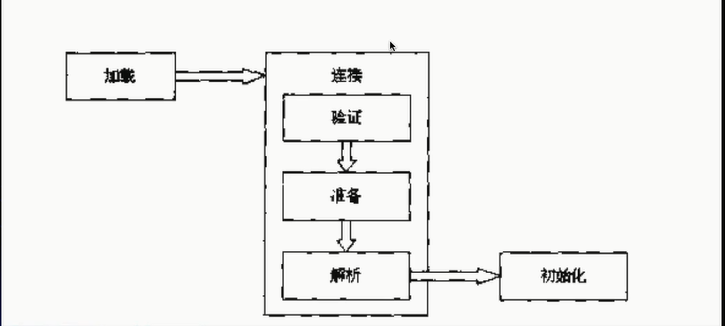
# Class加载流程

Class类型通常以文件的形式存在，只有被Java虚拟机装载的Class类型才能在程序中使用。

系统装载Class类型可以分为：加载，连接和初始化阶段，其中连接可分为：验证，准备和解析。



## 类加载

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在内存中创建一个java.lang.Class对象（规范并未说明Class对象位于哪里，HostSpot虚拟机将其放在了方法区）用于封装类在方法区的数据结构。

加载.class文件的方式：1.从本地系统中直接加载

2.通过网络下载.class文件

3.从zip,jar等归档文件中红加载.class文件

4.从专有数据库中提取.class文件

5.将Java源文件动态编译为.class文件

类加载器并需要在等到某个类被“首次主动使用”时再加载它。

JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）。如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误。

## 连接

类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中。

--验证：确保被加载的类的正确性。类的验证的内容：1）类文件的结构检查。2）语义检查。3）字节码验证。4）二进制兼容性的验证。

--准备：为类的静态变量分配内存，并将其初始化为默认值

--解析：在类型的常量池中寻找类，接口，字段和方法的符号引用，把这些符号引用转换为直接引用

## 初始化

类的初始化步骤：假如这个类还没有被加载和连接，那就先进行加载和连接

如果类存在直接父类，并且这个父类还没有被初始化，那就先初始化直接父类

如果类中存在初始化语句，那就一次执行这些初始化语句。

类的初始化时机：一个类或接口只有在初次使用前，必须要进行初始化。这里使用必须“主动使用”

主动使用：1.创建类的实例。2.访问某个类或接口的静态变量，或者对该静态变量赋值。

3.调用类的静态方法。4.反射（Class.forName()方法）。5.初始化一个类的子类。

6.Java虚拟机启动时被标明为启动类的类。7.JDK1.7开始提供的动态语言支持：java.lang.invoke.MethodHandle实例的解析结果REF\_getStatic，REF\_putStatic，REF\_invokeStatic句柄对应的类没有初始化则初始化。

### 接口的初始化

在一个接口在初始化时，并不要求父接口进行初始化。接口中的成员变量都是public static final

只有在真正使用到父接口的时候（如引用接口中所定义的常量时），才会初始化。

在初始化一个类时，并不会先初始化它所实现的接口。

如果一个接口定义的变量在编译器已经确定好了，则在加载时不会自动加载父接口。

如果一个接口定义的变量在编译器无法无法确定变量值，则在加载时会自动加载该接口和父接口。

#### 证明

## 案例

### 案例一

**public** **class** Singleton

{

**public** **static** **int** *num1* ;

**public** **static** **int** *num2* = 0 ;

**public** **static** Singleton *s* = **new** Singleton();

**private** Singleton()

{

*num1*++;

*num2*++;

}

**public** **static** Singleton getInstance()

{

**return** *s* ;

}

**public** **static** **void** main(String[] args) {

Singleton s = Singleton.*getInstance*();

System.***out***.println(s.*num1*);

System.***out***.println(s.*num2*);

}

}

运行结果：



调用类的静态方法会对类进行初始化，在初始化前的准备阶段，为静态变量赋予了默认值，即num1=0,num2=0,s=null。初始化阶段为会静态变量赋予我们给予的值，程序从上往下执行，首先num1没有直接给定值，num2给予了0值，然后s赋予新的Sinleton实例，创建Singleton实例会调用构造方法，从而num1=1,num2=1，初始化阶段完成，得出结果。

修改代码：

**public** **class** Singleton

{

**public** **static** **int** *num1* ;

**public** **static** Singleton *s* = **new** Singleton();

**private** Singleton()

{

*num1*++;

*num2*++;

}

**public** **static** **int** *num2* = 0 ;

**public** **static** Singleton getInstance()

{

**return** *s* ;

}

**public** **static** **void** main(String[] args) {

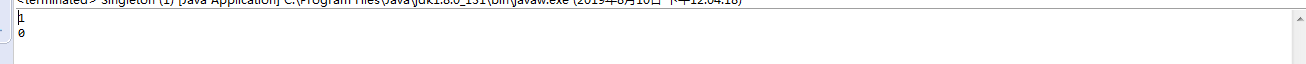
Singleton s = Singleton.*getInstance*();

System.***out***.println(s.*num1*);

System.***out***.println(s.*num2*);

}

}



调用类的静态方法会对类进行初始化，在初始化前的准备阶段，为静态变量赋予了默认值，即num1=0,num2=0,s=null。初始化阶段为会静态变量赋予我们给予的值，程序从上往下执行，即num1还是0, s赋予新的Singleton实例，创建Singleton实例会调用构造方法，从而使得num1=1,num2=1，在s赋予值之后，程序会继续执行，为num2赋予给定的值0，初始化完成，从而得到num1=1,num2=0的结果

## 双亲委托模型



在JDK1.2版本开始，类的加载过程采用父亲委托机制，这种机制更好地保证了Java平台的安全，在此委托机制中，除了Java虚拟机自带的根加载器以外，其余的类加载器都有且只有一个父加载器，当java程序请求加载器loader加载一个类时，loader首先委托给自己的父加载器去加载，如果父加载器可以加载成功，则有父加载器进行加载，否则才有加载器loader本身进行加载。

加载器之间的父子关系实际上是指加载器对象之间的包装关系，而不是类之间的继承关系。一对父子加载器可能是同一个加载器类的两个实例，也可能不是。在子加载器对象中包装了一个父加载器。

### 根加载器（Bootstrap）

根加载器（Bootstrap ClassLoader），使用C++编写（至今不确定），它主要加载JVM自身工作需要的类，这个ClassLoader完全有JVM自己控制，该加载器并不遵守类加载规则，它仅仅是一个类的加载工具，既没有更高一级的父加载器，也没有子加载器。

根加载器从系统属性sun.boot.class.path所指定的目录中加载类库

根加载又叫做启动类加载器。该加载器会在JVM启动时自动加载$JAVA\_HOME中的jre/lib/\*.jar包里所有的class文件。

内建于JVM中的启动类加载器会加载java.lang.ClassLoader以及其他的Java平台类，当JVM启动时，一块特殊的机器码会运行，它会加载扩展类加载器和系统类加载器，这块特殊的机器码叫做启动类加载器（Bootstrap）。

启动类加载器并不是Java类，而其他的加载器则是都是Java类，启动类加载器是特定于平台的机器指令，它负责开启真个加载过程

所有类加载器（除了启动类加载器）都被实现为Java类，不过，总归要有一个组件来加载第一个java类加载器，从而让整个加载过程能够顺利进行下去，加载第一个纯java类加载器就是启动类加载器的职责，启动类加载器还负责加载供JRE正常运行所需要的基本组件，这包括java.util和java.lang包中的类等。

#### 证明

**public** **static** **void** main(String[] args)

{

System.***out***.println(System.*getProperty*("sun.boot.class.path"));

}

运行结果：对于不同平台运行的结果是不相同的

C:\Program Files\Java\jdk1.8.0\_131\jre\lib\resources.jar;

C:\Program Files\Java\jdk1.8.0\_131\jre\lib\rt.jar;

C:\Program Files\Java\jdk1.8.0\_131\jre\lib\sunrsasign.jar;

C:\Program Files\Java\jdk1.8.0\_131\jre\lib\jsse.jar;

C:\Program Files\Java\jdk1.8.0\_131\jre\lib\jce.jar;

C:\Program Files\Java\jdk1.8.0\_131\jre\lib\charsets.jar;

C:\Program Files\Java\jdk1.8.0\_131\jre\lib\jfr.jar;

C:\Program Files\Java\jdk1.8.0\_131\jre\classes

在$JAVA\_HOME/jre目录下并没有classes目录，如果我们希望自爱JVM运行时就加载某些类，我们可以通过在该目录中创建classes目录，并将class文件放到对应的包目录中。

### 扩展类加载器（Extension）

扩展类加载器（Extension ClassLoader），使用Java代码实现，是java.lang.ClassLoader类的子类，但并不是JVM亲自实现的，该加载器服务的特定目标在System.getProperty(“java.ext.dirs”)目录下，或者从JDK的安装目录下，

注意：在我们使用java -Djava.ext.dirs=<path>设置扩展类加载器加载路径时，指定路径下的class文件必须打成jar包的形式，否则扩展类加载器就无法加载类

#### 证明

**public** **static** **void** main(String[] args)

{

System.***out***.println(System.*getProperty*("java.ext.dirs"));

}

运行结果：

C:\Program Files\Java\jdk1.8.0\_131\jre\lib\ext;

C:\Windows\Sun\Java\lib\ext

**public** **class** MyCat

{

**public** MyCat()

{

System.***out***.println("MyCat loaded by:"+**this**.getClass().getClassLoader());

}

}

**public** **class** ExtTest

{

**public** ExtTest()

{

System.***out***.println("ExtTest loaded by:"+**this**.getClass().getClassLoader());

**new** MyCat();

}

**public** **static** **void** main(String[] args) {

**new** ExtTest();

}

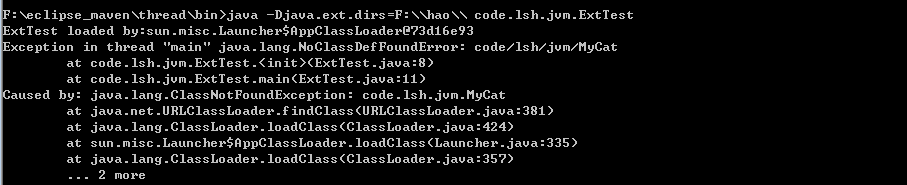
}

正常运行结果：ExtTest loaded by:sun.misc.Launcher$AppClassLoader@4e25154f

MyCat loaded by:sun.misc.Launcher$AppClassLoader@4e25154f

修改运行时JVM参数-Djava.ext.dirs=F:\\hao\\,并拷贝classpath路径中MyCat.class文件，到F:\\hao目录下，然后删除classpath路径下的MyCat.class文件。

运行结果：

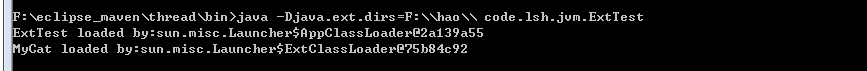


程序发生错误，但ExtTest应用类加载器成功加载，而加载ExtTest类的加载器委托父加载器，即扩展类加载器进行加载，我们指定了扩展类加载器的加载路径，并且在该路径中包含了MyCat.class文件，但扩展类加载器并没有进行加载。

将F:\hao\code\lsh\jvm\MyCat.class文件打成test.jar包。然后删除code/lsh/jvm/目录。

=

重新运行：java –Djava.ext.dirs=F:\\hao\\ code.lsh.jvm.ExtTests



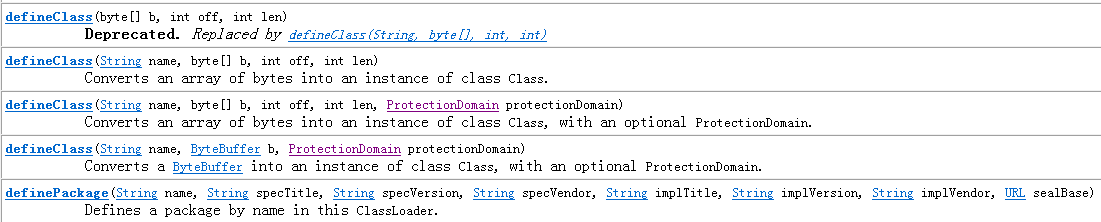
程序运行成功，并且MyCat类有扩展类加载器进行的加载。即扩展类加载器只能加载jar包。

### 应用类加载器（Application）

系统类加载器（APPClassLoader），使用Java代码实现，它的父类时扩展类加载器（ExtClassLoader），它服务目标是System.getProperty(“java.class.path”)目录下的类都可以被这个类加载器加载，这个目录是我们经常用到的classpath。

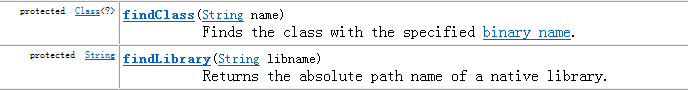
### 自定义类加载器

java.lang.ClassLoader方法：



defineClass用来将byte字节流解析成JVM能够识别的Class对象，这意味着我们不仅仅可以通过class文件实例化对象，还可以通过其他方式实例化对象。

off，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。该方法是受保护的，所以只能在自定义的ClassLoader子类中可以使用。



通常findClass方法和defineClass方法一起使用，我们通常直接覆盖ClassLoader父类的findClass方法来实现类的加载规则，从而取得要加载类的字节码，然后调用defineClass方法生成Class对象。

如果我们想要类被加载到JVM中时被链接，那么我们可以调用resolveClass方法。



在ClassLoader结构中，还有一个重要的字段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。

#### 实现自己的类加载器

**public** **class** MyClassLoader **extends** ClassLoader

{

**private** **static** **final** String ***EXT\_NAME*** = ".class" ;

**private** String classloaderName ;

**private** String path ;

**public** MyClassLoader(String cname)

{

**super**();

**this**.classloaderName = cname ;

}

**public** MyClassLoader(ClassLoader parent , String cname)

{

**super**(parent);

**this**.classloaderName = cname ;

}

**public** String getPath() {

**return** path;

}

**public** **void** setPath(String path) {

**this**.path = path;

}

@Override

**protected** Class<?> findClass(String className) **throws** ClassNotFoundException

{

**byte**[] data = **this**.loadClassForName(className);

**return** defineClass(className,data, 0, data.length);

}

**private** **byte**[] loadClassForName(String className)

{

InputStream input = **null** ;

ByteArrayOutputStream baos = **null** ;

**byte**[] data = **null** ;

**int** ch = 0 ;

**try**

{

className = className.replace(".", "\\");

String classPath = **this**.getPath()+className+***EXT\_NAME*** ;

System.***out***.println("classPath:"+classPath);

input = **new** FileInputStream(**new** File(classPath));

baos = **new** ByteArrayOutputStream();

**while**(-1!=(ch = input.read()))

{

baos.write(ch);

}

data = baos.toByteArray();

}

**catch**(Exception e) {e.printStackTrace();}

**finally**

{

**if**(input != **null**)

{

**try** {

input.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**if**(**null**!=baos)

{

**try** {

baos.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**return** data ;

}

## 命名空间

1. 每一个类加载器都有自己的命名空间，命名空间由该加载器及所有父加载器所加载的类组成

2. 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

3. 子加载器能够访问父加载器加载的类，而父加载器不能访问子加载器加载的类

4. 同一个命名空间内的类是相互可见的，子加载器的命名空间包含父加载器的命名空间，因此子加载器加载的类能看见父加载器加载的类，如：系统类加载器可以看见根类加载器。由父加载器加载的类不能看见子加载器加载的类。

5. 如果两个加载器之间没有直接或间接的父子关系，那么它们各自加载的类相互不可见。

### 证明

#### 证明1

创建MyPerson类。

**public** **class** MyPerson

{

**private** MyPerson myPerson ;

**public** MyPerson()

{

System.***out***.println("loaded by:"+**this**.getClass().getClassLoader());

}

**public** **void** setMyPerson(Object obj)

{

**this**.myPerson = (MyPerson)obj ;

}

}

这里我们使用我们自定的类加载器（MyClassLoader），加载MyPerson的class文件，获取对应的Class实例，从而创建MyPerson实例，我们创建了两个加载器loader1和loader2，分别创建了obj1和obj2的MyPerson实例,然后通过反射调用MyPerson中的setMyPerson方法。

**public** **class** MyTest

{

**public** **static** **void** main(String[] args) **throws** Exception

{

MyClassLoader loader1 = **new** MyClassLoader("loader1");

loader1.setPath("F:\\hao\\");

MyClassLoader loader2 = **new** MyClassLoader("loader2");

loader2.setPath("F:\\hao\\");

Class<?> clazz1 = loader1.loadClass("code.lsh.jvm.MyPerson");

Class<?> clazz2 = loader2.loadClass("code.lsh.jvm.MyPerson");

Object obj1 = clazz1.newInstance();

Object obj2 = clazz2.newInstance();

Method method = clazz1.getMethod("setMyPerson", Object.**class**);

method.invoke(obj1, obj2);

}

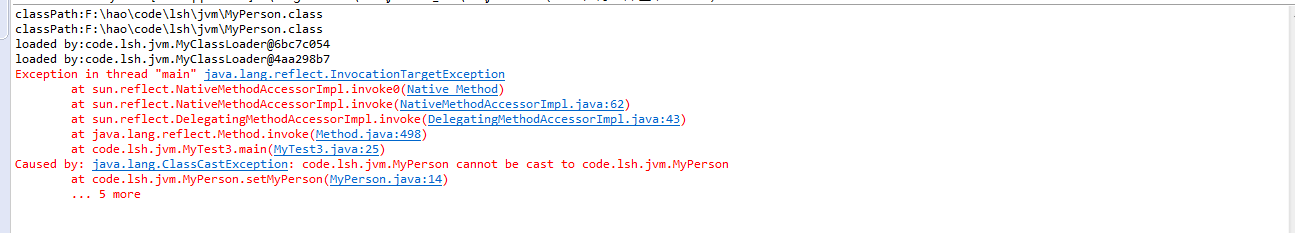
}

测试一：在classpath中有MyPerson类对应的class文件，运行结果：



该类是通过同一个加载器（应用类加载器）加载，无异常发生，在同一命名空间的类是相互可见的。

测试二：将classpath路径下的MyPerson对应的class文件拷贝到F:\\hao\\code\\lsh\\jvm\\路径下，拷贝后将classpath路径下的MyPerson.class文件删除。运行程序：



运行结果发生了异常，提示我们类型转换异常。MyPerson的setMyPerson方法将我们传入的对象转换为MyPerson对象，然会我们传入的就是MyPerson对象，这是为啥发生异常了呢？

我们从截图中可以发现，我们的创建的obj1和obj2对象是由我们自定义的类加载器进行加载的，即loader1和loader2加载，而loader1加载MyPerson.class文件创建类obj1实例，loader2加载的MyPerson.class文件创建了obj2实例，我们将obj2的实例传入到obj1实例方法setMyPerson中，在执行该方法的类型转换时发生异常java.lang.ClassCastException。我们也可以进行如下测试：clazz1==clazz2，测试结果为false。我们知道类对应的Class对象是唯一的，然而该测试结果为false，如果该结论是不正确的，这里我们不能说不正确，只能说不完善：在同一命名空间内，类对应的Class对象时唯一的，且只有一个，而在不同的命名空间内，有可能会出现类的完整名字（包括类的包名）相同的两个类，即一个类有多个Class对象。不同命名空间的类是不可见，不能相互转换。这就是异常发生的原因。

#### 证明2

## 分析Launcher源码

## 双亲委托模型的好处

1.可以确保Java核心库的类型安全：所有的Java类都会直接或间接继承了Object类，也就是说，在运行器java.lang.Object这个类会被加载到Java虚拟机中，如果这个加载过程是 由Java应用程序自己的加载器完成的，那么很可能就会在JVM中有多个版本的java.lang.Object类，而且这些类之间还是不兼容的，相互之间不可见（正是命名空间发挥着作用）借助于双亲委托机制，Java核心类库中的类的加载工作由启动类加载器进行加载，从而保证了Java应用程序所使用的都是同一个版本的Java核心类库，他们之间是相互兼容的。

2.可以确保Java核心库所提供的类不会被自定的类所代替。

3.不同的类加载器可以为相同名称（binary name）的类创建额外的命名空间。相同名称的类可以并存在JVM中，只需要不同的类加载器进行加载即可。不同类加载器所加载的类之间是不兼容的，这就相当于在Java虚拟机中内部创建一个有一个相互隔离的java类空间，这类技术在很多的矿建中都得到了实际应用。

## 双亲委托模型的坏处

我们知道子加载器可以访问父加载器加载的类和接口，而父加载器无法访问子加载器加载的类和接口。如果在系统类中，提供了一个接口，该接口的实现在应用中得以实现，该接口还绑定了一个工厂方法，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。拥有这种问题的组件有很多，比如JDBC，XmlParser等。

### SPI(Service Provider Interface)

SPI：服务提供者。

在Java平台中，启动类加载器加载核心库（rt.jar），然而在核心库有对外提供的服务接口或抽象类（如JDBC规范），而这些接口或抽象列可由应用层自行实现或第三方实现。而这些实现是添加到我们系统的ClassPath路径下的，由系统类加载器进行加载，由于JVM默认的双亲委托机制导致启动类加载器无法看见这些实现，从而导致无法对外提供服务，而JVM为我们提供了解决方法，详细介绍看：线程上下文加载器。

# 线程上下文加载器

在jdk1.2引入了线程上下文加载器，类Thread中的getContextClassLoader()和setContextClassLoader(ClassLoader cl)分别用于获取和设置上下文加载器。

如果没有通过setContextClassLoader(ClassLoader cl)进行设置的话，线程将继承其父线程的上下文加载器，Java应用运行时的初始线程的上下文类加载器是系统类加载器。在线程中运行的代码可以通过该类加载器来加载类与资源

父ClassLoader可以使用当先线程Thread.currentThread().getContextClassLoader()所指定的classloader加载的类。这能改变父classloader不能使用子classloader或者其他没有直接父子关系的classloader加载的类情况，即改变了双亲委托模型。

在双亲委托模型中，类的加载是由下至上的，即下层的类加载器会委托上层类加载器。但对于SPI(服务提供者)来说，有些接口是Java核心库提供的（java.sql.\*包中的类和接口），而Java核心库是由根加载器（启动加载器）来加载的，而这些接口的实现却是来自不同的jar包（不同厂商提供不同的实现），通常这些jar包是由应用类（系统类）加载器来加载的，Java启动类加载器不会加载其他来源的jar包，这样传统的双亲委托模型就无法满足SPI的要求，而通过给当前线程设置上下文类加载器，就可以由设置的上下文类加载器来实现对于接口实现类的加载。

## ServiceLoader类分析

API说明：

一个简单的服务提供者设施。众所周知，一个服务提供了一些列的接口或抽象类，而服务提供着是服务的特定实现。服务提供者可以以扩展的方式安装到Java平台上，也可以将jar文件放在任何扩展目录中，也可以将提供者添加到应用的classpath路径中或者通过其他某些特定与平台的方式使其可用。

为了加载，服务由单个类型表示，也就是单个接口或抽象类。一个给定服务的提供者包含一个或多个具体类，这些类扩展了此服务类型，具有特定与提供者的数据和代码。提供者通常不包含它本身，而是一个包含足够多信息的代理，用于决定提供者是否能够满足特定的请求，还包含了可以根据需要创建市级提供者的代码。提供者类的详细信息高度特定于服务，任何单个类或接口都不能统一它们，因此这里没有定义任何这种类型。唯一强制要求的是，提供者必须具有不带参数的构造方法，以便它们可以在加载中被实例化。

通过在资源目录META-INF/services下放置提供者配置文件来标识提供者。该配置文件名称必须是服务类型的全限定名二进制名字。该配置文件包含了具体提供者类的完全限定二进制名称列表，每行一个。忽略各名称周围的空格，制表符和空行。注释字符为’#’,文件必须以UTF-8编码。

如果一个特殊的提供者的名字在多个配置文件中出现，或者在配置文件中出现多次，这些多余的会被忽略。

对多线程并发执行，该类的实例是不安全的。

## 案例分析

### 案例一

首先，将MySQL的驱动添加到我们项目的classpath路径中。

**public** **class** Provider

{

**public** **static** **void** main(String[] args)

{

//Thread.currentThread().setContextClassLoader(Provider.class.getClassLoader().getParent());

ServiceLoader<Driver> loader = ServiceLoader.*load*(Driver.**class**);

System.***out***.println("ServiceLoader loaded by:"+loader.getClass().getClassLoader());

Iterator<Driver> iterator = loader.iterator();

**while**(iterator.hasNext())

{

Driver driver = iterator.next();

System.***out***.println("driver:"+driver.getClass()+",loader:"+driver.getClass().getClassLoader());

}

System.***out***.println("--------------------------------------------------");

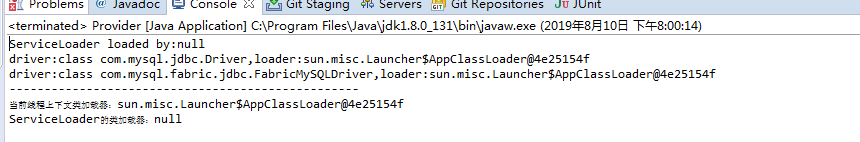
System.***out***.println("当前线程上下文类加载器："+Thread.*currentThread*().getContextClassLoader());

System.***out***.println("ServiceLoader的类加载器："+ServiceLoader.**class**.getClassLoader());

}

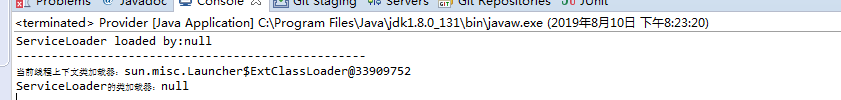
}

测试一：运行结果



从运行结果看：ServiceLoader是由根加载器加载的，在调用ServiceLoader的load方法时Driver类并不是有根加载器加载的，而是有系统类加载器加载的。

测试二：将我们注释的代码取消注释，运行程序，运行结果：



我们设置了线程上下文类加载器为扩展类加载器，ServiceLoader还是有跟加载器进行加载，然而结果并没有打印driver信息。这是为什么呢？要想解释产生该结果的原因，我们需要探讨ServiceLoader源码？

#### ServiceLoader源码分析

**public** **static** <S> ServiceLoader<S> load(Class<S> service) {

ClassLoader cl = Thread.*currentThread*().getContextClassLoader();

**return** ServiceLoader.*load*(service, cl);

}

从源码中，我们可以发现该方法首先后去线程上下文类加载器，然后调用load重载的方法将cl传入。

**public** **static** <S> ServiceLoader<S> load(Class<S> service,ClassLoader loader)

{

**return** **new** ServiceLoader<>(service, loader);

}

重载的load会创建一个ServiceLoader实例。调用ServiceLoader的私有构造方法。

**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();

}

在构造方法中会判断我们出入的cl是否为空，如果为空，则将loader变量赋予系统类加载器，否则将我们传入的类加载器赋予loader变量，这里loader变量时ServiceLoader的一个成员变量。然后调用reload()方法。

**public** **void** reload() {

providers.clear();

lookupIterator = **new** LazyIterator(service, loader);

}

reload()方法首先会清除缓存（这里的缓存是：**private** LinkedHashMap<String,S> providers = **new** LinkedHashMap<>()ServiceLoader定义的集合，用于缓存提供者），然后创建一个LazyIterator对象，将我们的Class对象和加载器传入。对于测试一来说，service=Driver.class,loader=Application ClassLoader。

这里的LazyIterator是ServiceLoader的一个私有内部类，该类实现了Iterator接口。

在我们调用loader的iterator()方法后，iterator方法是LazyIterator的方法：

**public** Iterator<S> iterator() {

**return** **new** Iterator<S>() {

Iterator<Map.Entry<String,S>> knownProviders

= providers.entrySet().iterator();

**public** **boolean** hasNext() {

**if** (knownProviders.hasNext())

**return** **true**;

**return** lookupIterator.hasNext();

}

**public** S next() {

**if** (knownProviders.hasNext())

**return** knownProviders.next().getValue();

**return** lookupIterator.next();

}

**public** **void** remove() {

**throw** **new** UnsupportedOperationException();

}

};

}

通过源码我们可以看出该方法会创建一个Iterator内部类，在内部类实现了hasNext，next和remove方法。

**public** **boolean** hasNext() {

**if** (acc == **null**) {

**return** hasNextService();

} **else** {

PrivilegedAction<Boolean> action = **new** PrivilegedAction<Boolean>() {

**public** Boolean run() { **return** hasNextService(); }

};

**return** AccessController.*doPrivileged*(action, acc);

}

}

在我们调用iterator方法创建的内部类Iterator的hasNext方法时，首先从缓存中获取，如果获取到就返回true，否则调用LazyIterator的hasNext方法：

**public** **boolean** hasNext() {

**if** (acc == **null**) {

**return** hasNextService();

} **else** {

PrivilegedAction<Boolean> action = **new** PrivilegedAction<Boolean>() {

**public** Boolean run() { **return** hasNextService(); }

};

**return** AccessController.*doPrivileged*(action, acc);

}

}

**private** **boolean** hasNextService() {

**if** (nextName != **null**) {

**return** **true**;

}

**if** (configs == **null**) {

**try** {

String fullName = ***PREFIX*** + service.getName();

**if** (loader == **null**)

configs = ClassLoader.*getSystemResources*(fullName);

**else**

configs = loader.getResources(fullName);

} **catch** (IOException x) {

*fail*(service, "Error locating configuration files", x);

}

}

**while** ((pending == **null**) || !pending.hasNext()) {

**if** (!configs.hasMoreElements()) {

**return** **false**;

}

pending = parse(service, configs.nextElement());

}

nextName = pending.next();

**return** **true**;

}

分析hasNextSerice源码可以得出，在方法是使用我们传入的加载器的getResource方法加载资源文件META-INF/services目录下的文件（测试一是META-INF/services/com.sql.Driver），返回一个实现了Enumeration接口对象，

如果设置线程上下文类加载器为扩展类加载器，扩展类加载器调用getResources()方法 去加载META-INF/services/目录下的资源文件，

方法的执行结果就是找不到，因此在while循环直接返回false。而系统类加载器调用getResources()方法会正确返回。

### 分析MySQL驱动加载

Class.fromName(“com.mysql.jdbc.Driver”)加载Mysql驱动，我们对Class类中的fromName方法的理解，该方法会使用调用该方法的类加载器加载Driver类，并且进行初始化。我们看看Driver类初始化过程，初始化过程会执行类中的静态代块和对静态变量赋予初始值，我们观察Driver类的源代码，发现该类只有一个静态代码块。

static {  
 try {  
 DriverManager.registerDriver(new Driver());  
 } catch (SQLException var1) {  
 throw new RuntimeException("Can't register driver!");  
 }  
}

在执行该代码块时，调用了DriverManager类的静态方法registerDriver，从而会加载DriverManager类，并进行初始化。我们看看DriverManager源码：初始化时会执行该静态代码块

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

LoadInitalDrivers源码：

private static void loadInitialDrivers() {

String drivers;

try {

drivers = AccessController.doPrivileged(new PrivilegedAction<String>() {

public String run() {

return System.getProperty("jdbc.drivers");

}

});

} catch (Exception ex) {

drivers = null;

}

AccessController.doPrivileged(new PrivilegedAction<Void>() {

public Void run() {

try{

while(driversIterator.hasNext()) {

driversIterator.next();

}

} catch(Throwable t) {

}

return null;

}

});

println("DriverManager.initialize: jdbc.drivers = " + drivers);

if (drivers == null || drivers.equals("")) {

return;

}

String[] driversList = drivers.split(":");

println("number of Drivers:" + driversList.length);

for (String aDriver : driversList) {

try {

println("DriverManager.Initialize: loading " + aDriver);

Class.forName(aDriver, true,

ClassLoader.getSystemClassLoader());

} catch (Exception ex) {

println("DriverManager.Initialize: load failed: " + ex);

}

}

}

**分析源码我得到如下结果：**首先获取系统属性：jdbc.drivers，来加载驱动，如果获取失败，则通过ServiceLoader.load()方法来加载（会记在META-INF/Service目录下文件，从而找到驱动实现类）。加载成功后，会使用系统类加载器进行加载和初始化。

**DriverManager.getConnection(url,user,pwd)**:获取数据库连接。DriverManager会尝试的从注册过的驱动列表中选择一个合适的驱动

getConnection源码分析：

caller参数是调用者对应的Class对象。

private static Connection getConnection(String url, java.util.Properties info, Class<?> caller) throws SQLException {

//获取调用者对应的ClassLoader对象（差不多为Application ClassLoader）

ClassLoader callerCL = caller != null ? caller.getClassLoader() : null;

synchronized(DriverManager.class) {

if (callerCL == null) {

callerCL = Thread.currentThread().getContextClassLoader();

}

}

if(url == null) {

throw new SQLException("The url cannot be null", "08001");

}

println("DriverManager.getConnection(\"" + url + "\")");

SQLException reason = null;

//遍历已经加载的驱动

for(DriverInfo aDriver : registeredDrivers) {

//下面解释

if(isDriverAllowed(aDriver.driver, callerCL)) {

try {

println(" trying " + aDriver.driver.getClass().getName());

Connection con = aDriver.driver.connect(url, info);

if (con != null) {

// Success!

println("getConnection returning " + aDriver.driver.getClass().getName());

return (con);

}

} catch (SQLException ex) {

if (reason == null) {

reason = ex;

}

}

} else {

println(" skipping: " + aDriver.getClass().getName());

}

}

if (reason != null) {

println("getConnection failed: " + reason);

throw reason;

}

println("getConnection: no suitable driver found for "+ url);

throw new SQLException("No suitable driver found for "+ url, "08001");

}

}

isDriverAllowed源码分析：

private static boolean isDriverAllowed(Driver driver, ClassLoader classLoader)

{

boolean result = false;

if(driver != null) {

Class<?> aClass = null;

try {

aClass = Class.forName(driver.getClass().getName(), true, classLoader);

} catch (Exception ex) {

result = false;

}

result = ( aClass == driver.getClass() ) ? true : false;

}

我们知道在调用Class.fromName()方法时，会使用SPI方式（线程上下文类加载器）加载和初始化数据库驱动。而这里该方法又一次加载和初始化数据库驱动，这是因为命名空间的问题，使用SPI方法加载类，我们可以通过设置

Thread.currentThread.setContextClassLoader(ClassLoader loader)方法来设置我们上下文类加载器，如果我们改变了类加载器了，那么SPI加载的数据库驱动类就在它的命名空间中，而该方法主要验证了数据库驱动类的加载是不是在同一个命名空间中，即是不是同一个类加载器加载的。对于同一类，如果在不同的命名空间内，那么即使它们的名字完全一样，它们之间的相互转换是无法进行的，抛出异常（这里我们在命名空间已经讲解过了）。