第十二章

执行

本章指定在程序执行期间发生的活动。它围绕 Java 虚拟机以及构成程序的类、接口和对象的生命周期进行组织。

Java 虚拟机通过加载指定的类或接口，然后调用该指定的类或接口中的 main 方法来启动。12.1 节概述了执行 main 所涉及的加载、链接和初始化步骤，作为对本章中概念的介绍。其他章节详细说明了加载(§12.2)、链接(§12.3)和初始化(§12.4)的细节。

本章继续说明了创建新类实例的过程(§12.5);以及类实例的终结(§12.6)。最后，它描述了类的卸载(§12.7)和程序退出后的过程(§12.8)。

12.1 Java 虚拟机启动

Java 虚拟机通过调用某个指定类或接口的 main 方法来开始执行，并向它传递一个单一的参数，该参数是一个字符串数组。在本规范的示例中，第一个类通常称为 Test。

Java 虚拟机启动的精确语义在 Java SE 19 版 Java 虚拟机规范的第 5 章中给出。在这里，我们从 Java 编程语言的角度来概述这个过程。

向 Java 虚拟机指定初始类或接口的方式超出了本规范的范围，但在使用命令行的主机环境中，将类或接口的完全限定名指定为命令行参数，并将以下命令行参数用作字符串，作为 main 方法的参数提供。

> 例如，在 UNIX 实现中，命令行:
>
> ```
> java Test reboot Bob Dot Enzo
> ```
>
> 通常会通过调用类 Test（未命名包中的类）的 main 方法来启动 Java 虚拟机，并将包含四个字符串“reboot”、“Bob”、“Dot”和“Enzo”的数组传递给它。

我们现在概述了 Java 虚拟机执行 Test 可能采取的步骤，作为加载、链接和初始化过程的示例，将在后面的章节中进一步描述。

12.1.1 加载类 `Test`

最初尝试执行 Test 类的方法 main 时，发现没有加载 Test 类--也就是说，Java 虚拟机当前不包含该类的二进制表示。然后，Java 虚拟机使用类加载器来尝试找到这样的二进制表示。

如果此过程失败，则会抛出错误。该加载过程将在§12.2 中进一步说明。

12.1.2 链接 `Test`: 验证, 准备, (可选的) 解析

加载 Test 后，必须先初始化它，然后才能调用 main。和所有类和接口一样，Test 在初始化之前必须链接。链接涉及验证、准备和（可选的）解析。链接在§12.3 中有进一步描述。

验证检查加载的 Test 表示形式是否正确，以及符号表是否正确。验证还检查实现 Test 的代码是否符合 Java 编程语言和 Java 虚拟机的语义要求。如果在验证过程中检测到问题，则抛出错误。验证将在§12.3.1 中进一步说明。

准备包括分配静态存储和 Java 虚拟机实现在内部使用的任何数据结构，如方法表。准备工作在第 12.3.2 节中有进一步说明。

解析是检查从 Test 到其他类和接口的符号引用的过程，方法是加载提到的其他类和接口，并检查引用是否正确。

在初始链接时，解析步骤是可选的。实现可以解析来自很早就被链接的类或接口的符号引用，甚至可以解析来自被进一步递归引用的类和接口的所有符号引用。 (此解析可能会导致这些进一步的加载和链接步骤产生错误。) 这种实现选择代表了一个极端，类似于多年来在简单的 C 语言实现中所做的那种“静态”链接。(在这些实现中，编译后的程序通常表示为包含该程序的完全链接版本的“a.out”文件，包括到该程序所使用的库例程的完全解析的链接。这些库例程的副本包含在“a.out”文件中。)

相反，实现可能只在符号引用被积极使用时才选择解析它；对所有符号引用一致地使用这一策略将代表“最懒惰”的解析形式。在这种情况下，如果 Test 有几个对另一个类的符号引用，则这些引用可能会在使用时一次解析一个，或者如果在程序执行期间从未使用过这些引用，则可能根本不解析。

关于何时执行解析的唯一要求是，在解析过程中检测到的任何错误都必须在程序中的某个点上被抛出，在该点上，程序可能直接或间接地需要链接到错误所涉及的类或接口。使用上面描述的“静态”示例实现选择，如果加载和链接错误涉及类 Test 中提到的类或接口或任何进一步递归引用的类和接口，则加载和链接错误可能在程序执行之前发生。在实现“最懒惰”解析的系统中，只有在积极地使用错误的符号引用时才会抛出这些错误。

解析过程将在§12.3.3 中进一步说明。

12.1.3 初始化 Test: 执行初始化器

在接下来的示例中，Java 虚拟机仍在尝试执行类 Test 的 main 方法。只有在类已初始化时才允许这样做（§12.4.1）。

初始化包括按文本顺序执行任何类变量初始化器和类 Test 的静态初始化器。但在初始化 Test 之前，必须递归地初始化它的直接超类，以及它的直接超级类的直接超类等。在最简单的情况下，Test 将 Object 作为其隐式直接超类；若 Object 类尚未初始化，则必须在初始化 Test 之前对其进行初始化。Object 类没有超类，因此递归在此终止。

如果类 Test 有另一个类 Super 作为其超类，则 Super 必须在 Test 之前。如果类 Test 有另一个类 Super 作为其超类，则必须在 Test 之前初始化 Super。这需要加载、验证和准备 Super（如果尚未完成），并且根据实现情况，还可能涉及递归地解析来自 Super 的符号引用等。

因此，初始化可能导致加载、链接和初始化错误，包括涉及其他类和接口的此类错误。

初始化过程在§12.4 中进一步描述。

12.1.4 调用 `Test.main`

最后，在完成类 Test 的初始化之后（在此期间可能会发生其他相应的加载、链接和初始化），将调用测试的 main 方法。最后，在完成类 Test 的初始化之后（在此期间可能会发生其他相应的加载、链接和初始化），将调用 Test 的 main 方法。

main 方法必须被声明为 public, static, 和 void。它必须指定一个形式参数(§8.4.1)，其声明类型为字符串数组。因此，以下任何一种声明都是可以接受的：

```
public static void main(String[] args)
public static void main(String... args)
```

12.2 加载类和接口

加载是指找到具有特定名称的类或接口的二进制形式的过程，可能是通过动态计算，但更典型的是通过检索 Java 编译器先前从源代码计算出的二进制表示，并从该二进制形式构造表示类或接口的 Class 对象(§1.4)。

加载的精确语义在 Java SE 19 版本的 Java 虚拟机规范的第 5 章中给出。在这里，我们从 Java 编程语言的角度来概述这个过程。

类或接口的二进制表示形式通常是 Java SE 19 版《Java 虚拟机规范》第 4 章中描述的 class 文件格式，但其他表示形式也可以，只要它们满足§13.1 中规定的要求。

12.2.1 加载过程

加载过程是由类 ClassLoader 及其子类实现的。ClassLoader 类的 defineClass 方法可以用来用 class 文件格式的二进制表示来构造 Class 对象(§1.4)。

ClassLoader 的不同子类可以实现不同的加载策略。特别是，类加载器可以缓存类和接口的二进制表示，根据预期的用法预取它们，或者加载一组相关的类。这些活动对于正在运行的应用程序可能不是完全透明的，例如，如果由于类加载器缓存了旧版本而没有找到类的新编译版本。然而，类加载器的责任是只在程序中不需要预取或组加载就可能出现加载错误的地方反映加载错误。

如果在类加载期间发生错误，则在(直接或间接)使用所请求的类或接口的程序中的任何一点，将抛出类 LinkageError 的下列子类之一的实例：

- ClassCircularityError: 请求的类或接口无法加载，因为它是自己的超类或超接口 (§8.1.4, §9.1.3, §13.4.4)。
- ClassFormatError: 声称指定请求的已编译类或接口的二进制数据格式不正确。
- NoClassDefFoundError: 通过相关的类加载器找不到请求的类或接口的定义。

因为加载涉及到新数据结构的分配，所以它可能会失败，并出现 OutOfMemoyError。

12.2.2 类加载器的一致性

行为良好的类加载器维护这些属性：

- 给定相同的名称，类加载器应该始终返回相同的 Class 对象。
- 如果类加载器 L_1 将类或接口 C 的加载委托给另一个加载器 L_2，则对于由 C 的直接超类类型、C 的直接超接口类型、C 中的字段类型、C 中的方法或构造函数的形参类型、或 C 中的方法的返回类型命名的任何类或接口 D，L_1 和 L_2 应为 D 返回相同的类对象。

恶意的类加载器可能会违反这些属性。但是，它不能破坏类型系统的安全性，因为 Java 虚拟机可以防范这一点。

> 有关这些问题的进一步讨论，请参阅 Java 虚拟机规范，Java SE 19 版和由盛亮和 Gilad Bracha 在 1998 年 10 月出版的 ACM SIGPLAN 通知第 33 卷，第 10 号，第 36-44 页上发表的论文 Dynamic Class Loading in the Java Virtual Machine，该论文发表在 OOPSLA'98 的论文集上。Java 编程语言设计的一个基本原则是，运行时类型系统不能被用 Java 编程语言编写的代码颠覆，甚至不能被 ClassLoader 和 SecurityManager 等其他敏感系统类的实现颠覆。

12.3 链接类和接口

链接是采用类或接口的二进制形式并将其组合到 Java 虚拟机的运行时状态中，以便可以执行的过程。类或接口始终在链接之前加载。

链接的精确语义在 Java SE 19 版的 Java 虚拟机规范第 5 章中给出。在这里，我们从 Java 编程语言的角度对该过程进行概述。

链接涉及三个不同的活动：验证、准备和解析符号引用。

该规范允许在链接活动（以及由于递归而导致的加载）发生时实现灵活性，前提是尊重 Java 编程语言的语义，在初始化类或接口之前完全验证和准备好类或接口，并且在链接期间检测到的错误被抛出到程序中的某个点，在该点上程序采取了可能需要链接到错误所涉及的类或接口的某些操作。

例如，实现可以选择仅在使用时单独解析类或接口中的每个符号引用（懒加载或延迟解析），或者在验证类时一次性解析所有符号引用（静态解析）。这意味着在某些实现中，在类或接口已被初始化之后，解析过程可以继续。

由于链接涉及新数据结构的分配，因此它可能会失败，并出现 OutOfMemoyError。

12.3.1 验证二进制表示形式

验证确保类或接口的二进制表示在结构上是正确的。例如，它检查每条指令是否具有有效的操作码；每条分支指令是否分支到其他指令的开头，而不是指令的中间；每种方法是否提供了结构上正确的签名；以及每条指令是否遵守 Java 虚拟机的类型规则。

如果在验证过程中发生错误，则将在程序中导致类被验证的点处抛出类 LinkageError 的以下子类的实例：

- VerifyError: 类或接口的二进制定义未能通过一组必需的检查，以验证它是否遵守 Java 虚拟机语言的语义，并且它不能违反 Java 虚拟机的完整性。（有关一些示例，请参见§13.4.2、§13.4.4、§13.4.9 和§13.4.17。）

12.3.2 准备类或接口

准备包括为类或接口创建静态字段（类变量和常量），并将这些字段初始化为默认值（§4.12.5）。 这不需要执行任何源代码；静态字段的显式初始化器作为初始化的一部分执行（§12.4），而不是准备。

> Java 虚拟机的实现可以在准备时预计算额外的数据结构，以便使类或接口上的后续操作更有效。一种特别有用的数据结构是“方法表”或其他数据结构，它允许在类的实例上调用任何方法，而不需要在调用时搜索超类。

12.3.3 符号引用解析

类或接口的二进制表示使用其他类和接口的二进制名称（§13.1），象征性地引用其他类和接口及其字段、方法和构造函数。对于字段和方法，这些符号引用包括字段或方法所属的类或接口的名称，以及字段或方法本身的名称以及适当的类型信息。

在可以使用符号引用之前，必须对其进行解析，其中检查符号引用是否正确，并且通常用直接引用替换，如果重复使用该引用，则可以更有效地处理该直接引用。

如果在解析过程中发生错误，则将引发错误。最典型的情况是，这将是类 IncompatibleClassChangeError 的以下子类之一的实例，但它也可能是 IncompatibleClassChangeError 其他子类的实例，甚至是类 IncompatibleClassChangeError 本身的实例。此错误可能在程序中使用符号引用的任何点直接或间接引发：

- IllegalAccessError: 遇到了一个符号引用，该符号引用指定了字段的使用或赋值、方法的调用或类实例的创建，包含引用的代码无法访问该字段或方法，因为该字段和方法是通过私有、受保护或包访问（非公共）声明的，或者因为该类在导出或打开到包含引用的代码的包中未声明为公共。

  > 例如，如果一个最初声明为公共的字段在引用该字段的另一个类被编译后变为私有的，则可能会发生这种情况（§13.4.7）；或者如果声明了公共类的包在引用该类的另一个模块被编译后停止由其模块导出（§13.3）。

- InstantiationError: 遇到了在类实例创建表达式中使用的符号引用，但无法创建实例，

因为该引用指向接口或抽象类。

例如，如果一个原本不是抽象类的类在引用该类的另一个类被编译后变为抽象类，则可能发生这种情况（§13.4.1）。

- NoSuchFieldError: 遇到了引用特定类或接口的特定字段的符号引用，但该类或接口不包含该名称的字段。

  例如，如果在编译引用该字段的另一个类之后从类中删除了字段声明，则可能会发生这种情况（§13.4.8）。

- NoSuchMethodError：遇到了引用特定类或接口的特定方法的符号引用，但该类或接口不包含该签名的方法。

  例如，如果在编译引用该方法的另一个类之后从类中删除了方法声明，则可能会发生这种情况（§13.4.12）。

此外，如果一个类声明了一个无法找到实现的 native 方法，则可能会抛出一个 UnsatifiedLinkError，它是 LinkageError 的子类。如果使用或更早使用该方法，则会发生错误，具体取决于 Java 虚拟机实现使用的是哪种解析策略（§12.3）。

12.4 初始化类或接口

类的初始化包括执行其静态初始化器和类中声明的静态字段（类变量）的初始化器。

接口的初始化包括对接口中声明的字段（常量）执行初始化器。

12.4.1 初始化何时发生

类或接口 T 将在以下任何一种情况首次出现之前立即初始化：

- T 是一个类，并创建了 T 的一个实例。
- 调用由 T 声明的静态方法。
- 赋值由 T 声明的静态字段。
- 使用由 T 声明的静态字段，并且该字段不是常量变量(§4.12.4)。

当一个类被初始化时，它的超类被初始化(如果它们以前没有被初始化)，以及声明任何默认方法(§9.4.3)的任何超接口(§8.1.5)(如果它们以前没有被初始化)。接口的初始化本身不会导致其任何超接口的初始化。

对静态字段的引用(§8.3.1.1)导致初始化实际声明它的类或接口，即使它可能通过子类、子接口或实现接口的类的名称被引用。

在类 Class 和包 java.lang.reflect 中调用某些反射方法也会导致类或接口初始化。

类或接口在任何其他情况下都不会被初始化。

请注意，编译器可能会在接口中生成合成的默认方法，即既不显式也不隐式声明的默认方法(§13.1)。这样的方法将触发接口的初始化，尽管源代码没有给出接口应该被初始化的指示。

其目的是让类或接口具有一组初始化器，使其处于一致状态，并且此状态是其他类观察到的第一个状态。静态初始化器和类变量初始化器是按文本顺序执行的，不能引用类中声明的类变量，这些类变量的声明在使用后以文本形式出现，即使这些类变量在作用域内(§8.3.3)。此限制旨在在编译时检测大多数循环或其他格式错误的初始化。

初始化代码不受限制这一事实允许构造示例，其中在计算初始化表达式之前，当类变量仍具有其初始默认值时，可以观察到该类变量的值，但此类示例在实践中很少。(此类示例也可用于实例变量初始化（§12.5）。) Java 编程语言的全部功能在这些初始化器中可用；程序员必须谨慎行事。这种能力给代码生成器带来了额外的负担，但这种负担在任何情况下都会出现，因为 Java 编程语言是并发的（§12.4.2）。

例子 12.4.1-1. 超类在子类之前初始化

```
class Super {
    static { System.out.print("Super "); }
}
class One {
    static { System.out.print("One "); }
}
class Two extends Super {
    static { System.out.print("Two "); }
}
class Test {
    public static void main(String[] args) {
        One o = null;
        Two t = new Two();
        System.out.println((Object)o == (Object)t);
    }
}
```


该程序产生以下输出：

```
Super Two false
```

类 One 从来不会被初始化，因为它不被主动使用，因此永远不会链接到。 类 Two 初始化在其超类 Super 初始化之后。

例子 12.4.1-2. 只有声明静态字段的类被初始化

```
class Super {
    static int taxi = 1729;
}
class Sub extends Super {
    static { System.out.print("Sub "); }
}
class Test {
    public static void main(String[] args) {
       System.out.println(Sub.taxi);
    }
}
```

此程序仅打印：

```
1729
```

因为类 Sub 从来不会被初始化；对 Sub.taxi 的引用是对在类 Super 中实际声明的字段的引用，并且不会触发类 Sub 的初始化。

例子 12.4.1-3. 接口初始化并不初始化超接口

```
interface I {
   int i = 1, ii = Test.out("ii", 2);
}
interface J extends I {
   int j = Test.out("j", 3), jj = Test.out("jj", 4);
}
interface K extends J {
   int k = Test.out("k", 5);
}
class Test {
    public static void main(String[] args) {
       System.out.println(J.i);
       System.out.println(K.j);
    }
    static int out(String s, int i) {
       System.out.println(s + "=" + i);
       return i;
    }
}
```

程序产生如下输出：

```
1
j=3
jj=4
3
```

对 J.i 的引用是指向一个常量变量的字段(§4.12.4)；因此，它不会导致 I 被初始化(§13.4.9)。

对 K.j 的引用是对接口 J 中实际声明的不是常量变量的字段的引用；这导致初始化接口 J 的字段，但不初始化其超接口 I 的字段，也不初始化接口 K 的字段。

尽管名称 K 被用来引用接口 J 的字段 j，但是接口 K 没有被初始化。

12.4.2 详细的初始化过程

因为 Java 编程语言是多线程的，所以类或接口的初始化需要仔细同步，因为其他线程可能正在尝试同时初始化相同的类或接口。还有一种可能性是，作为类或接口的初始化的一部分，可以递归地请求类或接口的初始化；例如，类 A 中的变量初始化器可能调用无关类 B 的方法，而类 B 又可能调用类 A 的方法。Java 虚拟机的实现通过使用以下过程负责同步和递归初始化。

该过程假定 Class 对象已经过验证和准备阶段，并且 Class 对象包含指示以下四种情况之一的状态：

- Class 对象已验证并准备好，但未初始化。
- Class 对象正在由某个特定的线程 T 初始化。
- Class 对象已经完全初始化，可以使用了。
- Class 对象处于错误状态，可能是因为尝试初始化但失败。

对于每个类或接口 C，都有一个唯一的初始化锁 LC。从 C 到 LC 的映射由 Java 虚拟机实现决定。从 C 到 LC 的映射由 Java 虚拟机实现决定。初始化 C 的过程如下：

1. 在 C 的初始化锁 LC 上同步。这涉及到等待，直到当前线程可以获取 LC。
2. 如果 C 的 Class 对象指示其他线程正在进行 C 的初始化，则释放 LC 并阻塞当前线程，直到通知正在进行的初始化已经完成，此时重复此步骤。
3. 如果 C 的 Class 对象指示当前线程正在进行 C 的初始化，那么这一定是一个递归的初始化请求。释放 LC 并正常完成。
4. 如果 C 的 Class 对象指示 C 已经初始化，则不需要进一步的操作。释放 LC 并正常完成。
5. 如果 C 的 Class 对象处于错误状态，则不可能进行初始化。释放 LC 并抛出 NoClassDefFoundError。
6. 否则，记录当前线程正在进行 C 的 Class 对象的初始化这一事实，并释放 LC。

   然后，初始化 C 的静态字段，它们是常量变量 (§4.12.4, §8.3.2, §9.3.1)。
7. 接下来，如果 C 是一个类而不是一个接口，那么让 SC 作为它的超类，并让 SI_1，...，SI_n 是 C 的所有至少声明了一个默认方法的超接口。超接口的顺序由 C 直接实现的每个接口的超接口层次结构上的递归枚举给出(按照 C 的 implements 子句从左到右的顺序)。对于由 C 直接实现的每个接口 I，在返回 I 之前，枚举在 I 的超接口上递归(按照 I 的 extends 子句从左到右的顺序)。

   对于列表 [SC, SI_1, ..., SI_n]中的每个 S, 如果 S 尚未初始化，则递归地为 S 执行整个过程。如有必要，首先验证并准备 S。

   如果 S 的初始化由于抛出的异常而突然完成，则获取 LC，将 C 的 Class 对象标记为错误，通知所有等待的线程，释放 LC，然后突然完成，引发与初始化 S 相同的异常。
8. 接下来，通过查询 C 的定义类加载器来确定是否为 C 启用了断言(§14.10)。
9. 接下来，按文本顺序执行类的类变量初始化器和类的静态初始化器，或接口的字段初始化器，就像它们是单个块一样。
10. 如果初始化器的执行正常完成，则获取 LC，将 C 的 Class 对象标记为完全初始化，通

知所有等待的线程，释放 LC，并正常完成此过程。

11. 否则，初始化器必须抛出某个异常 E 而突然完成。如果 E 的类不是 Error 或其子类之一，则创建 ExceptionInInitializerError 类的新实例，并使用 E 作为参数，并在接下来的步骤中使用该对象代替 E。如果因为发生 OutOfMemoyError 而无法创建 ExceptionInInitializerError 的新实例，则在下一步中使用 OutOfMemoyError 对象代替 E。

12. 获取 LC，将 C 的 Class 对象标记为错误，通知所有等待线程，释放 LC，并突然完成此过程，原因为 E 或上一步确定的替换原因。

    当一个实现可以确定类的初始化已经完成时，它可以通过在步骤 1 中取消锁获取(并在步骤 4/5 中释放)来优化该过程，前提是，就存储器模型而言，在执行优化时，如果锁被获取则将存在的所有 happens-before 的排序仍然存在。

    代码生成器需要保留类或接口的可能初始化点，插入对上述初始化过程的调用。如果此初始化过程正常完成，并且 Class 对象已完全初始化并准备使用，则不再需要调用初始化过程，并且可以从代码中消除它——例如，通过打补丁或以其他方式重新生成代码。

    在某些情况下，如果可以确定一组相关类和接口的初始化顺序，则编译时分析可能能够从生成的代码中消除许多关于类或接口已被初始化的检查。然而，这种分析必须充分考虑并发性和初始化代码不受限制的事实。

12.5 创建新的类实例

当类实例创建表达式（§15.9）的求值导致类被实例化时，显式创建新的类实例。

在以下情况下，可以隐式创建新的类实例：

- 加载包含字符串字面量（§3.10.5）或文本块（§3.10.6）的类或接口可能会创建一个新的字符串对象，以表示字符串字面量或文本块表示的字符串。(如果表示与字符串字面量或文本块表示的字符串相同的 Unicode 代码点序列的字符串实例先前已被占用，则不会创建此对象。)
- 执行导致装箱转换的操作(§5.1.7)。装箱转换可以创建与原生类型之一相关联的包装类(Boolean、Byte、Short、Character、Integer、Long、Float、Double)的新对象。
- 字符串连接运算符+(§15.18.1)的执行不是常量表达式(§15.29)的一部分，它总是创建一个新的字符串对象来表示结果。字符串连接操作符还可以为原生类型的值创建临时包装对象。
- 方法引用表达式(§15.13.3)或 lambda 表达式(§15.27.4)的求值可能需要创建实现函数式接口类型(§9.8)的类的新实例。

这些情况中的每一种都标识了要在类实例创建过程中使用指定参数(可能没有)调用的特定构造函数(§8.8)。

每当创建新的类实例时，都会为其分配内存空间，为类中声明的所有实例变量和类的每个超类中声明的所有实例变量，包括可能隐藏的所有实例变量留出空间(§8.3)。

如果没有足够的空间为对象分配内存，则类实例的创建会突然完成，并出现OutOfMemoyError。否则，新对象中的所有实例变量，包括在超类中声明的变量，都被初始化为它们的缺省值(§4.12.5)。

在将对新创建的对象的引用作为结果返回之前，使用以下过程处理所指示的构造函数以初始化新对象：

1. 将构造函数的参数赋给为这个构造函数调用新创建的参数变量。
2. 如果这个构造函数以一个同类的另一个构造函数的显式构造函数调用(§8.8.7.1)开始(使用 this)，然后计算参数，并使用相同的五个步骤递归地处理这个构造函数调用。如果那个构造函数调用突然结束，那么这个过程也会因为同样的原因突然结束;否则，继续步骤 5。
3. 此构造函数并不以对同一类中的另一个构造函数的显式构造函数调用(使用 this)开始。如果此构造函数用于 Object 以外的类，则此构造函数将以显式或隐式调用超类构造函数(使用 super)开始。计算参数并使用相同的五个步骤递归处理超类构造函数调用。如果那个构造函数调用突然结束，那么这个过程也会因为同样的原因突然结束。否则，继续步骤 4。
4. 执行该类的实例初始化器和实例变量初始化器，将实例变量初始化器的值赋值给相应的实例变量，按照它们在类的源代码中文本显示的从左到右的顺序。如果执行这些初始化器中的任何一个导致异常，那么就不会再处理其他初始化器，并且这个过程会突然结束，并出现相同的异常。否则，继续步骤 5。
5. 执行该构造函数体的其余部分。如果那个执行突然结束，那么这个过程也会因为同样的原因突然结束。否则，该过程正常完成。

与 C++不同的是，Java 编程语言在创建新类实例期间没有为方法分派指定更改规则。如果调用的方法在被初始化的对象的子类中被重写，那么这些重写的方法将被使用，甚至在新对象完全初始化之前。

例子 12.5-1. 实例创建评估

```
class Point {
    int x, y;
    Point() { x = 1; y = 1; }

}
class ColoredPoint extends Point {
    int color = 0xFF00FF;
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint(); System.out.println(cp.color);
    }
}
```

这里，创建了一个新的 ColoredPoint 实例。首先，为新的 ColoredPoint 分配空间，以保存字段 x、y 和

color。然后将所有这些字段初始化为它们的默认值(在本例中，每个字段为 0)。接下来，首先调用没有参数的 ColoredPoint 构造函数。由于 ColoredPoint 没有声明构造函数，因此隐式声明了以下形式的默认构造函数:

```
ColoredPoint() { super(); }
```

然后，这个构造函数调用不带参数的 Point 构造函数。Point 构造函数并不以调用构造函数开始，因此 Java 编译器提供了对其超类构造函数的不带参数的隐式调用，就像已经编写的那样:

```
Point() { super(); x = 1; y = 1; }
```

因此，将调用不带参数的 Object 构造函数。

类 Object 没有父类，因此递归到此结束。接下来，调用 Object 的任何实例初始化器和实例变量初始化器。接下来，执行不带参数的 Object 构造函数体。Object 中没有声明这样的构造函数，所以 Java 编译器提供了一个默认构造函数，在这个特殊情况下是:

```
Object() { }
```

此构造函数执行无效并返回。

接下来，执行类 Point 的实例变量的所有初始化器。当它发生时，x 和 y 的声明不提供任何初始化表达式，因此示例的这一步不需要任何操作。然后执行 Point 构造函数体，将 x 设为 1，将 y 设为 1。

接下来，执行 ColoredPoint 类的实例变量的初始化器。这一步将值 0xFF00FF 分配给 color。 最后，执行 ColoredPoint 构造函数体的其余部分(调用 super 之后的部分)；在主体的其他部分中碰巧没有语句，因此不需要进一步的操作，初始化完成。

例子 12.5-2. 实例创建期间的动态调度

```
class Super {
    Super() { printThree(); }
    void printThree() { System.out.println("three"); }
}
class Test extends Super {
    int three = (int)Math.PI; // That is, 3
    void printThree() { System.out.println(three); }

    public static void main(String[] args) {
        Test t = new Test();
        t.printThree();
    }
}
```

这个程序产生输出:

```
0
3
```

这表明在类 Super 的构造函数中调用 printThree 并没有调用类 Super 中 printThree 的定义，而是调用了类 Test 中 printThree 的重写定义。因此，该方法在 Test 的字段初始化器执行之前运行，这就是为什么第一个输出值是 0, Test 的字段 three 被初始化为默认值。之后在方法 main 中对 printThree 调用了

printThree 的相同定义，但是此时已经执行了实例变量 three 的初始化器，因此输出了值 3。

12.6 类实例的终结

从 Java SE 19 开始，Java SE 平台规范允许在 Java SE 平台的实现中禁用类实例的终结，期望在 Java SE 平台的未来版本中删除终结。

类 Object 有一个 protected 的方法叫做 finalize; 这个方法可以被其他类重写。可为对象调用的 finalize 的特定定义称为该对象的终结器。在垃圾回收器回收对象的存储之前，Java 虚拟机将调用该对象的终结器。

终结器提供了释放自动存储管理器无法自动释放的资源的机会。在这种情况下，仅仅回收对象使用的内存并不能保证回收对象所持有的资源。

Java 编程语言没有指定调用终结器的时间，只是说将在重用对象的存储之前调用终结器。

Java 编程语言没有指定哪个线程将为任何给定对象调用终结器。

需要注意的是，许多终结器线程可能是活动的(在大型共享内存多处理器上有时需要这样做)，而且如果一个大型连接的数据结构变成了垃圾，那么该数据结构中每个对象的所有 finalize 方法都可以同时被调用，每个终结器调用都在不同的线程中运行。

Java 编程语言对 finalize 方法调用没有要求排序。终结器可以以任何顺序调用，甚至可以并发调用。

例如，如果循环链接的一组未结束的对象变得不可访问(或终结器可访问)，那么所有对象可能一起成为可结束的。最终，这些对象的终结器可以以任何顺序调用，甚至可以使用多个线程并发调用。如果自动存储管理器后来发现对象不可达，那么可以回收它们的存储。

当所有对象都变得不可访问时，可以直接实现这样一个类，它将导致以指定的顺序为一组对象调用一组类似终结器的方法。定义这样的类留给读者作为练习。

可以保证调用终结器的线程在调用终结器时不会持有任何用户可见的同步锁。

如果在结束过程中抛出未捕获的异常，则忽略该异常并终止该对象的结束。

对象的构造函数的完成发生在它的 finalize 方法执行之前(形式意义上的 happens-before)。

在类 Object 中声明的 finalize 方法不采取任何操作。Object 类声明一个 finalize 方法这一事实意味着任何类的 finalize 方法都可以调用其超类的 finalize 方法。应该总是这样做，除非程序员有意取消超类中终结器的操作。(与构造函数不同，终结器不会自动调用父类的终结器;这样的调用必须显式编码。)

为了提高效率，实现可以跟踪那些不重写类 Object 的 finalize 方法的类，或者以一种简单的方式重写它。

例如:

```
protected void finalize() throws Throwable {
```

```
    super.finalize();
}
```

我们鼓励实现将此类对象视为具有未被重写的终结器，并更有效地结束它们，如§12.6.1 所述。

就像任何其他方法一样，可以显式调用终结器。

java.lang.ref 包描述了弱引用，它与垃圾收集和终结交互。与任何与 Java 编程语言有特殊交互的 API 一样，实现者必须认识到 java.lang.ref API 强加的任何要求。 本规范没有以任何方式讨论弱引用。读者可以参考 API 文档了解详细信息。

12.6.1 实现终结

每个对象都可以用两个属性来描述:它可以是可达的、终结器可达的或不可达的，它还可以是未终结的、可终结的或已终结的。

可达对象是可以从任何活动线程在任何潜在的持续计算中访问的任何对象。

终结器可达的对象可以通过某些引用链从某个可终结化对象访问，但不能从任何活动线程访问。

无法访问的对象无法通过这两种方法中的任何一种访问。

未终结的对象从未自动调用其终结器。

已终结的对象已自动调用其终结器。

可终结对象从未自动调用过其终结器，但 Java 虚拟机最终可能会自动调用其终结器。

对象 o 是不可终结的，直到它的构造函数调用了 o 上 Object 的构造函数，并且调用成功完成(也就是说，没有引发异常)。对对象字段的每个预终结写入对对象的结束必须是可见的。此外，该对象字段的预终结读取都不会看到在初始化该对象后发生的写入操作。

程序的优化转换可以设计成减少可到达对象的数量，使之少于那些天真地认为可到达的对象的数量。例如，Java 编译器或代码生成器可能会选择设置一个不再被用作 null 的变量或参数，从而使此类对象的存储可能更快地被回收。

另一个例子是对象字段中的值存储在寄存器中。然后程序可以访问寄存器而不是对象，并且不再访问该对象。这意味着该对象是垃圾。注意，只有当引用在栈上，而不是存储在堆中时，才允许这种优化。

例如，考虑终结器守护模式:

```
class Foo {
    private final Object finalizerGuardian = new Object() {
       protected void finalize() throws Throwable {
            /* finalize outer Foo object */
       }
    }
}
```

终结器守护者强制调用 super.finalize 如果子类重写 finalize 并且没有显式调用 super.finalize。

如果允许对存储在堆上的引用进行这些优化，则 Java 编译器可以检测到 finalizerGuardian 字段从未被读取，将其设为空，立即收集对象，并提前调用终结器。这与意图背道而驰：当 Foo 实例变得无法访问时，程序员可能想要调用 Foo 终结器。因此，这种转换是不合法的：只要外部类对象是可访问的，内部类对象就应该是可访问的。

这种类型的转换可能会导致调用 finaliz 方法的时间比预期的要早。为了允许用户防止这种情况，我们强制使用同步可以使对象保持活动状态的概念。如果对象的终结器可以在该对象上实现同步，则该对象必须是活动的，并且只要持有锁，该对象就被认为是可访问的。

请注意，这并不妨碍同步消除：只有在终结器可能在对象上同步的情况下，同步才会使对象保持活动状态。由于终结器发生在另一个线程中，因此在许多情况下，无论如何都无法删除同步。

12.6.2 与内存模型交互

内存模型(§17.4)必须能够决定何时可以提交终结器中发生的操作。这一节描述了终结化与内存模型的交互。

每次执行都有多个标记为 di 的可达性判定点。每个动作要么发生在 di 之前，要么发生在 di 之后。除了明确提到的以外，本节中描述的 comes-before 排序与内存模型中的所有其他排序无关。

如果读取 r 时看到写入 w, r 出现在 di 之前，那么 w 必须出现在 di 之前。

如果 x 和 y 是同一个变量或监视器上的同步动作，使得 so(x, y)(§17.4.4)和 y 出现在 di 之前，那么 x 必须出现在 di 之前。

在每个可达性决策点，一些对象集被标记为不可达，而这些对象的一些子集被标记为可终结。这些可达性决策点也是根据 java.lang.ref 包的 API 文档中提供的规则检查、加入队列和清除引用的点。

唯一被认为在 di 点绝对可达的对象是那些可以通过应用这些规则证明可达的对象:

- 如果类 C 的静态字段 v 存在写 w1，因此 w1 写的值是对 B 的引用，类 C 由可达的类加载器加载，并且不存在到 v 的写 w2，因此 hb(w2, w1)不为真，并且 w1 和 w2 都在 di 之前，则对象 B 在 di 是绝对可达的。
- 如果 A 的元素 v 有一个写 w1，且 w1 写的值是对 B 的引用，且不存在到 v 的写 w2，且 hb(w2, w1)不为真，且 w1 和 w2 都在 di 之前，则对象 B 在 di 是绝对可达的。
- 如果一个对象 C 绝对可以从一个对象 B 到达，对象 B 也绝对可以从一个对象 A 到达，那么 C 也绝对可以从一个对象 A 到达。

如果一个对象 X 在 di 被标记为不可达，那么:

- X 不能从静态字段绝对到达 di;而且
- 线程 t 中所有在 di 之后出现的 X 的活跃使用必须发生在对 X 的终结器调用中，或者是

线程 t 在 di 之后执行对 X 的引用而读取的结果;而且

- 所有从 di 后面来的读取，如果看到对 X 的引用，则必须看到在 di 处不可达的对象元素的写入，或者看到在 di 后面来的对象元素的写入。

动作 a 是对 X 的主动使用，当且仅当以下至少有一个为真:

- a 读取或写入 X 的元素
- a 锁定或解锁 X，并且在调用 X 的终结器之后发生了对 X 的锁定操作。
- a 写入 X 的引用
- a 是对对象 Y 的主动使用，而 X 绝对可以从 Y 到达

如果一个对象 X 在 di 被标记为可终结的，那么:

- X 必须在 di 处标记为不可达;并且
- di 必须是唯一将 X 标记为可终结的位置;而且
- 在终结器调用之后发生的动作必须在 di 之后发生。

12.7 卸载类和接口

Java 编程语言的实现可以卸载类。

当且仅当定义的类加载器被§12.6 所讨论的垃圾收集器回收时，类或接口才可以被卸载。

由引导程序加载器加载的类和接口不能被卸载。

> 类卸载是一种有助于减少内存使用的优化。显然，程序的语义不应该依赖于系统是否以及如何选择实现类卸载等优化。否则会损害程序的可移植性。因此，类或接口是否被卸载对程序来说应该是透明的。
>
> 但是，如果类或接口 C 在其定义的加载器可能可达的情况下被卸载，那么 C 可能会被重新加载。谁也不能保证这种情况不会发生。即使类没有被任何其他当前加载的类引用，它也可能被一些尚未加载的类或接口 D 引用。当 D 被 C 的定义加载器加载时，它的执行可能会导致 C 的重新加载。
>
> 例如，如果类具有静态变量(其状态将丢失)、静态初始化器(可能有副作用)或 native 方法(可能保留静态状态)，则重新加载可能不是透明的。此外，Class 对象的哈希值依赖于它的标识。因此，通常不可能以完全透明的方式重新加载类或接口。
>
> 因为我们永远不能保证卸载加载器可能可达的类或接口不会导致重新加载，而且重新加载永远不是透明的，但卸载必须是透明的，因此，当加载器可能可达时，一定不能卸载类或接口。类似的推理可以用来推断由引导程序加载器加载的类和接口永远不能被卸载。
>
> 人们还必须争论，如果类 C 的定义类加载器可以被回收，那么为什么卸载类 C 是安全的。如果定义的加载器可以被回收，那么就永远不会有任何对它的活动引用(这包括那些不是活动的引用，但可能会被终结器恢复)。反过来，这只能在永远不可能存在对该加载器定义的任何类(包括 C)的任何活动引用的情况下才成立，无论是从它们的实例还是从代码中。
>
> 类卸载是一种优化，它只对装入大量类并在一段时间后停止使用大部分类的应用程序有意义。这类应用程序的一个主要例子是 web 浏览器，但还有其他应用程序。这类应用程序的一个特点是，它们通过

显式使用类加载器来管理类。因此，上述政策对他们很有效。

严格地说，这个规范并不一定要讨论类卸载的问题，因为类卸载只是一种优化。但是，这个问题是非常微妙的，所以在这里提到它是为了澄清。

12.8 程序退出

当发生以下两种情况之一时，程序终止其所有活动并退出:

- 所有非守护线程的线程都会终止。
- 有些线程调用类 Runtime 或类 System 的退出方法，安全管理器并不禁止退出操作。