JVM 双亲委派模型及 SPI 实现原理分析

丙好像没跟大家聊过JVM相关的，最近618虽然很忙，但是我肝啊，在类加载机制我们了解之后，不可避免地提到了类加载器，和双亲委派模型，本文又从双亲委派模型讲解了 SPI 的相关实现。

双亲委派模型

我们知道类加载机制是将一个类从字节码文件转化为虚拟机可以直接使用类的过程，但是是谁来执行这个过程中的加载过程，它又是如何完成或者说保障了类加载的准确性和安全性呢?

答案就是类加载器以及双亲委派机制。

双亲委派模型的工作机制是：当类加载器接收到类加载的请求时，它不会自己去尝试加载这个类，而是把这个请求委派给父加载器去完成，只有当父类加载器反馈自己无法完成这个加载请求时，子加载器才会尝试自己去加载类。

我们可以从 JDK 源码中将它的工作机制一窥究竟。

ClassLoader#loadClass(String, boolean)

这是在 `jdk1.8` 的 `java.lang.ClassLoader` 类中的源码，这个方法就是用于加载指定的类。

```
public class ClassLoader {
    protected Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException{
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been loaded
            // 首先，检查该类是否已经被当前类加载器加载，若当前类加载未加载过该类，调用
父类的加载类方法去加载该类（如果父类为null的话交给启动类加载器加载）
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
```

```
                }
                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    // 如果父类未完成加载，使用当前类加载器去加载该类
                    long t1 = System.nanoTime();
                    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) {
                // 链接指定的类
                resolveClass(c);
            }
            return c;
        }
    }
}
```

看完了上面的代码，我们知道这就是双亲委派模型代码层面的解释：

1. 当类加载器接收到类加载的请求时，首先检查该类是否已经被当前类加载器加载；
2. 若该类未被加载过，当前类加载器会将加载请求委托给父类加载器去完成；
3. 若当前类加载器的父类加载器（或父类的父类……向上递归）为 null，会委托启动类加载器完成加载；
4. 若父类加载器无法完成类的加载，当前类加载器才会去尝试加载该类。

类加载器分类

在 JVM 中预定义的类加载器有3种：启动类加载器(Bootstrap ClassLoader)、扩展类加载器(Extension ClassLoader)、应用类/系统类加载器(App/System ClassLoader)，另外还有一种是用户自定义的类加载器，它们各自有各自的职责。

启动类加载器 `Bootstrap ClassLoader`

启动类加载器作为所有类加载器的"老祖宗"，是由C++实现的，不继承于 `java.lang.ClassLoader` 类。它在虚拟机启动时会由虚拟机的一段C++代码进行加载，所以它没有父类加载器，在加载完成后，它会负责去加载扩展类加载器和应用类加载器。

启动类加载器用于加载 Java 的核心类——位于 `<JAVA_HOME>\lib` 中，或者被 `-Xbootclasspath` 参数所指定的路径中，并且是虚拟机能够识别的类库（仅按照文件名识别,如 `rt.jar`、`tools.jar` ，名字不符合的类库即使放在lib目录中也不会被加载）。

拓展类加载器 `Extension ClassLoader`

拓展类加载器继承于 `java.lang.ClassLoader` 类，它的父类加载器是启动类加载器，而启动类加载器在 Java 中的显示就是 null。

> 引自 jdk1.8 ClassLoader#getParent() 方法的注释，这个方法是用于获取类加载器的父类加载器: Returns the parent class loader for delegation. Some implementations may use null to represent the bootstrap class loader. This method will return null in such implementations if this class loader's parent is the bootstrap class loader.

拓展类加载器负责加载 `<JAVA_HOME>\lib\ext` 目录中的，或者被 `java.ext.dirs` 系统变量所指定的路径的所有类。

需要注意的是扩展类加载器仅支持加载被打包为 `.jar` 格式的字节码文件。

应用类/系统类加载器 `App/System ClassLoader`

应用类加载器继承于 `java.lang.ClassLoader` 类，它的父类加载器是扩展类加载器。

应用类加载器负责加载用户类路径 `classpath` 上所指定的类库。

如果应用程序中没有自定义的类加载器，一般情况下应用类加载器就是程序中默认的类加载器。

自定义类加载器 `Custom ClassLoader`

自定义类加载器继承于 `java.lang.ClassLoader` 类，它的父类加载器是应用类加载器。

这是普某户籍自定义的类加载器,可加载指定路径的字节码文件。

自定义类加载器需要继承 `java.lang.ClassLoader` 类并重写 `findClass` 方法（下文有说明为什么不重写 `loadClass` 方法）用于实现自定义的加载类逻辑。

双亲委派模型的好处

1. 基于双亲委派模型规定的这种带有优先级的层次性关系，虚拟机运行程序时就能够避免类的重复加载。

   当父类类加载器已经加载过类时，如果再有该类的加载请求传递到子类类加载器，子类类加载器执行 `loadClass` 方法，然后委托给父类类加载器尝试加载该类，但是父类类加载器执行 `Class<?> c = findLoadedClass(name);` 检查该类是否已经被加载过这一阶段就会检查到该类已经被加载过，直接返回该类，而不会再次加载此类。

2. 双亲委派模型能够避免核心类篡改。一般我们描述的核心类是 `rt.jar、tools.jar` 这些由启动类加载器加载的类，这些类库在日常开发中被广泛运用，如果被篡改，后果将不堪设想。

   假设我们自定义了一个 `java.lang.Integer` 类，与好处1一样的流程，当加载类的请求传递到启动类加载器时，启动类加载器执行 `findLoadedClass(String)` 方法发现 `java.lang.Integer` 已经被加载过，然后直接返回该类，加载该类的请求结束。虽然避免核心类被篡改这一点的原因与避免类的重复加载一致，但这还是能够作为双亲委派模型的好处之一的。

双亲委派模型的不足

> 这里所说的不足也可以理解为打破双亲委派模型，当双亲委派模型不满足用户需求时，自然是由于其不足之处，也就促使用户将其打破。这里描述的也就是打破双亲委派模型的三种方式。

1. 由于历史原因（`ClassLoader` 类在 JDK1.0 时就已经存在，而双亲委派模型是在 JDK1.2 之后才引入的），在未引入双亲委派模型时，用户自定义的类加载器需要继承 `java.lang.ClassLoader` 类并重写 `loadClass()` 方法，因为虚拟机在加载类时会调用 `ClassLoader#loadClassInternal(String)`，而这个方法（源码如下）会调用自定义类加载重写的 `loadClass()` 方法。

   而在引入双亲委派模型后， `ClassLoader#loadClass` 方法实际就是双亲委派模型的实现，如果重写了此方法，相当于打破了双亲委派模型。为了让用户自定义的类加载器也遵从双亲委派模型， JDK 新增了 `findClass` 方法，用于实现自定义的类加载逻辑。

```
class ClassLoader {
    // This method is invoked by the virtual machine to load a class.
    private Class<?> loadClassInternal(String name) throws
ClassNotFoundException{
        // For backward compatibility, explicitly lock on 'this' when
        // the current class loader is not parallel capable.
        if (parallelLockMap == null) {
            synchronized (this) {
                 return loadClass(name);
            }
        } else {
            return loadClass(name);
        }
    }
// 其余方法省略......
}
```

2. 由于双亲委派模型规定的层次性关系，导致子类类加载器加载的类能访问父类类加载器加载的类，而父类类加载器加载的类无法访问子类类加载器加载的类。为了让上层类加载器加载的类能够访问下层类加载器加载的类，或者说让父类类加载器委托子类类加载器完成加载请求，JDK 引入了线程上下文类加载器，藉由它来打破双亲委派模型的屏障。

3. 当用户需要程序的动态性，比如代码热替换、模块热部署等时，双亲委派模型就不再适用，类加载器会发展为更为复杂的网状结构。

线程上下文类加载器

上面说到双亲委派模型的不足时提到了线程上下文类加载器 `Thread Context ClassLoader`，线程上下文类加载器是定义在 `Thread` 类中的一个 `ClassLoader` 类型的私有成员变量，它指向了当前线程的类加载器。

上文已经提到线程上下文类加载能够让父类类加载器委托子类类加载器完成加载请求，那么这是如何实现的呢？下面就来讨论一下。

SPI 在 JDBC 中的应用

我们知道 Java 提供了一些 `SPI(Service Provider Interface)` 接口，它允许服务商编写具体的代码逻辑来完成该接口的功能。

但是 Java 提供的 SPI 接口是在核心类库中，由启动类加载器加载的，厂商实现的具体逻辑代码是在 classpath 中，是由应用类加载器加载的，而启动类加载器加载的类无法访问应用类加载器加载的类，也就是说启动类加载器无法找到 SPI 实现类，单单依靠双亲委派模型就无法实现 SPI 的功能了，所以线程上下文类加载器应运而生。

在 Java 提供的 SPI 中我们最常用的可能就属 JDBC 了，下面我们就以 JDBC 为例来看一下线程上下文类加载器如何打破双亲委派模型。

回忆一下以前使用 JDBC 的场景，我们需要创建驱动，然后创建连接，就像下面的代码这样：

```
public class ThreadContextClassLoaderDemoOfJdbc {

    public static void main(String[] args) throws Exception {
        // 加载 Driver 的实现类
        Class.forName("com.mysql.jdbc.Driver");
        // 建立连接
        Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mysql", "root",
"admin");
    }
}
```

> 在 JDK1.6 以后可以不用写 `Class.forName("com.mysql.jdbc.Driver");`，代码依旧能正常运行。这是因为自带的 jdbc4.0 版本已支持 SPI 服务加载机制，只要服务商的实现类在 classpath 路径中，Java 程序会主动且自动去加载符合 SPI 规范的具体的驱动实现类，驱动的全限定类名在 `META-INF.services` 文件中。

所以，让我们把目光聚焦于建立连接的语句，这里调用了 `DriverManager` 类的静态方法 `getConnection` 。

在调用此方法前，根据类加载机制的初始化时机，调用类的静态方法会触发类的初始化，当 DriverManager 类被初始化时，会执行它的静态代码块。

```
    public class DriverManager {

        static {
            loadInitialDrivers();
            println("JDBC DriverManager initialized");
        }

        private static void loadInitialDrivers() {
            String drivers;
            // 省略代码：首先读取系统属性 jdbc.drivers

            // 通过 SPI 加载 classpath 中的驱动类
            AccessController.doPrivileged(new PrivilegedAction<Void>() {
                public Void run() {
                    // ServiceLoader 类是 SPI 加载的工具类
                    ServiceLoader<Driver> loadedDrivers =
    ServiceLoader.load(Driver.class);
                    Iterator<Driver> driversIterator =
    loadedDrivers.iterator();
                    try{
                        while(driversIterator.hasNext()) {
                            driversIterator.next();
                        }
                    } catch(Throwable t) {
                    // Do nothing
                    }
                    return null;
                }
            });
            // 省略代码：使用应用类加载器继续加载系统属性 jdbc.drivers 中的驱动类
        }

    }
```

从上面的代码中可以看到，程序时通过调用 ServiceLoader.load(Driver.class) 方法来完成自动加载 classpath 路径中具体的所有实现了 Driver.class 接口的厂商实现类，而在 ServiceLoader.load() 方法中，就是获取了当前线程上下文类加载器，并将它传递下去，将它作为类加载器去实现加载逻辑的。

```
public final class ServiceLoader<S> implements Iterable<S>{
    public static <S> ServiceLoader<S> load(Class<S> service) {
        // 获取当前线程的线程上下文类加载器 AppClassLoader，用于加载 classpath 中
的具体实现类
        ClassLoader cl = Thread.currentThread().getContextClassLoader();
        return ServiceLoader.load(service, cl);
    }
}
```

JDK 默认加载当前类的类加载器去加载当前类所依赖且未被加载的类，而 `ServiceLoader` 类位于 `java.util` 包下，自然是由启动类加载器完成加载，而厂商实现的具体驱动类是位于 classpath 下，启动类加载器无法加载 classpath 目录的类，而如果加载具体驱动类的类加载器变成了应用类加载器，那么就可以完成加载了。

通过跟踪代码，不难看出 `ServiceLoader#load(Class)` 方法创建了一个 `LazyIterator` 类同时返回了一个 `ServiceLoader` 对象，前者是一个懒加载的迭代器，同时它也是后者的一个成员变量，当对迭代器进行遍历时，就触发了目标接口实现类的加载。

```
private class LazyIterator implements Iterator<S> {

    public S next() {
        if (acc == null) {
            return nextService();
        } else {
            PrivilegedAction<S> action = new PrivilegedAction<S>() {
                public S run() { return nextService(); }
            };
            return AccessController.doPrivileged(action, acc);
        }
    }
}
```

在 `DriverManager#loadInitialDrivers` 方法，也就是 `DriverManager` 类的静态代码块所执行的方法中，有这样一段代码：

```
AccessController.doPrivileged(new PrivilegedAction<Void>() {
            public Void run() {
                ServiceLoader<Driver> loadedDrivers =
ServiceLoader.load(Driver.class);
                Iterator<Driver> driversIterator =
loadedDrivers.iterator();
                try{
```

```
                    while(driversIterator.hasNext()) {
                        driversIterator.next();
                    }
                } catch(Throwable t) {
                // Do nothing
                }
                return null;
            }
        });
```

这段代码返回了一个 `ServiceLoader` 对象，在这个对象中有一个 `LazyIterator` 迭代器类，用于存放所有厂商实现的具体驱动类，当我们对 `LazyIterator` 这个迭代器进行遍历时，就出发了类加载的逻辑。

```
    private S nextService() {
            if (!hasNextService())
                throw new NoSuchElementException();
            String cn = nextName;
            nextName = null;
            Class<?> c = null;
            try {
                // 不用写 Class.forName("com.mysql.jdbc.Driver"); 的原因就是在此处
    会自动调用这个方法
                c = Class.forName(cn, false, loader);
            } catch (ClassNotFoundException x) {
                fail(service, "Provider " + cn + " not found");
            }
            if (!service.isAssignableFrom(c)) {
                fail(service, "Provider " + cn  + " not a subtype");
            }
            try {
                S p = service.cast(c.newInstance());
                providers.put(cn, p);
                return p;
            } catch (Throwable x) {
                fail(service, "Provider " + cn + " could not be instantiated",
    x);
            }
            throw new Error();          // This cannot happen
        }
```

每次遍历都会调用 `Class.forName(cn, false, loader)` 方法对指定的类进行加载和实例化操作，这也是前文提到的在 jdk1.6 以后不用在写 `Class.forName("com.mysql.jdbc.Driver");` 的原因。

在这个方法 `Class.forName(cn, false, loader)` 中，传入的参数 cn 是全路径类名，false 是指不进行初始化，loader 则是指定完成 cn 类加载的类加载器。

在这里的 loader 变量，我们回顾一下前文的描述，在 `ServiceLoader.load(Driver.class)` 方法中是不是获取了线程上下文类加载器并传递下去?

不记得? 在回过头去看一遍!

而传入的线程上下文类加载器会作为参数传递给 `ServiceLoader` 类的构造方法

```
    private ServiceLoader(Class<S> svc, ClassLoader cl) {
            service = Objects.requireNonNull(svc, "Service interface cannot be
    null");
            loader = (cl == null) ? ClassLoader.getSystemClassLoader() : cl;
            acc = (System.getSecurityManager() != null) ?
    AccessController.getContext() : null;
            reload();
        }
```

而此处的 cl 变量就是调用 `DriverManager` 类静态方法的线程上下文类加载器，即应用类加载器。

也就是说，通过 `DriverManager` 类的静态方法，实现了由 `ServiceLoader` 类触发加载位于 `classpath` 的厂商实现的驱动类。前文已经说过， `ServiceLoader` 类位于 `java.util` 包中，是由启动类加载器加载的，而由启动类加载器加载的类竟然实现了"委派"应用类加载器去加载驱动类，这无疑是与双亲委派机制相悖的。而实现这个功能的，就是线程上下文类加载器。

至此，我们就分析完线程上下文类加载是如何实现 SPI 的了。

小结

1. 双亲委派模型的工作机制。
2. 类加载器的分类及各自的职责。
3. 双亲委派模型的好处。
4. 打破双亲委派模型的三种场景。
5. 线程上下文类加载器在是如何实现 SPI 的。