1. 类加载器
2. 概述

在类加载阶段，通过一个类的全限定来获取描述该类的二进制字节流这个动作实在Java虚拟机外部实现的，以便让程序自己决定从哪里加载类，实现这个动作的代码称为类加载器。

1. 类与类加载器

任意的一个类，都必须由加载它的类加载器和这个类本身一起共同确立其在Java虚拟机中的唯一性，每一个类加载器都有一个独立的类名称空间。若要比较两个类是否相等，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使两个类来自同一个Class文件，被同一个Java虚拟机加载，只要类加载器不同，必定是不相等的。这里所说的相等指的是Class对象的equals()方法，isAssignableFrom()方法，isInstance()方法的返回结果，还有使用instanceof关键字做对象所属关系判定的情况。

1. 双亲委派模型
2. 三层类加载器

从Java虚拟机的角度来看，只存在两种不同的类加载器，一种是启动类加载器，使用C++实现，是虚拟机的一部分；另一种是其他的两类加载器，由Java实现，独立存在于虚拟机之外，并且全部继承自java.lang.ClassLoader。

从开发者的角度看，类加载器应该划分地更加细致。JDK1.2之后，Java一直保持着三层类加载器，双亲委派的类加载架构。三层类加载器分别是启动类加载器，扩展类加载器，应用程序加载器，以下的加载器介绍基于JDK8及之前版本。

1. 启动类加载器

负责加载存放在<JAVA\_HOME>\lib目录，或者被-Xbootclasspath参数所指定的路径中存放的虚拟机能够识别的类库到虚拟机的内存中。启动类加载器无法直接被程序引用，用户在编写自定义类加载器时，如果需要将加载请求委派给引导类加载器处理，需要使用null来代替引导类加载器。

1. 扩展类加载器

负责加载<JAVA\_HOME>\lib\ext目录，或者被java.ext.dirs系统变量所指定的路径中的所有类库，这是一种Java系统类库的扩展机制，用户可以将通用性的类库放在ext目录中以扩展Java SE的功能，由于该扩展器由Java实现，因此可以直接在程序中使用扩展类加载器来来加载Class文件。

1. 应用程序类加载器

该类加载器是ClassLoader类中的getSystemClassLoader()方法的返回值，它负责加载用户类路径（ClassPath）上的所有类库，开发者可以直接在代码中使用这个类加载器，是程序的默认类加载器。

1. 双亲委派模型

启动类加载器在顶层，下一层是扩展类加载器，下一层是应用程序类加载器，最后一层是自定义类加载器。该模型要求除了顶层的启动类加载器之外，其余的类加载器都应该有自己的父类加载器，但是这里的父子关系并不是继承，而是使用组合关系来复用父加载器的代码。

该模型的工作过程是，当一个类加载器收到了类加载的请求后，首先它不会自己来尝试加载，而是将请求委派给父类加载器完成，每一层都是如此，因此所有的类加载请求最终都会传送到顶层的启动类加载器中，只有父加载器反馈自己无法完成这个加载请求时，子加载器才会尝试加载。

该模型的优点是让类的加载有一种优先级的层次关系，可以将基础的类都放在顶层，不论其他类的依赖关系如何，最终都会向上层请求加载，都能够加载到基础的类，而不是随意地加载，导致程序的依赖混乱。

1. 破坏双亲委派模型
2. 第一次破坏

JDK1.2引入双亲委派模型时，需要兼容之前的的代码，特别是loadClass()方法，该方法中之前没有向父类加载器委派请求的逻辑，需要添加该逻辑，并且需要在java.lang.ClassLoader中添加一个新的protected方法findClass()方法，该方法为用户自定义类加载器的加载逻辑。这样，既可以保证用户按照自己的意愿加载类，有保证了双亲委派规则。

1. 第二次破坏

双亲委派模型将越基础的类放到越上层的类加载器中加载，这样确实很好地保证了各个类加载器协作的高效加载。但是如果基础的类要加载下层的类，要怎么办呢？如JNDI服务，它的代码由启动类加载器加载，它存在的目的就是对资源进行查找和集中管理，需要调用其他厂商部署在程序的ClassPath下的JDNI服务提供者接口的代码。

为了解决这个问题，引入了线程上下文类加载器，这个类加载器可以通过Thread类的setContextLoader()方法设置，如果创建线程时还未设置，则会从父线程中继承一个，如果在应用程序的全局范围内都没有设置的话，那么这个类加载器就是默认的应用程序类加载器。有了线程上下文类加载器，父类加载器就可以请求子类加载器完成类加载的行为了，即逆向使用类加载器。

JDK6，提供了ServiceLoader类，以META-INF/services中的配置信息，辅以责任链模式，提供了更加合理的逆向加载方案。

1. 第三次破坏

由于程序动态性的需求，如代码热替换，模块热部署等，实现代码，模块在服务不停机的情况下更新的操作。在模块化规范的斗争中，IBM领导的OSGI战胜了Oracle公司的Jsgsaw，OSGI实现模块化热部署的关键是它自定义的类加载器机制的实现，每个程序模块（Bundle）都有一个自己的类加载器，当需要更换一个bundle时，就把bundle和类加载器一起换掉以实现代码的热替换。在OSGI环境下，类加载器不再是双亲委派的树状模型，而是网状结构，一层层地进行类搜索。

1. Java模块化系统
2. 概述

JDK9中引入的Java模块化系统（JPMS）实现了可配置的封装隔离机制，保证了模块之间的访问权限隔离。

1. 模块定义

Java中的模块不仅有代码，还有如下的内容：

1. 依赖其他模块的列表；
2. 导出的包列表，即其他的模块可以使用的列表；
3. 开放的包列表，即其他模块可反射访问模块的列表；
4. 使用的服务列表；
5. 提供服务的实现列表；
6. 封装隔离机制

要实现封装隔离机制，首先要解决JDK9之前基于类路径（ClassPath）来查找依赖的可靠性问题，此前如果类路径中缺失了运行时依赖的类型，就只能等程序运行到发生该类型的加载，链接时才会报出异常。在JDK9之后，采用模块化封装，模块可以声明对其他模块的显示依赖，这样Java虚拟机就能在启动时验证应用程序开发阶段设定好的依赖关系在运行期是否完备，如果有缺失则直接启动失败，避免了运行时异常。

该机制还解决了原来类路径上跨JAR文件的public类型的可访问性问题，模块化提供了更加细致的访问控制，必须声明哪些public的类型可以被哪些模块访问，访问控制是在类加载过程完成。

1. 模块的兼容性
2. 模块路径

为了让可配置的封装隔离机制能够兼容传统的类路径查找机制，JDK9提出了与类路径对应的模块路径。要辨别哪个类库是模块还是传统的JAR包，只取决于它存放的路径，只要在类路径上的JAR 文件，无论是否包含模块化信息（module-info.class文件），都会被当做传统的JAE包；同理，只要在模块路径上的JAR文件，即使没有JMOD后缀，或者module-info.class文件，仍然被当做一个模块。

那么JDK9之后是如何兼容传统类路径的呢？以下的规则保证了传统程序可以访问到所有标准类库模块中导出的包：

1. JAR文件在类路径的访问规则

所有类路径下的JAR以及资源文件，都被视为打包在一个匿名模块中，这个模块没有隔离，可以看到和使用类路径上的所有包，JDK系统模块中所有导出包，以及模块路径上所有模块的导出包。

1. 模块在模块路径的访问规则

模块路径上的有名模块只能访问到它依赖定义中列出的模块和包，匿名模块中的所有内容对于有名模块来说都是不可见的。

1. JAR文件在模块路径的访问规则

传统的，不包含模块定义的JAR放置在模块路径中，就会变成一个自动模块，该模块默认依赖整个模块路径中的所有模块，可以访问到所有模块导出的包，同时该模块也默认导出自己的所有包。

1. 模块的版本控制

如果同一个模块发行了多个不同的版本，只能由开发者在编译打包时人工选择正确版本的模块来保障依赖的正确性，Java模块系统目前不支持在模块定义中加入版本号来管理和约束依赖。

1. 模块化下的类加载器

扩展类加载器被平台类加载器代替，因为JDK是基于模块化构建的（原有的rt.jar和toos.jar被拆分为多个JMOD文件），Java类库已经支持了动态扩展的需求，因此无需保留扩展类加载器，因此无需保留<JAVA\_HOME>/lib/ext目录。同样的理由，<JAVA\_HOME>/jre目录也没有了，因为随时可以组合构建出程序运行所需的JRE。

平台类加载器和应用程序类加载器都不再派生自java.net.URLClassLoader，现在的启动类加载器，平台类加载器和应用程序类加载器都继承自jdk.internal.loader.BuiltinClassLoader。

JDK9中仍然维持着三层类加载器和双亲委派的架构，但是类加载的委派关系变化了，在委派给父加载器之前，先判断该类是否能够归属到某一个系统模块中，如果可以就优先委派给负责那个模块的加载器。Java模块化系统中明确规定了三个类加载器负责加载的模块。