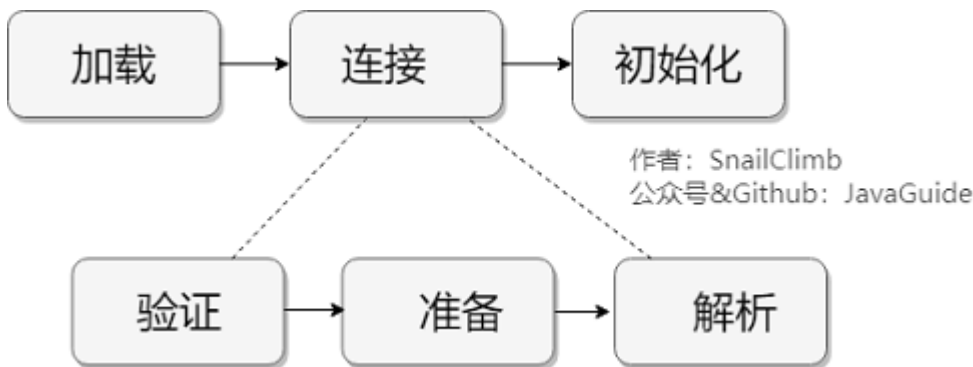


- [回顾一下类加载过程](#)
- [类加载器总结](#)
- [双亲委派模型](#)
 - [双亲委派模型介绍](#)
 - [双亲委派模型实现源码分析](#)
 - [双亲委派模型的好处](#)
 - [如果我们不想要双亲委派模型怎么办?](#)
- [自定义类加载器](#)
- [推荐](#)

公众号JavaGuide 后台回复关键字“1”，免费获取JavaGuide配套的Java工程师必备学习资源(文末有公众号二维码)。

回顾一下类加载过程

类加载过程：**加载**->**连接**->**初始化**。连接过程又可分为三步：**验证**->**准备**->**解析**。



一个非数组类的加载阶段（加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，这一步我们可以去完成还可以自定义类加载器去控制字节流的获取方式（重写一个类加载器的 `loadClass()` 方法）。数组类型不通过类加载器创建，它由 Java 虚拟机直接创建。

所有的类都由类加载器加载，加载的作用就是将 .class 文件加载到内存。

类加载器总结

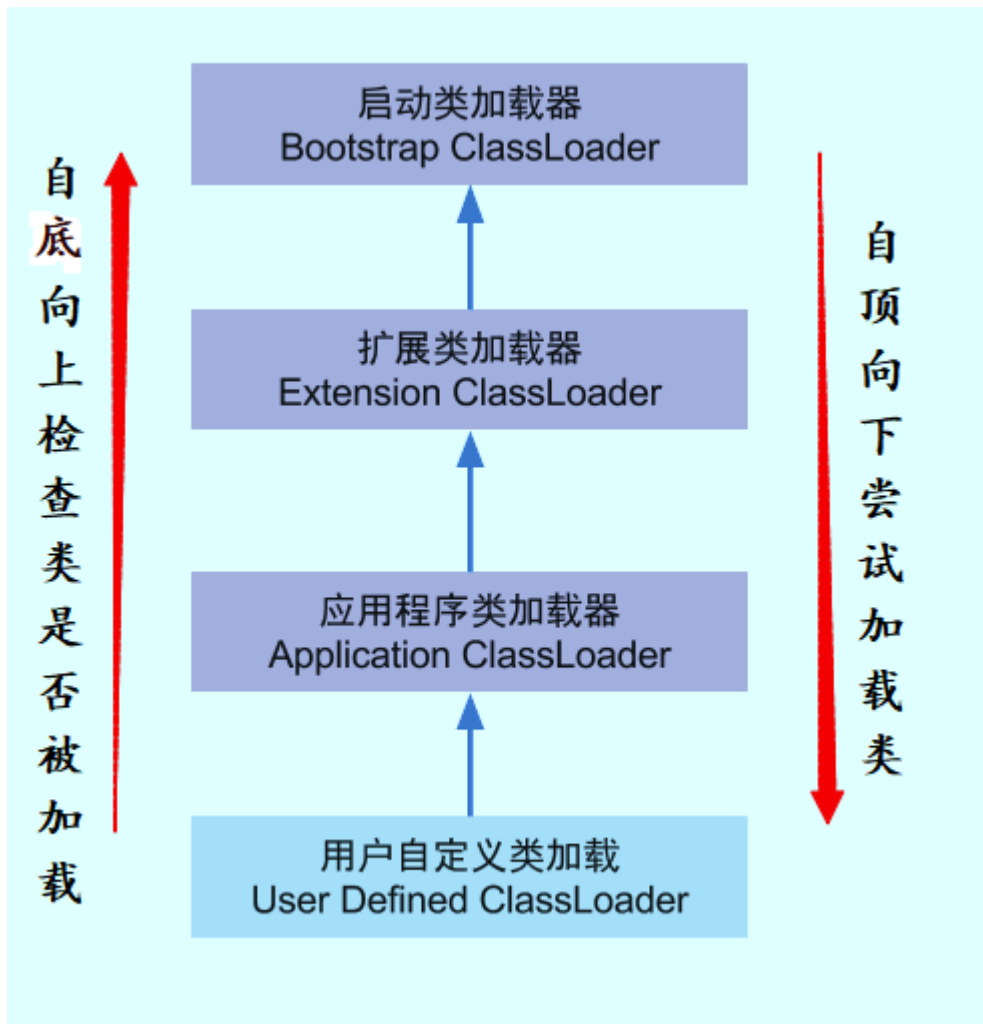
JVM 中内置了三个重要的 `ClassLoader`，除了 `BootstrapClassLoader` 其他类加载器均由 Java 实现且全部继承自 `java.lang.ClassLoader`：

1. **BootstrapClassLoader(启动类加载器)**：最顶层的加载类，由 C++ 实现，负责加载 `%JAVA_HOME%/lib` 目录下的 jar 包和类或者或被 `-Xbootclasspath` 参数指定的路径中的所有类。
2. **ExtensionClassLoader(扩展类加载器)**：主要负责加载目录 `%JRE_HOME%/lib/ext` 目录下的 jar 包和类，或被 `java.ext.dirs` 系统变量所指定的路径下的 jar 包。
3. **AppClassLoader(应用程序类加载器)**：面向我们用户的加载器，负责加载当前应用 classpath 下的所有 jar 包和类。

双亲委派模型

双亲委派模型介绍

每一个类都有一个对应它的类加载器。系统中的 ClassLoder 在协同工作的时候会默认使用 **双亲委派模型**。即在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载。加载的时候，首先会把该请求委派该父类加载器的 `loadClass()` 处理，因此所有的请求最终都应该传送到顶层的启动类加载器 `BootstrapClassLoader` 中。当父类加载器无法处理时，才由自己来处理。当父类加载器为 `null` 时，会使用启动类加载器 `BootstrapClassLoader` 作为父类加载器。



每个类加载都有一个父类加载器，我们通过下面的程序来验证。

```
public class ClassLoaderDemo {
    public static void main(String[] args) {
        System.out.println("ClassLodarDemo's ClassLoader is " +
ClassLoaderDemo.class.getClassLoader());
        System.out.println("The Parent of ClassLodarDemo's ClassLoader is " +
ClassLoaderDemo.class.getClassLoader().getParent());
        System.out.println("The GrandParent of ClassLodarDemo's ClassLoader is " +
ClassLoaderDemo.class.getClassLoader().getParent().getParent());
    }
}
```

Output

```

ClassLodarDemo's ClassLoader is sun.misc.Launcher$AppClassLoader@18b4aac2
The Parent of ClassLodarDemo's ClassLoader is
sun.misc.Launcher$ExtClassLoader@1b6d3586
The GrandParent of ClassLodarDemo's ClassLoader is null

```

`AppClassLoader` 的父类加载器为 `ExtClassLoader`，`ExtClassLoader` 的父类加载器为 `null`，`null` 并不代表 `ExtClassLoader` 没有父类加载器，而是 `BootstrapClassLoader`。

其实这个双亲翻译的容易让别人误解，我们一般理解的双亲都是父母，这里的双亲更多地表达的是“父母这一辈”的人而已，并不是说真的有一个 `Mother ClassLoader` 和一个 `Father ClassLoader`。另外，类加载器之间的“父子”关系也不是通过继承来体现的，是由“优先级”来决定。官方API文档对这部分的描述如下：

The Java platform uses a delegation model for loading classes. **The basic idea is that every class loader has a "parent" class loader.** When loading a class, a class loader first "delegates" the search for the class to its parent class loader before attempting to find the class itself.

双亲委派模型实现源码分析

双亲委派模型的实现代码非常简单，逻辑非常清晰，都集中在 `java.lang.ClassLoader` 的 `loadClass()` 中，相关代码如下所示。

```

private final ClassLoader parent;
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // 首先，检查请求的类是否已经被加载过
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) { // 父加载器不为空，调用父加载器loadClass()方
法处理
                    c = parent.loadClass(name, false);
                } else { // 父加载器为空，使用启动类加载器 BootstrapClassLoader 加
载
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // 抛出异常说明父类加载器无法完成加载请求
            }

            if (c == null) {
                long t1 = System.nanoTime();
                // 自己尝试加载
                c = findClass(name);

                // this is the defining class loader; record the stats
                sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 -
t0);
            }
        }
    }
}

```


```
sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
    sun.misc.PerfCounter.getFindClasses().increment();
    }
}
if (resolve) {
    resolveClass(c);
}
return c;
}
}
```

双亲委派模型的好处

双亲委派模型保证了Java程序的稳定运行，可以避免类的重复加载（JVM 区分不同类的方式不仅仅根据类名，相同的类文件被不同的类加载器加载产生的是两个不同的类），也保证了 Java 的核心 API 不被篡改。如果没有使用双亲委派模型，而是每个类加载器加载自己的话就会出现一些问题，比如我们编写一个称为 `java.lang.Object` 类的话，那么程序运行的时候，系统就会出现多个不同的 `Object` 类。

如果我们不想用双亲委派模型怎么办？

~~为了避免双亲委托机制，我们可以自己定义一个类加载器，然后重写 `loadClass()` 即可。~~

 **修正（参见：[issue871](#)）**：自定义加载器的话，需要继承 `ClassLoader`。如果我们不想打破双亲委派模型，就重写 `ClassLoader` 类中的 `findClass()` 方法即可，无法被父类加载器加载的类最终会通过这个方法被加载。但是，如果想打破双亲委派模型则需要重写 `loadClass()` 方法

自定义类加载器

除了 `BootstrapClassLoader` 其他类加载器均由 Java 实现且全部继承自 `java.lang.ClassLoader`。如果我们要自定义自己的类加载器，很明显需要继承 `ClassLoader`。

推荐阅读

- <https://blog.csdn.net/xyang81/article/details/7292380>
- <https://juejin.im/post/5c04892351882516e70dcc9b>
- <http://gityuan.com/2016/01/24/java-classloader/>