

1、概念

SPI(Service Provider Interface), 它是从 Java6 开始引入的, 是一种基于 ClassLoader 来发现并加载服务的机制。
一个标准的 SPI, 由 3 个组件构成, 分别是:

1. Service: 是一个公开的接口或者抽象类, 定义了一个抽象的功能模块
2. Service Provider: 则是 Service 接口的一个实现类
3. ServiceLoader: 是 SPI 机制中的核心组件, 负责在运行时发现并加载 Service Provider

2、原理

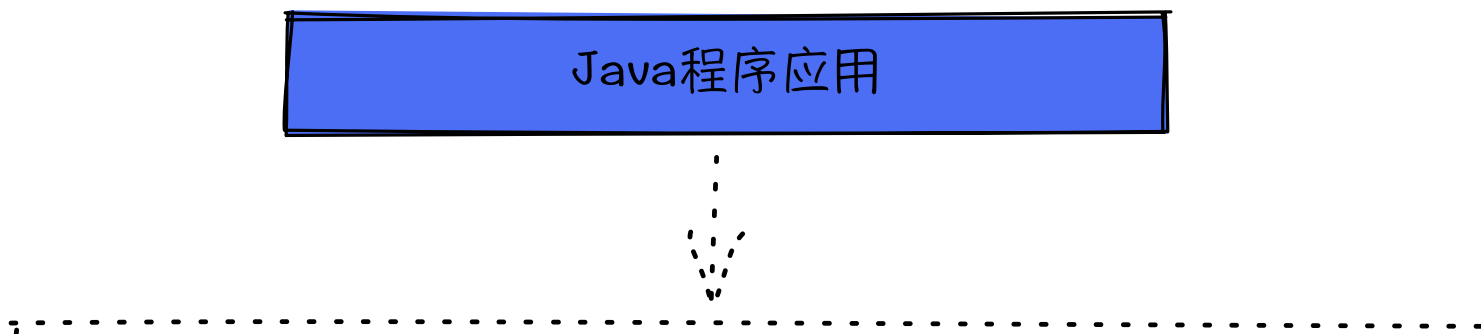
Application 应用程序先调用 ServiceLoader 中的 load 方法, Service 接口作为参数传入, 此时就会去加载当前应用中有关于这个接口的所有 Service Provider, 获取到 Service Provider 之后就可进行下一步操作了。

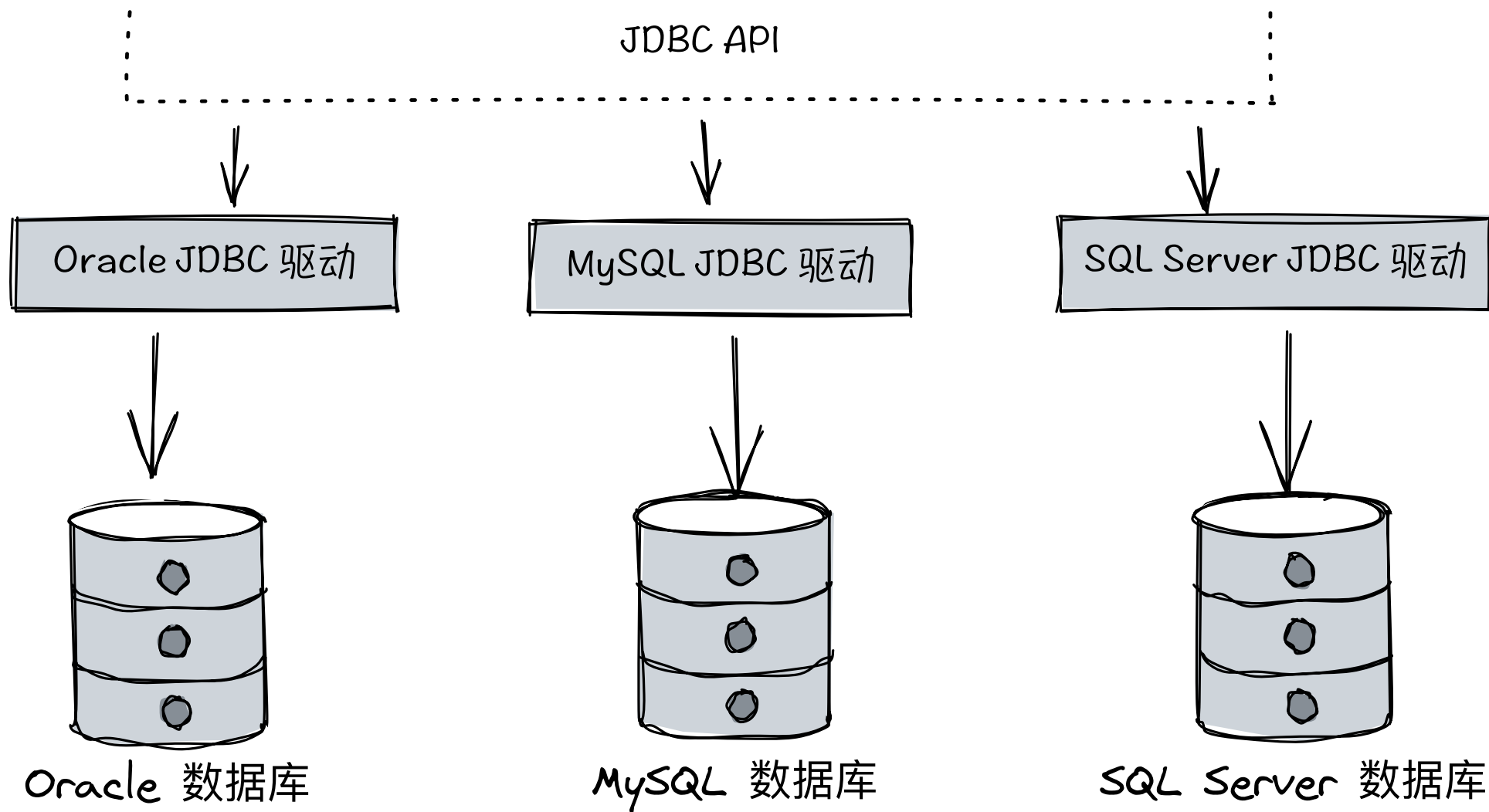
💡 值得注意的是, Application 应用程序不用关注 Service 的具体实现, 它只需和 Service 接口交互即可。

3、SPI 在 JDBC 中的应用

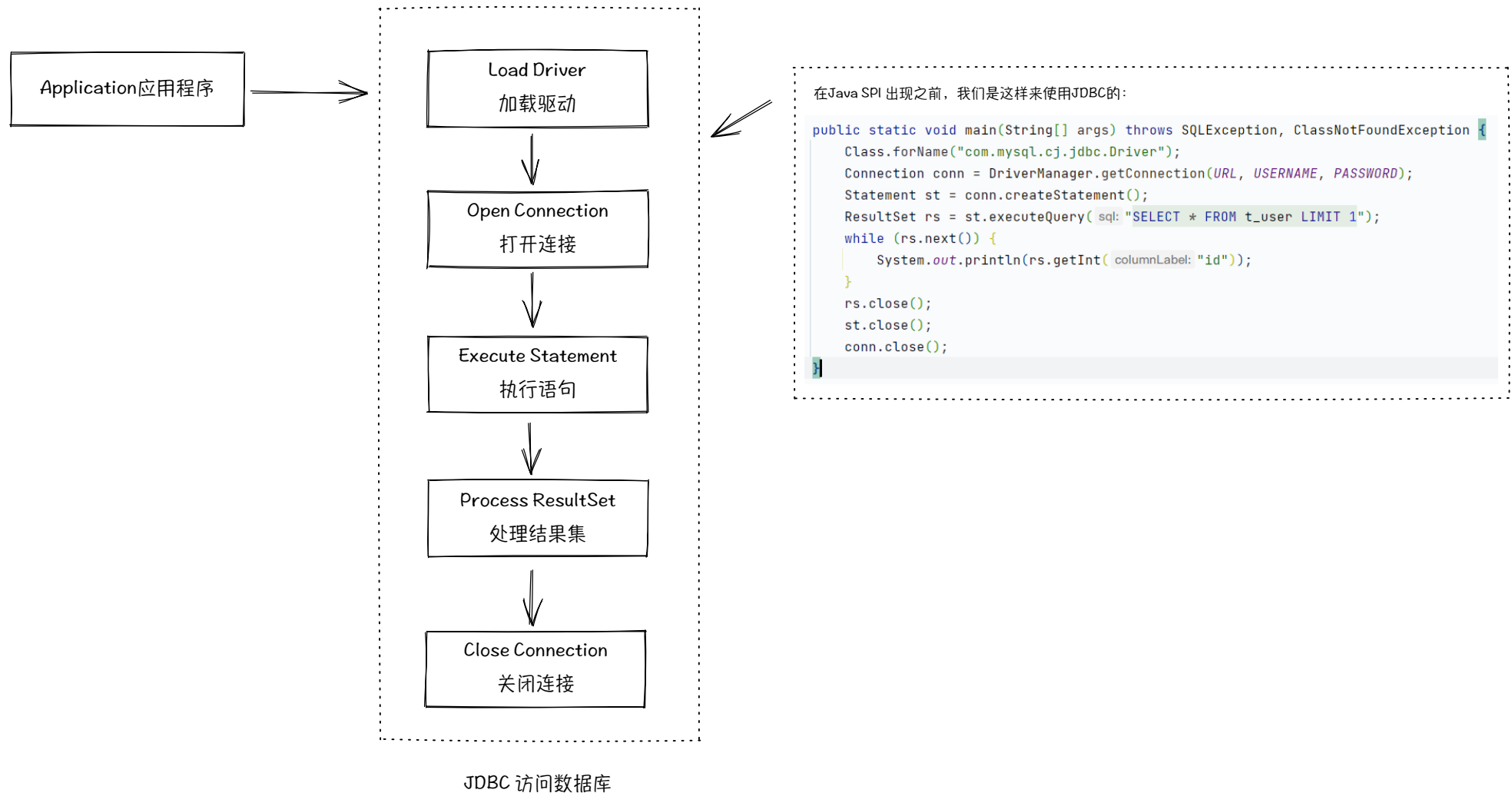
JDBC, 全称是 Java DataBase Connectivity。

- JDBC 即使用 Java 语言来访问数据库的一套 API
- 每个数据库厂商会提供各自的 JDBC 实现





JDBC 的调用流程:



在 Java SPI 出现之前，程序们使用 `Class.forName` 来加载数据库驱动：

```
// 加载 MySQL 数据库驱动
Class.forName("com.mysql.cj.jdbc.Driver");

// 加载 Oracle 数据库驱动
```

```
Class.forName("oracle.jdbc.driver.OracleDriver");

// 加载 SqlServer 数据库驱动
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

题外话：JDBC 要求 Driver 实现类在类加载的时候，能将自身的实例对象自动注册到 DriverManager 中，从而加载数据库驱动。
MySQL 中的 Driver 类源码，使用 `Class.forName("com.mysql.cj.jdbc.Driver");` 可以往 DriverManager 中注册 Driver。

```
package com.mysql.cj.jdbc;

import ...

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!");
        }
    }
}
```

看着这些硬编码的类名，一个有素养的程序员，自然而然就会想到。

细 B：咦！？这些类名是不是可以写到配置文件中呢？这样我更换数据库驱动时，就不用修改代码了。如：

```
dirver-name: com.mysql.cj.jdbc.Driver
```

细 B: 不过, 这好像还是不够完美。。。我还需要记住不同的数据库厂商提供的 `Driver` 的类名, 这也太麻烦了吧! 本来头发就已经不多了啦~

细 B: 能不能和数据库厂商商量一下, 干脆让他们把配置文件也一并提供得了? 这样一来, 程序员省事, 数据库厂商也省事, 程序员不用了解具体的驱动类名, 而厂商也可以启动升级驱动。

大 A: 听起来是个好主意! 问题是如果由厂商提供配置文件, 我们如何去读取它呢?

细 B: 还记得 `ClassLoader` 吗? 它除了可以加载类之外, 还提供了方法 `getResource/getResources`, 可以根据指定的路径, 读取 `classpath` 中对应的文件。我们可以用它来读取厂商放在 `jar` 包中的配置文件, 当然我们要事先约定好配置文件的路径和格式就行。

大 A: 你 TN 的还真是个天才!!! 这套机制, 我们就叫它 SPI 吧!

Important

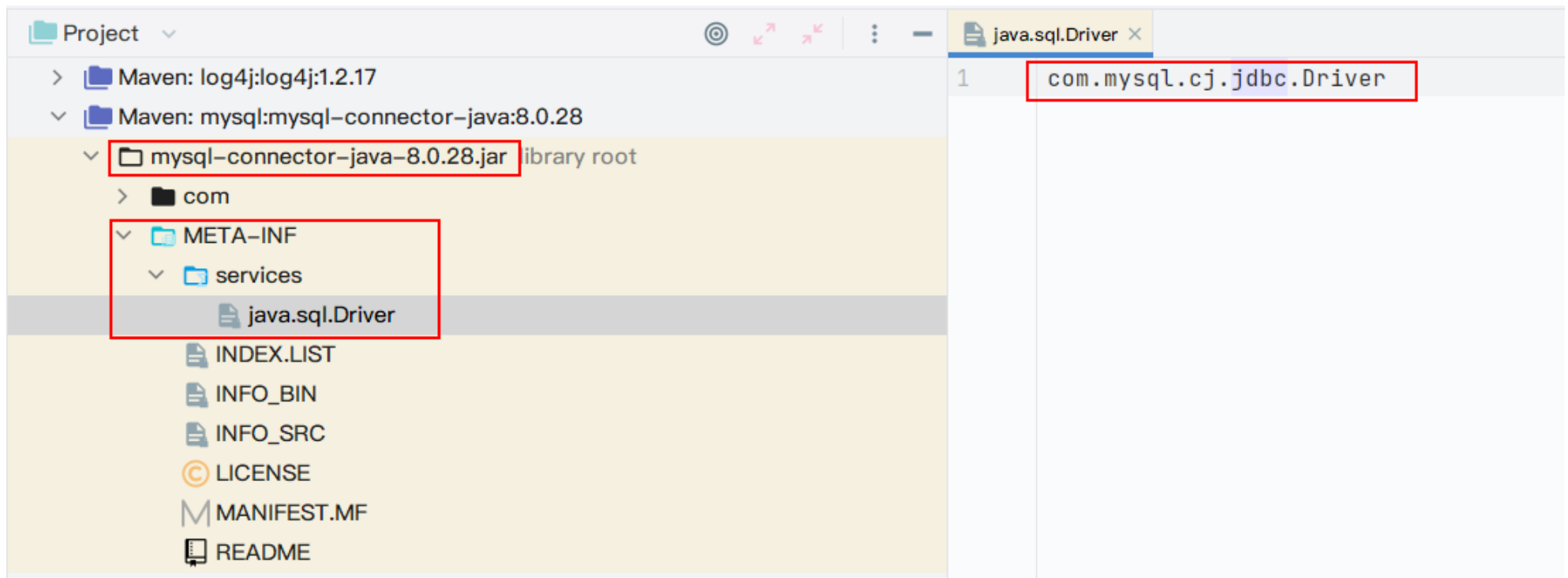
最终演化为: 使用 Java SPI 机制来加载数据库驱动, 这样做的好处就是无需再使用 `Class.forName` 来加载数据库驱动, 只需引入所需的数据库驱动 `jar` 包即可, 就算更换数据库, 也只需要更换所依赖的 `jar` 包就行, 而不需要修改代码

4、SPI 的三大规范要素

1、规范的配置文件

- 文件路径: 必须在 `jar` 包中的 `META-INF/services` 目录下
- 文件名称: `Service` 接口的全限定名
- 文件内容: `Service` 实现类的全限定名。如果有多个实现类, 那么每一个实现类在文件中单独占据一行

以 MySQL 为例, 查看 `mysql-connector-java` 的 `jar` 包, 可以看到目录 `META-INF/services` 下确实存在一个名为 `java.sql.Driver` 的配置文件, 而文件内容则是 MySQL 的数据库驱动类。



2、Service Provider 类必须具备无参构造方法

Service 接口的实现类，即 Service Provider 类，必须具备无参构造方法。因为随后会通过反射技术实例化它，是不带参数的。以 MySQL 为例，从上面 MySQL 驱动类的截图中可以看出，确实存在一个无参构造方法。

3、保证能加载到配置文件和 Service Provider 类

- 方式 1：将 Service Provider 的 jar 包放到 classpath 中(最常用)
- 方式 2：将 jar 包安装到 JRE 的扩展目录当中
- 方式 3：自定义一个 ClassLoader

以 MySQL 为例，我们只需通过 maven 将 mysql 的驱动 jar 包作为依赖引入后，JDBC 就会自动加载 mysql 的数据库驱动。

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.28</version>
</dependency>
```

5、总结

- 作用：提供一种**组件发现与注册**的方式，可以用于实现各种插件，或者灵活替换框架所使用的组件。
- 优点：基于面向接口编程，优雅地实现模块之间的**解耦**。
- 设计思想：**面向接口 + 配置文件 + 反射技术**，YYDS!!!
- 应用场景：JDBC、SLF4J、Servlet 容器初始化...

6、手撕一个 SPI 应用实例

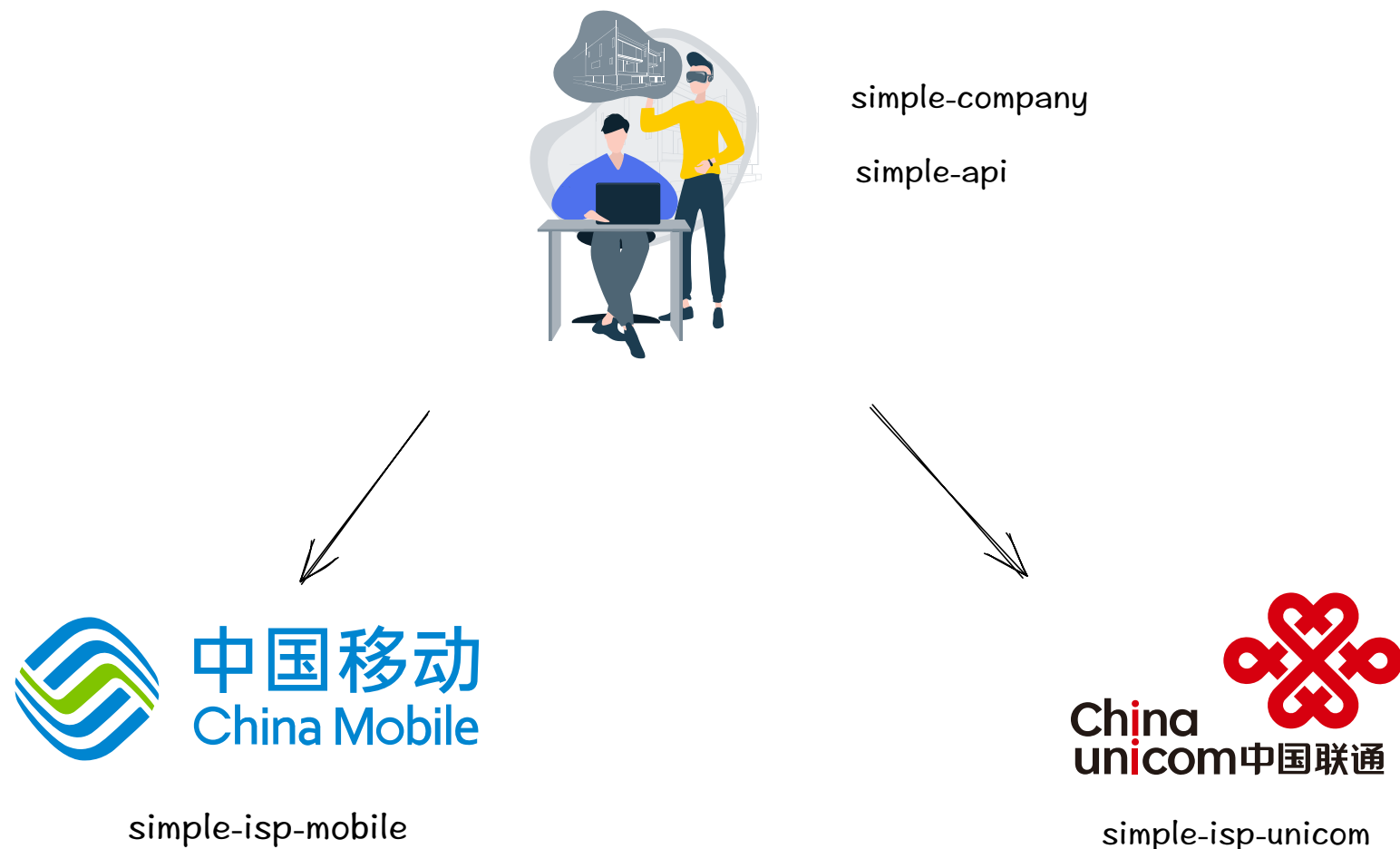
1、背景

我们从从开始手撸一个 SPI 应用，在正式撸码之前，我们先介绍一下背景：

假设有一家公司A，它需要连接互联网。它定义了一个连接网络的API，由中国移动和中国联通来提供网络服务。

那么这个场景设计到了三方：1. 公司A；2. 中国移动；3. 中国联通。

其中公司A负责开发项目simple-company和simple-api，simple-company代表的是业务应用程序，simple-api即SPI中的Service接口，而移动和联通则以jar包的形式分别提供各自的联网服务

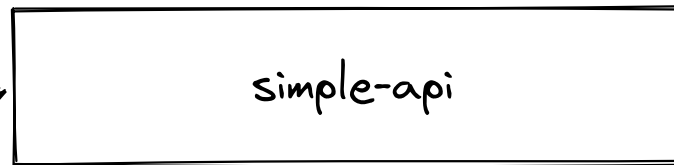


项目关系图如下：simple-company 调用 simple-api，而 simple-isp-mobile 和 simple-isp-unicom 则实现了 simple-api。

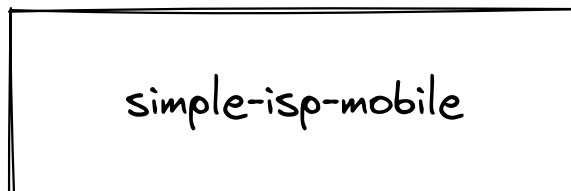
Application



Service



Service-Provider



2、代码编程：

Important

关于本案例的代码已上传自[github](#)，有需要的小伙伴可自行下载。

1、simple-api

整个项目只定义了一个 Service 接口，这就是 SPI 中的 Service 接口。

The screenshot displays an IDE interface with two main panels. The left panel shows the project structure for 'simple-spi-example'. A red box highlights the 'simple-api' directory, which contains 'src/main/java/top/xiaorang/spi/InternetService'. A red arrow points from this box to the right panel. The right panel shows the code for 'InternetService.java'. A red box highlights the 'public interface InternetService' declaration. A red arrow points from this box to the text '定义了SPI中的Service接口' (Defined the Service interface in SPI).

项目结构

```
package top.xiaorang.spi;

/**
 * @author liulei
 */
public interface InternetService {
    /**
     * 连接网络
     */
    void connectInternet();
}
```

定义了SPI中的Service接口

2、simple-isp-mobile

The screenshot displays an IDE with the project structure on the left and the source code on the right. The project structure shows a hierarchy starting from `simple-isp-mobile`, which contains `src`, `main`, `java`, `cn`, and `mobile`. The `mobile` package contains `BeijingChinaMobile` and `ChinaMobile`. The `resources` directory contains `META-INF`, which in turn contains `services` and the `top.xiaorang.spi.InternetService` interface. The `target` directory contains `pom.xml`. The code on the right shows two classes, `ChinaMobile` and `BeijingChinaMobile`, both implementing the `connectInternet` method of the `InternetService` interface. Red annotations highlight the project structure, the configuration file, and the two service provider classes.

项目结构

配置文件

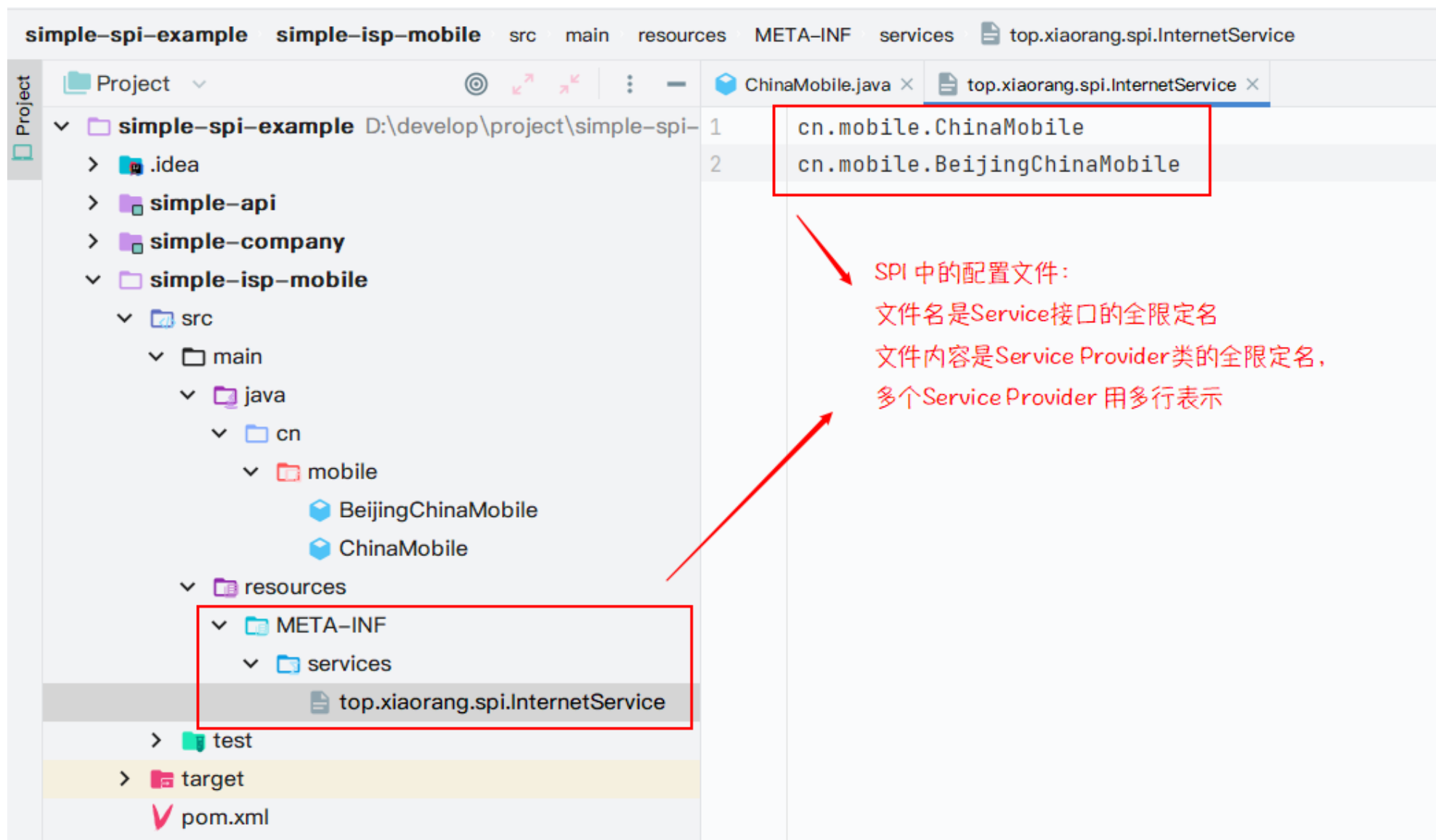
Service Provider类，实现了 InternetService 接口

```
1 package cn.mobile;
2
3 import top.xiaorang.spi.InternetService;
4
5 /**
6  * @author liulei
7  */
8 public class ChinaMobile implements InternetService {
9     @Override
10    public void connectInternet() {
11        System.out.println("connect internet by [China Mobile]");
12    }
13 }
14
```

```
1 package cn.mobile;
2
3 import top.xiaorang.spi.InternetService;
4
5 /**
6  * @author liulei
7  */
8 public class BeijingChinaMobile implements InternetService {
9     @Override
10    public void connectInternet() { System.out.println("connect internet by [Beijing China Mobile]"); }
11 }
12
```

移动提供的 simple-isp-mobile 也非常简单，从图中可以看出它提供了两个 Service Provider，一个是 ChinaMobile，另一个是 BeijingChinaMobile，这两个类都实现了 InternetService 接口中的 connectInternet 方法。

有一个非常关键的点，在创建 **META-INF/services** 目录的时候，一定得一级一级创建，不然很容易就只创建出一级目录，名字叫 **META-INF.services**，这个会导致后续使用的时候根本找不到。



然后再看一下配置文件，注意它是放在目录 META-INF/services 下的，文件名是 InternetService 接口的全限定名，文件内容则有两行，分别是 ChinaMobile 和 BeijingChinaMobile 的类名。

3、simple-isp-unicom

simple-spi-example simple-isp-unicom src main java cn unicom ChinaUnicom

Project

- simple-spi-example D:\develop\project\simple-spi-
 - .idea
 - simple-api
 - simple-company
 - simple-isp-mobile
 - simple-isp-unicom
 - src
 - main
 - java
 - cn
 - unicom
 - ChinaUnicom
 - resources
 - META-INF
 - services
 - top.xiaorang.spi.InternetService
 - test
 - target
 - pom.xml
 - External Libraries
 - Scratches and Consoles

```
1 package cn.unicom;
2
3 import top.xiaorang.spi.InternetService;
4
5 /**
6  * @author liulei
7  */
8 public class ChinaUnicom implements InternetService {
9     @Override
10    public void connectInternet() {
11        System.out.println("connect internet by [China Unicom]");
12    }
13 }
14
```

Service Provider, 实现类 InternetService 接口

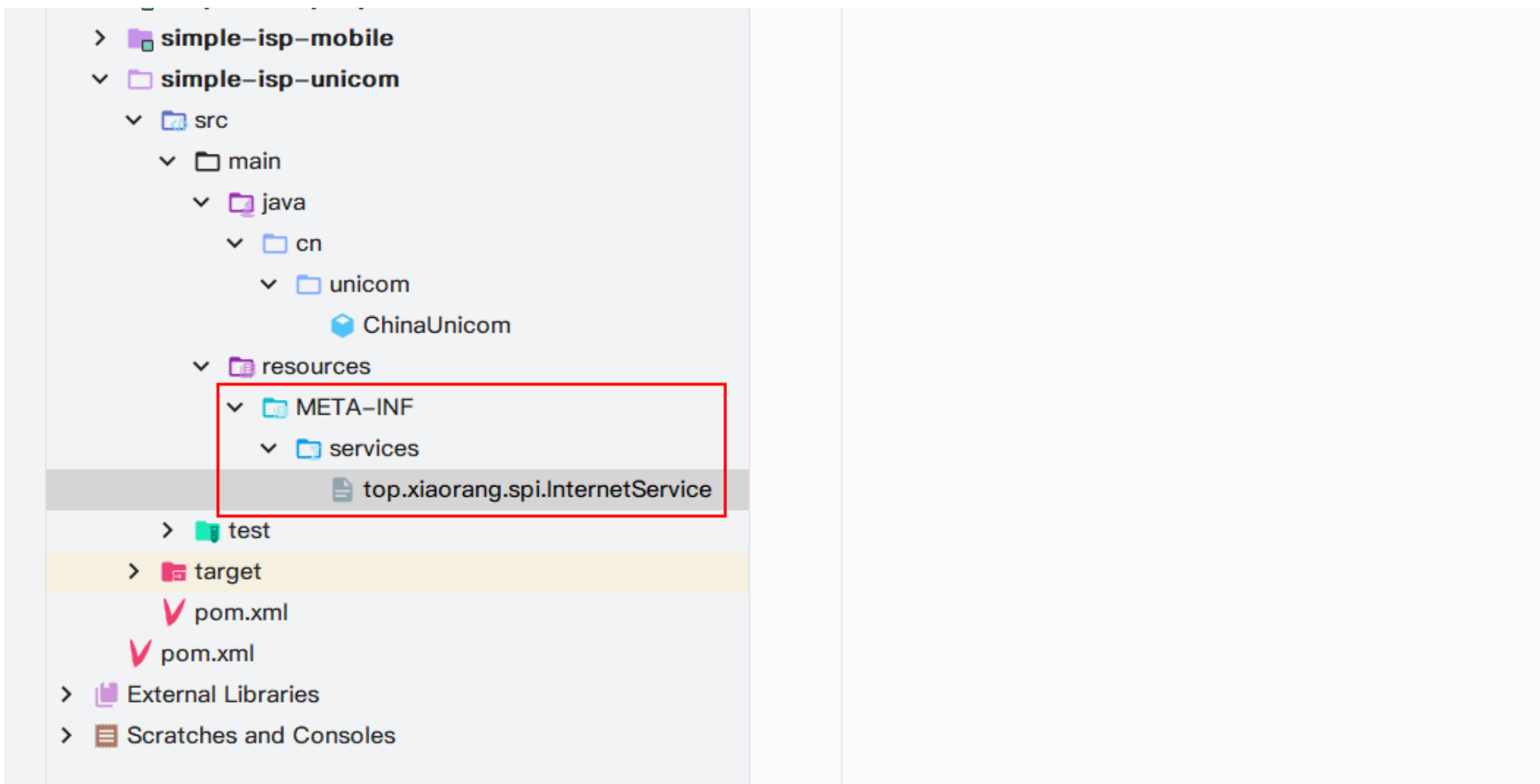
配置文件

simple-spi-example simple-isp-unicom src main resources META-INF services top.xiaorang.spi.InternetService

Project

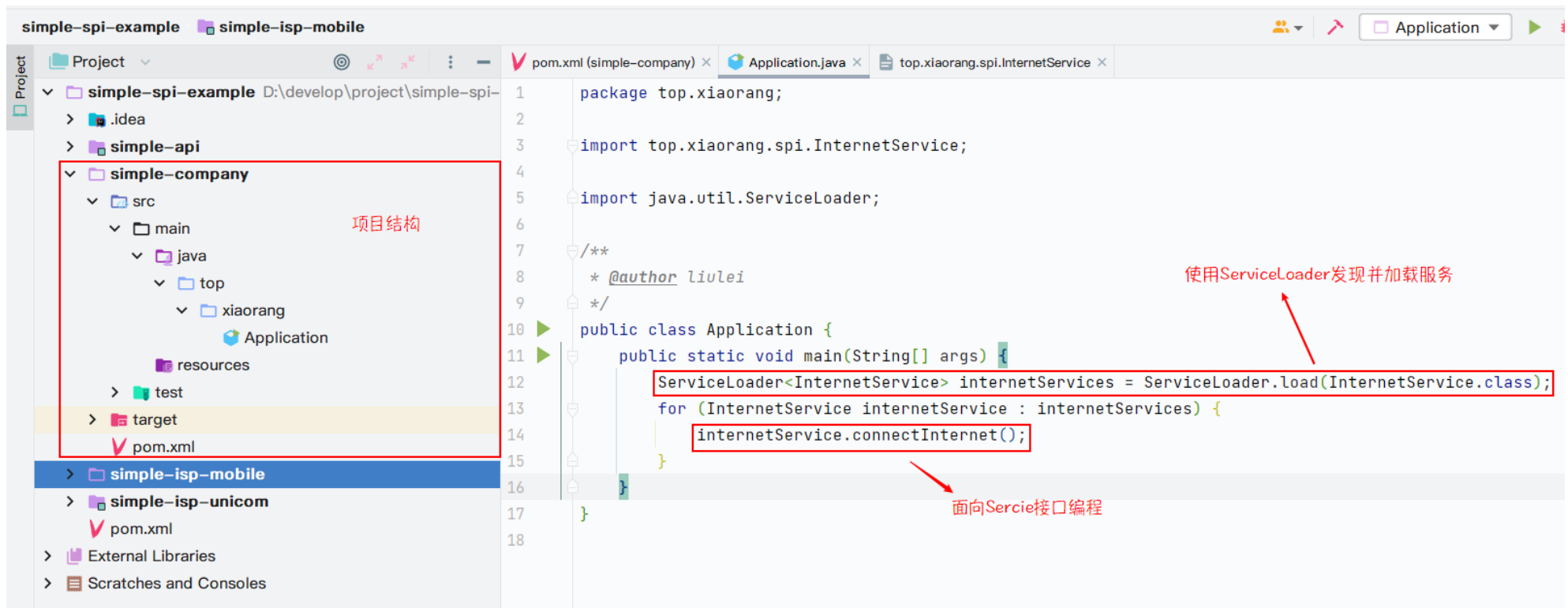
- simple-spi-example D:\develop\project\simple-spi-
 - .idea
 - simple-api
 - simple-company

```
1 cn.unicom.ChinaUnicom
```

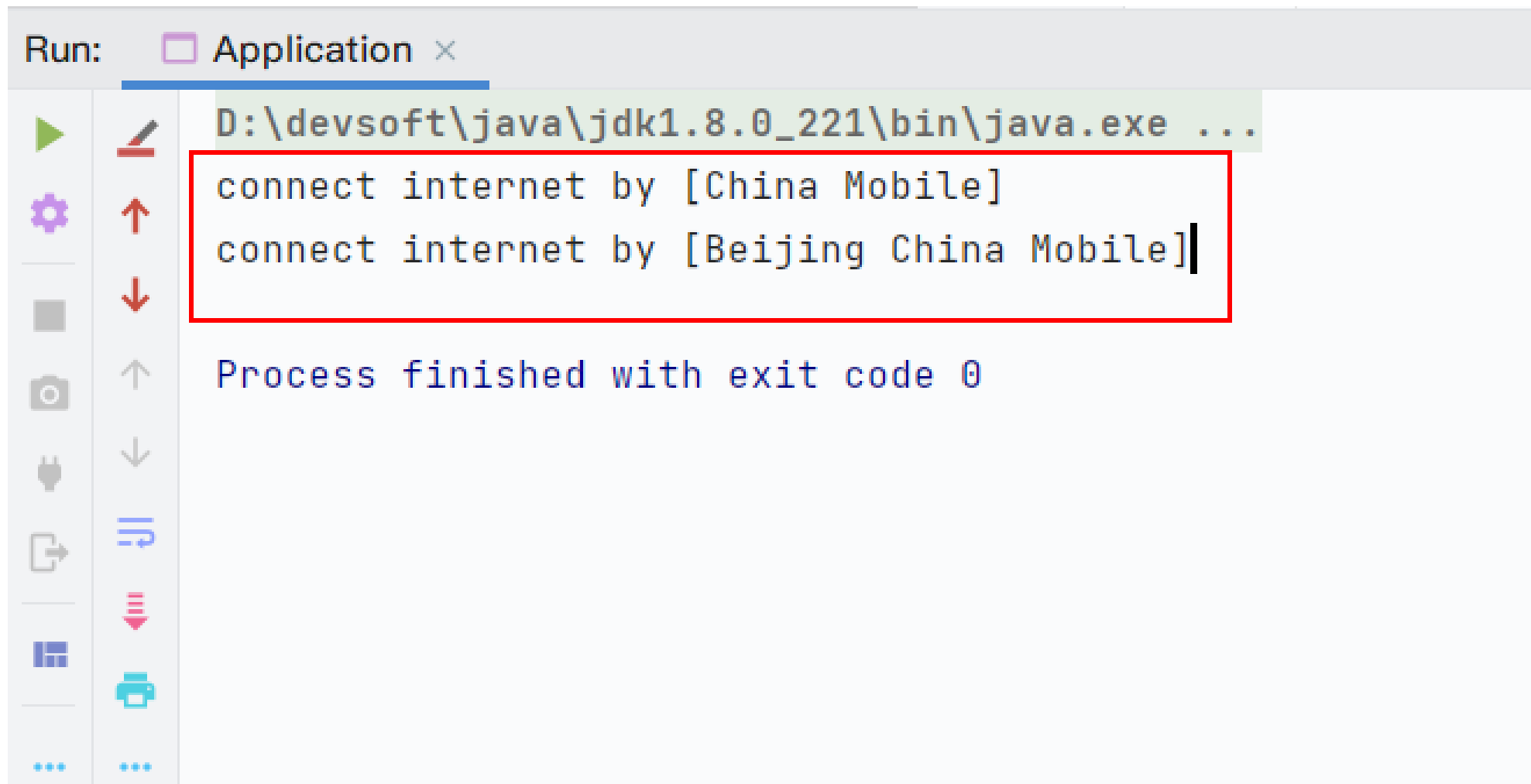
再来看看联通提供的 simple-isp-unicom，整个代码实现和移动的很相似，唯一不同的是联通只提供了一个 ServiceProvider，即类 ChinaUnicom。

4、simple-company



最后看一下 simple-company 的实现，也非常简单。在 main 方法中，先调用 ServiceLoader 的 load 方法获取所有的 Provider，然后逐一调用 connectInternet 方法就完事了。

在运行 simple-company 的时候，先只引入移动的 jar 包 simple-isp-mobile，程序运行完毕可以看到控制台打印：



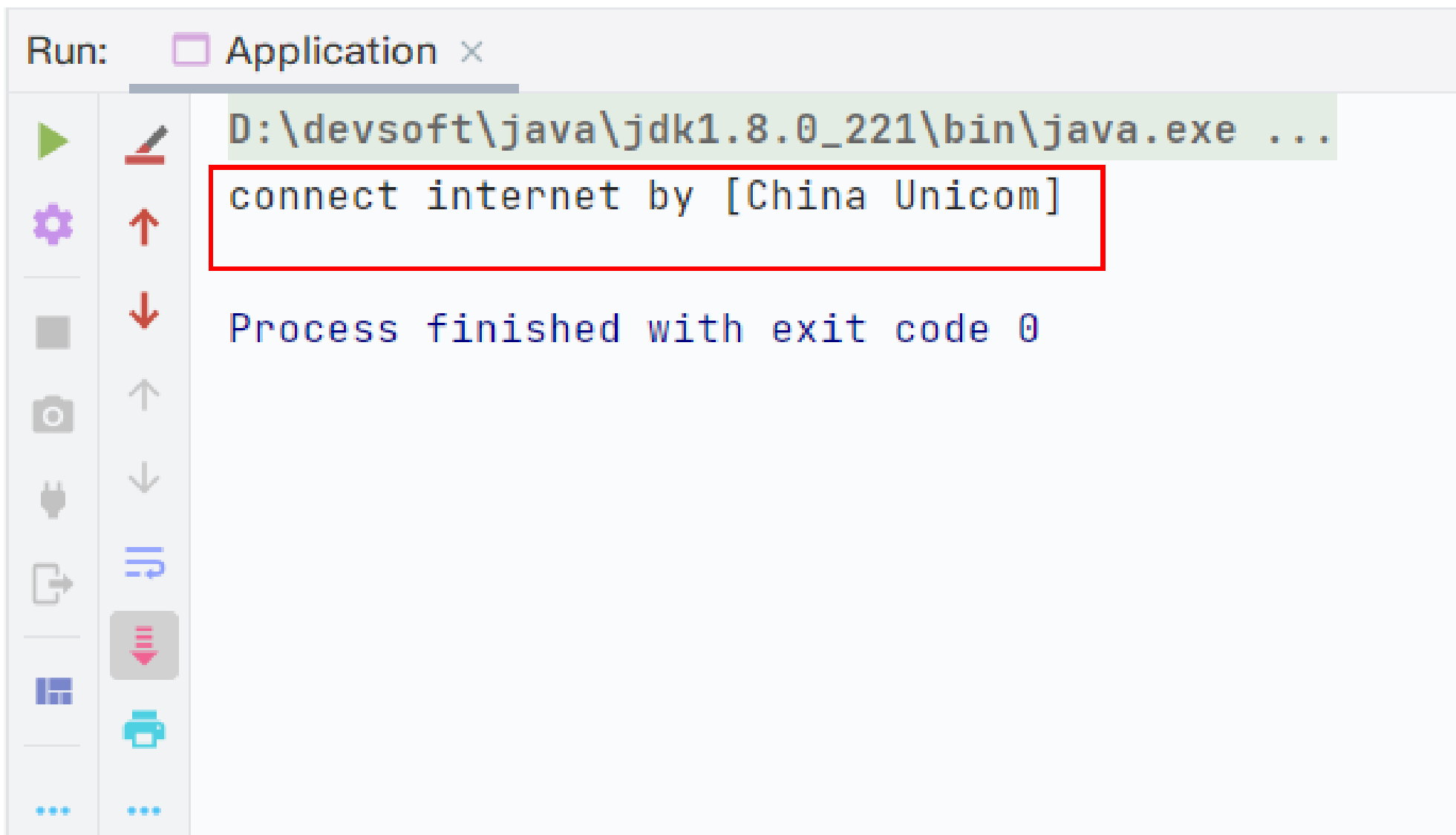
The screenshot shows an IDE's Run console window. At the top, there's a tab labeled 'Run: Application x'. Below the tab, on the left, is a vertical toolbar with icons for play, edit, settings, run, stop, restart, and other actions. The main area of the console displays the following text:

```
D:\devsoft\java\jdk1.8.0_221\bin\java.exe ...  
connect internet by [China Mobile]  
connect internet by [Beijing China Mobile]  
  
Process finished with exit code 0
```

The first two lines of output are enclosed in a red rectangular box.

分别表示，通过中国移动和北京移动联网成功！

接着我们将网络服务商替换成联通，也就是将依赖的jar包换成联通的 simple-isp-unicom，其余什么都不用改，运行程序后可以看到控制台打印：



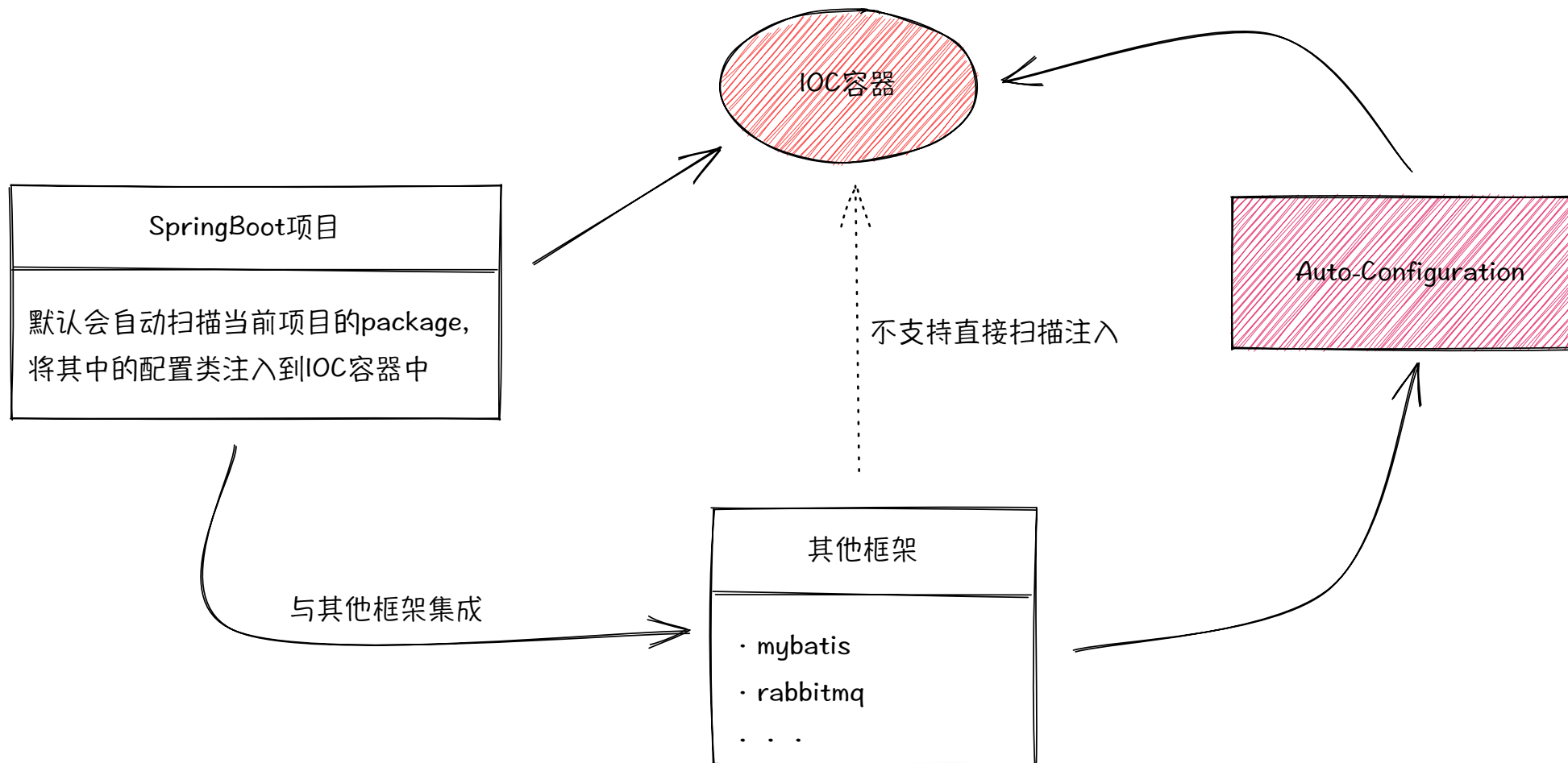
更换为中国联通联网成功了！

7、SPI 与 SpringBoot 自动配置

最后，让我们触类旁通，举一反三，以 SpringBoot 框架的自动配置为例，一起探讨一下 SPI 设计思想的发散和延伸。
SpringBoot 自动配置，即大名鼎鼎的 Auto-Configuration：

- 它是指基于你引入的依赖jar包，对SpringBoot应用进行自动配置
- 它提供了自动配置功能的依赖jar包，通常被称为 stater，如 mybatis-spring-boot-stater 等等

SpringBoot 项目启动的时候默认会自动扫描当前项目的 package，然后将其中的配置类注入到 IOC 容器中。但是，当我们与其他框架进行集成的时候，如 mybatis、rabbitmq 框架等等，SpringBoot 是不支持直接扫描其他框架的 package 的，这个时候则需要使用 **Auto-Configuration** 机制，基于你引入的依赖jar包对SpringBoot应用进行自动配置，换言之呢，就是将其他jar包的配置类注入到IOC容器中。



如果你是 SpringBoot 的开发人员，你会怎样实现 Auto-Configuration 呢？

大 A：作为 Leader，我先提几点要求：首先，不能脱离 SpringBoot 框架，我可不想重复造轮子！

细 B：同意。我们可以继续使用 @Configuration 等已有注解，然后将自动配置类注入到 Spring 的 IOC 容器中。

大 A：作为 Leader，我再提个问题，SpringBoot 框架默认是扫描当前项目的 package 的，那么如何将其他 jar 包中的配置类也注入到 IOC 容器中呢？

细 B：让用户使用注解 @ComponentScan 来扫描第三方的 package 吧！

大 A：听起来对用户很不友好啊！用户只想引入依赖的 jar 包就行。既然我们叫“自动配置”，那么能否实现全自动，而不要是半自动呢？

细 B：让我想想，这个需求听起来很耳熟。。。要不我们参考参考 Java SPI 的设计思想？

Java SPI 的设计思想

1 使用约定的配置文件

2 谁提供 jar 包，也负责提供配置文件：
· 高内聚低耦合，代码和配置一肩挑

3 使用 ClassLoader 的 getResource 和 getResources 方法来读取 classpath 中的配置文件

Java SPI 的设计思想

1 使用约定的配置文件：
1. 文件路径是 META-INF/spring.factories
2. 文件内容是 "key=value1,value2,...valueN" 的格式。
其中 key 是 "EnableAutoConfiguration" 的类名，value 是自动配置类的类名

2 提供自动配置类的 jar 包中，也同时提供配置文件
META-INF/spring.factories

3 和 SPI 一样，使用 ClassLoader 的 getResource 和 getResources 方法来读取 classpath 中的配置文件

我们来看一个 SpringBoot 自动配置案例：Mybatis，先在 github 上找到对应的 [源码](#)。

mybatis / spring-boot-starter

src / main / resources / META-INF / spring.factories

两个配置类

配置文件META-INF/spring.factories

1.注意左边配置文件路径

2.注意mybatis中定义了两个自动配置类

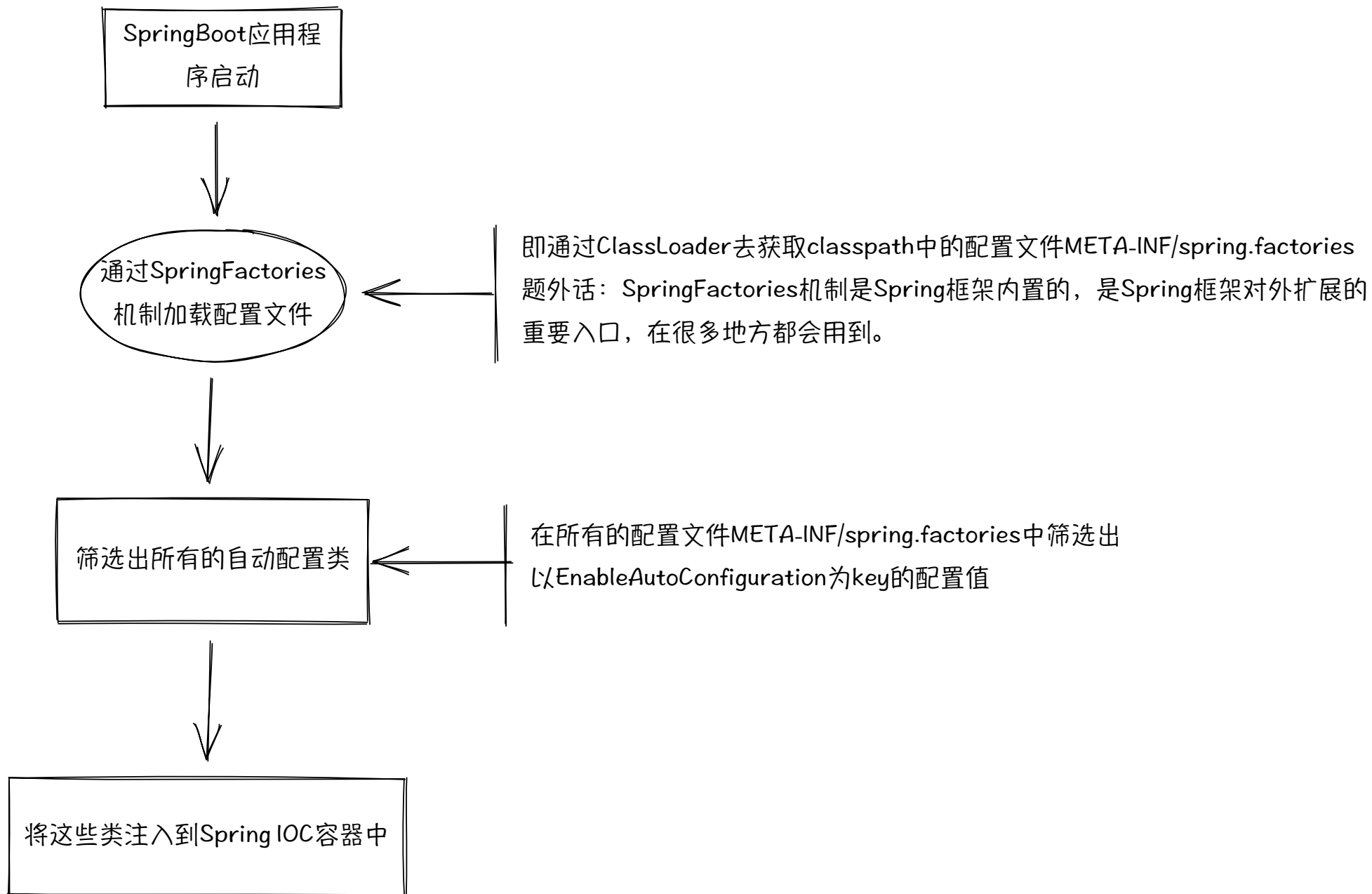
key

values

```
1 # Auto Configure
2 org.springframework.boot.autoconfigure.EnableAutoConfiguration
3 org.mybatis.spring.boot.autoconfigure.MybatisLanguageDriverAutoConfiguration,
4 org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration
```

可以看到，mybatis 总共有两个自动配置类，分别是 `MybatisAutoConfiguration` 和 `MybatisLanguageDriverAutoConfiguration`，然后在目录 `META-INF` 中也确实存在一个配置文件 `spring.factories`，里面的内容是 key-value 的格式，其中 key 是 `EnableAutoConfiguration` 的类名，value 则是两个自动配置类的类名，两个类名直接用逗号隔开。

简单总结一下 `SpringBoot` 自动配置的核心流程：



以上，就是 SpringBoot 自动配置的原理，它是不是和 SPI 的设计思想有着异曲同工之妙呢？

