文章已收录我的仓库：Java学习笔记

SPI 机制以及 JDBC 打破双亲委派

> 本文基于 jdk 11

SPI 机制

简介

何为 SPI 机制？

SPI 在Java中的全称为 Service Provider Interface，是JDK内置的一种**服务提供发现机制**，是Java提供的一套用来被**第三方实现或者扩展的API**，它可以用来启用框架扩展和替换组件。

例如本文的中心 jdbc，我们可以使用统一的接口 `Connection` 去操控**各种数据库**，但你有没有想过，难道 jdk 真的内置了所有的数据库驱动吗？

显然这是不可能的，我们平时使用需要我们自己导入对应 jar 包去使用不同的数据库，例如 MySQL、SqlServer 数据库等。**可是，我们使用的 `Connection conn = DriverManager.getConnection(url,user,pass)` 又是确确实实是 jdk 内置的一个接口**，其实这就是一种 SPI 机制，**即官方定义好一个接口，由不同的第三方服务者去实现这个接口。**

那么问题来了，SPI 机制是如何实现的呢？

答案很简单，**遵守官方的约定**！

jdk SPI 原理

SPI 可以有不同实现，但无论怎样核心思想都是**遵守官方规则**。官方不一定要是 jdk，例如 SpringBoot 就定义了一套规则，但我们这里仍然讲 jdk 的实现原理。

实现 SPI 机制的关键点在于要知道**接口的具体实现类是哪一个**，这些实现类是第三方服务提供的，必须得有一种方法让 JVM 知道使用哪个实现类，因此官方定义了如下规则：

- 服务提供者的类必须要实现官方提供的接口（或继承一个类）。
- 将该具体实现类的全限定名放置在资源文件下 META-INF/services/${interfaceClassName} 文件中，其中 ${interfaceClassName} 是接口的全限定名。
- 如果扫描到多个具体实现类，jdk 会初始化所有的这些类。

这就是 jdk 的约定，既然要求服务者提供的类名放在 META-INF/services/${interfaceClassName} 文件下，那么官方肯定要去扫描这个文件，官方提供了一个实现：`ServiceLoader.load` 方法。

获取具体类实例的伪代码如下：

```
ServiceLoader load = ServiceLoader.load(XXX.class);
for (XXX x : load) {
    // o 就是我们要获取的实例，XXX 是一个官方定义的接口接口
    System.out.println(x);
}
```

ServiceLoader.load 方法返回一个 ServiceLoader 实例，我们需要通过遍历其**迭代器**去获取所有可能的实例，核心代码就在迭代器中了。

我们来看看这个迭代器的源码，我们主要看 hasNext 方法，因为 next 方法本身依赖于 hasNext 方法：

```
public Iterator<S> iterator() {
    return new Iterator<S>() {
        int index;
        @Override
        public boolean hasNext() {
            if (index < instantiatedProviders.size())
                return true;
            return lookupIterator1.hasNext();
        }
    };
}
```

跳到了 lookupIterator1.hasNext() 方法中，lookupIterator1 是调用 newLookupIterator() 方法返回的，来看看这个方法：

```
private Iterator<Provider<S>> newLookupIterator() {
    Iterator<Provider<S>> first = new ModuleServicesLookupIterator<>();
    Iterator<Provider<S>> second = new LazyClassPathLookupIterator<>();
    return new Iterator<Provider<S>>() {
        @Override
```

```
        public boolean hasNext() {
            return (first.hasNext() || second.hasNext());
        }
        @Override
        public Provider<S> next() {
            if (first.hasNext()) {
                return first.next();
            } else if (second.hasNext()) {
                return second.next();
            } else {
                throw new NoSuchElementException();
            }
        }
    };
}
```

可以发现主要有两种加载方式，一种是模块化的加载，另一种是普通的懒加载方式，我们应该会进到 second.hasNext() 方法：

```
@Override
public boolean hasNext() {
    if (acc == null) {
        return hasNextService();
    } else {
        PrivilegedAction<Boolean> action = new PrivilegedAction<>() {
            public Boolean run() { return hasNextService(); }
        };
        return AccessController.doPrivileged(action, acc);
    }
}
```

second.hasNext() 方法又调用了 hasNextService() 方法，来看看这个方法的逻辑：

```
private boolean hasNextService() {
    while (nextProvider == null && nextError == null) {
        try {
            Class<?> clazz = nextProviderClass();
            if (clazz == null)
                return false;
            if (service.isAssignableFrom(clazz)) {
                Class<? extends S> type = (Class<? extends S>) clazz;
                Constructor<? extends S> ctor  = (Constructor<? extends
S>)getConstructor(clazz);
                ProviderImpl<S> p = new ProviderImpl<S>(service, type,
ctor, acc);
                nextProvider = (ProviderImpl<T>) p;
            }
        } catch (ServiceConfigurationError e) {
            nextError = e;
        }
    }
```

```
    return true;
}
```

这个方法首先判断 nextProvider 是不是空，如果不是的话说明已经有资源，直接返回；我们是初次加载，肯定为空，因此进入循环，可以看到主要的方法就是 nextProviderClass()，通过这个方法返回一个构造器，然后进行实包装，并设置 nextProvider 等于包装后的 ProviderImpl，有了 ProviderImpl，自然可以通过反射获取实例！

这里核心方法应该是 nextProviderClass() ：

```
static final String PREFIX = "META-INF/services/";
private Class<?> nextProviderClass() {
    if (configs == null) {
        // 路径名
        String fullName = PREFIX + service.getName();
        // 根据路径名取得资源，这个 loader 是线程上下文取得的
        configs = loader.getResources(fullName);
    }
    // pending 是一个 String 的迭代器
    while ((pending == null) || !pending.hasNext()) {
        if (!configs.hasMoreElements()) {
            return null;
        }
        pending = parse(configs.nextElement());
    }
    String cn = pending.next();
    try {
        return Class.forName(cn, false, loader);
    } catch (ClassNotFoundException x) {
        return null;
    }
}
```

可以看到在这一步通过 PREFIX + service.getName() 获取文件名，这个就是我们说的 `META-INF/services/${interfaceClassName}` 约定的文件，然后取得对应资源，pending 是一个 String 的迭代器，其内就是具体实现类的**全限定名**，如果这个迭代器存在，说明已经解析过了，则直接返回 `Class.forName(pending.next(), false, loader)` 。

如果 pending 迭代器到底了，那么会再一次进入循环，并调用 configs.nextElement() 再次读取，**这也就是说明如果有多个配置文件，那么所有的类都会被加载！**直到真的什么都没有了，那么抛出异常，返回 null，这里返回 null 的话，那么hashNext 方法就会返回 false。

如果是第一次解析，那么会进入到 `pending = parse(configs.nextElement());` 方法，来看看这个方法：

```
private Iterator<String> parse(URL u) {
    Set<String> names = new LinkedHashSet<>(); // preserve insertion
order
    URLConnection uc = u.openConnection();
    uc.setUseCaches(false);
    try (InputStream in = uc.getInputStream();
         BufferedReader r = new BufferedReader(new InputStreamReader(in,
UTF_8.INSTANCE))) {
        int lc = 1;
        while ((lc = parseLine(u, r, lc, names)) >= 0);
    }
    return names.iterator();
}
```

这个方法很简单，就是读取配置中的每一行，添加到 Set 中，然后返回其迭代器。所以我们将返回一个配置文件中所有的全限定名！

这就是 jdk 提供的 SPI 机制的具体实现原理！

> 注：上述源码经过了些许简化

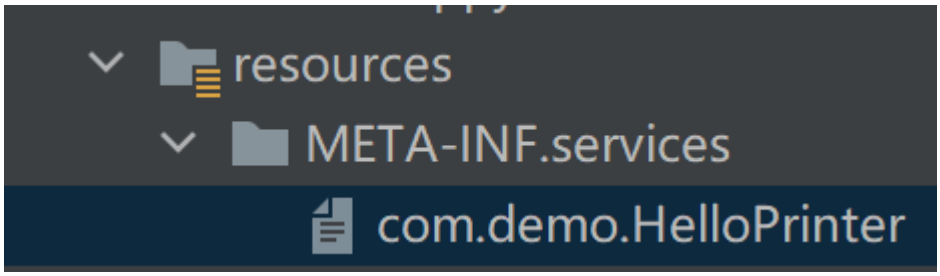
写一个 demo

假如我们有一个 HelloPrinter 的接口，这个接口需要第三方来提供，则可以这样编写来获取具体实现类：

```
public class Test {
    public static void main(String[] args) {
        ServiceLoader<HelloPrinter> load =
ServiceLoader.load(HelloPrinter.class);
        Iterator<HelloPrinter> iterator = load.iterator();
        while (iterator.hasNext()) {
            HelloPrinter helloPrinter = iterator.next();
            helloPrinter.hello();
        }
    }
}
```

试问，这里有没有出现任何关于具体实现类的信息？没有！具体实现类对客户来说是完全透明的，客户只知道 HelloPrinter 这个接口！现在什么都没做，这个代码什么也不会输出。

然后我们启动另一个项目写两个实现类 HelloPrinterImpl1 和 HelloPrinterImpl2，按照约定建立如下目录：



在这个文件中填写实现者的全限定名：

```
com.demo.HelloPrinterImpl1
com.demo.HelloPrinterImpl2
```

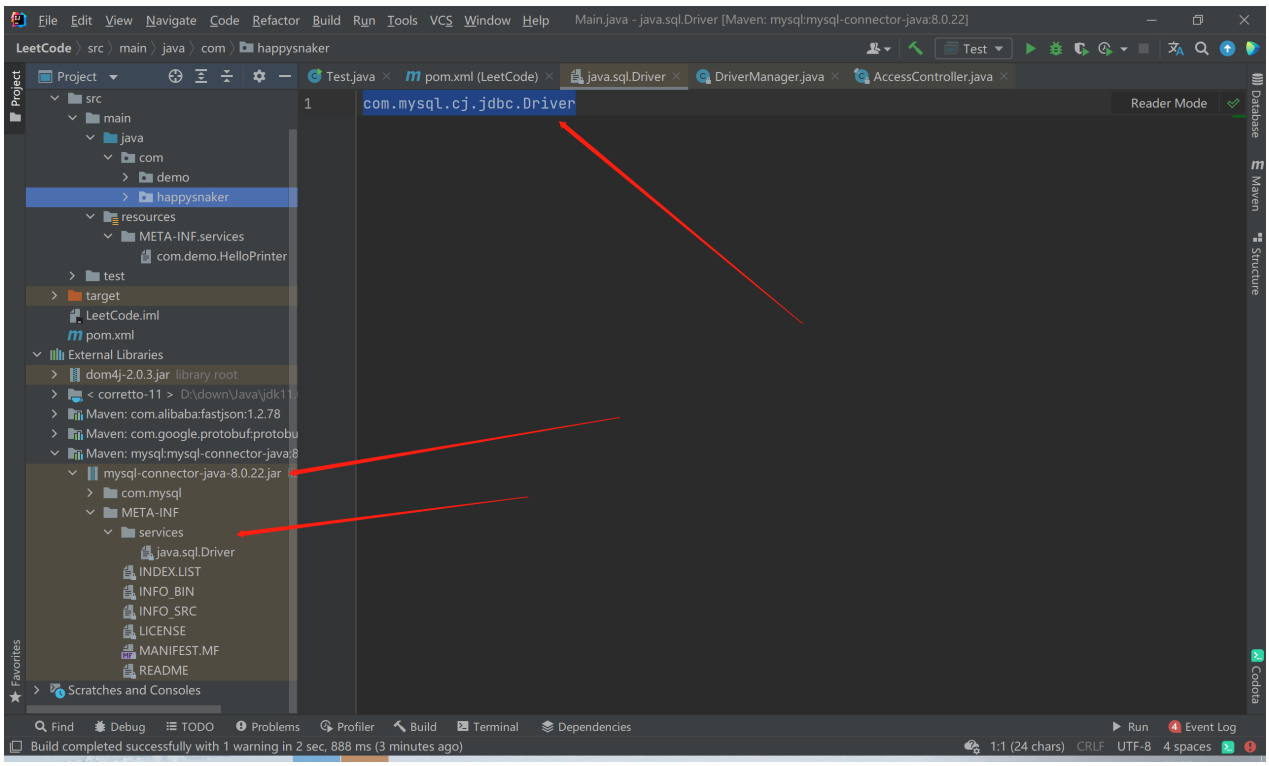
然后 maven 打包，让客户端引入这个 jar 包，重新运行，现在客户端的输出是：

```
我是第一个实现者！
我是第二个实现者！
```

JDBC 打破双亲委派模型

> 本节前置知识：类加载机制

JDBC 其实也是一种 SPI 机制，例如当我们引入 MYSQL 驱动的 jar 包时：



可以发现这与我们上面讲的一模一样，由此可见我们使用的驱动应该是“com.mysql.cj.jdbc.Driver”这个类。

当引入了 jar 包之后，则可以通过代码：

```
Connection connection =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/xxx?
serverTimezone=GMT", "root", "123456");
```

来获取数据库连接，Connection 是官方的接口，无论什么数据库都可以一样操作，非常方便。

但我们为什么说 jdbc 打破了双亲委派模型呢？

原因在于 DriverManager.getConnection 方法其实是加载了 com.mysql.cj.jdbc.Driver 这个类，然后这个驱动去创建连接。

问题是 DriverManager 是属于 jdk 的官方类，应当是由引导类加载器（jdk8 以前叫启动类加载器）加载的，而 com.mysql.cj.jdbc.Driver 这个类明显不能由引导类加载器加载，而是由应用类加载器（也叫系统类加载器）加载。

```
public class Test {
    public static void main(String[] args) throws Exception {

System.out.println(DriverManager.class.getClassLoader().getName());
        Class c = Class.forName("com.mysql.cj.jdbc.Driver");
        System.out.println(c.getClassLoader().getName());
    }
}
```

输出是：

```
platform
app
```

可见确实是由平台类加载器和应用类加载器去加载这两个类，问题是 DriverManager 要如何加载 com.mysql.cj.jdbc.Driver 驱动，直接使用 Class.forName 可行吗？

不可行！因为 Class.forName **默认会采用调用者自己的类加载器**去加载，DriverManager 的类加载器是平台类加载器，显然加载不到驱动类。

所以 DriveManager 必须要使用次一级的类加载器去加载，这里是使用了 SPI 机制，在 getConnection 方法中，我们调用了一行代码：

```
private static Connection getConnection(
    String url, java.util.Properties info, Class<?> caller) throws
SQLException {
    // 省略
    ensureDriversInitialized();
    // 省略
    return driver.connect(url, info);
```

```
}
```

ensureDriversInitialized() 这个方法：

```
private static void ensureDriversInitialized() {
    synchronized (lockForInitDrivers) {
        if (driversInitialized) {
            return;
        }
        String drivers;
        AccessController.doPrivileged(new PrivilegedAction<Void>() {
            public Void run() {
                ServiceLoader<Driver> loadedDrivers =
ServiceLoader.load(Driver.class);
                Iterator<Driver> driversIterator =
loadedDrivers.iterator();
                while (driversIterator.hasNext()) {
                    driversIterator.next();
                }
                return null;
            }
        });
        driversInitialized = true;
    }
}
```

可以看到在这个方法内调用了 loadedDrivers = ServiceLoader.load(Driver.class) 方法，然后遍历迭代器，driversIterator.next() 相当于实例化了 Drive，别看它好像啥也没做，但事实上在实例化的过程中，触发了 Drive 类的初始化语句块的调用：

```
public class Driver extends NonRegisteringDriver implements
java.sql.Driver {
    public Driver() throws SQLException {
    }

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

它向 DriverManager 注册了自己！因此 DriverManager 可以保存这个 Driver！

ServiceLoader 加载实现原理上面已经说了，但我故意没讲 ServiceLoader.load 方法：

```
@CallerSensitive
public static <S> ServiceLoader<S> load(Class<S> service) {
    ClassLoader cl = Thread.currentThread().getContextClassLoader();
    return new ServiceLoader<>(Reflection.getCallerClass(), service, cl);
}
```

load 方法内，设置默认的加载器为线程上下文加载器，这个线程上下文类加载器在上一篇文章已经详细说了，就不再赘述了。

由于我们在主线程调用的 DriverManager.getConnection 方法，现在已经很明显了，ServiceLoader 使用了主线程的类加载器去加载，这当然是应用加载器！

现在再来总结一下为什么说 jdbc 打破了双亲委派，我认为有两点：

1. DriverManager 属于 jdk 官方类，使用平台加载器，然而却使用了应用加载器去加载 Driver 类，这相当于是高层次的类使用了低层次的类加载器，这算是逆向打通了双亲委派模型。
2. 另外就是 ServiceLoader 使用线程上下文加载器直接获得类加载器，而没有一层一层向上调用，这肯定也是违反了双亲委派模型的。