

1.

1)maven 是一个项目管理工具软件，它基于项目对象模型（POM），可以通过一小段描述信息来管理项目的构建、报告和文档，是覆盖项目全生命周期（从初始化、编译、测试到打包、部署）的综合工具

POM（Project Object Model，项目对象模型）

`pom.xml` 是 Maven 项目的**唯一配置文件**，位于项目根目录，包含项目的所有信息：

- 项目基本信息（groupId、artifactId、version，即“GAV 坐标”）；
- 项目依赖（`<dependencies>` 标签）；
- 构建配置（如编译插件、输出目录，`<build>` 标签）；
- 父项目继承（`<parent>` 标签，用于复用配置）。

```
com.alibaba  
fastjson  
2.0.51
```

GAV坐标

Maven 通过 **GAV 三要素** 唯一标识一个项目（或依赖），如同“地址 + 门牌号”定位一个人：

- **groupId**：组织 / 公司标识，通常是域名反写（如 `com.apache`、`com.alibaba`），确保不同组织的项目不冲突；
- **artifactId**：项目 / 模块名称（如 `spring-core`、`mybatis`），在同一 groupId 下唯一；
- **version**：版本号，格式通常为 `主版本.次版本.修订号`（如 `5.3.20.RELEASE`），常见后缀含义：
 - `SNAPSHOT`：快照版（开发中，每次构建可能更新）；
 - `RELEASE`：稳定版（正式发布，版本固定）；
 - `RC`（Release Candidate）：候选发布版（接近正式版，可能修复少量 bug）。

仓库（Repository）：依赖的存储位置

本地仓库（Local）

远程仓库（Remote）如公司内部仓库、阿里云仓库

中央仓库（Central）Maven 官方仓库

生命周期Lifecycle

里面包含 `clean`、`compile`、`package`、`install` 等命令，日常使用中主要关注 **Clean 生命周期** 和 **Default 生命周期**，每个生命周期包含多个“阶段（Phase）”，阶段按顺序执行比如执行 `install` 会自动先执行 `compile` → `test` → `package`

依赖范围scope

`<scope>` 标签控制依赖在哪些生命周期阶段生效，避免不必要的依赖被打包到最终产物（如测试依赖无需打包到生产 JAR）。

范围（Scope）	生效阶段	依赖传递	说明
<code>compile</code> （默认）	编译、测试、运行	是	生产环境必需的依赖（如 Spring 核心包），会被打包到最终产物
<code>test</code>	仅测试阶段	否	测试用依赖（如 JUnit），不会打包到生产产物
<code>provided</code>	编译、测试	否	运行时由容器提供（如 Web 项目的 Servlet API，Tomcat 已包含），不会打包
<code>runtime</code>	测试、运行	是	编译时无需，运行时必需（如 JDBC 驱动），会被打包
<code>system</code>	编译、测试	否	依赖本地磁盘上的 Jar 包（需通过 <code><systemPath></code> 指定路径，不推荐使用）

- 2)jar包即 Java Archive 文件，是基于 zip 格式的压缩归档文件，用于存储 Java 类、资源及元数据。
- 3) 在 Maven 项目中，通过在 `pom.xml` 文件中声明依赖关系，Maven 会根据这些声明自动从中央仓库或其他配置的仓库下载对应的 JAR 包到本地仓库。

Maven 使用生命周期和插件的概念来定义构建过程，当执行到 `package` 阶段时，`maven-jar-plugin` 插件会将项目的编译产物（例如 `class` 文件）和其他资源打包成 JAR 文件。
- 4) Maven 可以自动管理项目的依赖关系，简化依赖管理；提供了一套标准化的构建生命周期，可以使用相同的方式进行构建项目；可以方便地扩展其功能，以方便地扩展其功能；Maven 项目的配置信息都存储在 `pom.xml` 文件中，便于团队成员之间的协作和沟通。

2.

Java

```
C:\Users\ZZF>java -version
java version "24.0.2" 2025-07-15
Java(TM) SE Runtime Environment (build 24.0.2+12-54)
Java HotSpot(TM) 64-Bit Server VM (build 24.0.2+12-54, mixed mode, sharing)

C:\Users\ZZF>|
```

Maven

```
Microsoft Windows [版本 10.0.26100.6584]
(c) Microsoft Corporation. 保留所有权利。

:\Users\ZZF>mvn -v
Apache Maven 3.9.11 (3e54c93a704957b63ee3494413a2b544fd3d825b)
Maven home: D:\M\apache-maven-3.9.11
Java version: 24.0.2, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-24
default locale: zh_CN, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"

:\Users\ZZF>
```

版本Apache Maven (版本 3.9.11)

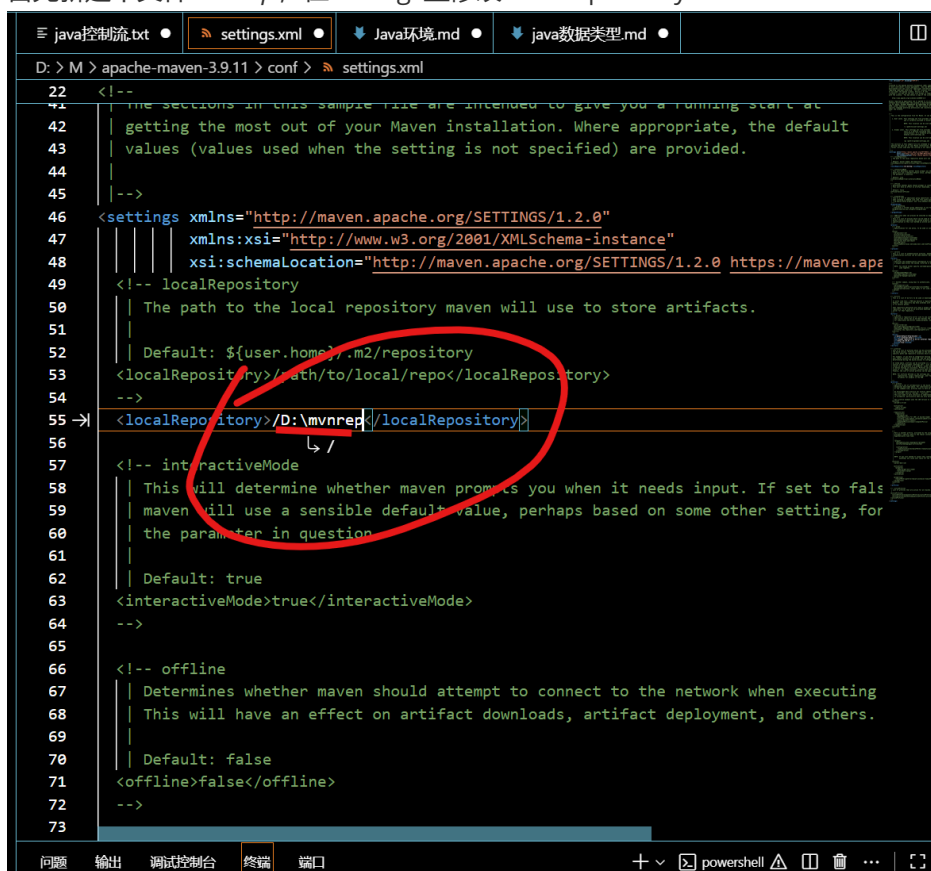
3.

1) 配置本地仓库

Maven 默认的本地仓库位置是用户目录下的 `.m2/repository`，首次使用需要手动创建如 `C:\Users\你的用户名\.m2\repository` 的文件。

若想修改本地仓库位置，一般是修改maven的conf文件夹的 `settings.xml` (Maven 的配置文件)

首先新建个文件`mvnrep`，在settings里修改`localRepository`



在这也可以配置一个阿里云镜像，下载依赖时更快

在settings中找到`errors`修改成

```
<mirrors>
  <mirror>
    <id>nexus</id>
    <name>Nexus Public Mirror</name>
    <url>http://nexus.example.com/content/groups/public</url>
    <mirrorOf>central</mirrorOf>
  </mirror>
</mirrors>
```

```
<mirrors>
  <mirror>
    <id>aliyunmaven</id>
    <name>阿里云公共仓库</name>
    <url>http://maven.aliyun.com/repository/public</url>
    <mirrorOf>central</mirrorOf>
  </mirror>
</mirrors>
```

2) 远程仓库

是“存 Jar 包的远程服务器”，主要分 **中央仓库** 和 **私服** 两类，作用完全不同：

3) 中央仓库：Maven 官方的“公共 Jar 包超市”

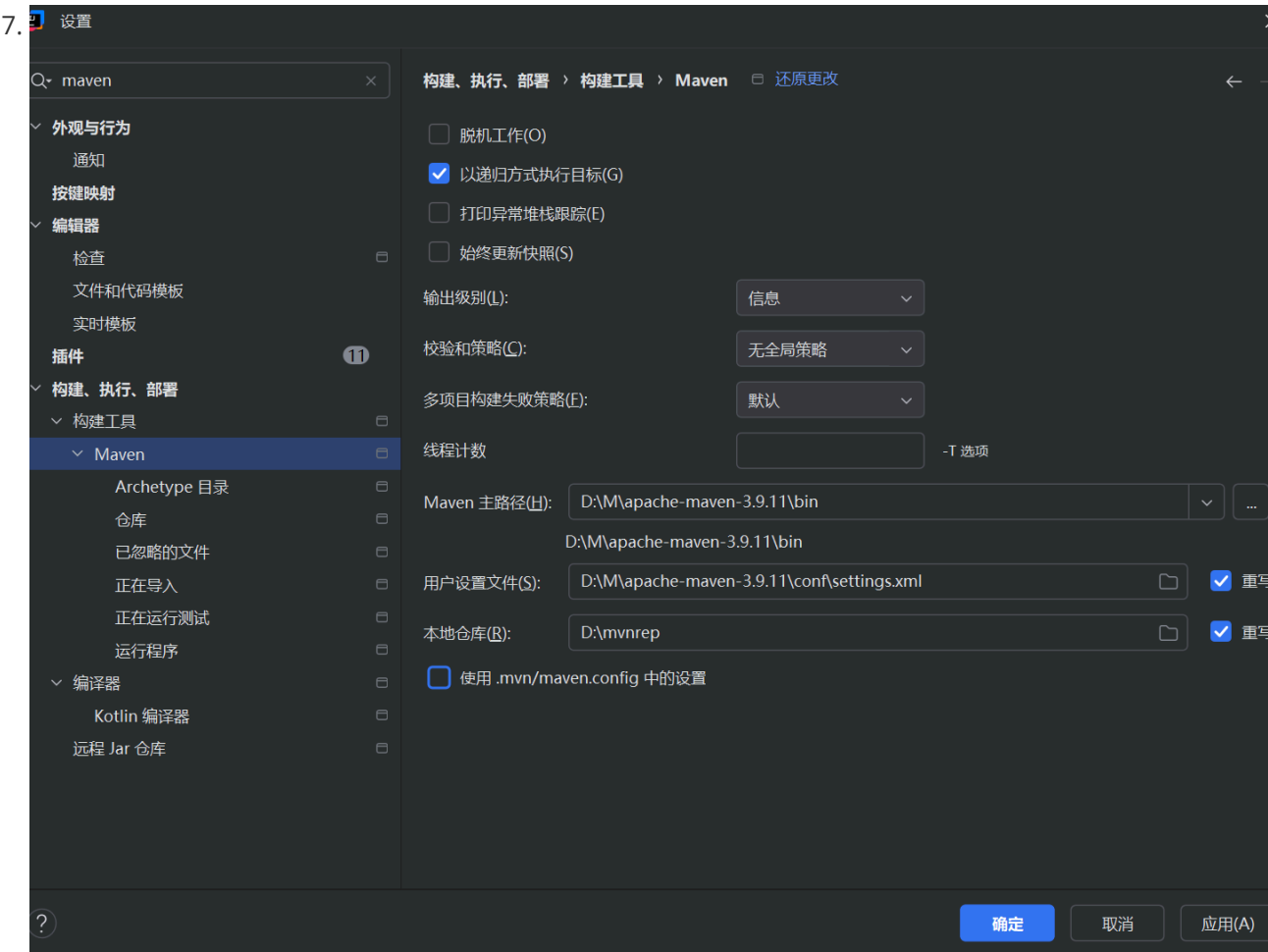
- **是什么**：由 Maven 官方维护的、全球公开的远程仓库（相当于“公共超市”），里面存放了几乎所有开源项目的 Jar 包（比如 Spring、MyBatis、FastJSON 等）。
- **特点**：
 - 公开免费，所有人都能访问；
 - 是 Maven 的“默认远程仓库”——如果你的项目没配置其他远程仓库，Maven 会自动去中央仓库下载本地缺少的 Jar 包；
 - 地址默认是 `https://repo.maven.apache.org/maven2`（但国内访问可能较慢，所以常换阿里云镜像）。

4) 私服：企业 / 团队的“内部 Jar 包仓库”

- **是什么**：由企业或团队自己搭建的、仅内部可访问的远程仓库（相当于“公司内部仓库”），不是公开的。
- **核心作用**（为什么企业要用）：
 - ① 缓存中央仓库的 Jar 包：企业里所有人都从私服下载，不用每个人都去中央仓库（国内访问快，还省外网流量）；
 - ② 存放企业“私有 Jar 包”：比如公司自己开发的、不对外公开的工具类 Jar 包（只能内部项目用，不能放中央仓库）；
 - ③ 统一管理依赖：企业可以控制私服里的 Jar 包版本，避免团队成员用错版本导致项目出错。

1) IDEA关联maven

- 1. **打开项目设置**：打开 IDEA，点击菜单栏中的 **File**（文件），选择 **Settings**（设置）（Windows 和 Linux 系统）
- 2. **找到 Maven 设置项**：在弹出的设置窗口中，找到 **Build, Execution, Deployment**（构建，执行，部署） -> **Build Tools**（构建工具） -> **Maven**。
- 3. **配置 Maven 主目录**：在 **Maven home directory**（Maven 主路径）处，点击右侧的 **...** 按钮，选择本地安装的 Maven 目录（即包含 **bin**、**conf** 等文件夹的目录）。
- 4. **指定 settings.xml 文件**：在 **User settings file**（用户设置文件）处，指定之前修改过本地仓库路径的 **settings.xml** 文件位置
- 5. **设置本地仓库路径**：**Local repository**（本地仓库）路径一般会根据 **settings.xml** 文件中的配置自动填充，如果没有填充正确，可以手动输入你设置的本地仓库路径
- 6. **应用设置**：点击 **OK** 保存设置。之后在创建或导入 Maven 项目时，IDEA 就会按照你配置的 Maven 相关信息进行操作。



2) 一个maven项目都需要配置如下参数：

项目基本信息

在项目的 **pom.xml** 文件中：

- **groupId**：项目组标识，一般是公司域名的反向，例如 **com.example**，用于唯一标识项目所属的组织。
- **artifactId**：项目的唯一名称，比如 **my-maven-project**，和 **groupId** 组合起来可以唯一确定一个项目。
- **version**：项目版本号，如 **1.0.0**，用于区分项目的不同版本。

- **packaging**: 打包方式, 常见的有 `jar` (用于 Java 类库或可执行的 Java 应用程序)、`war` (用于 Web 应用程序)、`pom` (用于多模块项目的父项目) 。

依赖管理

通过 `<dependencies>` 标签配置项目所需的依赖:

- **groupId**: 依赖的组标识。
- **artifactId**: 依赖的名称。
- **version**: 依赖的版本号。
- **scope**: 依赖的作用范围, 常见的有 `compile` (编译时有效, 默认值)、`test` (仅在测试时有效)、`runtime` (运行时有效) 等。

构建配置

在 `<build>` 标签中配置项目的构建相关信息:

- **plugins**: 配置构建过程中使用的插件, 如 `maven-compiler-plugin` 用于编译 Java 代码, `maven-jar-plugin` 用于打包 jar 文件等。可以设置插件的版本、配置参数等。
- **resources**: 指定项目中资源文件的位置, 比如配置文件 (`.properties`、`.xml` 等), 默认情况下, Maven 会处理 `src/main/resources` 目录下的资源文件, 但如果资源文件位置有变动, 就需要在这里配置。

3)项目结构

```
my-java-project/ # 项目根目录 (打开这里找 pom.xml)
├─ pom.xml       # 要找的核心配置文件 (直接在根目录)
├─ src/          # 源码目录 (无需进入)
│   ├─ main/
│   └─ test/
└─ target/       # 构建输出目录 (自动生成, 可选)
```

- **代码**: 一般存放在 `src/main/java` 目录下, 按照项目的包结构进行组织, 用于存放项目的主要 Java 源代码。
- **配置文件**: 项目的资源配置文件通常存放在 `src/main/resources` 目录下。此外, IDEA 还有一些项目级别的配置文件存放在项目根目录下的 `.idea` 文件夹中, 如 `encodings.xml` (编码设置)、`runConfigurations` (运行配置信息) 等。
- **测试代码**: 默认位于 `src/test/java` 目录, 同样会按照与主代码相同的包结构来组织, 用于编写对主代码进行测试的代码。
- **测试配置文件**: 存放于 `src/test/resources` 目录, 用于存储测试相关的配置信息, 如测试数据库配置等。
- **Maven 项目配置文件**: Maven 项目的核心配置文件为 `pom.xml`, 位于项目的根目录下。全局的 Maven 配置文件 `settings.xml` 通常位于 Maven 安装目录下的 `conf` 文件夹中, 也可以在用户目录下的 `.m2` 目录中找到用户级别的 `settings.xml` 配置文件。可通过依次点击 `File -> Settings -> Build, Execution, Deployment -> Build Tools -> Maven` 查看其配置路径。

4)pom.xml

1. pom.xml 是 Maven 项目的核心配置文件

2. 主要作用

1. 定义项目基本信息

- 提供 GAV 坐标 (`groupId`、`artifactId`、`version`) 唯一标识项目
- 包含项目名称、描述、开发者信息等元数据

2. 管理项目依赖

- 声明项目需要的第三方库 (如 Spring、JUnit)
- 自动处理依赖传递 (A 依赖 B, Maven 会自动下载 B)
- 解决版本冲突 (通过依赖仲裁机制)

3. 配置构建过程

- 指定打包类型 (`jar`、`war`、`pom` 等)
- 配置编译参数 (JDK 版本、编码等)
- 管理插件 (编译、测试、打包等插件)

4. 支持多模块项目

- 作为父 POM 统一管理子模块的依赖和配置
- 定义项目模块结构, 实现批量构建

5. 提供项目信息

- 配置项目文档生成
- 定义 SCM (源代码管理) 信息
- 指定 issue 跟踪系统

3. 核心基本信息:

- **GAV 坐标:** `groupId`、`artifactId`、`version` (项目唯一标识)
- **打包类型:** `packaging` (默认 `jar`, web 项目用 `war`, 父项目用 `pom`)
- **项目元数据:** 名称、描述、URL 等

4.

--直接在 `<dependencies>` 中声明版本

--使用 `<properties>` 集中管理版本号

--使用 `<dependencyManagement>` 统一管理版本 (推荐多模块项目)

5)

- 通过项目设置导入：
 - 打开项目，点击菜单栏的“File”，选择“Project Structure”（快捷键为“Ctrl + Shift + Alt + S”）。
 - 在弹出的窗口中，点击左侧的“Modules”，然后在右侧的“Dependencies”选项卡中，点击下方的“+”按钮，选择“JARs or directories”。
 - 在文件选择对话框中，找到要导入的 JAR 包，选中后点击“OK”。
 - 最后点击“Apply”和“OK”完成导入。
- 通过快速导入：
 - 打开项目，在项目视图找到“External Libraries”。
 - 右键单击“External Libraries”，选择“Add as Library”。
 - 在弹出的文件对话框中，浏览并选择要导入的 JAR 文件，然后单击“OK”即可。
- 使用依赖管理工具导入：如果项目是基于 Maven 或 Gradle 构建的，可通过编辑项目的构建文件来添加 JAR 包依赖。对于 Maven 项目，编辑 pom.xml 文件，添加相应的 <dependency> 标签

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>example - library</artifactId>
  <version>1.0.0</version>
</dependency>
```

对于 Gradle 项目，编辑 build.gradle 文件，添加类似 implementation 'com.example:example - library:1.0.0' 的依赖语句。添加后，构建工具会自动下载并将 JAR 包添加到项目中。

5.










1)

命令	核心作用	常用场景
mvn clean	清理项目编译生成的文件（如 target 目录下的 class 文件、Jar 包等）	编译报错时（可能是旧文件冲突）、重新编译前，先清理旧产物。
mvn compile	编译项目的主代码（src/main/java 下的 Java 文件），生成 class 文件	需要单独编译主代码，或验证主代码语法是否正确时。
mvn test	编译测试代码（src/test/java）并执行所有测试用例（如 JUnit 测试）	开发完功能后，执行测试用例验证功能是否正常。
mvn package	打包项目（根据 pom.xml 配置，生成 Jar 包或 War 包），存放在 target 目录	需要生成项目的可执行包（如 Jar 包给其他项目引用，或 War 包部署到服务器）。
mvn install	将项目打包后的产物（Jar/War 包）安装到本地仓库（默认 ~/.m2/repository）	本地多个项目复用当前项目（如把自己写的工具类项目安装到本地仓库，其他项目依赖）。









命令	核心作用	常用场景
<code>mvn clean install</code>	组合命令：先清理（clean），再编译、测试、打包（package），最后安装到本地仓库	本地开发时“一键构建 + 安装”，确保最新代码被其他项目引用。
<code>mvn dependency:tree</code>	查看项目的 依赖树 （所有直接 / 间接依赖的层级关系）	排查依赖冲突（如两个依赖引用了同一个 Jar 包的不同版本）。

demo-project




▼ 生存期

-  clean
-  validate
-  compile
-  test
-  package
-  verify
-  install
-  site
-  deploy


▼ 插件

- >  clean (org.apache.maven.plugins:maven-clean-plugin:3.2.0)
- >  compiler (org.apache.maven.plugins:maven-compiler-plugin:3.10.1)
- >  deploy (org.apache.maven.plugins:maven-deploy-plugin:3.1.1)
- >  install (org.apache.maven.plugins:maven-install-plugin:3.1.1)
- >  jar (org.apache.maven.plugins:maven-jar-plugin:3.4.1)
- >  resources (org.apache.maven.plugins:maven-resources-plugin:3.3.1)
- >  site (org.apache.maven.plugins:maven-site-plugin:3.12.1)
- >  surefire (org.apache.maven.plugins:maven-surefire-plugin:3.0.0-M5)

▼ 依赖项

- ▼  com.alibaba:fastjson:2.0.51
 - ▼  com.alibaba.fastjson2:fastjson2-extension:2.0.51
 -  com.alibaba.fastjson2:fastjson2:2.0.51

▼ 仓库

-  local (D:\mvnrep)
-  central (<https://repo.maven.apache.org/maven2>)

2)

m mvn

最近

m mvn install

Maven 目标

m mvn clean

m mvn compile

m mvn deploy

m mvn package

m mvn site

m mvn test

m mvn validate

m mvn verify

m mvn clean:clean

m mvn clean:help

m mvn compiler:compile

m mvn compiler:help

m mvn compiler:testCompile

m mvn deploy:deploy

m mvn deploy:deploy-file

m mvn deploy:help

m mvn install:help

m mvn install:install

m mvn install:install-file

m mvn jar:help

m mvn jar:jar

m mvn jar:test-jar

m mvn resources:copy-resources

m mvn resources:help

m mvn resources:resources

m mvn resources:testResources

m mvn site:attach-descriptor

m mvn site:deploy

按 向上箭头 或 向下箭头 浏览建议列表

6.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>demo-project</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
    <maven.compiler.source>24</maven.compiler.source>
    <maven.compiler.target>24</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>fastjson</artifactId>
        <version>2.0.51</version>
    </dependency>
</dependencies>
</project>
```

```
import com.alibaba.fastjson.JSON;

public class User {
    private String name;
    private int age;

    // 无参构造函数
    public User() {
    }

    // 有参构造函数
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // name 的 getter 和 setter
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    // age 的 getter 和 setter
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```

    }

    public static void main(String[] args) {
        // 创建 User 对象
        User user = new User("Alice", 25);
        // 将 User 对象转换为 JSON 字符串
        String jsonStr = JSON.toJSONString(user);
        System.out.println("对象转JSON: " + jsonStr);

        // 定义 JSON 字符串
        String json = "{\"name\":\"Alice\",\"age\":25}";
        // 将 JSON 字符串转换为 User 对象
        User userFromJson = JSON.parseObject(json, User.class);
        System.out.println("JSON转对象, name: " + userFromJson.getName() + ", age: " +
userFromJson.getAge());
    }
}

```

客户端

```

package org.example;
import java.io.IOException;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;

public class Client {
    private static final String SERVER_URL = "http://localhost:8000/query";

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        while (true) {
            // 1. 读取用户输入
            System.out.print("请输入快递单号: ");
            String trackingNumber = scanner.nextLine();
            System.out.print("请输入手机号: ");
            String phone = scanner.nextLine();

            // 如果输入 exit 则退出循环
            if ("exit".equalsIgnoreCase(trackingNumber)) {
                break;
            }

            try {
                // 2. 用 FastJSON 构造 JSON
                JSONObject json = new JSONObject();
                json.put("trackingNumber", trackingNumber);
                json.put("phone", phone);
            }

```

```

        String jsonBody = json.toString();

        // 3. 创建 HTTP POST 请求
        URL url = new URL(SERVER_URL);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("POST");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Type", "application/json; charset=UTF-8");

        // 4. 发送 JSON 数据
        try (OutputStream os = connection.getOutputStream()) {
            byte[] input = jsonBody.getBytes(StandardCharsets.UTF_8);
            os.write(input, 0, input.length);
        }

        // 5. 读取响应
        if (connection.getResponseCode() == HttpURLConnection.HTTP_OK) {
            // 读取响应 JSON
            try (Scanner responseScanner = new Scanner(connection.getInputStream(),
                StandardCharsets.UTF_8.name())) {
                String responseJson = responseScanner.useDelimiter("\\A").next();

                // 6. 用 FastJSON 解析响应
                JSONObject respObj = JSON.parseObject(responseJson);
                Integer pickCode = respObj.getInteger("pick_code");
                String msg = respObj.getString("msg");

                if (pickCode != null) {
                    System.out.println("取件码: " + pickCode);
                } else {
                    System.out.println("提示: " + msg);
                }
            }
        } else {
            System.out.println("查询失败, 状态码: " + connection.getResponseCode());
        }
    } catch (IOException e) {
        System.out.println("请求发生错误: " + e.getMessage());
    }
}

scanner.close();
System.out.println("客户端已退出");
}
}

```

服务端

```

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;
import com.alibaba.fastjson.JSON;

```

```

import com.alibaba.fastjson.JSONObject;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class Server {
    private static final int PORT = 8000;
    private static final Map<String, String> expressMap = new HashMap<>();

    public static void main(String[] args) throws IOException {
        // 初始化一些测试数据
        initializeExpressData();
        // 创建HTTP服务器，监听指定端口
        HttpServer server = HttpServer.create(new InetSocketAddress(PORT), 0);
        // 设置路由和处理程序
        server.createContext("/query", new QueryHandler());
        // 启动服务器
        server.start();
        System.out.println("Server started on port " + PORT);
    }

    private static void initializeExpressData() {
        // 添加一些测试数据
        // 键的构成是 快递单号_手机号
        expressMap.put("SF123456789_13005433678", "1234");
        expressMap.put("JD987654321_19805433168", "5678");
        expressMap.put("YT456789123_13905479698", "9012");
        expressMap.put("ZT789123456_18505433664", "3456");
    }

    static class QueryHandler implements HttpHandler {
        @Override
        public void handle(HttpExchange exchange) throws IOException {
            if ("POST".equals(exchange.getRequestMethod())) {
                // 读取请求体
                InputStream requestBody = exchange.getRequestBody(); //? 为什么无法用
                // 虽然它们俩功能一样的
                Scanner scanner = new Scanner(requestBody, StandardCharsets.UTF_8);
                String requestBodyStr = scanner.useDelimiter("\\A").next();
                scanner.close();

                try {
                    // 解析JSON请求获得单号和手机号
                    JSONObject requestJson = JSON.parseObject(requestBodyStr);
                    String trackingNumber = requestJson.getString("trackingNumber");
                    String phone = requestJson.getString("phone");
                } catch (JSONException e) {
                    // 处理JSON解析异常
                }
            }
        }
    }
}

```

```
// 在expressMap中查询取件码
String key = trackingNumber + "_" + phone;
String pickCode = expressMap.get(key);

// 构建响应的json
JSONObject responseJson = new JSONObject();
if (pickCode != null) {
    responseJson.put("pick_code", pickCode);
    responseJson.put("msg", "success");
} else {
    responseJson.put("pick_code", null);
    responseJson.put("msg", "快递单号或手机号不正确，未找到取件码");
}
String responseBody = responseJson.toJSONString();

// 发送响应
exchange.getResponseHeaders().set("Content-Type", "application/json;
charset=UTF-8");

exchange.sendResponseHeaders(200,
responseBody.getBytes(StandardCharsets.UTF_8).length);
OutputStream responseBodyStream = exchange.getResponseBody();
responseBodyStream.write(responseBody.getBytes(StandardCharsets.UTF_8));
responseBodyStream.close();
} catch (Exception e) {
    // 处理异常，返回400状态码(Bad Request)
    exchange.sendResponseHeaders(400, -1);
}
} else {
    // 非POST请求返回405 Method Not Allowed
    exchange.sendResponseHeaders(405, -1);
}
}
}
}
```