

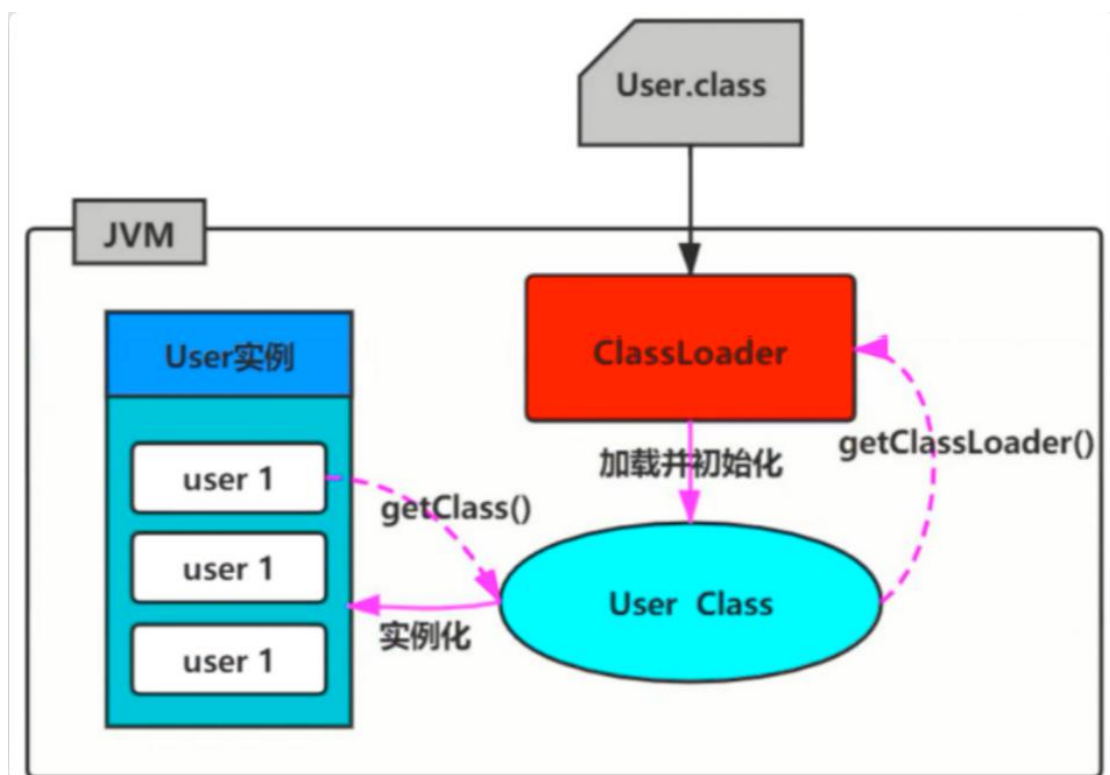
再谈类的加载器

第一节 概述

类加载器是 JVM 执行类加载机制的前提

ClassLoader 的作用：

ClassLoader 是 Java 的核心组件，所有的 Class 都是由 ClassLoader 进行加载的，ClassLoader 负责通过各种方式将 Class 信息的二进制数据流读入 JVM 内部，转换为一个与目标类对应的 `java.lang.Class` 对象实例。然后交给 Java 虚拟机尽心链接、初始化等操作。因此，ClassLoader 在整个装载阶段，只能影响到类的加载，而无法通过 ClassLoader 去改变类的链接和初始化行为。至于它是否可以运行，则由 Execution Engine 决定



类加载器最早出现在 Java 1.0 版本中，那个时候只是单纯地为了满足 Java Applet 应用而被研发出来，但如今类加载器却在 OSGI、字节码加解密领域大放异彩。这主要归功于 Java 虚拟机的设计者们当初在设计类加载器的时候，并没有考虑将它绑定在 JVM 内部，这样做的好处就是能够更加灵活和动态地执行类加载操作

1. 大厂面试题

➤ 蚂蚁金服：

深入分析 ClassLoader，双亲委派机制

类加载器的双亲委派模型是什么？

一面：双亲委派机制及使用原因

➤ 百度：

都有哪些类加载器，这些类加载器都加载哪些文件？

手写一个类加载器 Demo

Class 的 `forName("java.lang.String")` 和 Class 的 `getClassLoader()` 的 `loadClass("java.lang.String")` 有什么区别？

➤ 腾讯：

什么是双亲委派模型？

类加载器有哪些？

➤ 小米：

双亲委派模型介绍一下

➤ 滴滴：

简单说说你了解的类加载器

一面：讲一下双亲委派模型，以及其优点

➤ **字节跳动：**

什么事类加载器，类加载器有哪些？

➤ **京东：**

类加载器的双亲委派模型是什么？

双亲委派机制可以打破吗？为什么？

2. 类加载的分类

类的加载分类：显式加载 vs 隐式加载

Class 文件的显式加载与隐式加载的方式是指 JVM 加载 Class 文件到内存的方式

- 显式加载指的是在代码中通过调用 `ClassLoader` 加载 Class 对象，如直接使用 `Class.forName(name)` 或 `this.getClass().getClassLoader().loadClass()` 加载 Class 对象
- 隐式加载则是不直接在代码中调用 `ClassLoader` 的方法加载 Class 对象，而是通过虚拟机自动加载到内存中，如在加载某个类的 Class 文件时，该类的 Class 文件中引用了另外一个类的对象，此时额外引用的类将通过 JVM 自动加载到内存中

在日常开发中以上两种方式一般会混合使用

3. 类加载器的必要性

一般情况下，Java 开发人员并不需要在程序中显式地使用类加载器，但是了解类加载器的加载机制却显得至关重要。从以下几个方面说：

- 避免在开发中遇到 `java.lang.ClassNotFoundException` 异常或 `java.lang.NoClassDefFoundError` 异常时手足无措。只有了解类加载器的加载

机制才能够在出现异常的时候快速地根据错误异常日志定位问题和解决问题

- 需要支持类的动态加载或需要对编译后的字节码文件进行加解密操作时，就需要与类加载器打交道了
- 开发人员可以在程序中编写自定义类加载器来重新定义类的加载规则，以便实现一些自定义的处理逻辑

4. 命名空间

1) 何为类的唯一性？

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确认其在 Java 虚拟机中的唯一性。每一个类加载器，都拥有一个独立的类名称空间：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，即使这两个类源自同一个 Class 文件，被同一个虚拟机加载，只要加载他们的类加载器不同，那这两个类就必定不相等

2) 命名空间

- 每个类加载器都有自己的命名空间，命名空间由该加载器所有的父加载器所加载的类组成
- 在同一命名空间中，不会出现类的完整名字(包括类的包名)相同的两个类
- 在不同的命名空间中，有可能会出现类的完整名字(包括类的包名)相同的两个类

在大型应用中，我们往往借助这一特性，来运行同一个类的不同版本

5. 类加载机制的基本特征

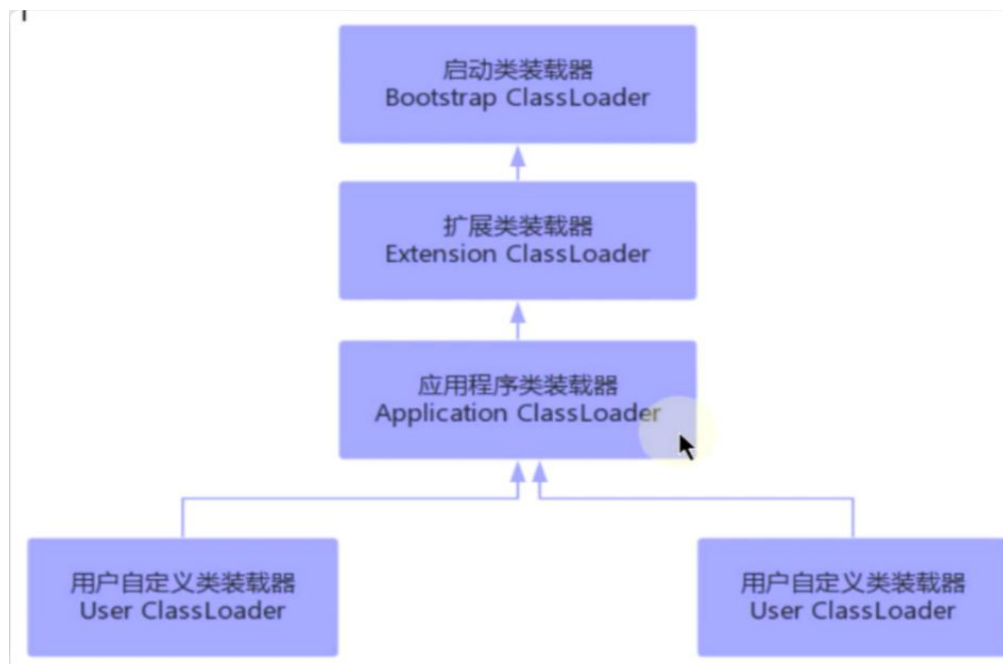
通常类加载机制有三个基本特征：

-
- 双亲委派模型。但不是所有类加载都遵守这个模型，有的时候，启动类加载器所加载的类型，是可能要加载用户代码的，比如 JDK 内部的 `ServiceProvider/ServiceLoader` 机制，用户可以在标准 API 框架上，提供自己的实现，JDK 也需要提供些默认的实现。例如，Java 中 JNDI、JDBC、文件系统、Cipher 等很多方面，都是利用的这种机制，这种情况就不会用双亲委派模型去加载，而是利用所谓的上下文加载器
 - 可见性，子类加载器可以访问父加载器加载的类型，但是反过来是不允许的。不然，因为缺少必要的隔离，我们就没有办法利用类加载器去实现容器的逻辑
 - 单一性，由于父加载器的类型对于子加载器是可见的，所以父加载器中加载过的类型，就不会在子加载器中重复加载。但是注意，类加载器"邻居"间，同一类型仍然可以被加载多次，因为相互并不可见

第二节 复习：类的加载器分类

JVM 支持两种类型的类加载器，分别为引导类加载器(Bootstrap ClassLoader)和自定义类加载器(User-Defined ClassLoader)

从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是 Java 虚拟机规范却没有这么定义，而是将所有派生于抽象类 `ClassLoader` 的类加载器都划分为自定义类加载器。无论类加载器的类型如何划分，在程序中我们最常见的类加载器结构主要是如下情况：



- 除了顶层的启动类加载器外，其余的类加载器都应当有自己的"父类"加载器
- 不同类加载器看似是继承(Inheritance)关系，实际上是包含关系。在下层加载器中，包含着上层加载器的引用

```
class ClassLoader {  
  
    ClassLoader parent; //父类加载器  
  
    public ClassLoader(ClassLoader parent) {  
  
        this.parent = parent;  
  
    }  
}  
  
class ParentClassLoader extends ClassLoader {  
  
    public ParentClassLoader(ClassLoader parent) {  
  
        super(parent);  
  
    }  
}
```

```
}  
  
class ChildClassLoader extends ClassLoader {  
  
    public ChildClassLoader(ClassLoader parent) {  
  
        //parent = new ParentClassLoader();  
  
        super(parent);  
  
    }  
  
}
```

1. 引导类加载器

启动类加载器(引导类加载器)

- 这个类加载使用 C/C++ 语言实现的，嵌套在 JVM 内部
- 它用来加载 Java 的核心库(JAVA_HOME/jre/lib/rt.jar 或 sun.boot.class.path 路径下的内容)。用于提供 JVM 自身需要的类
- 并不继承自 java.lang.ClassLoader，没有父加载器
- 出于安全考虑，Bootstrap 启动类加载器之加载包名为 java、javax、sun 等开头的类
- 加载扩展类和应用程序类加载器，并指定为他们的父类加载器

```
[Loaded java.net.URL from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]  
[Loaded java.util.jar.Manifest from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]  
[Loaded sun.misc.Launcher from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]  
[Loaded sun.misc.Launcher$AppClassLoader from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]  
[Loaded sun.misc.Launcher$ExtClassLoader from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]  
[Loaded java.security.CodeSource from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]  
[Loaded java.lang.StackTraceElement from D:\develop_tools\jdk\jdk1.8.0_131\jre\lib\rt.jar]
```

使用 -XX:+TraceClassLoading 参数

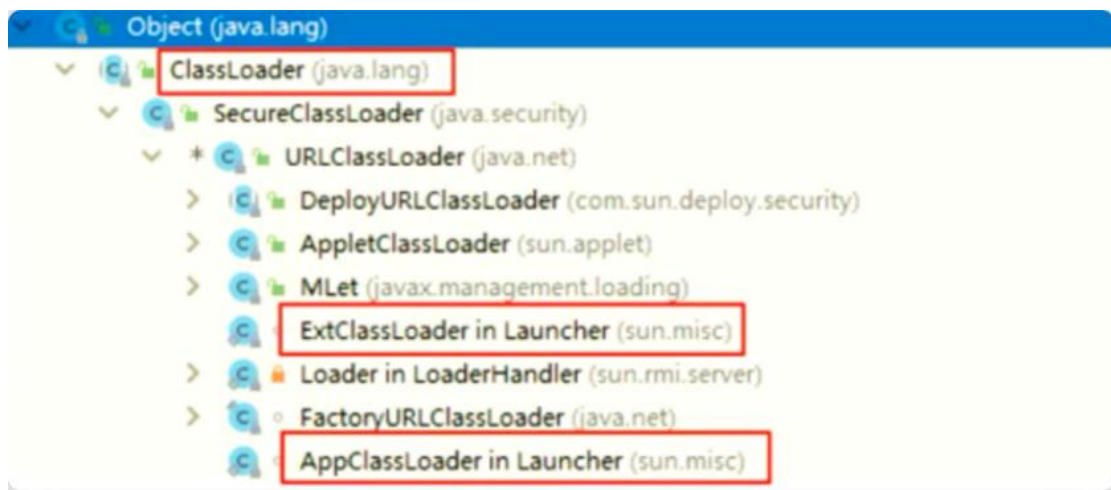
启动类加载器使用 C++ 编写的？Yes！

- C/C++：指针函数 & 函数指针、C++ 支持多继承、更加高效
- Java：由 C++ 演变而来，(C++)-- 版，单继承

2. 扩展类加载器

扩展类加载器(Extension ClassLoader)

- Java 语言编写，由 `sun.misc.Launcher$ExtClassLoader` 实现
- 继承于 `ClassLoader` 类
- 父类加载器为启动类加载器
- 从 `java.ext.dirs` 系统属性所指定的目录中加载类库，或从 JDK 的安装目录的 `jre/lib/ext` 子目录下加载类库。如果用户创建的 JAR 放在此目录下，也会自动由扩展类加载器加载



3. 系统类加载器

应用程序类加载器(系统类加载器，AppClassLoader)

- Java 语言编写，由 `sun.misc.Launcher$AppClassLoader` 实现

-
- 继承于 `ClassLoader` 类
 - 父类加载器为扩展类加载器
 - 它负责加载环境变量 `classpath` 或系统属性 `java.class.path` 指定路径下的类库
 - 应用程序中的类加载器默认是系统类加载器
 - 它是用户自定义类加载器的默认父加载器
 - 通过 `ClassLoader` 的 `getSystemClassLoader()` 方法可以获取到该类加载器

4. 用户自定义类加载器

- 在 Java 的日常应用程序开发中，类的加载几乎是由上述 3 种类加载器相互配合执行的。在必要时，我们还可以自定义类加载器，来定制类的加载方式
- 体现 Java 语言强大生命力和巨大魅力的关键因素之一便是，Java 开发者可以自定义类加载器来实现类库的动态加载，加载源可以是本地的 JAR 包，也可以是网络上的远程资源
- 通过类加载器可以实现非常绝妙的插件机制，这方面的实际应用案例不胜枚举。例如，著名的 OSGI 组件框架，再如 Eclipse 的插件机制。类加载器为应用程序提供了一种动态增加新功能的机制，这种机制无需重新打包发布应用程序就能实现
- 同时，自定义加载器能够实现应用隔离，例如 Tomcat、Spring 等中间件和组件框架都在内部实现了自定义的加载器，并通过自定义加载器隔离不同的组件模块。这种机制比 C/C++ 程序要好太多，想不修改 C/C++ 程序就能为其新增功能，几乎是不可能的，仅仅一个兼容性便能阻挡所有美好的设想
- 自定义类加载器通常需要继承于 `ClassLoader`

第三节 测试不同的类加载器

每个 Class 对象都会包含一个定义它的 ClassLoader 的一个引用

获取 ClassLoader 的途径

获取当前类的 ClassLoader

```
clazz.getClassLoader();
```

获得当前线程上下文的 ClassLoader

```
Thread.currentThread().getContextClassLoader();
```

获得系统的 ClassLoader

```
ClassLoader.getSystemClassLoader();
```

说明:

站在程序的角度看,引导类加载器与另外两种类加载器(系统类加载器和扩展类加载器)并不是同一个层次意义上的加载器,引导类加载器是使用 C++ 语言编写而成的,而另外两种类加载器则是使用 Java 语言编写的。由于引导类加载器压根儿就不是一个 Java 类,因此在 Java 程序中只能打印出空值

数组类的 Class 对象,不是由类加载器去创建的,而是在 Java 运行期 JVM 根据需要自动创建的。对于数组类的类加载器来说,是通过 `Class.geetClassLoader()` 返回的,与数组当中元素类型的类加载器是一样的:如果数组当中的元素类型是基本数据类型,数组类是没有类加载器的

```
String[] strArr = new String[6];
```

```
System.out.println(strArr.getClass().getClassLoader());
```

```
//运行结果: null
```

```
ClassLoaderTest[] test = new ClassLoaderTest[1];

System.out.println(test.getClass().getClassLoader());

//运行结果: sun.misc.Launcher$AppClassLoader@18b4aac2

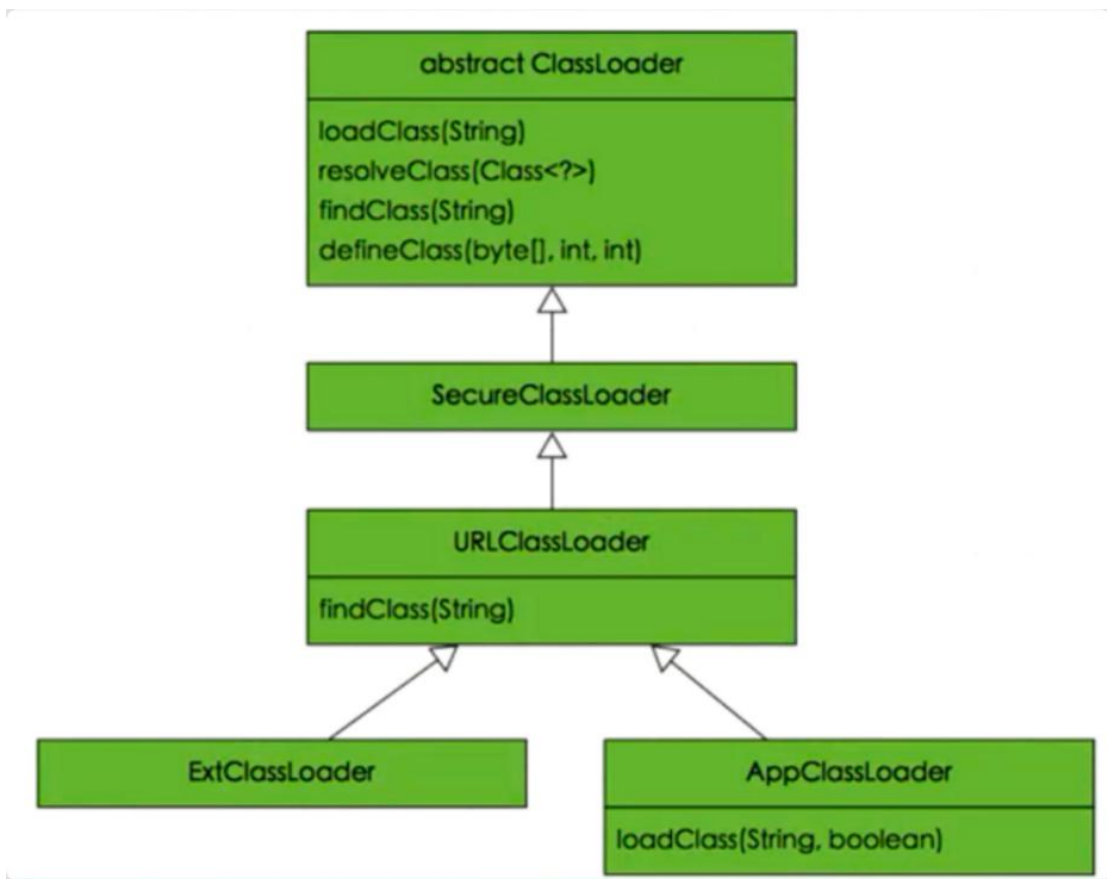

int[] inst = new int[2];

System.out.println(inst.getClass().getClassLoader());

//运行结果: null
```

第四节 ClassLoader 源码解析

ClassLoader 与现有类加载的关系:



除了以上虚拟机自带的加载器外，用户还可以定制自己的类加载器。Java 提供了抽象类 `java.lang.ClassLoader`，所有用户自定义的类加载器都应该继承 `ClassLoader` 类

1. ClassLoader 的主要方法

抽象类 `ClassLoader` 的主要方法：(内部没有抽象方法)

- `public final ClassLoader getParent()`

返回该类加载器的超类加载器

- `public Class<?> loadClass(String name) throws ClassNotFoundException`

加载名称为 `name` 的类，返回结果为 `java.lang.Class` 类的实例。如果找不到类，则返回 `ClassNotFoundException` 异常。该方法中的逻辑就是双亲委派模式的实现

➤ `protected Class<?> findClass(String name) throws ClassNotFoundException`

查找二进制名称为 `name` 的类，返回结果为 `java.lang.Class` 类的实例。这是一个受保护的方法，JVM 鼓励我们重写此方法，需要自定义加载器遵循双亲委派机制，该方法会在检查完父类加载器之后被 `loadClass()` 方法调用

在 JDK 1.2 之前，在自定义类加载时，总会去继承 `ClassLoader` 类并重写 `loadClass` 方法，从而实现自定义的类加载类。但是在 JDK 1.2 之后已不再建议用户去覆盖 `loadClass()` 方法，而是建议把自定义的类加载逻辑写在 `findClass()` 方法中，从前面的分析可知，`findClass()` 方法是在 `loadClass()` 方法中被调用的，当 `loadClass()` 方法中父加载器加载失败后，则会调用自己的 `findClass()` 方法来完成类加载，这样就可以保证自定义的类加载器也符合双亲委派机制

需要注意的是 `ClassLoader` 类中并没有实现 `findClass()` 方法的具体代码逻辑，取而代之的是抛出 `ClassNotFoundException` 异常，同时应该知道的是 `findClass()` 方法通常是和 `defineClass()` 方法一起使用的。一般情况下，在自定义类加载器时，会直接覆盖 `ClassLoader` 的 `findClass()` 方法并编写加载规则，取得要加载类的字节码后转换成流，然后调用 `defineClass()` 方法生成类的 `Class` 对象

➤ `protected final Class<?> defineClass(String name, byte[] b, int off, int len)`

根据给定的字节数组 `b` 转换为 `Class` 的实例，`off` 和 `len` 参数表示实际 `Class` 信息在 `byte` 数组中的位置和长度，其中 `byte` 数组 `b` 是 `ClassLoader` 从外部获取的。这是受保护的方法，只有在自定义 `ClassLoader` 子类中可以使用

`defineClass()` 方法是用来将 `byte` 字节流解析成 JVM 能够识别的 `Class` 对象(`ClassLoader` 中已实现该方法逻辑)，通过这个方法不仅能够通过 `Class` 文件实例化 `Class` 对象，也可以通过其它方式实例化 `Class` 对象，如通过网络中接收一个类的字节码，然后转换为 `byte` 字节流创建对应的 `Class` 对象

`defineClass()` 方法通常与 `findClass()` 方法一起使用，一般情况下，在自定义类加载器时，会直接覆盖 `ClassLoader` 的 `findClass()` 方法并编写加载规则，取

得要加载类的字节码后转换成流，然后调用 `defineClass()` 方法生成类的 `Class` 对象

简单举例：

```
protected Class<?> findClass(String name) throws ClassNotFoundException {  
    //获取类的字节数组  
    byte[] classData = getClassData(name);  
    if (classData == null) {  
        throw new ClassNotFoundException();  
    } else {  
        //使用 defineClass 生成 Class 对象  
        return defineClass(name, classData, 0, classData.length);  
    }  
}
```

➤ `protected final void resolveClass(Class<?> c)`

链接指定的一个 Java 类。使用该方法可以使用类的 `Class` 对象创建完成的同时也被解析。前面我们说链接阶段主要是对字节码进行验证，为类变量分配内存并设置初始值同时将字节码文件中的符号引用转换为直接引用

➤ `protected final Class<?> findLoadedClass(String name)`

查找名称为 `name` 的已经被加载过的类，返回结果为 `java.lang.Class` 类的实例。这个方法是 `final` 方法，无法被修改

➤ `private final ClassLoader parent;`

它也是一个 `ClassLoader` 的实例，这个字段所表示的 `ClassLoader` 也称为这个 `ClassLoader` 的双亲。在类加载的过程中，`ClassLoader` 可能会将某些请求交予自

己的双亲处理

① loadClass() 的剖析

测试代码：

```
ClassLoader.getSystemClassLoader().loadClass("com.atguigu.java.User")
```

```
/**
 * Loads the class with the specified binary name. The
 *
 * default implementation of this method searches for classes in the
 *
 * following order:
 *
 *
 * <ol>
 *
 *
 * <li><p> Invoke the findLoadedClass\(String\) to check if the class
 *
 * has already been loaded. </p></li>
 *
 *
 *
 * <li><p> Invoke the loadClass\(String\) loadClass} method
 *
 * on the parent class loader. If the parent is null the class
 *
 * loader built into the virtual machine is used, instead. </p></li>
 *
 *
 *
 * <li><p> Invoke the findClass\(String\) method to find the
 *
 * class. </p></li>
 *
 *
 *
 *
```

```

* </ol>

*

* <p> If the class was found using the above steps, and the

* {@code resolve} flag is true, this method will then invoke the {@link

* #resolveClass(Class)} method on the resulting {@code Class} object.

*

* <p> Subclasses of {@code ClassLoader} are encouraged to override {@link

* #findClass(String)}, rather than this method. </p>

*

* <p> Unless overridden, this method synchronizes on the result of

* {@link #getClassLoadingLock getClassLoadingLock} method

* during the entire class loading process.

*

* @param name

*         The <a href="#binary-name">binary name</a> of the class

*

* @param resolve

*         If {@code true} then resolve the class

*

* @return The resulting {@code Class} object

*

* @throws ClassNotFoundException

*         If the class could not be found

```



```
*/

protected Class<?> loadClass(String name, boolean resolve) //resolve:true 加载 Class 的
时进行解析操作

    throws ClassNotFoundException

{

    synchronized (getClassLoadingLock(name)) { //同步操作，保证只能加载一次

        // 首先，在缓存中判断是否已经加载同名类

        // First, check if the class has already been loaded

        Class<?> c = findLoadedClass(name);

        if (c == null) {

            long t0 = System.nanoTime();

            try {

                // 获取当前类加载器的父类加载器

                if (parent != null) {

                    // 如果存在父类加载器，则调用父类加载器进行类加载

                    c = parent.loadClass(name, false);

                } else { // parent 为 null : 父类加载器是引导类加载器

                    c = findBootstrapClassOrNull(name);

                }

            } catch (ClassNotFoundException e) {

                // ClassNotFoundException thrown if class not found

                // from the non-null parent class loader

            }

        }

    }

}
```

```
        if (c == null) { // 当前类的加载器的父类加载未加载其此类 or 当前类的加载器未
加载此类

            // If still not found, then invoke findClass in order

            // to find the class.

            long t1 = System.nanoTime();

            c = findClass(name);

            // this is the defining class loader; record the stats

            PerfCounter.getParentDelegationTime().addTime(t1 - t0);

            PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);

            PerfCounter.getFindClasses().increment();

        }

    }

    if (resolve) { //是否进行解析操作

        resolveClass(c);

    }

    return c;

}

}
```

2. SecureClassLoader 与 URLClassLoader

接着 SecureClassLoader 扩展了 ClassLoader, 新增了几个与使用相关的代码

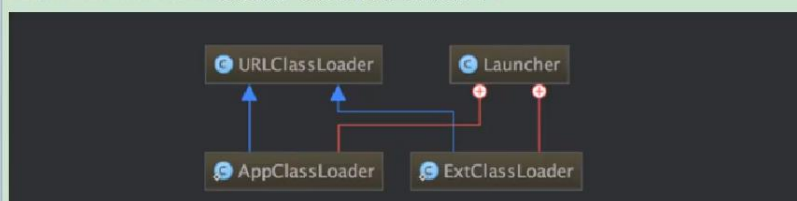
源(对代码源的位置及其证书的验证)和权限定义类验证(主要针对 Class 源码的访问权限)的方法，一般我们不会直接跟这个类打交道，更多的是与它的子类 `URLClassLoader` 有所关联

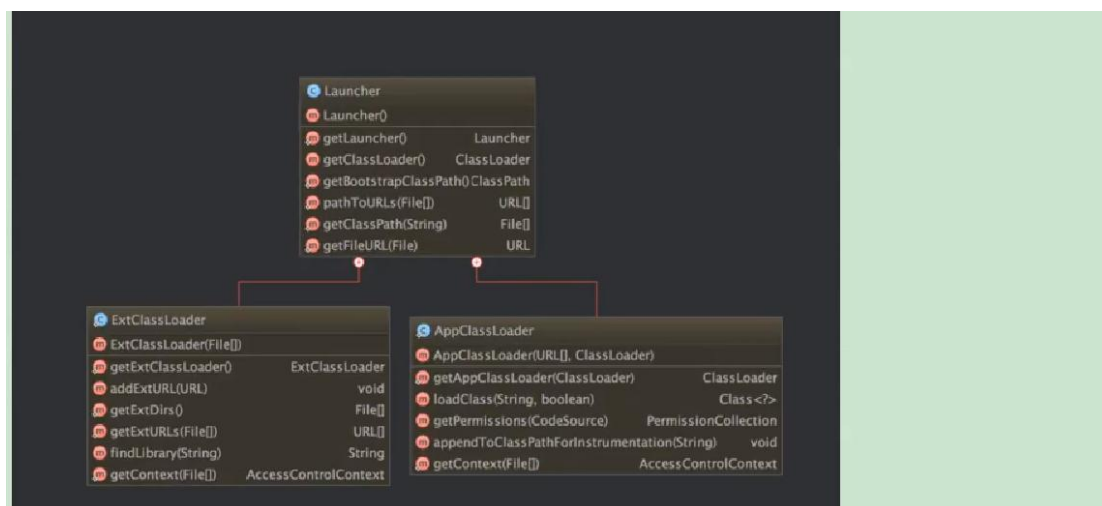
前面说过，`ClassLoader` 是一个抽象类，很多方法是空的没有实现，比如 `findClass()`、`findResource()` 等。而 `URLClassLoader` 这个实现类为这些方法提供了具体的实现。并新增了 `URLClassPath` 类协助取得 Class 字节码流等功能。在编写自定义类加载器时，如果没有太过于复杂的需求，可以直接继承 `URLClassLoader` 类，这样就可以避免自己去编写 `findClass()` 方法及其获取字节码流的方式，使自定义类加载器编写更加简洁

3. `ExtClassLoader` 与 `AppClassLoader`

`ExtClassLoader` 并没有重写 `loadClass()` 方法，这足以说明其遵循双亲委派模式，而 `AppClassLoader` 重载了 `loadClass()` 方法，但最终调用的还是父类 `loadClass()` 方法，因此依然遵循双亲委派模式

了解完 `URLClassLoader` 后接着看看剩余的两个类加载器，即拓展类加载器 `ExtClassLoader` 和系统类加载器 `AppClassLoader`，这两个类都继承自 `URLClassLoader`，是 `sun.misc.Launcher` 的静态内部类。`sun.misc.Launcher` 主要被系统用于启动主应用程序，`ExtClassLoader` 和 `AppClassLoader` 都是由 `sun.misc.Launcher` 创建的，其类主要类结构如下：





4. Class.forName() 与 ClassLoader.loadClass()

- `Class.forName()`：是一个静态方法，最常用的是 `Class.forName(String className)`；根据传入的类的权限定名返回一个 `Class` 对象。****该方法在将 Class 文件加载到内存的同时，会执行类的初始化。****如：

`Class.forName("com.atguigu.java.HelloWorld");`

- `ClassLoader.loadClass()` 这是一个实例方法，需要一个 `ClassLoader` 对象来调用该方法。该方法将 `Class` 文件加载到内存时，并不会执行类的初始化，直到这个类第一次使用时才进行初始化。该方法因为需要得到一个 `ClassLoader` 对象，所以可以根据需要指定使用哪个类加载器，如：
`ClassLoader c1 =; c1.loadClass("com.atguigu.java.HelloWorld");`

第五节 双亲委派模型

1. 定义与本质

类加载器用来把类加载到 Java 虚拟机中。从 JDK 1.2 版本开始，类的加载

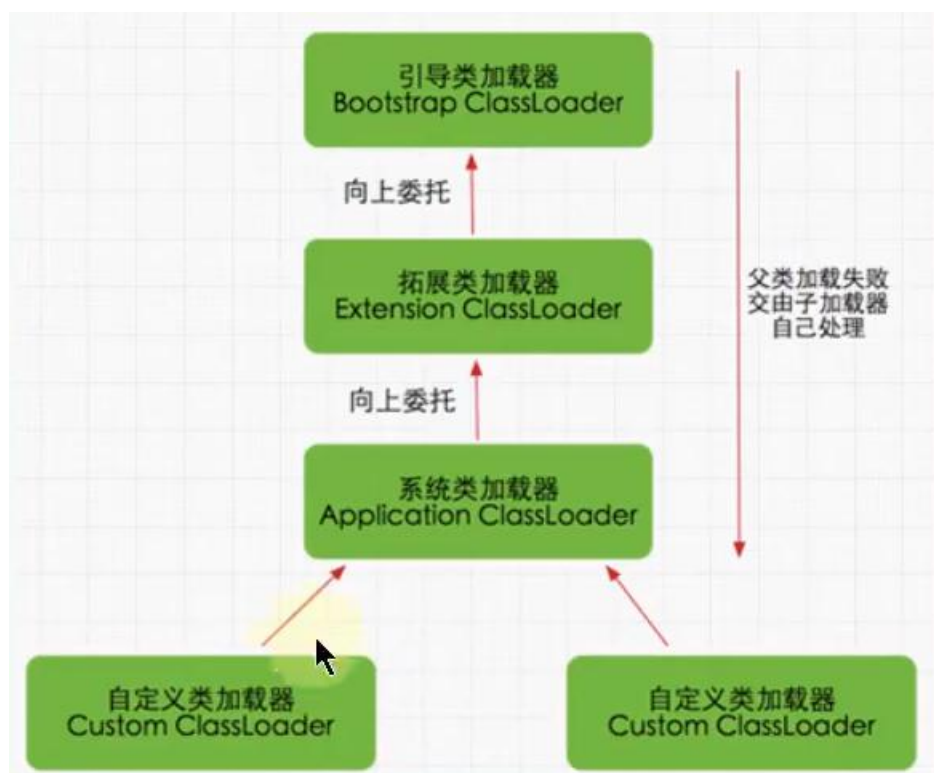
过程采用双亲委派机制，这种机制能更好地保证 Java 平台的安全

1. 定义

如果一个类加载器在接到加载类的请求时，它首先不会自己尝试去加载这个类，而是把这个请求任务委托给父类加载器去完成，依次递归，如果父类加载器可以完成类加载任务，就成功返回。只有父类加载器无法完成此加载任务时，才自己去加载

2. 本质

规定了类加载的顺序是：引导类加载器先加载，若加载不到，由扩展类加载器加载，若还加载不到，才会由系统类加载器或自定义的类加载器进行加载



2. 优势与劣势

1. 双亲委派机制优势

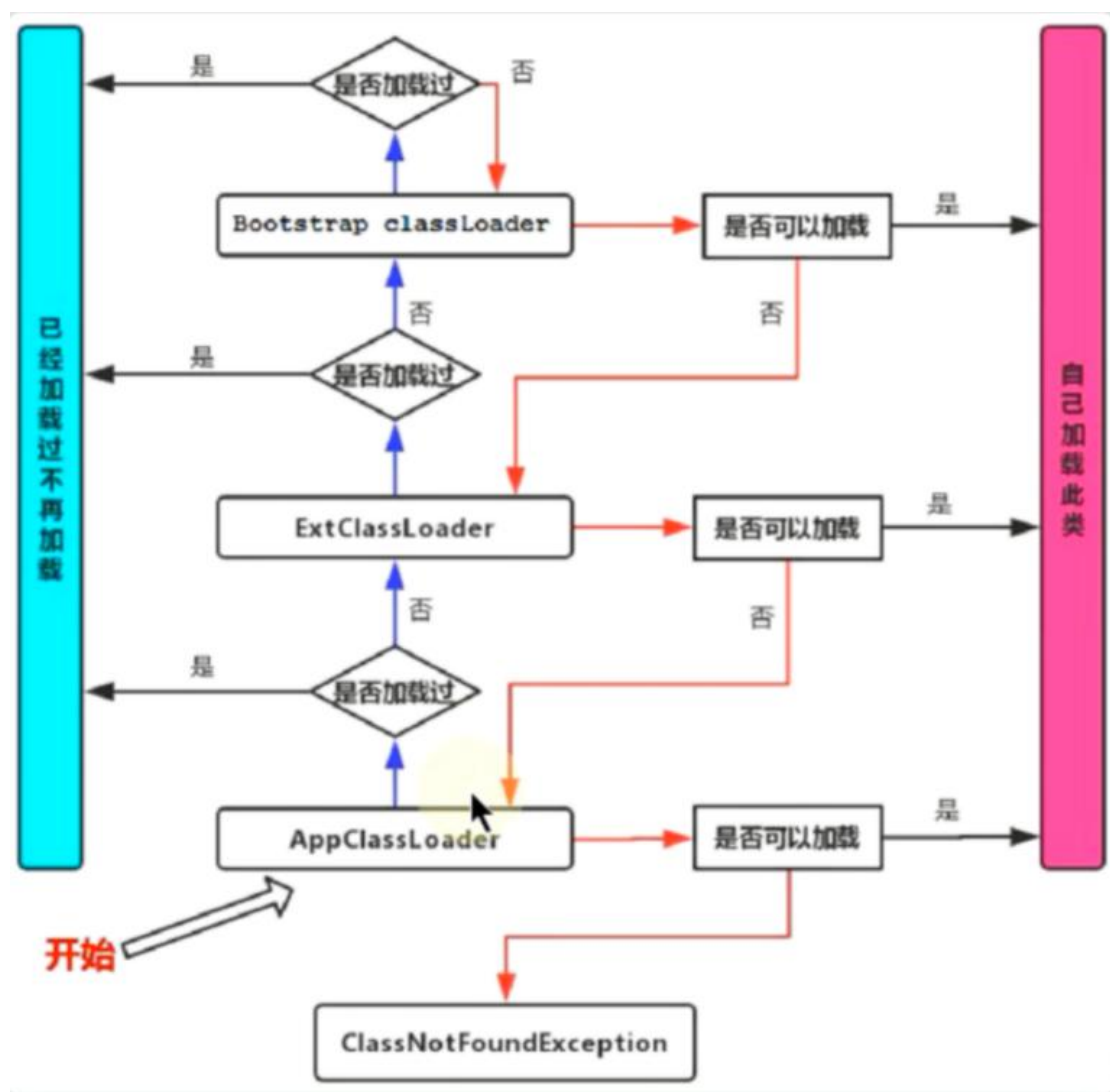
- 避免类的重复加载，确保一个类的全局唯一性

Java 类随着它的类加载器一起具备了一种带有优先级的层级关系，通过这种层级关系可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子 ClassLoader 再加载一次

- 保护程序安全，防止核心 API 被随意篡改

2. 代码支持

双亲委派机制在 `java.lang.ClassLoader.loadClass(String, boolean)` 接口中体现。该接口的逻辑如下：



先在当前加载器的缓存中查找有无目标类，如果有，直接返回

- 判断当前加载器的父加载器是否为空，如果不为空，则调用 `parent.loadClass(name, false)` 接口进行加载
- 反之，如果当前加载器的父类加载器为空，则调用 `findBootstrapClassOrNull(name)` 接口，让引导类加载器进行加载
- 如果通过以上 3 条路径都没能成功加载，则调用 `findClass(name)` 接口进行加载。该接口最终会调用 `java.lang.ClassLoader` 接口的 `defineClass` 系列的 `native` 接口加载目标 Java 类

双亲委派的模型就隐藏在第 2 和第 3 步中

3. 举例

假设当前加载的是 `java.lang.Object` 这个类，很显然，该类属于 JDK 中核心的不能再核心的一个类，因此一定只能由引导类加载器进行加载。当 JVM 准备加载 `java.lang.Object` 时，JVM 默认会使用系统类加载器去加载，按照上面 5 步加载的逻辑，在第 1 步从系统类的缓存中肯定查找不到该类，于是进入第 2 步。由于从系统类加载器的父类加载器是扩展类加载器，于是扩展类加载器继续从第 1 步开始重复。由于扩展类加载器的缓存中也一定查找不到该类，因此进入第 2 步。扩展类的父加载器是 `null`，因此系统调用 `findClass(String)`，最终通过引导类加载器进行加载

4. 思考

如果在自定义的类加载器中重写 `java.lang.ClassLoader.loadClass(String)` 或 `java.lang.ClassLoader.loadClass(String, boolean)` 方法，抹去其中的双亲委派机制，仅保留上面这 4 步中的第 1 步和第 4 步，那么是不是就能够加载核心类库了呢？

这也不行！因为 JDK 还为核心类库提供了一层保护机制。不管是自定义的类加载器，还是系统类加载器抑或扩展类加载器，最终都必须调用 `java.lang.ClassLoader.defineClass(String, byte[], int, int, ProtectionDomain)` 方法，而该方法会执行 `preDefineClass()` 接口，该接口中提供了对 JDK 核心类库的保

护

5. 双亲委派模式的弊端

检查类是否加载的委派过程是单向的，这个方式虽然从结构上说比较清晰，使各个 `ClassLoader` 的职责非常明确，但是同时会带来一个问题，即顶层的 `ClassLoader` 无法访问底层的 `ClassLoader` 所加载的类

通常情况下，启动类加载器中的类为系统核心类，包括一些重要的系统接口，而在应用类加载器中，为应用类。按照这种模式，**应用类访问系统类自然是没有问题，但是系统类访问应用类就会出现问题。**比如在系统类中提供了一个接口，该接口需要在应用类中得以实现，该接口还绑定一个工厂方法，用于创建该接口的实例，而接口和工厂方法都在启动类加载器中。这时，就会出现该工厂方法无法创建由应用类加载器加载的应用实例的问题

6. 结论

由于 Java 虚拟机规范并没有明确要求类加载器的加载机制一定要使用双亲委派模型，只是建议采用这种方式而已。比如 Tomcat 中，类加载器所采用的加载机制就和传统的双亲委派模型有一定区别，当缺省的类加载器接收到一个类的加载任务时，首先会由它自行加载，当它加载失败时，才会将类的加载任务委派给它的超类加载器去执行，这同时也是 Servlet 规范推荐的一种做法

3. 破坏双亲委派机制

① 破坏双亲委派机制 1

双亲委派模型并不是一个具有强制性约束的模型，而是 Java 设计者推荐给开发者们的类加载器实现方式

在 Java 的世界中大部分的类加载器都遵循这个模型，但也有例外情况，直到 Java 模块化出现为止，双亲委派模型主要出现过 3 次较大规模"被破坏"的情况

第一次破坏双亲委派机制：

双亲委派模型的第一次"被破坏"其实发生在双亲委派模型出现之前——即 JDK 1.2 面世以前的"远古"时代

由于双亲委派模型在 JDK 1.2 之后才被引入,但是类加载器的概念和抽象类 `java.lang.ClassLoader` 则在 Java 的第一个版本中就已经存在,面对已经存在的用户自定义类加载器的代码,Java 设计者们引入双亲委派模型时不得不做出一些妥协,为了兼容这些已有的代码,无法再以技术手段避免 `loadClass()` 被子类覆盖的可能性,只能在 JDK 1.2 之后的 `java.lang.ClassLoader` 中添加一个新的 `protected` 方法 `findClass()`,并引导用户编写的类加载逻辑时尽可能去重写这个方法,而不是在 `loadClass()` 中编写代码。上节我们已经分析过 `loadClass()` 方法,双亲委派的具体逻辑就实现在这里面,按照 `loadClass()` 方法的逻辑,如果父类加载失败,会自动调用自己的 `findClass()` 方法来完成加载,这样既不影响用户按照自己的意愿去加载类,又可以保证新写出来的类加载器是符合双亲委派规则的

② 破坏双亲委派机制 2

第二次破坏双亲委派机制:线程上下文类加载器

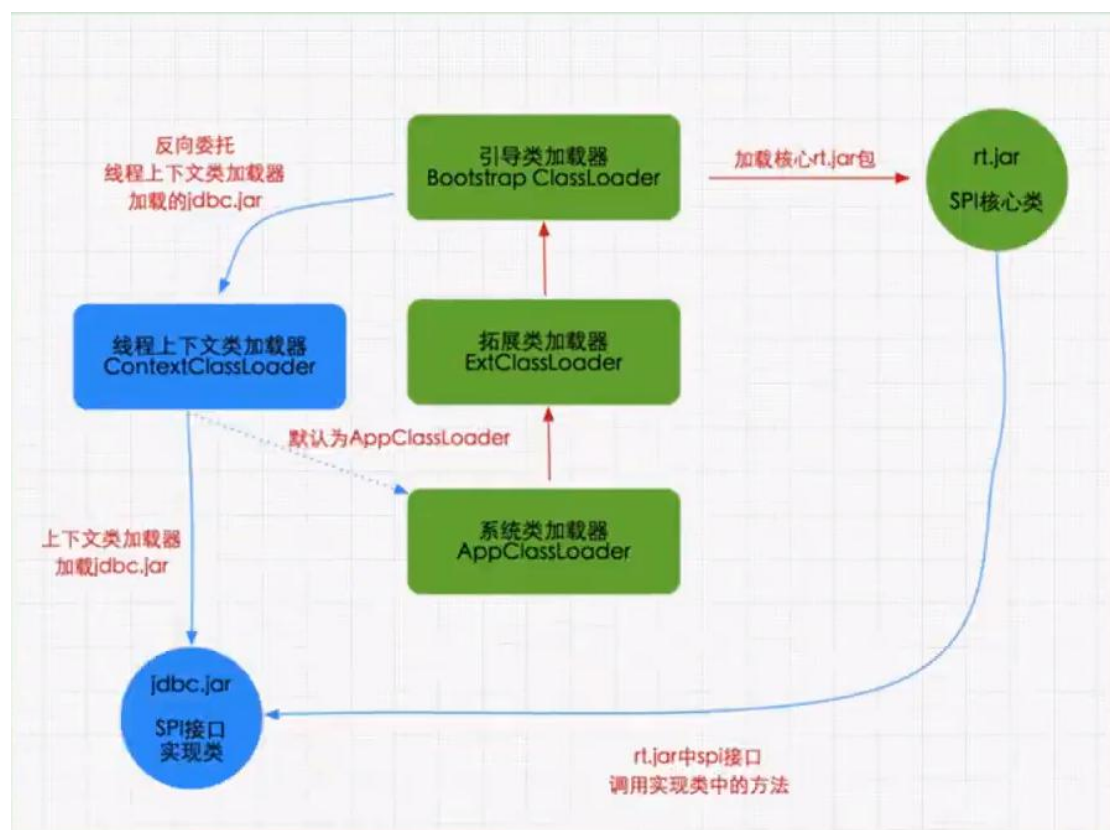
双亲委派模型的第二次"被破坏"是由这个模型自身的缺陷导致的,双亲委派很好地解决了各个类加载器协作时基础类型的一致性问题(越基础的类由越上层的加载器进行加载),基础类型之所以被称为"基础",是因为它们总是作为被用户代码继承、调用的 API 存在,但程序设计往往没有绝对不变的完美规则,如果有基础类型又要调用回用户代码,那该怎么办?

这并非是不可能出现的事情,一个典型的例子便是 JNDI 服务,JNDI 现在已经是 Java 的标准服务,它的代码由启动类加载器来完成加载(在 JDK 1.3 时加入到 `rt.jar`),肯定属于 Java 中很基础的类型了。但 JNDI 存在的目的就是ForResource 进行查找和集中管理,它需要调用由其它厂商实现并部署在应用程序的 `ClassPath` 下的 JNDI 服务提供者接口(Service Provider Interface. SPI)的代码,现在问题来了,启动类加载器时绝对不可能认识、加载这些代码的,那该怎么办?(SPI:在 Java 平台中,通常把核心类 `rt.jar` 中提供外部服务、可由应用层自行

实现的接口称为 SPI)

为了解决这个困境，Java 的设计团队只好引入了一个不太优雅的设计：****线程上下文类加载器(Thread Context ClassLoader)****。这个类加载器可以通过 `java.lang.Thread` 类的 `setContextClassLoader()` 方法进行设置，如果创建线程时还未设置，它将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过的话，那这个类加载器默认就是应用程序类加载器

有了线程上下文类加载器，程序就可以做一些"舞弊"的事情了。JNDI 服务使用这个线程上下文类加载器去加载所需的 SPI 服务代码。这是一种负累加载器去请求子类加载器完成类加载的行为，这种行为实际上是打通了双亲委派模型的层次结构来逆向使用类加载器，已经违背了双亲委派模型的一般性原则，但也是无可奈何的事情。Java 中涉及 SPI 的加载基本上都采用这种方式来完成，例如 JNDI、JDBC、JCE、JAXB 和 JBI 等。不过，当 SPI 的服务提供者多于一个的时候，代码就只能根据具体提供者的类型来硬编码判断，为了消除这种极不优雅的方式，在 JDK 6 时，JDK 提供了 `java.util.ServiceLoader` 类，以 `META-INF/Services` 中的配置信息，辅以责任链模式，这才算是给 SPI 的加载提供了一种相对合理的解决方案



默认上下文加载器就是应用类加载器，这样以上下文加载器为中介，使得启动类加载器中的代码也可以访问应用类加载器中的类

③ 破坏双亲委派机制 3

第三次破坏双亲委派机制：

双亲委派模型的第三次"被破坏"是由于用户对程序动态性的追求而导致的。
如：代码热替换(Hot Swap)、**模块热部署(Hot Deployment)**等

IBM 公司主导的 JSR-291(即 OSGI R4.2)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(OSGI 中称为 Bundle)都有一个自己的类加载器，当需要更换一个 Bundle 时，就把 Bundle 连同类加载器一起换掉以实现代码的热替换。在 OSGI 环境下，类加载器不再双亲委派模型推荐的树状结构，而是进一步发展为更加复杂的网状结构

当收到类加载请求时，OSGI 将按照下面的顺序进行类搜索：

-
- 1) 将以 `java.` 开头的类，委派给父类加载器加载*
 - 2) 否则，将委派列表名单内的类，委派给父类加载器加载
 - 3) 否则，将 `Import` 列表中的类，委派给 `Export` 这个类的 `Bundle` 的类加载器加载
 - 4) 否则，查找当前 `Bundle` 的 `ClassPath`，使用自己的类加载器加载
 - 5) 否则，查找类是否在自己的 `Fragment Bundle` 中，如果在，则委派给 `Fragment Bundle` 的类加载器加载
 - 6) 否则，查找 `Dynamic Import` 列表的 `Bundle`，委派给对应 `Bundle` 的类加载器加载
 - 7) 否则，类查找失败
 - 8) 说明：只有开头两点仍然符合双亲委派模型的原则，其余的类查找都是在平级的类加载器中进行的

小结：

这里，我们使用了"被破坏"这个词来形容上述不符合双亲委派模型原则的行为，但这里"被破坏"并不一定是带有贬义的。只要有明确的目的和充分的理由，突破旧有原则无疑是一种创新

正如：`OSGI` 中的类加载器的设计不符合传统的双亲委派的类加载器架构，且业界对其为了实现热部署而带来的额外的高复杂度还存在不少争议，但对这方面有了解的技术人员基本还是能达成一个共识，认为 `OSGI` 中对类加载器的运用是值得学习的，完全弄懂了 `OSGI` 的实现，就算是掌握了类加载器的精髓

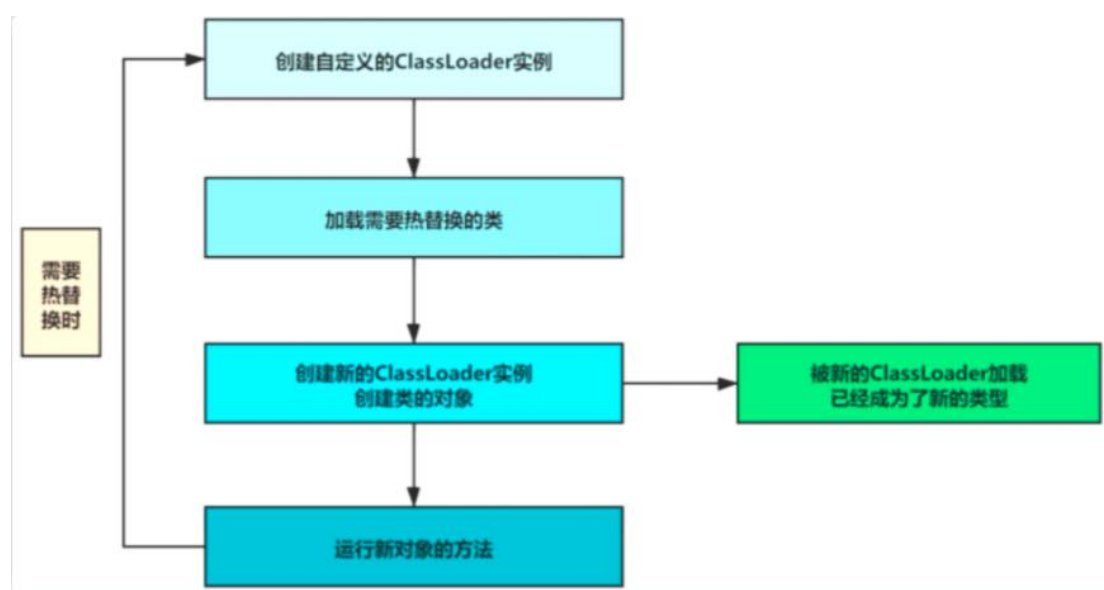
4. 热替换的实现

热替换是指在程序运行过程中，不停止服务，只通过替换程序文件来修改程序的行为。热替换的关键需求在于服务不能中断，修改必须立即表现正在运行的系统之中。基本上大部分脚本语言都是天生支持热替换的，比如：`PHP`，只要替换了 `PHP` 源文件，这种改动就会立即生效，而无需重启 `Web` 服务器

但对 Java 来说，热替换并非天生就支持，如果一个类已经加载到系统中，通过修改类文件，并无法让系统再来加载并重定义这个类。因此，在 Java 中实现这一功能的一个可行的方法就是灵活运用 ClassLoader

注意：由不同 ClassLoader 加载的同名类属于不同的类型，不能相互转换和兼容。即两个不同的 ClassLoader 加载同一个类，在虚拟机内部，会认为这 2 个类是完全不同的

根据这个特点，可以用来模拟热替换的实现，基本思路如下图所示：



第六节 沙箱安全机制

- 保护程序安全
- 保护 Java 原生的 JDK 代码

Java 安全模型的核心就是 Java 沙箱(Sandbox)，什么是沙箱？沙箱就是一个限制程序运行的环境

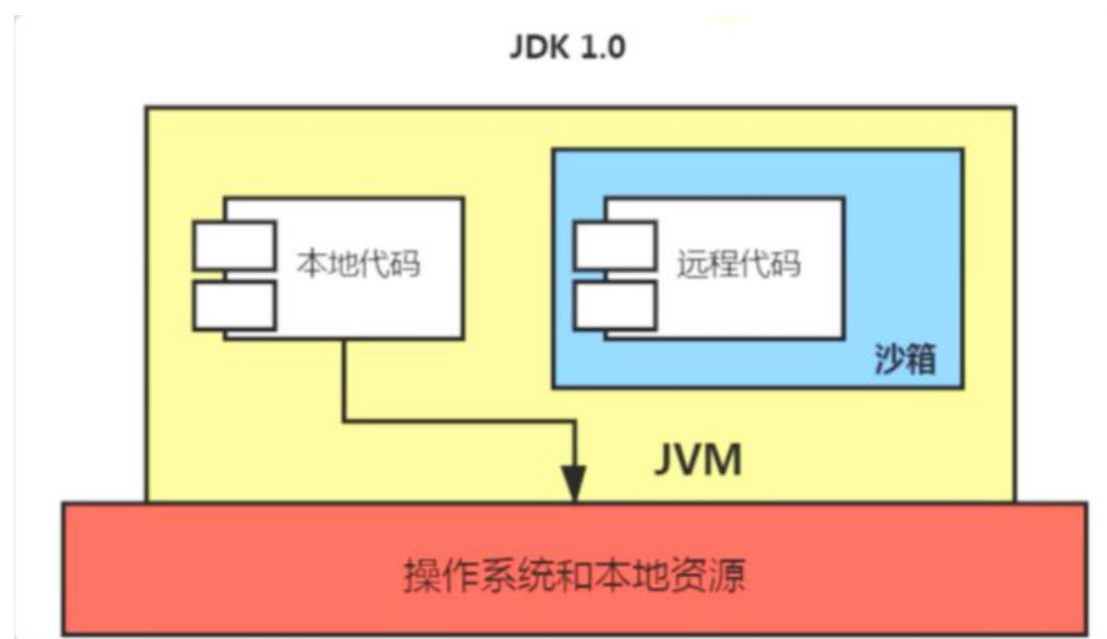
沙箱机制就是将 Java 代码**限定在虚拟机(JVM)特定的运行范围中，并且严格限制代码对本地系统资源访问。**通过这样的措施来保证对代码的有限隔离，防止对本地系统造成破坏

沙箱主要限制系统资源访问，那系统资源包括什么？CPU、内存、文件系统、网络。不同级别的沙箱对这些资源访问的限制也可以不一样

所有的 Java 程序运行都可以指定沙箱，可以定制安全策略

1. JDK 1.0 时期

在 Java 中将执行程序分成本地代码和远程代码两种，本地代码默认视为可信任的，而远程代码则被看作是不受信的。对于授信的本地代码，可以访问一切本地资源。而对于非授信的远程代码在早期的 Java 实现中，安全依赖于沙箱 (Sandbox) 机制。如下图所示 JDK 1.0 安全模型

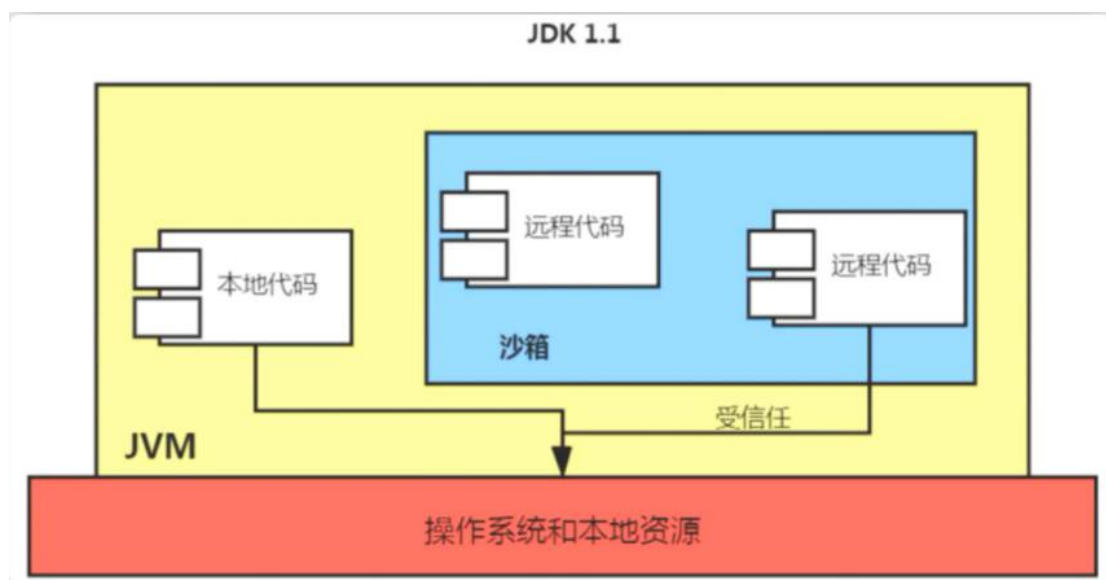


2. JDK 1.1 时期

JDK 1.0 中如此严格的安全机制也给程序的功能扩展带来障碍，比如当用户希望远程代码访问本地系统的文件时候，就无法实现

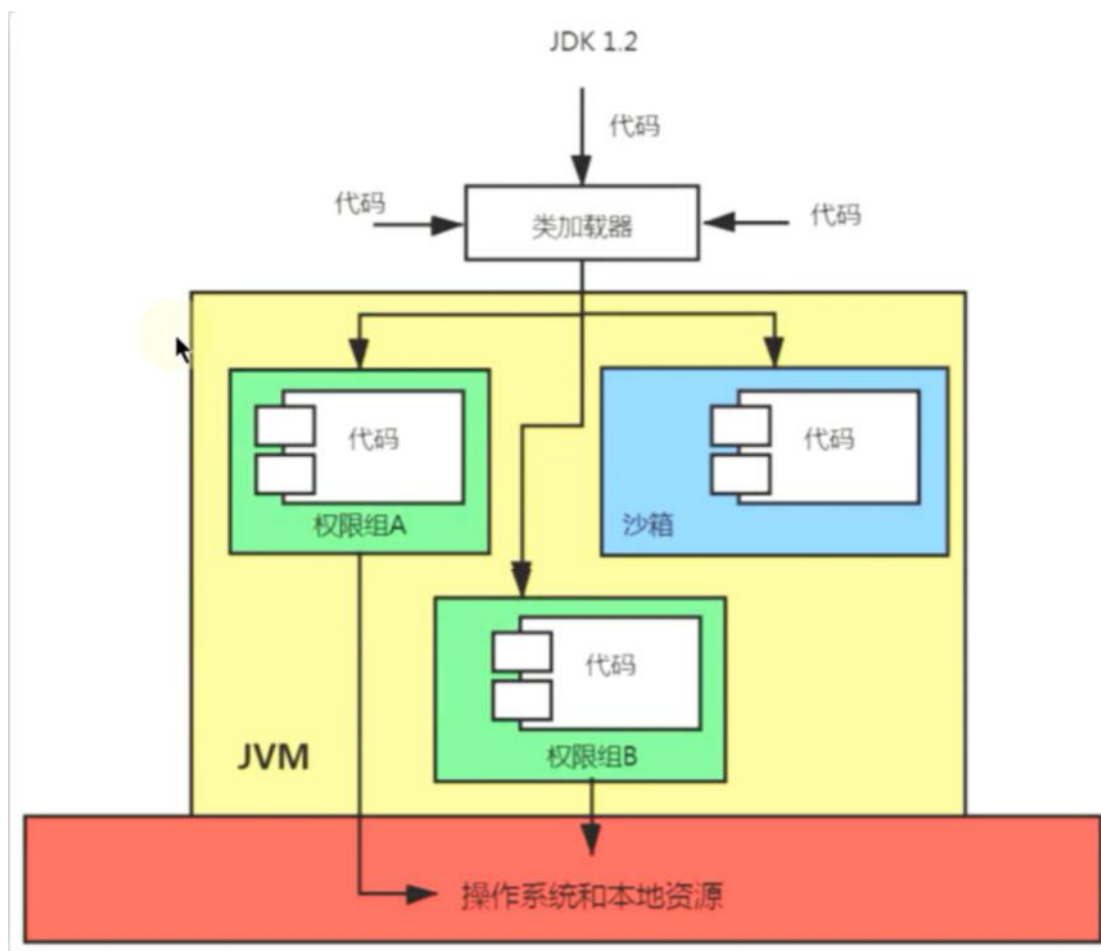
因此在后续的 JDK 1.1 版本中，针对安全机制做了改进，增加了安全策略。允许用户指定代码对本地资源的访问权限

如下图所示 JDK 1.1 安全模型



3. JDK 1.2 时期

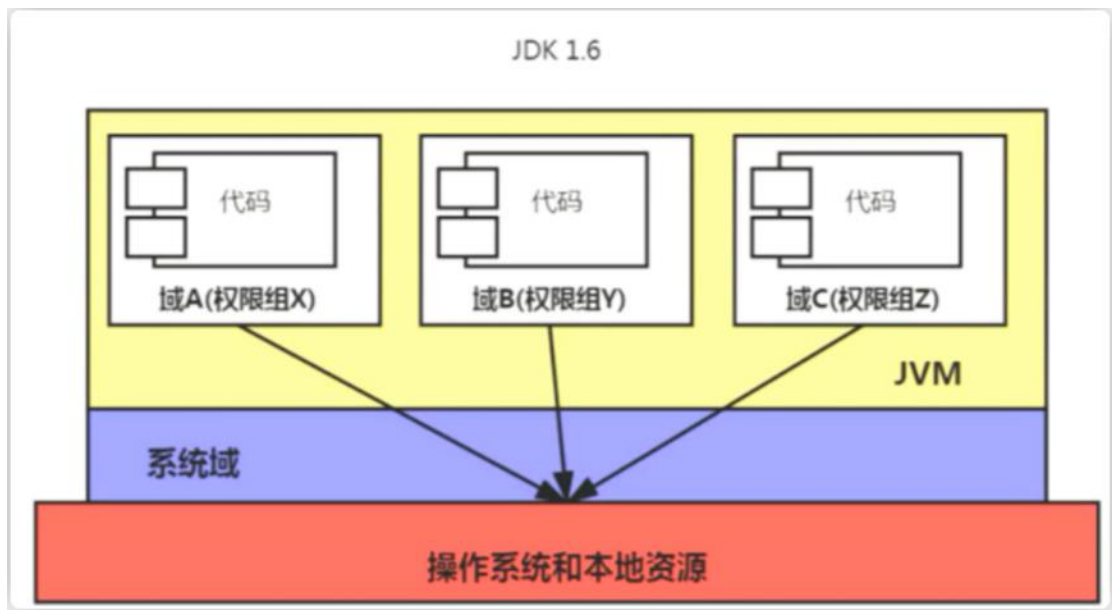
在 JDK 1.2 版本中，再次改进了安全机制，增加了代码签名。不论本地代码或是远程代码，都会按照用户的安全策略设定，由类加载器加载到虚拟机中权限不同的运行空间，来实现差异化的代码执行权限控制。如下图所示 JDK 1.2 安全模型：



4. JDK 1.6 时期

当前最新的安全机制实现，则引入了**域(Domain)**的概念

虚拟机会把所有代码加载到不同的系统域和应用域。系统域部分专门负责与关键资源进行交互，而各个应用域部分则通过系统域的部分代理来对各种需要的资源进行访问。虚拟机中不同的受保护域(Protected Domain)，对应不一样的权限(Permission)。存在于不同域中的类文件就具有了当前域的全部权限，如下图所示，最新的安全模型(JDK 1.6)



第七节 自定义类的加载器

1. 为什么要自定义类加载器？

➤ 隔离加载类

在某些框架内进行中间件与应用的模块隔离，把类加载到不同的环境。比如：阿里内某容器框架通过自定义类加载器确保应用中依赖的 jar 包不会影响到中间件运行时使用的 jar 包。再比如：Tomcat 这类 Web 应用服务器，内部自定义了好几种类加载器，用于隔离同一个 Web 应用服务器上的不同应用程序。(类的仲裁 --> 类冲突)

➤ 修改类加载的方式

类的加载模型并非强制，除 Bootstrap 外，其他的加载并非一定要引入，或者根据实际情况在某个时间点按需进行动态加载

➤ 扩展加载源

比如从数据库、网络、甚至是电视机机顶盒进行加载

➤ 防止源码泄露

Java 代码容易被编译和篡改, 可以进行编译加密。那么类加载也需要自定义, 还原加密的字节码

2. 常见的场景

实现类似进程内隔离, 类加载器实际上用作不同的命名空间, 以提供类似容器、模块化的效果。例如, 两个模块依赖于某个类库的不同版本, 如果分别被不同的容器加载, 就可以互不干扰。这个方面的集大成者是 Java EE 和 OSGI、JPMS 等框架

应用需要从不同的数据源获取类定义信息, 例如网络数据源, 而不是本地文件系统。或者是需要自己操纵字节码, 动态修改或者生成类型

3. 注意

在一般情况下, 使用不同的类加载器去加载不同的功能模块, 会提高应用程序的安全性。但是, 如果涉及 Java 类型转换, 则加载器反而容易产生不美好的事情。在做 Java 类型转换时, 只有两个类型都是由同一个加载器所加载, 才能进行类型转换, 否则转换时会发生异常

1. 实现方式

用户通过定制自己的类加载器, 这样可以重新定义类的加载规则, 以便实现一些自定义的处理逻辑

1. 实现方式

- Java 提供了抽象类 `java.lang.ClassLoader`, 所有用户自定义的类加载器都应该继承 `ClassLoader` 类
- 在自定义 `ClassLoader` 的子类时候, 我们常见的会有两种做法:
 - 方式一: 重写 `loadClass()` 方法
 - 方式二: 重写 `findClass()` 方法

2. 对比

这两种方法本质上差不多, 毕竟 `loadClass()` 也会调用 `findClass()`, 但是从

逻辑上讲我们最好不要直接修改 `loadClass()` 的内部逻辑。建议的做法是只在 `findClass()` 里重写自定义类的加载方法，根据参数指定类的名字，返回对应的 `Class` 对象的引用

- `loadClass()` 这个方法是实现双亲委派模型逻辑的地方，擅自修改这个方法会导致模型被破坏，容易造成问题。因此我们最好是在双亲委派模型框架内进行小范围的改动，不破坏原有的稳定结构。同时，也避免了自己重写 `loadClass()` 方法的过程中必须写双亲委托的重复代码，从代码的复用性来看，不直接修改这个方法始终是比较好的选择
- 当编写好自定义类加载器后，便可以在程序中调用 `loadClass()` 方法来实现类加载操作

3. 说明

- 其父类加载器是系统类加载器
- JVM 中的所有类加载都会使用 `java.lang.ClassLoader.loadClass(String)` 接口 (自定义类加载器并重写 `java.lang.ClassLoader.loadClass(String)` 接口的除外)，连 JDK 的核心类库也不能例外

第八节 Java 9 新特性

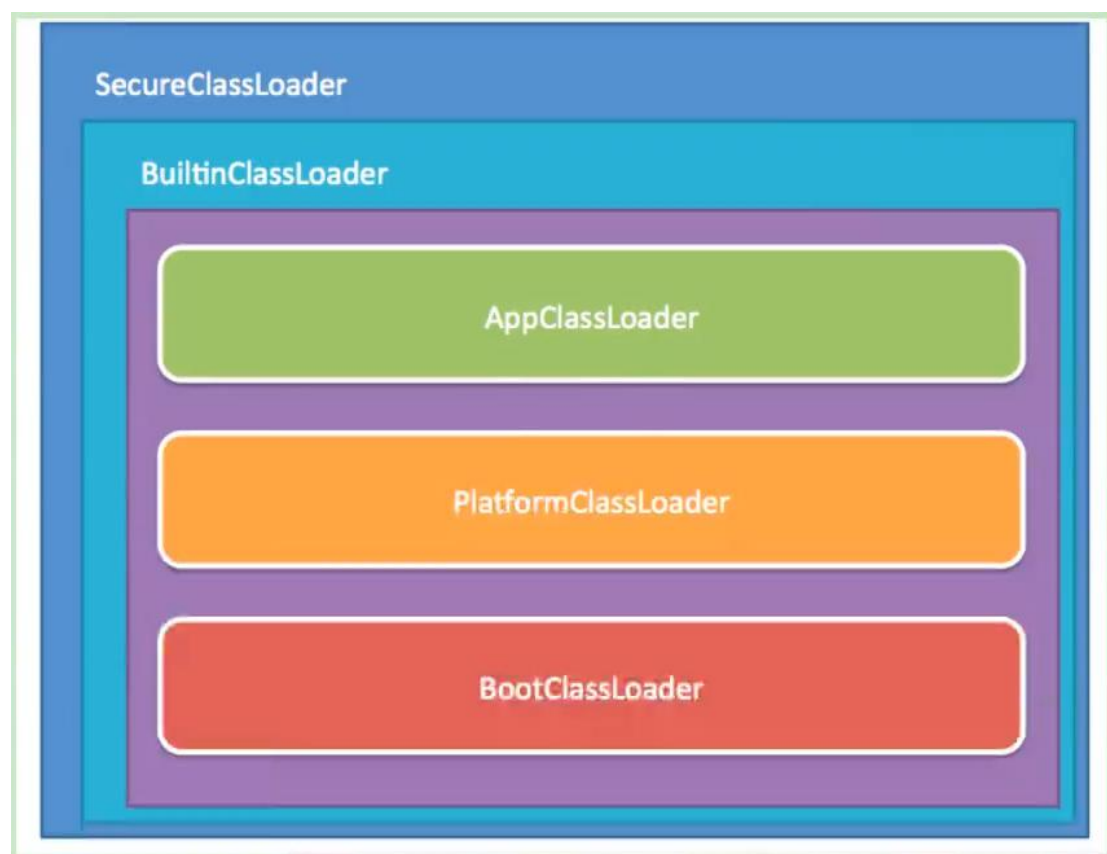
为了保证兼容性，JDK 9 没有从根本上改变三层类加载器架构和双亲委派模型，但为了模块化系统的顺利运行，仍然发生了一些值得被注意的变动

1. 扩展机制被移除，扩展类加载器由于向后兼容性的原因被保留，不过被重命名为平台类加载器(Platform Class Loader)。可以通过 `ClassLoader` 的新方法 `getPlatformClassLoader()` 来获取

JDK 9 时基于模块化进行构建(原来的 `rt.jar` 和 `tools.jar` 被拆分成数十个 JMOD 文件)，其中的 Java 类库就已天然地满足了可扩展的需求，那自然无需再保留 `<JAVA_HOME>\lib\ext` 目录，此前使用这个目录或者 `java.ext.dirs` 系统变量来扩展 JDK 功能的机制已经没有继续存在的价值了

2. 平台类加载器和应用程序类加载器都不再继承自 `java.net.URLClassLoader`

现在启动类加载器、平台类加载器、应用程序类加载器全都继承于 `jdk.internal.loader.BuiltinClassLoader`

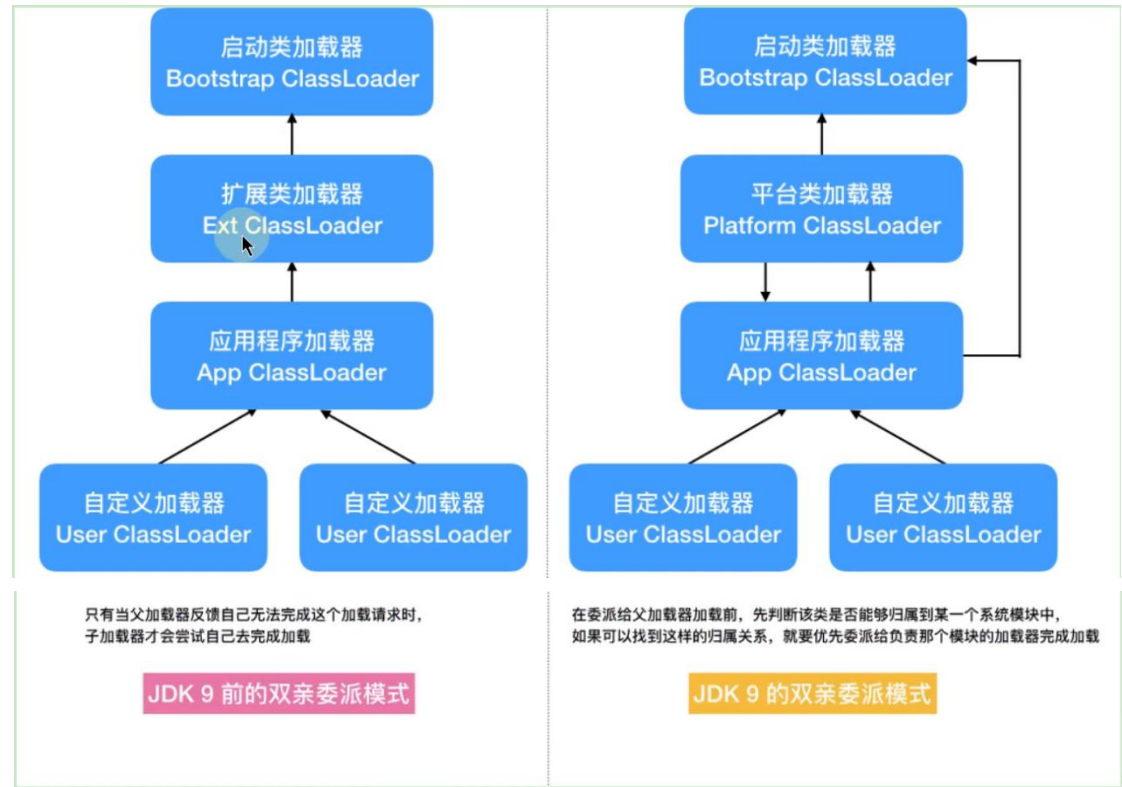


如果有程序直接依赖了这种继承关系，或者依赖了 `URLClassLoader` 类的特定方法，那代码很可能会在 **JDK 9** 及更高版本的 **JDK** 中崩溃

3. 在 **Java 9** 中，类加载器有了名称。该名称在构造方法中指定，可以通过 `getName()` 方法来获取。平台类加载器的名称是 **Platform**，应用类加载器的名称是 **App**。类加载器的名称在调试与类加载器相关的问题时会非常有用
4. 启动类加载器现在是在 **JVM** 内部和 **Java** 类库共同协作实现的类加载器 (以前是 **C++** 实现)，但为了与之前代码兼容，在获取启动类加载器的场景中仍然会返回 `null`，而不会得到 `BootClassLoader` 实例
5. 类加载的委派关系也发生了变动

当平台及应用程序类加载器收到类加载请求，在委派给父加载器加载前，要先判断该类是否能够归属到某一个系统模块中，如果可以找到这样的归属关系，就要优先委派给负责哪个模块的加载器完成加载

双亲委派模式示意图



附加：

在 Java 模块化系统明确规定了三个类加载器负责各自加载的模块：

- 启动类加载器负责加载的模块

java.base	java.security.sasl
java.datatransfer	java.xml
java.desktop	jdk.httpserver
java.instrument	jdk.internal.vm.ci
java.logging	jdk.management
java.management	jdk.management.agent
java.management.rmi	jdk.naming.rmi
java.naming	jdk.net
java.prefs	jdk.sctp
java.rmi	jdk.unsupported

- 平台类加载器负责加载的模块

java.activation*	jdk.accessibility
java.compiler*	jdk.charsets
java.corba*	jdk.crypto.cryptoki
java.scripting	jdk.crypto.ec
java.se	jdk.dynalink
java.se.se	jdk.incubator.httpclient
java.security.jgss	jdk.internal.vm.compiler*
java.smartcardio	jdk.jsobject
java.sql	jdk.localedata
java.sql.rowset	jdk.naming.dns
java.transaction*	jdk.scripting.nashorn

java.xml.bind*	jdk.security.auth
java.xml.crypto	jdk.security.jgss
java.xml.ws*	jdk.xml.dom
java.xml.ws.annotation*	jdk.zipfs

● 应用程序类加载器负责加载的模块

jdk.aot	jdk.jdeps
jdk.attach	jdk.jdi
jdk.compiler	jdk.jdwp.agent
jdk.editpad	jdk.jlink
jdk.hotspot.agent	jdk.jshell
jdk.internal.ed	jdk.jstatd