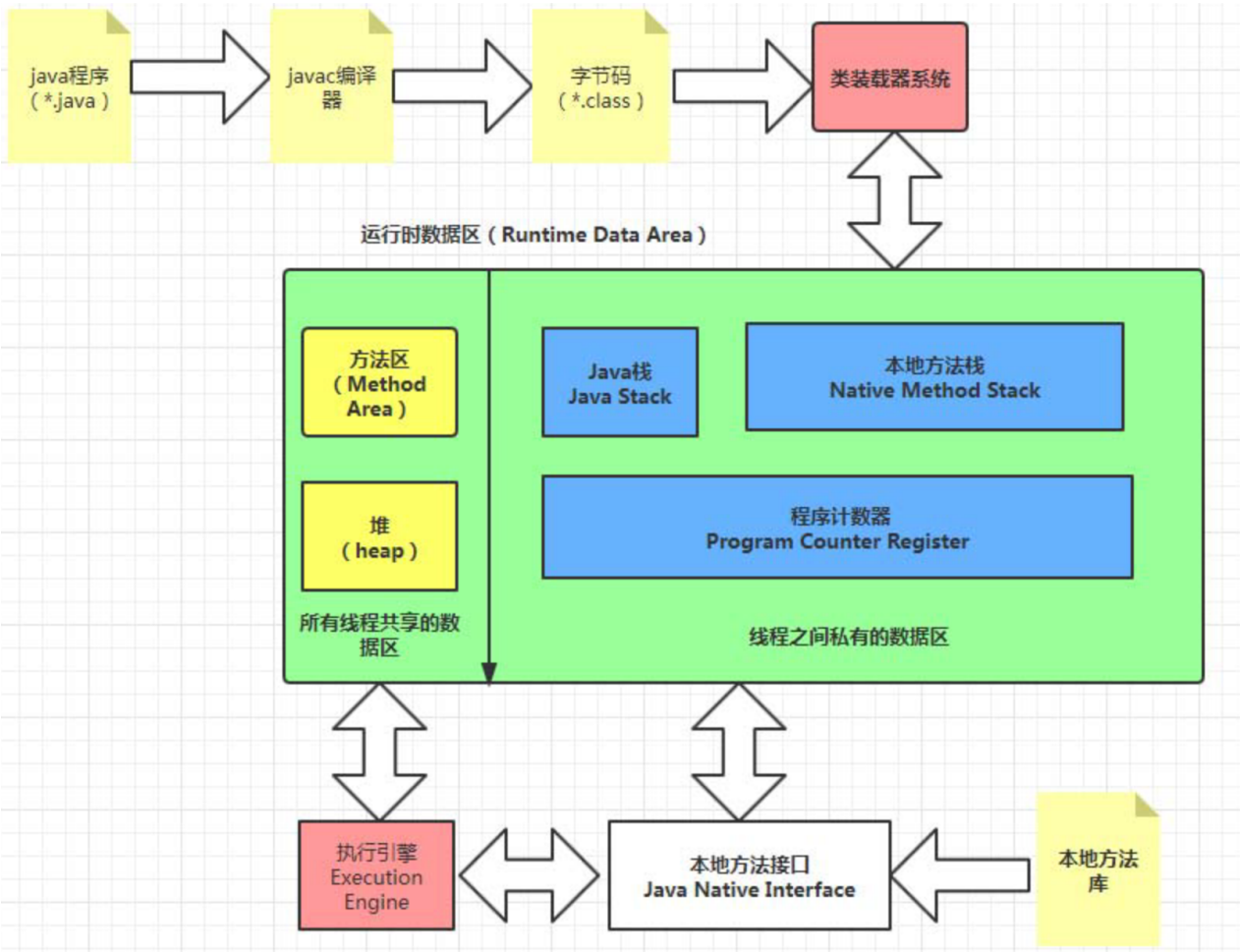
文章已收录我的仓库：Java学习笔记与免费书籍分享

类加载机制

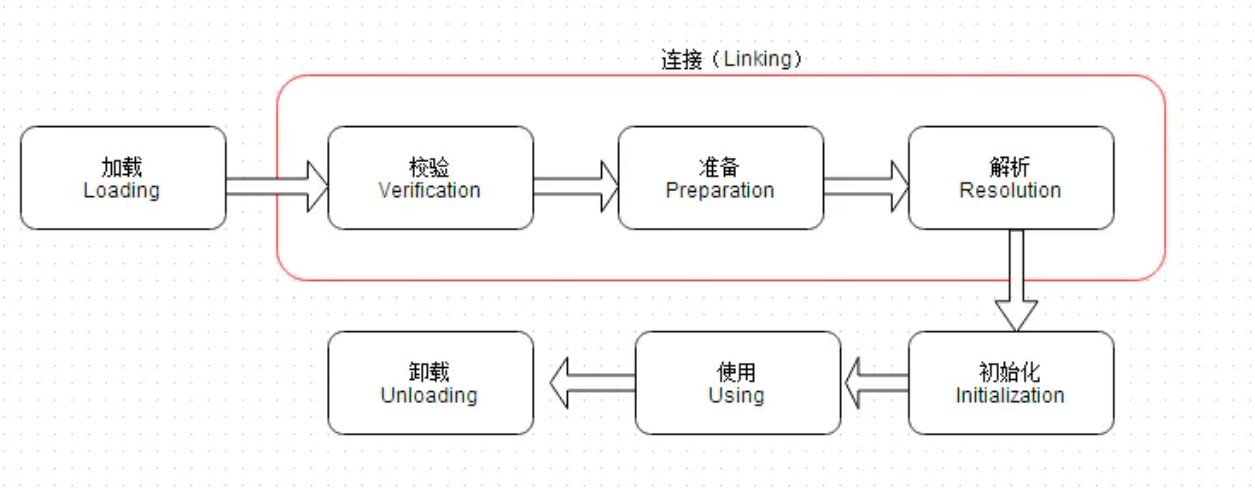
前言



前端编译后，.java文件已经变成了字节码存储在.class文件中，问题是，运行时虚拟机是如何将一个静态文件读入JVM中作为运行时数据的一部分呢？这就是本文章的中心 ——

类加载机制。

类加载的过程有哪些步骤呢？—— 家(加载) 宴(验证) 贝(准备) 斯(解析) 化(初始化)!



加载过程

加载

加载过程主要完成三件事情：

1. 通过类的全限定名来获取定义此类的二进制字节流(这包括了父类以及父接口)。
2. 将这个类字节流代表的静态存储结构转为方法区的运行时数据结构。
3. 在堆中生成一个代表此类的java.lang.Class对象，作为访问方法区这些数据结构的入口。

总的来说，就是在JVM运行时**方法区**中开辟一系列空间，以存储类的相关的类信息(方法、字段、类名等)，而类的相关信息是从 .class 文件读入的，JVM允许我们自己扩展加载器类以从不同位置读取 .class 文件，例如可以从网络中读取 .class 文件，亦或是从 ZIP 文件中读取，这提供了极大的灵活性，也为后来 JAR 包的发展奠定了基础。

如果需要自定义类加载器，我们只需继承系统的类加载器，并且重写 findClass 方法即可，请注意我们通常仅仅只能控制获取 .class 流的获取途径，步骤 2、3 还是需要交给系统类加载器完成，这也是为什么我们只需重写 findClass 的原因：

```
package com.happysnaker;


import java.io.File;
import java.io.FileInputStream;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;
```

```java
/**
 * @author Happysnaker
 * @description
 * @date 2021/10/23
 * @email happysnaker@foxmail.com
 */
public class Temp {
    public static void main(String[] args) throws ClassNotFoundException
{
        Class<?> dp = new
MyClassLoader("C://Tools.class").loadClass("com.happysnaker.Tools");
        System.out.println("类的名字是: " + dp.getSimpleName());

        Field[] fields = dp.getFields();

        for (Field field : fields) {
            System.out.println("该类有字段: " + field.getName());
        }

        Method[] methods = dp.getMethods();

        for (Method method : methods) {
            System.out.println("该有类方法: " + method.getName());
        }
    }
}

class MyClassLoader extends ClassLoader {
    //加载的文件路径
    String path;

    public MyClassLoader(String path) {
        this.path = path;
    }

    //name 是类的全限定名称
    @Override
    protected Class<?> findClass(String name) throws
ClassNotFoundException {
        File file = new File(path);
        List<Byte> bytes = new ArrayList<>();
        //读取 .class 文件的二进制流
        try (FileInputStream in = new FileInputStream(file)) {
            int b = 0;
            while ((b = in.read()) != -1) {
                bytes.add((byte) b);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        byte[] b = new byte[bytes.size()];
        for (int i = 0; i < b.length; i++) {
            b[i] = bytes.get(i);
        }
        //依赖系统类加载器帮我们完成其他操作
```

```
            return super.defineClass(name, b, 0, b.length);
        }


        /**
         * 重写 loadClass 打破双亲委派模型
         * @param name
         * @param resolve
         * @return
         * @throws ClassNotFoundException
         */

        /**
         * 重写 loadClass 打破双亲委派模型
         * @param name
         * @param resolve
         * @return
         * @throws ClassNotFoundException
         */
        @Override
        protected Class<?> loadClass(String name, boolean resolve) throws
    ClassNotFoundException {
            if (name.indexOf("java.") != -1) {
                return super.loadClass(name, resolve);
            }
            System.out.println("使用自定义类加载器加载！");
            return findClass(name);
        }
    }
```

输出：

```
使用自定义类加载器加载！
类的名字是: Tools
该类有字段: val
```

注意，所有包名以 java 开头的都会经过安全检查，这里我们加载的类继承了 Object，Object 类应该由父加载器加载，因此我们检查任何以 java. 开头的包名，将其交给父类加载器加载。

要注意经过加载过程后，仅仅只是类**的方法、字段、类名等元数据被加载至方法区中**，真正的**实例还没有被加载**出来，有必要区分 类加载过程 与 加载实例 的区别，**一个类通常只会被加载一次，加载后类元数据就已经存放在JVM方法区中了，后续便可以直接引用**。后面讲类加载模型时还会提到类加载器。

验证

Java是相对安全的语言，原因之一就是因为每个类在被加载前都经过相对严格的验证。此阶段主要确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚

拟机的自身安全。

1. 文件格式验证：
    - 验证字节码是否以魔术 0xCAFEBABE 开头(标识Java类文件)。
    - 编译、运行版本是否可接受。
    - …
2. 元数据验证：
    - 这个类是否存在父类，如果不存在则一定不合法。
    - 这个类是否继承了由 final 标识的类。
    - 这个类是否覆写了父类的 final 方法。
    - …
3. 字节码验证：
    - 字节码是否合法。
    - 字节码是否配对。
    - …
4. 符号引用验证：
    - 根据字符串描述的全限定名是否能找到对应的类。
    - 是否引用了符号引用类中的 private 等不可访问方法。

凡是有一个不合法的地方，JVM都会无情的拒绝加载并且抛出异常，我们在解析过程会介绍一下什么是符号引用。

准备

准备阶段JVM将为类变量分配内存，并将其初始化为默认值，这个阶段将正式在JVM堆中为对象实例开辟内存。

这里仅仅是为类字段分配内存，并赋默认值，这说明字段 a、b `int a = 123; String b = "123";` ;在这一阶段会被赋值为 0 和 null，当然这是有例外的，例如 `fianl int a = 123` ，此时 a 的值会被赋值 123，所有不可变常量在这一阶段将会被正常赋值，并且后续无法修改。

解析

解析阶段JVM将常量池中的符号引用转换为直接引用，**解析的主要目的是为了解析类或接口。**

这句话可能很难理解，我们一步一步来理解，首先，符号引用是什么？

- **符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中，只要能唯一的标识目标即可**。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。
- **直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄**。如果有了直接引用，那引用的目标必定已经在内存中存在

举个栗子：

```
class T {
    private String a = new String();
}
```

这里 a 引用了一个 String 对象(注意解析阶段并不真正分配对象，这是初始化阶段做的事)，但问题是，JVM根本不知道 String 类的具体信息(它只认识基本类型)，这导致JVM根本无法对 String 分配实例。

如果正常来说，**JVM应该要知道 String 类信息在方法区的具体位置，这就是对 String 类信息的直接引用**，知道了该直接引用，准备阶段就可以真正分配一个 String 实例。但此时 JVM 根本不知道 String 类在方法区中的具体位置，，**因此 JVM为 String 类信息分配一个符号引用 java.lang.String，该符号引用唯一的标识了 String 类，起到一个过渡的作用**。该符号引用被放置在类的常量池中。

解析阶段就是将符号引用转换为内存中的直接引用，通过上述例子你应该可以发现，符号引用仅仅只是针对于类和接口的元数据信息，解析阶段不会在这一过程去真正的分配对象实例。

解析有如下几种：

1. 类或接口的解析：
    - 如果要解析的类不是数组类型，则将符号引用的全限定名传递给当前类的类加载器去加载。注意这里的加载仅仅只是类加载机制中的第一步，并不是整个加载过程，解析阶段仅仅只需要解析出类的元数据信息所在的位置，而并不需要去加载类的实例。如果该类已经被加载了，那么类加载器会直接返回直接引用，否则，会经历我们上面所说的加载过程。
    - 如果是数组类型，则按照第一点去加载数组的元素类型(如果是基本类型，则转换成包装类型)。
2. 类字段解析（例如`int a = Test.a`，这里引用了 Test 的字段 a）：
    - 先对字段所处的类进行解析，在该类的方法区中查找字段，如果存在字段并且可访问，则返回这个字段的直接引用。
    - 否则，如果该类实现了接口，则按照继承关系从下向上搜索父接口，如果存在字段并且可访问，则返回这个字段的直接引用。

- 否则，如果该类不是Obeject，则按照继承关系从下向上搜索父类，如果存在字段并且可访问，则返回这个字段的直接引用。
- 否则，解析失败。

在这一步骤中，如果父接口或者父类未被解析，则会递归的去解析父接口或者父类。

3. 类方法解析：
    - 与类字段解析类似。
4. 接口方法解析：
    - 与类字段解析类似。

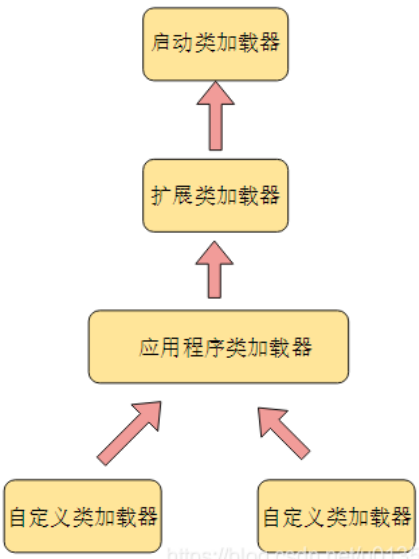
初始化阶段

经过解析阶段后，所有必备的信息都已经准备好了，例如代码`private String a = new String();`，现在JVM已经知道了 String 类信息在方法区的直接引用，JVM可以根据 String 类的信息去加载 String 对象实例了(不会经过加载步骤，类只会被加载一次)。

这一阶段 JVM 会按照顺序(这意味着父类的变量初始化肯定先于子类)收集**所有的变量赋值方法以及静态语句块的方法**，合并为`<clinit>()`方法，然后一起执行，此时一个类实例就算正式分配完成。

要注意初始化阶段可能发送在解析阶段之前，这是为了支持 Java 语言的动态绑定特性，这意味类可能提前返回而此时还未真正分配完成，volatile可以禁止此类指令重排，这也是为什么单例模式双重验证需要加 volatile 关键字。

双亲委派模型



在加载过程我们讲到可以从不同的地方加载二进制流，这虽然提高了灵活性，但如果不加以限制，也会带来糟糕的后果，例如，我们可以自定义加载器加载一个不知名的 Object

类，这下完蛋了，所有类到底要继承哪个 Object 呢？这不被 JVM 所允许!

为了解决这种问题，官方提出了这种双亲委派的加载模型，其工作原理是：

- 如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此。

  因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

使用这种模型后，即使我们自定义加载器去加载 Object 类，也会被委派给系统类加载器，**系统类加载器则会加载默认的 Object 类**，这样即可解决歧异。

从 Java 虚拟机的角度来讲，只存在以下两种不同的类加载器：

- **启动类加载器**（Bootstrap ClassLoader），使用 C++ 实现，是虚拟机自身的一部分
- **所有其它类的加载器**，继承自抽象类 java.lang.ClassLoader，使用Java实现，覆写 findClass 方法。

从 Java 开发人员的角度看，**类加载器可以划分得更细致一些**：

- **启动类加载器**（Bootstrap ClassLoader）：这个类加载器负责将存放在 <JRE_HOME>\lib 目录中的，或者被 -Xbootclasspath 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给启动类加载器，直接使用 null 代替即可。
- **扩展类加载器**（Extension ClassLoader，继承于启动类加载器）：这个类加载器是由 ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现的。它负责将 <JAVA_HOME>/lib/ext 或者被 java.ext.dir 系统变量所指定路径中的所有类库加载到内存中，开发者可以直接使用扩展类加载器。
- **应用程序类加载器**（Application ClassLoader，继承于应用程序类加载器）：这个类加载器是由 AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，因此一般称为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

> 在 Java 9 之后，为了支持模块之间的类加载，类加载器被重新划分：
>
> - 引导或启动类加载器：定义核心Java SE和JDK模块。
> - 平台类加载器：定义部分Java SE和JDK模块。
> - 应用或系统类加载器：定义CLASSPATH上的类和模块路径中的模块。

> 现在都是 jar 包的形式，jar 包仍然是处于我们的类路径下，扩展类路径几乎用不上了，因此扩展类加载器相当于被移除了，同时，jdk9 之后将 Java SE 和 JDK 模块进行了更细致的划分，例如 jdbc 由平台类加载器加载而非引导类加载器加载，有关于 jdbc 知识请参考我的另一篇文章 [SPI 与 jdbc](./SPI 机制以及 jdbc 打破双亲委派.pdf)，这篇文章将会基于 jdk11 讲解。
>
> 这些知识点其实都是很重要的，毕竟我们平时应该也不会使用 jdk8 进行开发，现在都已经更新到 jdk18了，但奈何 jdk 迭代过快，并且 jdk8 最为经典，绝大多书籍仍然是基于 jdk8 进行讲解的。

我们自己编写的类是由应用程序类加载器加载的，应用程序类加载器首先会递归向上查询，如果启动类加载器和扩展类加载器均不能加载，则由应用程序类加载器默认加载。

来看看 ClassLoader 中 loadClass 的源码实现：

```
    protected Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been loaded
            //第一，检查该类是否已经被加载过了
            Class<?> c = findLoadedClass(name);
            //如果为被加载，则加载，否则直接返回
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    //如果 parent == null，说明该类就是启动类加载器，则尝试自己
加载，否则交由父类加载
                    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
                }
                //如果父类仍然无法加载，则尝试自己加载
                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    long t1 = System.nanoTime();

                    //这是自己加载的核心代码，我们通常只需要覆写 findClass 方法
即可
                    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) {
                resolveClass(c);
            }
            return c;
        }
    }
```

源码理解起来十分简单，为了不破坏双亲委派，只推荐重写 findClass 方法(模板方法设计模式)。

双亲委派模型可以被打破吗？

> 例如 JDBC 连接可能会使用其他的 Drive 类，而 JDBC 肯定算是基础类了，正常是由启动器去加载 home 路径下的 Driver，而连接可能会使用用户指定的 Driver 类，这就必须破坏双亲委派模型。

当然可以，最暴力的方法就是直接覆写 loadClass 方法，这当然有效。不过在 defineClass 代码中，JVM 仍然会检查全限定名是否以 java 开头，如果是的话，仍然会交给启动类。

> 参考我们上面自定义的类加载器，所有类都继承了 java.lang.Object，如果不检查 java 开头的包名将会报错。
>
> ```
> 抛出:       ClassFormatError – 如果数据不包含有效的类
>             IndexOutOfBoundsException – 如果off或len为负数，或者off+len大于b.
>             length 。
>             SecurityException – 如果尝试将此类添加到包含由与此类不同的证书集（未签
>             名）签名的类的包中，或者如果name以“ java. ”开头。
> 自从:       1.1
> 也可以看看: loadClass(String, boolean) , resolveClass(Class) , CodeSource ,
>             java.security.SecureClassLoader
> protected final Class<?> defineClass(String name, byte[] b, int off, int len)
> ```

再就是通过线程上下文加载器(ContextClassLoader)去加载，大部分 SPI 都是利用线程上下文加载器去加载的，ContextClassLoader有 set 和 get 方法可以对加载器进行设置，如果线程创建时还未设置，则从父进程继承而来。

```
package com.happysnaker;
import javax.management.loading.MLet;

public class Temp {
    public static void main(String[] args) throws ClassNotFoundException,
InterruptedException {
        var t1 = Thread.currentThread().getContextClassLoader();
        System.out.println(t1); //默认是应用文加载器

        //设置成自定义加载器
        Thread.currentThread().setContextClassLoader(new
```

```
MyClassLoader());
        var t2 = Thread.currentThread().getContextClassLoader();
        System.out.println(t2); //打印自定义加载器

        var thread = new Thread(new Runnable() {
            @Override
            public void run() {
                //从父线程继承，默认是自定义加载器
                var t3 = Thread.currentThread().getContextClassLoader();
                System.out.println(t3);

                Thread.currentThread().setContextClassLoader(new MLet());
            }
        });

        thread.start();
    }
}
class MyClassLoader extends ClassLoader {
}
```

在多数SPI加载过程中，子类设置线程上下文加载器，一步一步传递给高层，最终高层获取到了子类的加载器，进行加载，即逆向打通了双亲委派模型，现在高层用的是子类的加载器了。

> 不过本质仍然是改写 loadClass 方法。

线程上下文类加载器的适用场景：

- 当高层提供了统一接口让低层去实现，同时又要是在高层加载（或实例化）低层的类时，必须通过线程上下文类加载器来帮助高层的ClassLoader找到并加载该类。
- 当使用本类托管类加载，然而加载本类的ClassLoader未知时，为了隔离不同的调用者，可以取调用者各自的线程上下文类加载器代为托管。

> SPI机制简介 SPI的全名为Service Provider Interface，主要是应用于厂商自定义组件或插件中。在java.util.ServiceLoader的文档里有比较详细的介绍。简单的总结下java SPI机制的思想：我们系统里抽象的各个模块，往往有很多不同的实现方案，比如日志模块、xml解析模块、jdbc模块等方案。面向的对象的设计里，我们一般推荐模块之间基于接口编程，模块之间不对实现类进行硬编码。一旦代码里涉及具体的实现类，就违反了可拔插的原则，如果需要替换一种实现，就需要修改代码。为了实现在模块装配的时候能不在程序里动态指明，这就需要一种服务发现机制。
> Java SPI就是提供这样的一个机制：为某个接口寻找服务实现的机制。有点类似IOC的思想，就是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要。
> SPI具体约定 Java SPI的具体约定为：当服务的提供者提供了服务接口的一种实现之后，在jar包的META-INF/services/目录里同时创建一个以服务接口命名的文件。该文件里就是实现该服务接口的具体实现类。而当外部程序装配这个模块的时候，就能通过该jar包META-INF/services/里的配置文件找到具体的实现类名，并装载实例

> 化，完成模块的注入。基于这样一个约定就能很好的找到服务接口的实现类，而不需要再代码里制定。jdk提供服务实现查找的一个工具类：java.util.ServiceLoader。

在下一篇博客中我将讲解 SPI 机制以及 jdbc 如何破坏双亲委派模型。