目标。

```
instructionIsTypeSafe(goto(Target), Environment, _Offset, StackFrame,
                      afterGoto, ExceptionStackFrame) :-
    targetIsTypeSafe(Environment, StackFrame, Target),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

goto_w 指令是类型安全的，当且仅当等效的 goto 指令是类型安全的。

```
instructionHasEquivalentTypeRule(goto_w(Target), goto(Target)).
```

i2b, i2c, i2d, i2f, i2l, i2s

i2b 指令是类型安全的，当且仅当等效的 ineg 指令是类型安全的。

```
instructionHasEquivalentTypeRule(i2b, ineg).
```

i2c 指令是类型安全的，当且仅当等效的 ineg 指令是类型安全的。

```
instructionHasEquivalentTypeRule(i2c, ineg).
```

如果可以有效地从传入操作数栈中弹出 int 并将其替换为 double，从而生成传出类型状态，则 i2d 指令是类型安全的。

```
instructionIsTypeSafe(i2d, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [int], double,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

如果可以有效地从传入操作数栈中弹出 int 并将其替换为 float，从而生成传出类型状态，则 i2f 指令是类型安全的。

```
instructionIsTypeSafe(i2f, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [int], float,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

如果可以有效地从传入操作数栈中弹出 int，并将其替换为 long，从而生成传出类型状态，则 i2l 指令是类型安全的。

```
instructionIsTypeSafe(i2l, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [int], long,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

i2s 指令是类型安全的，当且仅当等效的 ineg 指令是类型安全的。

```
instructionHasEquivalentTypeRule(i2s, ineg).
```

iadd

iadd 指令是类型安全的，当且仅当可以有效地将传入操作数栈上匹配 int 和 int 的类型替换为产生输出类型状态的 int。

```
instructionIsTypeSafe(iadd, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [int, int], int,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

iaload

iaload 指令是类型安全的，当且仅当可以有效地将传入操作数栈上匹配 int 和 int 数组的类型替换为产生输出类型状态的 int。

```
instructionIsTypeSafe(iaload, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [int, arrayOf(int)], int,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

iand

iand 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(iand, iadd).
```

iastore

iastore 指令是类型安全的，当且仅当可以有效地从传入操作数栈中弹出匹配 int, int 和 int 数组的类型，从而生成传出类型状态。

```
instructionIsTypeSafe(iastore, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame)
    canPop(StackFrame, [int, int, arrayOf(int)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

iconst_<i>

iconst_m1 指令是类型安全的，如果可以有效地将类型 int 推送到传入操作数栈上，从而产生传出类型状态。

```
instructionIsTypeSafe(iconst_m1, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

iconst 的其他变体的规则是等效的。

```
instructionHasEquivalentTypeRule(iconst_0, iconst_m1).
instructionHasEquivalentTypeRule(iconst_1, iconst_m1).
instructionHasEquivalentTypeRule(iconst_2, iconst_m1).
instructionHasEquivalentTypeRule(iconst_3, iconst_m1).
instructionHasEquivalentTypeRule(iconst_4, iconst_m1).
instructionHasEquivalentTypeRule(iconst_5, iconst_m1).
```

idiv

idiv 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(idiv, iadd).
```

if_acmp<cond>

if_acmpeq 指令是类型安全的，当且仅当可以有效地弹出与传入操作数栈上的 reference 和 reference 匹配的类型，从而生成传出类型状态 NextStackFrame，并且指令的操作数 Target 是假设传入类型状态为 NextStackFrame 的有效分支目标。

```
instructionIsTypeSafe(if_acmpeq(Target), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     canPop(StackFrame, [reference, reference], NextStackFrame),
```

```
        targetIsTypeSafe(Environment, NextStackFrame, Target),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

if_acmpne 的规则是相同的。

```
instructionHasEquivalentTypeRule(if_acmpne(Target), if_acmpeq(Target)).
```

if_icmp<cond>

if_icmpeq 指令是类型安全的，当且仅当可以有效地在传入操作数栈上弹出匹配 int 和 int 的类型，从而生成传出类型状态 NextStackFrame，并且指令的操作数 Target 是假设传入类型状态为 NextStackFrame 的有效分支目标。

```
instructionIsTypeSafe(if_icmpeq(Target), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [int, int], NextStackFrame),
    targetIsTypeSafe(Environment, NextStackFrame, Target),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

if_icmp < cond > 指令的所有其他变体的规则是相同的。

```
instructionHasEquivalentTypeRule(if_icmpge(Target),if_icmpeq(Target)).
instructionHasEquivalentTypeRule(if_icmpgt(Target),if_icmpeq(Target)).
instructionHasEquivalentTypeRule(if_icmple(Target),if_icmpeq(Target)).
instructionHasEquivalentTypeRule(if_icmplt(Target),if_icmpeq(Target)).
instructionHasEquivalentTypeRule(if_icmpne(Target),if_icmpeq(Target)).
```

if<cond>

ifeq 指令是类型安全的，当且仅当可以有效地从传入操作数栈中弹出类型匹配 int，从而生成传出类型状态 NextStackFrame，并且指令的操作数 Target 是假设传入类型状态为 NextStackFrame 的有效分支目标。

```
instructionIsTypeSafe(ifeq(Target), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [int], NextStackFrame),
    targetIsTypeSafe(Environment, NextStackFrame, Target),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

if < cond > 指令的所有其他变体的规则是相同的。

```
instructionHasEquivalentTypeRule(ifge(Target), ifeq(Target)).
instructionHasEquivalentTypeRule(ifgt(Target), ifeq(Target)).
instructionHasEquivalentTypeRule(ifle(Target), ifeq(Target)).
instructionHasEquivalentTypeRule(iflt(Target), ifeq(Target)).
instructionHasEquivalentTypeRule(ifne(Target), ifeq(Target)).
```

ifnonnull, ifnull

ifnonnull 指令是类型安全的，当且仅当可以有效地从传入操作数栈中弹出类型匹配 reference，生成传出类型状态 NextStackFrame，并且指令的操作数 Target 是假设传入类型状态为 NextStackFrame 的有效分支目标。

```
instructionIsTypeSafe(ifnonnull(Target), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [reference], NextStackFrame),
    targetIsTypeSafe(Environment, NextStackFrame, Target),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ifnull 指令是类型安全的，当且仅当等效的 ifnonnull 指令是类型安全的。

```
instructionHasEquivalentTypeRule(ifnull(Target), ifnonnull(Target)).
```

iinc

带有第一个操作数 Index 的 iinc 指令是类型安全的，当且仅当 L_{Index} 的类型为 int。iinc 指令不会改变类型状态。

```
instructionIsTypeSafe(iinc(Index, _Value), _Environment, _Offset,
                      StackFrame, StackFrame, ExceptionStackFrame)
    StackFrame = frame(Locals, _OperandStack, _Flags),
    nth0(Index, Locals, int),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

iload, iload_<n>

带有操作数 Index 的 iload 指令是类型安全的，并产生出 NextStackFrame 类型状态，如果带有操作数 Index 和类型 int 的加载指令是类型安全的，并生成传出类型状态 NextStackFrame。

```
instructionIsTypeSafe(iload(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 iload_<n>, 对 0 < n < 3,是类型安全的，当且仅当等效的 iload 指令是类型安全的。

```
instructionHasEquivalentTypeRule(iload_0, iload(0)).
instructionHasEquivalentTypeRule(iload_1, iload(1)).
instructionHasEquivalentTypeRule(iload_2, iload(2)).
instructionHasEquivalentTypeRule(iload_3, iload(3)).
```

imul

imul 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(imul, iadd).
```

ineg *ineg*

ineg 指令是类型安全的，当且仅当在传入的操作数栈上有一个匹配 int 的类型。ineg 指令不改变类型状态。

```
instructionIsTypeSafe(ineg, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

instanceof

具有操作数 CP 的 instanceof 指令是类型安全的，当且仅当 CP 引用一个常量池条目，该条目表示类或数组，并且可以有效地将传入操作数栈上的 Object 类型替换为 int 类型，从而产生传出的类型状态。


```
instructionIsTypeSafe(instanceof(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    (CP = class(_, _) ; CP = arrayOf(_)),
    isBootstrapLoader(BL),
    validTypeTransition(Environment, [class('java/lang/Object', BL)], int,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

invokedynamic

invokedynamic 指令是类型安全的，当且仅当以下所有都为 true:

- 它的第一个操作数 CP 引用一个常量池条目，该条目表示一个动态调用站点，名称为 CallSiteName，描述符为 Descriptor。
- CallSiteName 不是<init>。
- CallSiteName 不是<clinit>。
- 可以有效地将传入操作数栈上的 Descriptor 中给出的参数类型替换为 Descriptor 中给出的返回类型，从而产生传出的类型状态。

```
instructionIsTypeSafe(invokedynamic(CP,0,0), Environment, _Offset,
                      StackFrame, NextStackFrame, ExceptionStackFrame)
    CP = dmethod(CallSiteName, Descriptor),
    CallSiteName \= '<init>',
    CallSiteName \= '<clinit>',
    parseMethodDescriptor(Descriptor, OperandArgList, ReturnType),
    reverse(OperandArgList, StackArgList),
    validTypeTransition(Environment, StackArgList, ReturnType,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

invokeinterface

invokeinterface 指令是类型安全的，当且仅当以下所有都为 true:

• 它的第一个操作数 CP 引用一个常量池条目，表示名为 MethodName 的接口方法，描述符 Descriptor 是接口 MethodIntfName 的成员。

• MethodName 不是<init>。

• MethodName 不是<clinit>。

• 它的第二个操作数 Count 是一个有效的计数操作数(见下文)。

• 可以有效地将与传入操作数栈上的 MethodIntfName 类型和在 Descriptor 中给出的参数类型相匹配的类型替换为在 Descriptor 中给出的返回类型，从而产生传出的类型状态。

```
instructionIsTypeSafe(invokeinterface(CP, Count, 0), Environment, _Offset,
                      StackFrame, NextStackFrame, ExceptionStackFrame) :-
    CP = imethod(MethodIntfName, MethodName, Descriptor),
    MethodName \= '<init>',
    MethodName \= '<clinit>',
    parseMethodDescriptor(Descriptor, OperandArgList, ReturnType),
    currentClassLoader(Environment, CurrentLoader),
    reverse([class(MethodIntfName, CurrentLoader) | OperandArgList],
            StackArgList),
    canPop(StackFrame, StackArgList, TempFrame),
    validTypeTransition(Environment, [], ReturnType,
                        TempFrame, NextStackFrame),
    countIsValid(Count, StackFrame, TempFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

如果 invokeinterface 指令的 Count 操作数等于该指令参数的大小，则该操作数有效。这等于 InputFrame 和 OutputFrame 的大小之差。

```
countIsValid(Count, InputFrame, OutputFrame) :-
    InputFrame = frame(_Locals1, OperandStack1, _Flags1),
    OutputFrame = frame(_Locals2, OperandStack2, _Flags2),
    length(OperandStack1, Length1),
    length(OperandStack2, Length2),
```

```
    Count =:= Length1 - Length2.
```

invokespecial

invokespecial 指令是类型安全的，当且仅当以下所有都为 true:

• 它的第一个操作数 CP 引用一个常量池条目，该条目表示一个名为 MethodName 的方法，其描述符 Descriptor 是 MethodClassName 类的成员。

• 并且:

- MethodName 不是<init>。
- MethodName 不是<clinit>。
- 可以有效地将与当前类和传入操作数栈上的 Descriptor 中给出的参数类型相匹配的类型替换为 Descriptor 中给出的返回类型，从而产生传出的类型状态。
- 可以有效地将与 MethodclassName 类和传入操作数栈上的 Descriptor 中给出的参数类型匹配的类型替换为 Descriptor 中给出的返回类型。

```
instructionIsTypeSafe(invokespecial(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    CP = method(MethodClassName, MethodName, Descriptor),
    MethodName \= '<init>',
    MethodName \= '<clinit>',
    parseMethodDescriptor(Descriptor, OperandArgList, ReturnType),
    thisClass(Environment, class(CurrentClassName, CurrentLoader)),

    isAssignable(class(CurrentClassName, CurrentLoader),
                 class(MethodClassName, CurrentLoader)),
    reverse([class(CurrentClassName, CurrentLoader) | OperandArgList],
            StackArgList),
    validTypeTransition(Environment, StackArgList, ReturnType,
                        StackFrame, NextStackFrame),
    reverse([class(MethodClassName, CurrentLoader) | OperandArgList],
            StackArgList2),
    validTypeTransition(Environment, StackArgList2, ReturnType,
                        StackFrame, _ResultStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

isAssignable 子句强制结构约束，除了实例初始化方法之外，invokespecial 必须在当前类/接口或超类/超接口中命名方法。

第一个 validTypeTransition 子句强制结构约束，除了实例初始化方法之外，invokespecial 的目标是当前类或更深层的接收方对象，而不是实例初始化方法。要了解原因，请考虑 StackArgList 模拟方法所期望的操作数栈上的类型列表，从当前类(执行 invokespecial 的类)开始。操作数栈的实际类型在 StackFrame 中。validTypeTransition 的效果是从 StackFrame 的操作数栈中弹出第一个类型，并检查它是否是 StackArgList 的第一项的子类型，即当前类。因此，实际的接收器类型与当前的类是兼容的。

眼尖的读者可能会注意到，强制执行这个结构约束取代了与受保护方法的 invokespecial 相关的结构约束。因此，上面的 Prolog 代码没有引用 passesProtectedCheck(§4.10.1.8)，而 Prolog 代码用于实例初始化方法的 invokespecial 时，使用 passesProtectedCheck 来确保在命名特定的受保护实例初始化方法时，实

际的接收器类型与当前类兼容。

第二个 validTypeTransition 子句强制一个结构约束，即任何方法调用指令都必须以一个接收方对象为目标，该对象的类型与指令命名的类型兼容。要了解原因，请考虑 StackArgList2 模拟方法所期望的操作数栈上的类型列表，从指令命名的类型开始。同样，操作数栈上的实际类型在 StackFrame 中，validTypeTransition 的作用是检查 StackFrame 中实际的接收器类型是否与 StackArgList2 中指令命名的类型兼容。

- 或者:

- MethodName 是<init>。

- Descriptor 指定一个 void 返回类型。

- 可以有效地从传入操作数栈中弹出与 Descriptor 中给定的参数类型和未初始化类型 UninitializedArg 匹配的类型，从而生成 OperandStack。

- 输出的类型状态是从传入的类型状态派生出来的，首先用 OperandStack 替换传入的操作数栈，然后用正在初始化的实例类型替换 UninitializedArg 的所有实例。

- 如果该指令在先前的 new 指令创建的类实例上调用实例初始化方法，并且该方法是受保护的，则使用符合管理访问受保护成员的特殊规则（§4.10.1.8）。

```
instructionIsTypeSafe(invokespecial(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     CP = method(MethodClassName, '<init>', Descriptor),
     parseMethodDescriptor(Descriptor, OperandArgList, void),
     reverse(OperandArgList, StackArgList),
     canPop(StackFrame, StackArgList, TempFrame),
     TempFrame = frame(Locals, [uninitializedThis | OperandStack], Flags),
     currentClassLoader(Environment, CurrentLoader),
     rewrittenUninitializedType(uninitializedThis, Environment,
                                class(MethodClassName, CurrentLoader), This),
     rewrittenInitializationFlags(uninitializedThis, Flags, NextFlags),
     substitute(uninitializedThis, This, OperandStack, NextOperandStack),
     substitute(uninitializedThis, This, Locals, NextLocals),
     NextStackFrame = frame(NextLocals, NextOperandStack, NextFlags),
     ExceptionStackFrame = frame(Locals, [], Flags).


instructionIsTypeSafe(invokespecial(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     CP = method(MethodClassName, '<init>', Descriptor),
     parseMethodDescriptor(Descriptor, OperandArgList, void),
     reverse(OperandArgList, StackArgList),
     canPop(StackFrame, StackArgList, TempFrame),
     TempFrame = frame(Locals, [uninitialized(Address) | OperandStack], Flags),
     currentClassLoader(Environment, CurrentLoader),
     rewrittenUninitializedType(uninitialized(Address), Environment,
                                class(MethodClassName, CurrentLoader), This),
     rewrittenInitializationFlags(uninitialized(Address), Flags, NextFlags),
     substitute(uninitialized(Address), This, OperandStack, NextOperandStack),
     substitute(uninitialized(Address), This, Locals, NextLocals),
     NextStackFrame = frame(NextLocals, NextOperandStack, NextFlags),
     ExceptionStackFrame = frame(Locals, [], Flags),
     passesProtectedCheck(Environment, MethodClassName, '<init>',
                          Descriptor, NextStackFrame).
```

要计算需要重写未初始化参数的类型，有两种情况：

- 如果我们在构造函数中初始化一个对象，它的类型最初是 uninitializedThis。此类型将重写为 < init > 方法的类的类型。

- 第二种情况产生于初始化由 new 创建的对象的初始化。未初始化的参数类型被重写为 MethodClass，这是<init>的方法持有者的类型。我们检查 Address 是否真的有一条 new 指令。

```
rewrittenUninitializedType(uninitializedThis, Environment,
                           MethodClass, MethodClass) :-
    MethodClass = class(MethodClassName, CurrentLoader),
    thisClass(Environment, MethodClass).

rewrittenUninitializedType(uninitializedThis, Environment,
                           MethodClass, MethodClass) :-
    MethodClass = class(MethodClassName, CurrentLoader),
    thisClass(Environment, class(thisClassName, thisLoader)),
    superclassChain(thisClassName, thisLoader, [MethodClass | Rest]).

rewrittenUninitializedType(uninitialized(Address), Environment,
                           MethodClass, MethodClass) :-
    allInstructions(Environment, Instructions),
    member(instruction(Address, new(MethodClass)), Instructions).

rewrittenInitializationFlags(uninitializedThis, _Flags, []).
rewrittenInitializationFlags(uninitialized(_), Flags, Flags).

substitute(_Old, _New, [], []).
substitute(Old, New, [Old | FromRest], [New | ToRest]) :-
    substitute(Old, New, FromRest, ToRest).
substitute(Old, New, [From1 | FromRest], [From1 | ToRest]) :-
    From1 \= Old,
    substitute(Old, New, FromRest, ToRest).
```


<init>方法的 invokespecial 的规则是返回一个明显异常栈帧的唯一动机。问题在于，当在构造函数中初始化对象时，invokspecial 可能会导致超类<init>方法被调用，并且调用可能会失败，从而使 this 未初始化。这种情况不能使用 Java 编程语言中的源代码来创建，但可以通过直接使用字节码编程来创建。

在这种情况下，原始帧在局部变量 0 中保存着一个未初始化的对象，并带有标志 flagThisUninit。invokspecial 的正常终止会初始化未初始化的对象，并关闭 flagThisUninit 标志。但是，如果<init>方法的调用引发异常，未初始化的对象可能会处于部分初始化状态，需要使其永久不可用。它由一个异常帧表示，该异常帧包含被破坏的对象(local 的新值)和 flagThisUninit 标志(旧标志)。没有办法从一个明显初始化的带有 flagThisUninit 标志的对象获得一个正确初始化的对象，因此该对象是永久不可用的。

如果不是这种情况，异常栈帧的标志将始终与输入栈帧的标志相同。

invokestatic

invokestatic 指令是类型安全的，当且仅当以下所有都为 true:

• 它的第一个操作数 CP 引用一个常量池条目，表示一个名为 MethodName 的方法，该方法带有描述符 Descriptor。

• MethodName 不是<init>。

• MethodName 不是<clinit>。

• 可以有效地将传入操作数栈上的 Descriptor 中给出的参数类型替换为 Descriptor 中给出的返回类型，从而产生传出的类型状态。

```
instructionIsTypeSafe(invokestatic(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     CP = method(_MethodClassName, MethodName, Descriptor),
     MethodName \= '<init>',
     MethodName \= '<clinit>',
     parseMethodDescriptor(Descriptor, OperandArgList, ReturnType),
     reverse(OperandArgList, StackArgList),
     validTypeTransition(Environment, StackArgList, ReturnType,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

invokevirtual

invokevirtual 指令是类型安全的，当且仅当以下所有都为 true:

• 它的第一个操作数 CP 引用一个常量池条目，该条目表示一个名为 MethodName 的方法，其描述符 Descriptor 是 MethodClassName 类的成员。

• MethodName 不是<init>。

• MethodName 不是<clinit>。

• 可以有效地将与传入操作数栈上的 MethodClassName 类和在 Descriptor 中给出的参数类型匹配的类型替换为在 Descriptor 中给出的返回类型，从而产生传出的类型状态。

• 如果该方法是受保护的，则使用符合管理访问受保护成员的特殊规则（§4.10.1.8）。

```
instructionIsTypeSafe(invokevirtual(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    CP = method(MethodClassName, MethodName, Descriptor),
    MethodName \= '<init>',
    MethodName \= '<clinit>',
    parseMethodDescriptor(Descriptor, OperandArgList, ReturnType),
    reverse(OperandArgList, ArgList),
    currentClassLoader(Environment, CurrentLoader),
    reverse([class(MethodClassName, CurrentLoader) | OperandArgList],
            StackArgList),
    validTypeTransition(Environment, StackArgList, ReturnType,
            StackFrame, NextStackFrame),
```

```
    canPop(StackFrame, ArgList, PoppedFrame),
    passesProtectedCheck(Environment, MethodClassName, MethodName,
            Descriptor, PoppedFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ior, irem

ior 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(ior, iadd).
```

irem 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(irem, iadd).
```

ireturn

如果封闭方法声明的返回类型为 int，并且可以从传入的操作数栈中有效地弹出与 int 匹配的类型，那么 ireturn 指令就是类型安全的。

```
instructionIsTypeSafe(ireturn, Environment, _Offset, StackFrame,
                      afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, int),
    canPop(StackFrame, [int], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```


ishl, ishr, iushr

ishl 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的

```
instructionHasEquivalentTypeRule(ishl, iadd).
```

ishr 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的

```
instructionHasEquivalentTypeRule(ishr, iadd).
```

iushr 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(iushr, iadd).
```

istore, istore_<n>

具有操作数 Index 的 istore 指令是类型安全的，并生成传出类型状态 NextStackFrame，如果具有操作数 Index 和类型 int 的存储指令是类型安全的，并产生传出类型状态 NextStackFrame。

```
instructionIsTypeSafe(istore(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, int, StackFrame, NextStackFrame),
```

```
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 istore_<n>,对 0 < n < 3,是类型安全的，当且仅当等效的 istore 指令是类型安全的。

```
instructionHasEquivalentTypeRule(istore_0, istore(0)).
instructionHasEquivalentTypeRule(istore_1, istore(1)).
instructionHasEquivalentTypeRule(istore_2, istore(2)).
instructionHasEquivalentTypeRule(istore_3, istore(3)).
```

isub, ixor

isub 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(isub, iadd).
```

ixor 指令是类型安全的，当且仅当等效的 iadd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(ixor, iadd).
```

l2d, l2f, l2i

如果可以有效地从传入操作数栈中弹出 long，并将其替换为 double，从而产生传出的类型状态，那么 l2d 指令就是类型安全的。

```
instructionIsTypeSafe(l2d, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [long], double,
                         StackFrame,NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

如果可以有效地从传入的操作数栈中弹出 long，并将其替换为 float，从而产生传出的类型状态，那么 l2f 指令就是类型安全的。

```
instructionIsTypeSafe(l2f, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [long], float,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

如果可以有效地从传入的操作数栈中弹出 long，并将其替换为 int，从而产生传出的类型状态，那么 l2i 指令就是类型安全的。

```
instructionIsTypeSafe(l2i, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [long], int,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ladd

ladd 指令是类型安全的，当且仅当可以有效地将传入操作数栈上匹配 long 和 long 的类型

替换为生成传出类型状态的 long。

```
instructionIsTypeSafe(ladd, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [long, long], long,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

laload

laload 指令是类型安全的，当且仅当可以有效地将传入操作数栈上匹配 int 和 long 数组的类型替换为生成传出类型状态的 long。

```
instructionIsTypeSafe(laload, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     validTypeTransition(Environment, [int, arrayOf(long)], long,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

land

land 指令是类型安全的，当且仅当等效的 ladd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(land, ladd).
```


lastore

lastore 指令是类型安全的，当且仅当可以有效地从传入操作数栈中弹出匹配 long,int 和 long 数组的类型，从而生成传出类型状态。

```
instructionIsTypeSafe(lastore, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame)
    canPop(StackFrame, [long, int, arrayOf(long)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lcmp

lcmp 指令是类型安全的，当且仅当可以有效地将传入操作数栈上与 long 和 long 匹配的类型替换为 int，从而生成传出类型状态。

```
instructionIsTypeSafe(lcmp, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame)
    validTypeTransition(Environment, [long, long], int,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lconst_<l>

lconst_0 指令是类型安全的，如果可以有效地将类型 long 推送到传入操作数栈上，从而产生传出类型状态。

```
instructionIsTypeSafe(lconst_0, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame)
    validTypeTransition(Environment, [], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lconst_1 指令是类型安全的，当且仅当等效的 lconst_0 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lconst_1, lconst_0).
```

ldc, ldc_w, ldc2_w

操作数为 CP 的 ldc 指令是类型安全的，当且仅当 CP 引用一个常量池条目，该条目表示类型为 Type 的实体，其中 Type 是可加载的(§4.4)，但不是 long 或 double，并且可以有效地将 Type 推入传入的操作数栈，从而产生传出的类型状态。

```
instructionIsTypeSafe(ldc(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     loadableConstant(CP, Type),
     Type \= long,
     Type \= double,
     validTypeTransition(Environment, [], Type, StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).


loadableConstant(CP, Type) :-
     member([CP, Type], [
        [int(_),int],
        [float(_), float],
        [long(_),  long],
        [double(_), double]
     ]).


loadableConstant(CP, Type) :-
     isBootstrapLoader(BL),
     member([CP, Type], [
        [class(_),class('java/lang/Class', BL)],
        [string(_),class('java/lang/String', BL)],
        [methodHandle(_,_), class('java/lang/invoke/MethodHandle', BL)],
        [methodType(_,_), class('java/lang/invoke/MethodType', BL)]
     ]).


loadableConstant(CP, Type) :-
    CP = dconstant(_, FieldDescriptor),
```

```
    parseFieldDescriptor(FieldDescriptor, Type).
```

ldc_w 指令是类型安全的，当且仅当等效的 ldc 指令是类型安全的。

```
instructionHasEquivalentTypeRule(ldc_w(CP), ldc(CP))
```

操作数为 CP 的 ldc2_w 指令是类型安全的，当且仅当 CP 引用一个常量池条目，该条目表示类型为 Type 的实体，其中 Type 为 long 或 double，可以有效地将 Type 推入传入操作数栈，从而产生传出的类型状态。

```
instructionIsTypeSafe(ldc2_w(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    loadableConstant(CP, Type),
    (Type = long ; Type = double),
    validTypeTransition(Environment, [], Type, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ldiv

ldiv 指令是类型安全的，当且仅当等效的 ladd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(ldiv, ladd).
```

lload, lload_<n>

具有操作数 Index 的 lload 指令是类型安全的，并生成传出类型状态 NextStackFrame，如果具有操作数 Index 和类型 long 的加载指令为类型安全的并生成传出的类型状态 NextStackFrame。

```
instructionIsTypeSafe(lload(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
  loadIsTypeSafe(Environment, Index, long, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 lload_<n>, 对 0 < n < 3,是类型安全的，当且仅当等效的 lload 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lload_0, lload(0)).
instructionHasEquivalentTypeRule(lload_1, lload(1)).
instructionHasEquivalentTypeRule(lload_2, lload(2)).
instructionHasEquivalentTypeRule(lload_3, lload(3)).
```

lmul

lmul 指令是类型安全的，当且仅当等效的 ladd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lmul, ladd).
```

lneg

lneg 指令是类型安全的，当且仅当传入操作数栈上存在类型匹配 long。lneg 指令不改变类型状态。

```
instructionIsTypeSafe(lneg, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame)
    validTypeTransition(Environment, [long], long,
                      StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lookupswitch

lookupswitch 指令是类型安全的，如果它的键是排过序的，可以有效地从传入的操作数栈中弹出 int，产生一个新的类型状态 BranchStackFrame，并且所有指令的目标都是有效的分支目标，假设 BranchStackFrame 是它们的传入类型状态。

```
instructionIsTypeSafe(lookupswitch(Targets, Keys), Environment, _, StackFrame,
                      afterGoto, ExceptionStackFrame) :-
  sort(Keys, Keys),
  canPop(StackFrame, [int], BranchStackFrame),
  checklist(targetIsTypeSafe(Environment, BranchStackFrame), Targets),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```


lor, lrem

lor 指令是类型安全的，当且仅当等效的 ladd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lor, ladd).
```

lrem 指令是类型安全的，当且仅当等效的 ladd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lrem, ladd).
```

lreturn

如果封闭方法声明的返回类型为 long，那么 lreturn 指令是类型安全的，并且可以从传入的操作数栈中有效地弹出匹配 long 的类型。

```
instructionIsTypeSafe(lreturn, Environment, _Offset, StackFrame,
                      afterGoto, ExceptionStackFrame) :-
  thisMethodReturnType(Environment, long),
  canPop(StackFrame, [long], _PoppedStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lshl, lshr, lushr

如果可以有效地将传入操作数栈上的 int 和 long 类型替换为产生传出类型状态的 long 类型，那么 lshl 指令就是类型安全的。

```
instructionIsTypeSafe(lshl, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
  validTypeTransition(Environment, [int, long], long,
                      StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lshr 指令是类型安全的，当且仅当等效的 lshl 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lshr, lshl).
```

lushr 指令是类型安全的，当且仅当等效的 lshl 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lushr, lshl).
```

lstore, lstore_<n>

具有操作数 Index 的 lstore 指令是类型安全的，并生成传出类型状态 NextStackFrame，如果具有操作数 Index 和类型 long 的存储指令为类型安全的并生成传出型状态 NextStackFrame。

```
instructionIsTypeSafe(lstore(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

指令 lstore_<n>,对 0 < n < 3,是类型安全的，当且仅当等效的 lstore 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lstore_0, lstore(0)).
instructionHasEquivalentTypeRule(lstore_1, lstore(1)).
instructionHasEquivalentTypeRule(lstore_2, lstore(2)).
instructionHasEquivalentTypeRule(lstore_3, lstore(3)).
```

lsub, lxor

lsub 指令是类型安全的，当且仅当等效的 ladd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lsub, ladd).
```

lxor 指令是类型安全的，当且仅当等效的 ladd 指令是类型安全的。

```
instructionHasEquivalentTypeRule(lxor, ladd).
```

monitorenter, monitorexit

monitorenter 指令是类型安全的，当且仅当可以有效地从传入操作数栈弹出一个类型匹配 reference，从而产生传出的类型状态。

```
instructionIsTypeSafe(monitorenter, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [reference], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

monitorexit 指令是类型安全的，当且仅当等效的 monitorenter 指令是类型安全的。

```
instructionHasEquivalentTypeRule(monitorexit, monitorenter).
```

multianewarray

具有操作数 CP 和 Dim 的 multianewarray 指令是类型安全的，当且仅当 CP 引用一个常量池条目，该条目表示维数大于或等于 Dim 的数组类型，Dim 严格为正，可以有效地将传入操作数栈上的 Dim int 类型替换为 CP 表示的类型，从而生成传出类型状态。

```
instructionIsTypeSafe(multianewarray(CP, Dim), Environment, _Offset,
                      StackFrame, NextStackFrame, ExceptionStackFrame) :-
     CP = arrayOf(_),
     classDimension(CP, Dimension),
     Dimension >= Dim,
     Dim > 0,
     /* Make a list of Dim ints */
     findall(int, between(1, Dim, _), IntList),
     validTypeTransition(Environment, IntList, CP,
                         StackFrame, NextStackFrame),
     exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

组件类型也是数组类型的数组类型的维度比其组件类型的维度大一。

```
classDimension(arrayOf(X), Dimension) :-
   classDimension(X, Dimension1),
   Dimension is Dimension1 + 1.

classDimension(_, Dimension) :-
   Dimension = 0.
```

new

操作数 CP 位于偏移量 Offset 的 new 指令是类型安全的，当且仅当 CP 引用表示类或接口类型的常量池条目，类型 uninitialized(Offset)不会出现在传入操作数栈中，可以有效地将 uninitialized(Offset)推送到传入操作数栈中，并在传入局部变量中用 top 替换 uninitialized(Offset)，从而生成传出类型状态。

```
instructionIsTypeSafe(new(CP), Environment, Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
     StackFrame = frame(Locals, OperandStack, Flags),
     CP = class(_, _),
     NewItem = uninitialized(Offset),
```

```
        notMember(NewItem, OperandStack),
        substitute(NewItem, top, Locals, NewLocals),
        validTypeTransition(Environment, [], NewItem,
                            frame(NewLocals, OperandStack, Flags),
                            NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

substitute 谓词在 invokespecial (§invokespecial)规则中定义。

newarray

带有操作数 TypeCode 的 newarray 指令是类型安全的，当且仅当 TypeCode 对应于原生类型 ElementType，可以有效地将传入操作数栈上的类型 int 替换为类型“ElementType 的数组”，从而生成传出类型状态。

```
    instructionIsTypeSafe(newarray(TypeCode), Environment, _Offset, StackFrame,
                          NextStackFrame, ExceptionStackFrame) :-
        primitiveArrayInfo(TypeCode, _TypeChar, ElementType, _VerifierType),
        validTypeTransition(Environment, [int], arrayOf(ElementType),
                            StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

类型代码和原生类型之间的对应关系由以下谓词指定：

```
primitiveArrayInfo(4,  0'Z, boolean, int).
primitiveArrayInfo(5,  0'C, char,    int).
primitiveArrayInfo(6,  0'F, float,   float).
primitiveArrayInfo(7,  0'D, double,  double)
primitiveArrayInfo(8,  0'B, byte,    int).
primitiveArrayInfo(9,  0'S, short,   int).
primitiveArrayInfo(10, 0'I, int,     int).
primitiveArrayInfo(11, 0'J, long,    long).
```

nop

nop 指令始终是类型安全的。nop 指令不影响类型状态。

```
    instructionIsTypeSafe(nop, _Environment, _Offset, StackFrame,
                          StackFrame, ExceptionStackFrame) :-
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

pop, pop2

pop 指令是类型安全的，当且仅当可以有效地从传入操作数栈中弹出类别 1 类型，从而生成传出类型状态。

```
    instructionIsTypeSafe(pop, _Environment, _Offset, StackFrame,
```

```
                        NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, [Type | Rest], Flags),
    popCategory1([Type | Rest], Type, Rest),
    NextStackFrame = frame(Locals, Rest, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

pop2 指令是类型安全的，当且仅当它是 pop2 指令的类型安全形式。

```
instructionIsTypeSafe(pop2, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    pop2SomeFormIsTypeSafe(InputOperandStack, OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

pop2 指令是 pop2 指令的类型安全形式，当且仅当它是类型安全形式 1 pop2 指令或类型安全形式 2 pop2 指令。

```
pop2SomeFormIsTypeSafe(InputOperandStack, OutputOperandStack) :-
    pop2Form1IsTypeSafe(InputOperandStack, OutputOperandStack).

pop2SomeFormIsTypeSafe(InputOperandStack, OutputOperandStack) :-
    pop2Form2IsTypeSafe(InputOperandStack, OutputOperandStack).
```

pop2 指令是类型安全形式 1 pop2 指令，当且仅当可以有效地从传入操作数栈中弹出两种大小为 1 的类型，从而生成传出类型状态。

```
pop2Form1IsTypeSafe([Type1, Type2 | Rest], Rest) :-
    popCategory1([Type1 | Rest], Type1, Rest),
    popCategory1([Type2 | Rest], Type2, Rest).
```

pop2 指令是类型安全形式 2 pop2 指令，当且仅当可以有效地从传入操作数栈中弹出大小为 2 的类型，从而生成传出类型状态。

```
pop2Form2IsTypeSafe([top, Type | Rest], Rest) :-
    popCategory2([top, Type | Rest], Type, Rest).
```

putfield

具有操作数 CP 的 putfield 指令是类型安全的，当且仅当以下所有都为 true:

- 它的第一个操作数 CP 引用一个常量池条目，该条目表示一个字段，该字段的声明类型为 FieldType，在类 FieldClassName 中声明。FieldClassName 不能是数组类型。
- 并且:
  - 可以有效地从传入操作数栈中弹出与 FieldType 和 FieldClassName 匹配的类型，从而生成传出类型状态。
  - 受保护字段需接受额外检查（§4.10.1.8）。

```
instructionIsTypeSafe(putfieid(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    CP = fieid(FieidCiassName, FieidName, FieidDescriptor),
```

```
    parseFieidDescriptor(FieidDescriptor, FieidType),
    canPop(StackFrame, [FieidType], PoppedFrame),
    passesProtectedCheck(Environment, FieidCiassName, FieidName,
    FieidDescriptor, PoppedFrame),
    currentCiassLoader(Environment, CurrentLoader),
    canPop(StackFrame, [FieidType, ciass(FieidCiassName, CurrentLoader)],
           NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

- 或者:
  - 如果指令发生在类 FieldClassName 的实例初始化方法中，则可以有效地将匹配 FieldType 和 uninitiaiizedThis 类型从传入操作数栈中弹出，从而生成传出类型状态。这允许在完成 this 的初始化之前赋值在当前类中声明的 this 的实例字段。

```
instructionIsTypeSafe(putfieid(CP), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    CP = fieid(FieidCiassName, _FieidName, FieidDescriptor),
    parseFieidDescriptor(FieidDescriptor, FieidType),
    Environment = environment(CurrentCiass, CurrentMethod, _, _, _, _),
    CurrentCiass = ciass(FieidCiassName, _),
    isInit(CurrentMethod),
    canPop(StackFrame, [FieidType, uninitiaiizedThis], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

putstatic

操作数为 CP 的 putstatic 指令是类型安全的，当且仅当 CP 引用一个常量池条目，它表示声明类型为 FieldType 的字段，可以从传入操作数栈中有效地弹出与 FieldType 匹配的类型，从而产生传出的类型状态。

```
instructionIsTypeSafe(putstatic(CP), _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    CP = field(_FieldClassName, _FieldName, FieldDescriptor),
    parseFieldDescriptor(FieldDescriptor, FieldType), canPop(StackFrame, [FieldType],
    NextStackFrame), exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

return

如果封闭方法声明了一个 void 返回类型，return 指令是类型安全的，并且:

- 封闭方法不是一个 <init>方法，或者
- this 在指令发生的地方已经完全初始化了。

```
instructionIsTypeSafe(return, Environment, _Offset, StackFrame,
                      afterGoto, ExceptionStackFrame) :-
  thisMethodReturnType(Environment, void),
```

```
    StackFrame = frame(_Locals, _OperandStack, Flags),
    notMember(flagThisUninit, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

saload

saload 指令是类型安全的，当且仅当可以有效地将输入操作数栈上匹配 int 和 short 数组的类型替换为 int，从而生成输出类型状态。

```
instructionIsTypeSafe(saload, Environment, _Offset, StackFrame,
                      NextStackFrame,ExceptionStackFrame) :-
  validTypeTransition(Environment, [int, arrayOf(short)], int,
                      StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

sastore

sastore 指令是类型安全的，当且仅当可以有效地从传入操作数栈中弹出匹配 int, int 和 short 数组的类型，从而生成传出类型状态。

```
instructionIsTypeSafe(sastore, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame)
    canPop(StackFrame, [int, int, arrayOf(short)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```


sipush

sipush 指令是类型安全的，当且仅当可以有效地将类型 int 推送到传入操作数栈上，从而生成传出类型状态。

```
instructionIsTypeSafe(sipush(_Value), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

swap

swap 指令是类型安全的，当且仅当可以有效地用类型 Type2 和 Type1 替换传入操作数栈上的两个类别 1 类型 Type1 和 Type2，从而生成传出类型状态。

```
instructionIsTypeSafe(swap, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
  StackFrame = frame(_Locals, [Type1, Type2 | Rest], _Flags),
  popCategory1([Type1 | Rest], Type1, Rest),
```

```
    popCategory1([Type2 | Rest], Type2, Rest),
    NextStackFrame = frame(_Locals, [Type2, Type1 | Rest], _Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

tableswitch

如果 tableswitch 指令的键是有序的，那么它是类型安全的，可以从传入的操作数栈中有效地弹出 int，产生一个新的类型状态 BranchStackFrame，并且所有指令的目标都是有效的分支目标，假设 BranchStackFrame 是它们的传入类型状态。

```
instructionIsTypeSafe(tableswitch(Targets, Keys), Environment, _Offset,
                      StackFrame, afterGoto, ExceptionStackFrame) :-
  sort(Keys, Keys),
  canPop(StackFrame, [int], BranchStackFrame),
  checklist(targetIsTypeSafe(Environment, BranchStackFrame), Targets),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

wide

wide 指令与它们加宽的指令遵循相同的规则。

```
instructionHasEquivalentTypeRule(wide(WidenedInstruction),
                                 WidenedInstruction).
```

4.10.2 通过类型推断验证

不包含 StackMapTable 属性(必须具有 49.0 或更低的版本号)的 class 文件必须使用类型推断进行验证。

4.10.2.1 类型推断验证的过程

在链接过程中，验证器通过对每个方法执行数据流分析来检查 class 文件中每个方法的 Code 属性的 code 数组。验证器确保在程序中的任何给定点，无论采用何种代码路径到达该点，以下所有项都为 true:

- 操作数栈总是具有相同的大小并包含相同类型的值。
- 除非已知局部变量包含适当类型的值，否则不会访问它。
- 方法是用适当的参数调用的。
- 仅使用适当类型的值对字段赋值。
- 所有操作码在操作数栈和局部变量数组中都有适当类型的参数。

出于效率的原因，在原则上可以由验证器执行的某些测试被延迟到方法的代码第一次实际调用时才执行。这样做，验证器避免加载 class 文件，除非它必须加载。

> 例如，如果一个方法调用了另一个返回类 A 实例的方法，并且该实例只赋值给相同类型的字段，验证器就不会费心检查类 A 是否实际存在。但是，如果它被分配给类型为 B 的字段，A 和 B 的定义都必须加载进来，以确保 A 是 B 的子类。

4.10.2.2 字节码验证器

每个方法的代码都是独立验证的。首先，组成代码的字节被分解成一个指令序列，每个指令开始的 code 数组的索引被放置在一个数组中。然后验证器会第二次遍历代码并解析指令。在此传递过程中，将构建一个数据结构来保存方法中关于每个 Java 虚拟机指令的信息。检查每条指令的操作数(如果有的话)，以确保它们是有效的。例如:

- 分支必须在方法的 code 数组范围内。
- 所有控制流指令的目标都是指令的开始。在 wide 指令的情况下，wide 操作码被认为是指令的开始，而给出由该 wide 指令修改的操作的操作码不被认为是指令的开始。不允许在指令中间进行分支。
- 任何指令都不能访问或修改索引处的局部变量，该索引大于或等于其方法指示其分配的局部变量的数量。
- 对常量池的所有引用都必须指向适当类型的条目。(例如，指令 getfield 必须引用一个字段。)
- 代码不会在指令中间结束。
- 执行不能离开代码的末尾。

- 对于每个异常处理程序，受处理程序保护的代码的起始点和结束点必须在指令的开始处，或者在结束点的情况下，直接超过代码的结束点。起点必须在终点之前。异常处理程序代码必须从有效指令开始，而不能从被 wide 指令修改的操作码开始。

对于该方法的每条指令，验证器在执行该指令之前记录操作数栈的内容和局部变量数组的内容。对于操作数栈，它需要知道栈高度和其中每个值的类型。对于每个局部变量，它需要知道该局部变量的内容类型，或者该局部变量包含一个不可用的或未知的值(它可能没有初始化)。字节码验证器在确定操作数栈上的值类型时不需要区分整型(如 byte, short, char)。

接下来，初始化一个数据流分析器。对于方法的第一条指令，表示参数的局部变量最初包含由方法的类型描述符指定的类型值;操作数栈为空。所有其他局部变量都包含非法值。对于尚未检查的其他指令，没有关于操作数栈或局部变量的信息。

最后，运行数据流分析器。对于每条指令，一个“改变”位表示该指令是否需要查看。 最初，“更改”位只对第一条指令设置。数据流分析器执行以下循环:

1. 选择一条设置了“更改”位的 Java 虚拟机指令。如果没有设置“改变”位的指令存在，则该方法已成功验证。否则，关闭所选指令的“更改”位。
2. 通过以下操作来模拟该指令对操作数栈和局部变量数组的影响:
   - 如果指令使用来自操作数栈的值，请确保栈上有足够数量的值，并且栈的顶部值是适当的类型。否则，验证失败。
   - 如果指令使用局部变量，请确保指定的局部变量包含适当类型的值。否则，验证失败。
   - 如果指令将值压入操作数栈，请确保操作数栈上有足够的空间容纳新值。将指示的类型添加到模拟操作数栈的顶部。
   - 如果指令修改了局部变量，请记录该局部变量现在包含新类型。
3. 确定可遵循当前指令的指令。后续指令可以是以下指令之一:
   - 下一条指令，如果当前指令不是无条件控制转移指令（例如，goto,return 或 throw）。如果有可能“分开”该方法的最后一条指令，则验证失败。
   - 条件或无条件分支或切换的目标。
   - 此指令的任何异常处理程序。
4. 在当前指令执行结束时，将操作数栈和局部变量数组的状态合并到每个后续指令中，如下所示:
   - 如果这是第一次访问后继指令，记录在 step2 中计算的操作数栈和局部变量值是在执行后继指令之前操作数栈和局部变量数组的状态。为后继指令设置“更改”位。

- 如果后继指令之前已经看到过，那么将步骤 2 中计算的操作数栈和局部变量值合并到已经存在的值中。如果对值有任何修改，则设置“更改”位。

在将控制传递给异常处理程序的特殊情况下:

- 记录由异常处理程序指定的异常类型的单个对象是操作数栈在执行后续指令之前的状态。操作数栈上必须有足够的空间容纳这个值，就像一条指令推入了它一样。
- 记录第 2 步之前的局部变量值是执行后续指令之前的局部变量数组的状态。步骤 2 中计算的局部变量值是不相关的。

5. 继续步骤 1。

要合并两个操作数栈，每个栈上的值的数量必须相同。然后比较两个栈对应的值，计算合并后栈的值，如下所示:

- 如果一个值是原生类型，那么相应的值必须是相同的原生类型。合并后的值是原始类型。
- 如果一个值是非数组引用类型，那么相应的值必须是引用类型(数组或非数组)。合并的值是对两个引用类型的第一个公共超类型实例的引用。 (这样的引用类型总是存在，因为 Object 类型是所有类、接口和数组类型的超类型。)

  例如，Object 和 String 可以合并;结果是 Object。类似地，Object 和 String[]可以合并;结果仍然是 Object。甚至 Object 和 int[]也可以合并，String 和 int[]也可以合并;结果都是 Object。

- 如果对应的值都是数组引用类型，则检查它们的维数。如果数组类型具有相同的维度，则合并值是对数组类型实例的引用，该数组类型是两个数组类型的第一个公共超类型。(如果其中一种或两种数组类型都具有基本元素类型，则改用 Object 作为元素类型。) 如果数组类型具有不同的维数，则合并后的值是对维数较小的（那一方）数组类型实例的引用;如果较小的数组类型为 Cloneable 或 java.io.Serializable，则元素类型为 Cloneable 或 java.io.Serializable，否则为 Object。

  例如，Object []和 String []可以合并;结果为 Object[]。Cloneable[]和 String[]可以合并，或者 java.io.Serializable[]和 String[]可以合并;结果分别是 Cloneable[]和 java.io.Serializable[]。即使是 int[]和 String[]也可以合并;结果是 Object[]，因为在计算第一个公共超类型时使用 Object 而不是 int。

  由于数组类型可以有不同的维度，Object[]和 String[][]可以合并，或者 Object[][]和 String[]也可以合并;这两种情况的结果都是 Object[]。Cloneable[]和 String[][]可以合并;结果是 Cloneable[]。最后，可克隆[][]和 String[]可以合并;结果为 Object[]。最后，Cloneable[][]和 String[]可以合并;结果为 Object[]。

如果操作数栈不能合并，则方法验证失败。

为了合并两个局部变量数组状态，对对应的局部变量对进行比较。合并的局部变量的值是使用上面的规则计算的，除了对应的值允许是不同的原生类型。在这种情况下，验证器记录合并的局部变量包含一个不可用的值。

如果数据流分析器运行在一个方法上而没有报告验证失败，那么这个方法已经被 class 文件验证器成功验证了。

某些指令和数据类型使数据流分析器复杂化。现在，我们将更详细地研究其中的每一个。

4.10.2.3 long 和 double 类型的值

long 和 double 类型的值在验证过程中被特别处理。

当 long 或 double 类型的值移动到索引 n 处的局部变量时，索引 n+1 会被特别标记，表示它已被索引 n 处的值保留，不能用作局部变量索引。之前索引 n+1 处的任何值都变得不可用了。

当一个值移动到索引 n 处的局部变量时，将检查索引 n-1 是否为 long 类型或 double 类型值的索引。如果是这样，索引 n-1 处的局部变量将被更改，以表明它现在包含一个不可用的值。由于索引 n 处的局部变量已被重写，索引 n-1 处的局部变量不能表示 long 或 double 类型的值。

在操作数栈上处理 long 或 double 类型的值更简单;验证器将它们视为栈上的单个值。例如，dadd 操作码(添加两个 double 值)的验证代码检查栈上最上面的两个 double 类型的项。当计算操作数栈长度时，long 和 double 类型的值的长度为 2。

操作操作数栈的非类型化指令必须将 long 和 double 类型的值视为原子(不可分割)。例如，如果栈上的最上面的值是一个 double，并且遇到了像 pop 或 dup 这样的指令，验证器就会报告失败。必须使用 pop2 或 dup2 指令来代替。

4.10.2.4 实例初始化方法和新创建的对象

创建一个新的类实例是一个多步骤的过程。语句:

```
new myClass(i, j, k);
...
```

可以通过以下方法实现:

```
new #1            // Allocate uninitialized space for myClass
dup               // Duplicate object on the operand stack
iload_1           // Push i
iload_2           // Push j
iload_3           // Push k
invokespecial #5  // Invoke myClass.<init>
...
```

该指令序列将新创建并初始化的对象留在操作数栈的顶部。(§3 (Java 虚拟机的编译)给出了 Java 虚拟机指令集的其他编译例子。

myClass 的实例初始化方法(§2.9.1)将未初始化的新对象视为局部变量 0 中的 this 参数。在该方法调用 myClass 的另一个实例初始化方法或它的 this 的直接超类之前，该方法可以对 this 执行的唯一操作是赋值 myClass 中声明的字段。

当对实例方法进行数据流分析时，验证器初始化局部变量 0 以包含当前类的对象，或者，对于实例初始化方法，局部变量 0 包含一个特殊类型，指示一个未初始化的对象。在此对

象上调用适当的实例初始化方法(从当前类或其直接超类)后，操作数栈的验证器模型和局部变量数组中出现的所有这种特殊类型都将被当前类的类型替换。验证器拒绝在初始化新对象之前使用该对象或多次初始化该对象的代码。此外，它确保方法的每次正常返回都调用了该方法的类或直接超类中的实例初始化方法。

类似地，作为 Java 虚拟机指令 new 的结果，创建一个特殊类型并将其压入操作数栈的验证器模型。特殊类型指示创建类实例的指令和创建未初始化的类实例的类型。当在未初始化的类实例的类中声明的实例初始化方法在该类实例上调用时，所有出现的特殊类型都被类实例的预期类型替换。随着数据流分析的进行，这种类型的更改可能会传播到后续指令。

指令号需要作为特殊类型的一部分存储，因为操作数栈上可能同时存在一个类的多个尚未初始化的实例。例如，Java 虚拟机指令序列实现:

```
new InputStream(new Foo(), new InputStream("foo"))
```

可以同时在操作数栈上有两个未初始化的 InputStream 实例。当在类实例上调用实例初始化方法时，只有在操作数栈或局部变量数组中与类实例相同对象的特殊类型才会被替换。

4.10.2.5 异常和 finally

为了实现 try-finally 构造，生成版本号为 50.0 或更低版本的 class 文件的 Java 编程语言编译器可以使用异常处理工具和两个特殊指令:jsr(“跳转到子例程”)和 ret(“从子例程返回”)。finally 子句在 Java 虚拟机代码中被编译为其方法的子例程，非常类似于异常处理程序的代码。当调用子例程的 jsr 指令被执行时，它会将其返回地址(位于正在执行的 jsr 之后的指令地址)作为 returnAddress 类型的值推入操作数栈。子例程的代码将返回地址存储在一个局部变量中。在子例程的末尾，ret 指令从局部变量获取返回地址，并将控制转移到返回地址处的指令。

可以通过几种不同的方式将控制转移到 finally 子句(可以调用 finally 子例程)。如果 try 子句正常完成，则在计算下一个表达式之前，通过 jsr 指令调用 finally 子例程。在 try 子句内将控制转移到 try 子句外的 break 或 continue 首先对 finally 子句的代码执行 jsr。如果 try 子句执行一个 return，编译后的代码会执行以下操作:

1. 在局部变量中保存返回值(如果有的话)。
2. 对 finally 子句的代码执行 jsr。
3. 从 finally 子句返回时，返回保存在局部变量中的值。

编译器设置一个特殊的异常处理程序，它捕获 try 子句引发的任何异常。如果在 try 子句中抛出异常，这个异常处理程序会执行以下操作:

1. 将异常保存在局部变量中。
2. 对 finally 子句执行 jsr。

3. 从 finally 子句返回时，重新抛出异常。

关于 try-finally 结构实现的更多信息，参见§3.13。

finally 子句的代码给验证者带来了一个特殊的问题。通常情况下，如果一条指令可以通过多条路径到达，而某个局部变量在多条路径上包含不兼容的值，那么该局部变量就不可用了。但是，finally 子句可以从不同的地方调用，产生几种不同的情况:

- 来自异常处理程序的调用可能具有某个包含异常的局部变量。
- 实现 return 的调用可能有一些包含返回值的局部变量。
- 来自 try 子句底部的调用可能在同一个局部变量中具有不确定的值。

finally 子句本身的代码可能会通过验证，但是在完成对 ret 指令的所有后继程序的更新之后，验证器会注意到异常处理程序希望保存异常的局部变量，或者希望保存返回值的返回代码，现在包含一个不确定的值。

验证包含 finally 子句的代码很复杂。其基本思想如下:

- 每条指令都跟踪到达该指令所需的 jsr 目标列表。对于大多数代码，此列表为空。对于 finally 子句的代码中的指令，其长度为 1。对于多重嵌套的 finally 代码（非常罕见！），它可能比 1 长。
- 对于每个指令和到达该指令所需的每个 jsr，将维护自执行 jsr 指令以来访问或修改的所有局部变量的位向量。
- 当执行 ret 指令时，该指令的实现从子程序返回，必须只有一个可能的子程序可以返回指令。两个不同的子例程不能将它们的执行“合并”到单个 ret 指令中。
- 要对 ret 指令执行数据流分析，需要使用特殊过程。由于验证器知道指令必须从哪个子例程返回，它可以找到所有调用子例程的 jsr 指令，并将 ret 指令时操作数栈和局部变量数组的状态合并到 jsr 之后的操作数栈和局部变量数组中。
  - 对于位向量(上面构造的)表示已被子例程访问或修改的任何局部变量，使用 ret 时的局部变量类型。
  - 对于其他局部变量，在 jsr 指令之前使用局部变量的类型。

4.11 Java 虚拟机的限制

Java 虚拟机的以下限制隐含在 class 文件格式中:

- 每个类或每个接口的常量池被 ClassFile 结构的 16 位 constant_pool_count 字段限制为 65535 项(§4.1)。这对单个类或接口的总体复杂性起到内部限制的作用。
- 类或接口可以声明的字段数量被 ClassFile 结构中 fields_count 项的大小限制为 65535(§4.1)。

  注意，ClassFile 结构的 fields_count 项的值不包括从超类或超接口继承的字段。

- 类或接口可以声明的方法的数量被 ClassFile 结构中 methods_count 项的大小限制为 65535(§4.1)。

  注意，ClassFile 结构的 methods_count 项的值不包括从超类或超接口继承的方法。

- 类或接口的直接超接口的数量被 ClassFile 结构中 interface_count 项的大小限制为 65535(§4.1)。

- 在调用方法时(§2.6)创建的帧的局部变量数组中，局部变量的最大数量被限制在 65535 以内，这取决于给出方法代码的 Code 属性(§4.7.3)的 max_locals 项的大小，以及 Java 虚拟机指令集的 16 位局部变量索引。

  注意，long 和 double 类型的值都被认为保留了两个局部变量，并为 max_locals 值贡献了两个单元，因此使用这些类型的局部变量进一步减少了这一限制。

- 一个帧中操作数栈的大小(§2.6)被 Code 属性的 max_stack 字段(§4.7.3)限制为 65535 个值。

  注意，long 和 double 类型的值都被认为为 max_stack 值贡献了两个单元，因此在操作数栈上使用这些类型的值进一步减少了这一限制。

- 方法描述符(§4.3.3)的定义将方法参数的数量限制在 255 个，在实例或接口方法调用的情况下，这个限制包括 this 的一个单元。

  注意，方法描述符是根据方法参数长度的概念定义的，其中 long 或 double 类型的参数为长度贡献两个单位，因此这些类型的参数进一步减少了限制。

- 字段和方法名、字段和方法描述符以及其他常量字符串值(包括那些被 ConstantValue(§4.7.2)属性引用的值)的长度被 CONSTANT_Utf8_info 结构(§4.4.7)的 16 位无符号 length 项限制为 65535 个字符。

  注意，限制是针对编码中的字节数，而不是编码的字符数。UTF-8 使用两个或三个字节对某些字符进行编码。因此，包含多字节字符的字符串被进一步限制。

- 数组中的维数限制为 255，这取决于 multianewarray 指令的维数操作码的大小以及对 multianewarray, anewarray 和 newarray 指令施加的限制（§4.9.1、§4.8.2）。