第 3 章

项目与模块

我们知道，Eclipse 采用工作空间(Workspace)与项目(Project)的方式进行管理，在同一工作空间下可以有多个项目。Eclipse 中的项目不可再划分，如果需要使用其他外部的功能，则通常以引入外部项目归档文件(Java Archive)的方式进行，也就是引入外部 Jar 文件。

在这种情况下项目之间缺乏关联与结构化管理，同时项目本身可能由于应用功能的不断增加而变得庞大。对于单体结构的项目来讲，这将是一场噩梦。

IntelliJ IDEA 中不再有工作空间的概念，同时在项目(Project)下使用了模块(Module)来对其进行划分，因此可以将一个项目划分为多个模块，以不同的模块来管理不同的功能。

读者可能会认为，一个工作空间下有多个项目与一个项目下有多个模块，既然都是一对多的管理方式，那么它们本身应该没有太大的差别。事实并不是这样，因为在 Eclipse 中项目是运行时的单位，而 IntelliJ IDEA 中模块才是运行时的基本单位，它对项目进行了更加细化的运行时管理，而且模块间的依赖关系也变得更加灵活和高效。

在最简单的情况下，一个模块就是一个项目。模块作为一种组成项目的构件，它既可以独立运行，也可以与其他模块组合在一起使用，只需维护好这些模块之间的依赖关系。

3.1 项目结构

项目结构(Project Structure)用于对项目进行设置，如 Modules、Libraries、Facets、Artifacts 和 SDK。了解项目结构可以帮助开发者更好地管理、分析与调试项目。

项目结构窗口只有在项目处于打开状态时才可以进行访问并操作，如图 3.1 所示。

要打开 Project Structure 窗口，可以采用如下方式：

(1) 选择菜单 File→Project Structure 命令或使用快捷键 Ctrl+Shift+Alt+S。

(2) 单击工具栏上的 按钮。

在 Project Structure 窗口中包含了与项目配置(Project Settings)和平台配置(Platform Settings)相关的众多选项，接下来逐一进行说明。

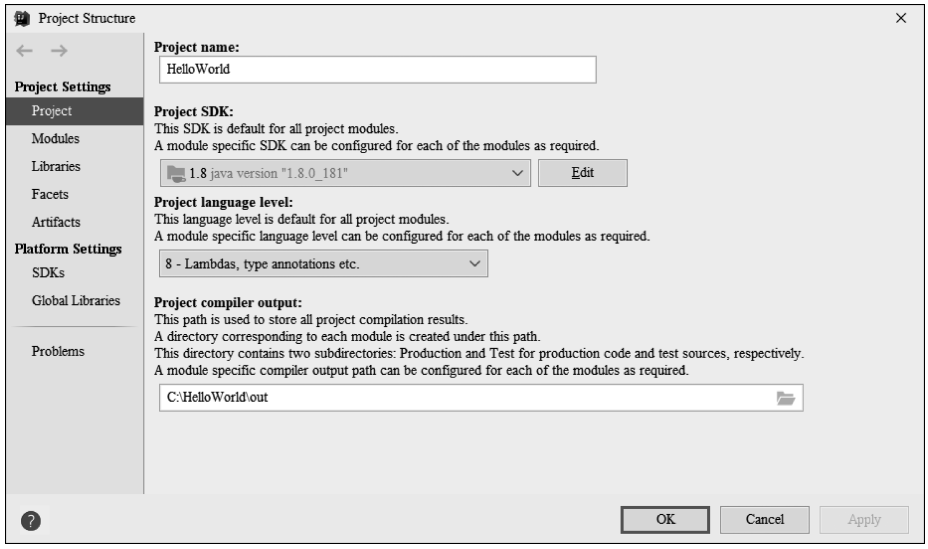

图 3.1　Project Structure

3.1.1　工程

在工程(Project)选项卡中主要包含以下内容：

(1) Project name 指定项目的名称。

(2) Project SDK 指定项目运行需要的 SDK。

(3) Project language level 指定 SDK 的语言级别。

Java JDK 的每个新版本都会有新特性发布，而新版本一般也会向下兼容旧版本的特性。IntelliJ IDEA 中列出了对 JDK 新特性的支持，如图 3.2 所示。

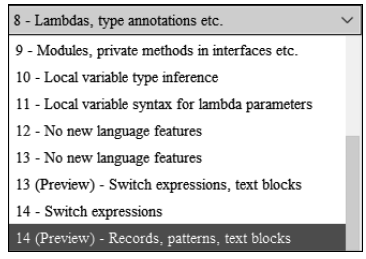

图 3.2　支持的 JDK 特性

可以看到，最新版本的 IntelliJ IDEA 已经支持 JDK 14 的特性。通常在 SDK 选取完成之后，IntelliJ IDEA 会提供一个 SDK 默认的语言级别。如果目标级别没有明确定义，则认为它与源语言级别相同。

例如，如果使用的是 JDK 1.8，则只能兼容 JDK 1.8 及其以下的特性，此时语言级别为 SDK default(8-Lambdas, type annotations etc.)，如图 3.3 所示。

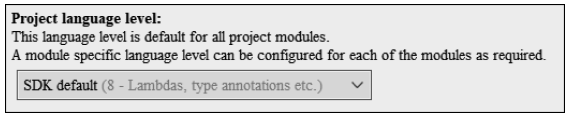

图 3.3　默认语言级别

如果项目配置了 JDK 1.8，但是只使用了 JDK 1.7 的特性，则语言级别可以选择为 7-Diamonds，ARM，multi-catch etc.。如果项目中使用了 JDK1.8 的新特性（如 Lambda 语法），但是使用的却是 JDK 1.7，即使 Project language level 选择了 8-Lambdas，type annotations etc. 也是没意义的，同样会产生编译错误。

所以 Project language level 用来限定项目编译检查时最低要求的 JDK 特性。具体使用哪种语言特性不仅取决于当前项目的 JDK 依赖，还需要开发者对 JDK 特性有比较深入的了解。

Project compiler output 用于指定当前项目编译结果的全局输出目录，同时各模块还可以自定义编译输出目录。如果模块不进行编译输出目录的定义，则将继承并使用全局的编译输出目录。

在上述内容中，Project SDK、Project language level 和 Project compiler output 都是基于项目级别进行的定义。在大型且复杂的项目中通常包含了很多模块，并且每个模块使用的 Project SDK 和 language level 很有可能是不同的，因此可以对不同模块进行更细级别的配置来覆盖项目级别的配置以实现兼容。

如果项目中使用了 Maven 构建管理工具并且在 pom.xml 文件中指定了编译器版本，则将会对项目中（包括模块内）指定的编译器级别产生限定，这种强制性的指定有可能会导致编译错误，此时需要修改 pom.xml 文件并重新刷新 Maven 配置。

3.1.2　模块

模块（Modules）选项卡用于对项目中的模块进行管理，它可以覆盖 Project 级别的选项配置并将配置的粒度细化，如图 3.4 所示。

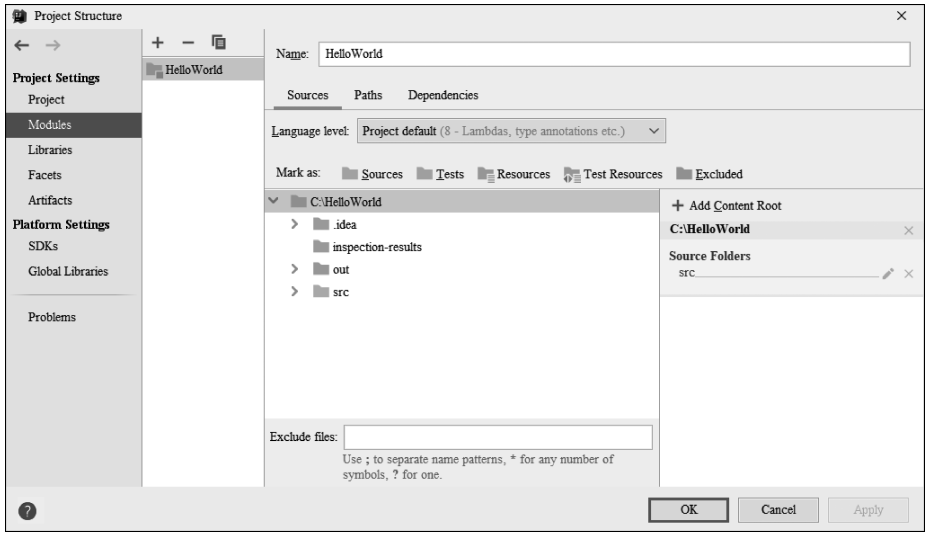

图 3.4　Modules 模块管理

Modules 列出了当前项目管理的所有模块，默认新建的项目就是一个模块。

用户可以将其他项目引入当前项目作为一个子模块，也可以在当前项目中新建一个子模块(项目)。这样做是有好处的，通过模块化管理可以轻松地将各种功能独立开来，这种类似单元组装的方式可以最大限度地实现组件的复用，也便于对项目进行更好的组织管理。后续会讲解如何创建与管理模块。

选择模块列表中的某一模块(如 HelloWorld)后，窗口右侧会展示该模块的具体信息。对于项目来讲，其子模块所在的存储位置并没有统一的要求，通常组成项目的各模块都位于当前工程目录下的多个子目录中，子模块与项目在磁盘上的存储位置没有必要的联系。

在右侧的模块信息中，首先展示了模块在当前项目中的名称。如果一个模块出现在多个项目中，则它可能在这些项目中具有不同的名称，但事实上还是同一个模块所对应的项目。

Sources 选项卡下指定了模块使用的语言级别，Project 标签下设置的语言级别是整个项目的语言级别，而当前模块下可以对项目级别的设置进行覆盖，毕竟很多时候一个项目所需要的各个模块可能是在不同版本或不同语言级别的 JDK 下开发完成的。

Sources 选项卡下还提供了 Mark 标记管理。这些标记用于标识工程中的各种内容，IntelliJ IDEA 根据这些标记进行相应内容的识别，如哪些目录用于存放源代码、哪些目录用于存放静态文件、哪些目录用于存放测试代码、哪些目录被排除编译等。

在 Mark 标记主区域展示了模块的工程结构，通过选择区域内的某一目录并单击区域上方(Mark as 部分)的不同标签，可以将其标记为某个指定用途的目录，打标成功的目录会变为与标记相同的颜色。

例如，对于标记为 Sources 的目录，IntelliJ IDEA 会使用 javac 命令去编译其中的源码文件。如果模块中无 Sources 标记，则其中的源码文件通常会显示为 图标，这表明源码没有加入 IntelliJ IDEA 的 Source 管理，即 Java class located out of the source root 状态，此时源文件将无法编译或运行。

被标记的目录会展示在右侧的 Content Root 列表下，各种标记的含义如下：

(1) Source Roots/Source Folders：标记为此类的目录会作为构建过程的一部分进行编译，Source Roots 的子文件夹代表 Java 的包结构。

(2) Resource Roots/Resource Folders：标记为此类的目录用于管理资源文件，如图片、xml 配置文件和 properties 属性文件等。在构建过程中，所有 Resources 标记目录下的资源会被复制到 Output 文件夹，并且在项目打包时会被复制到 jar 或 war 中，同时忽略标记为 Excluded 的内容。

(3) Excluded Roots：标记为此类的目录会被 IntelliJ IDEA 忽略，同时在进行搜索时 IntelliJ IDEA 也不会去查找这个目录下的内容。将一些不重要的目录标记为 Excluded Roots 可以提高 IntelliJ IDEA 的用户体验。

用户可以单击右侧的修改路径图标或删除图标来管理标记，也可以通过 Mark as 按钮组标记或取消。有些时候用户新建或导入的项目在内容标记上是不完整的，因此需要手工进行处理。

Paths 选项卡指定了模块的编译输出路径，用户可以保持继承自项目的默认配置，也可以手工指定项目类与测试类的编译输出路径，如图 3.5 所示。

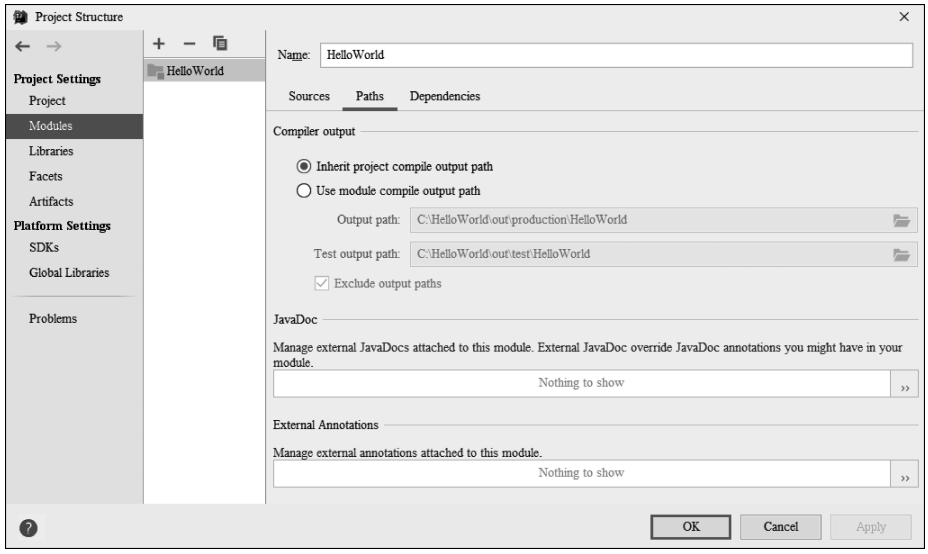

图 3.5　Paths 路径配置

在 Compiler output 选项下，Inherit project compile output path 会继承项目的编译输出路径，也就是在 Project 选项中设置过的 Output 路径。Use module compile output path 用于手工指定源码与测试类的编译输出路径，其中 Output path 代表源码的编译输出路径。Test output path 代表测试代码的编译输出路径。Exclude output paths 可以排除输出目录。

JavaDoc 用于管理与模块关联的外部存储位置的 JavaDoc 列表，并对模块内的 JavaDoc 进行覆盖。

External Annotations 用于管理外部注解以将其与模块关联。

Dependencies 选项卡中 Module SDK 用于从系统配置的 SDK 中进行选择并指定模块级别的 SDK 版本。如果需要的 SDK 不在列表中，则可执行菜单 Add SDK 命令，然后选择所需的 SDK 类型，如图 3.6 所示。

以添加 JDK 为例，在打开的对话框中选择 JDK 主目录，然后单击 OK 按钮确定，如图 3.7 所示。

单击右侧的 Edit 按钮可以查看或编辑 SDK。还可以对依赖项及依赖应用范围进行配置，如图 3.8 所示。

用户可以在 Scope 选项列指定依赖的应用范围以使其在不同运行环境下生效，如编译、测试或运行时等，其具体含义如下。

(1) Compile：对项目类和测试类来讲，编译和运行都有效。

(2) Test：仅对测试类来讲，编译和运行都有效。

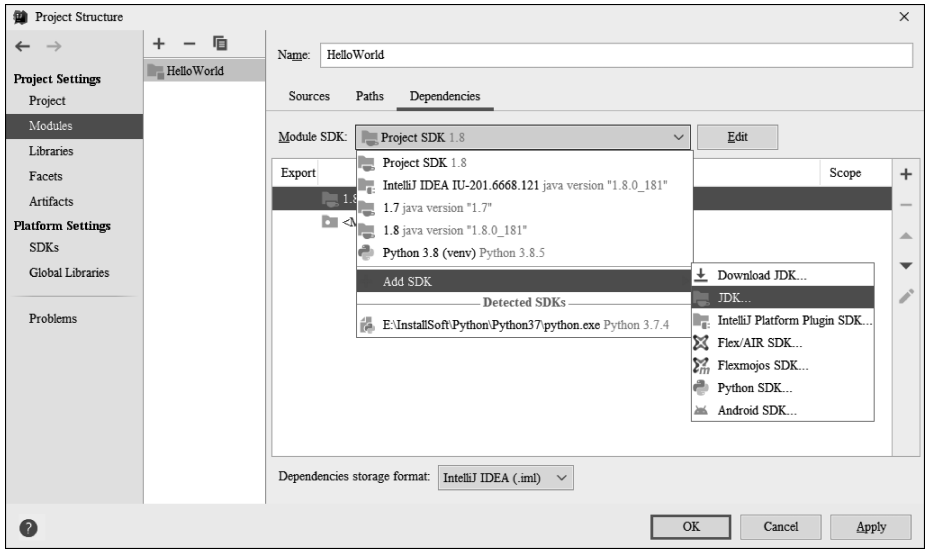

图 3.6　配置模块 SDK

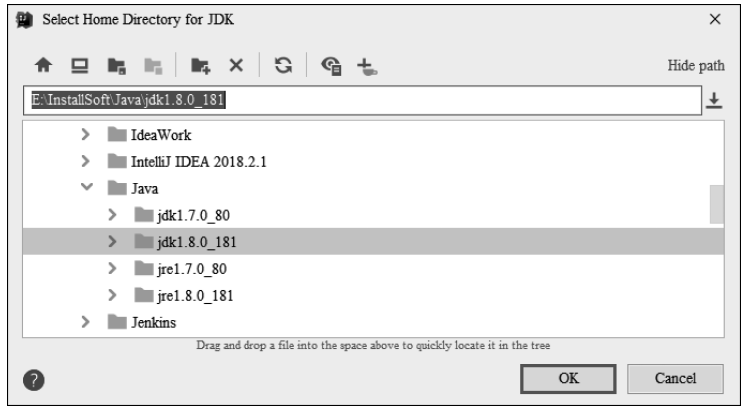

图 3.7　添加 JDK

(3) RunTime：对项目类和测试类来讲，仅运行时有效。

(4) Provided：对项目类来讲，仅编译时有效，而对测试类来讲，构建和运行时有效。

最后在 Dependencies storage format 中选择用于存储依赖关系的格式(IntelliJ IDEA 模块或 Eclipse 项目)，该选项会对使用不同开发工具的团队提供帮助。

3.1.3　类库

类库(Libraries)选项卡列出了当前项目使用的外部资源类库，这些资源既可以直接加载使用，也可以通过项目管理工具(如 Apache Maven)来管理并使用，如图 3.9 所示。

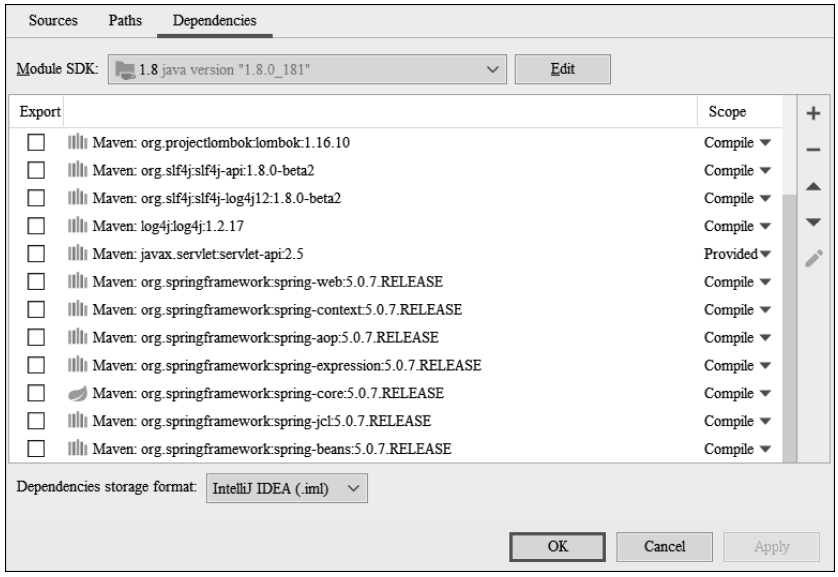

图 3.8　管理依赖(非 HelloWorld 项目截图)

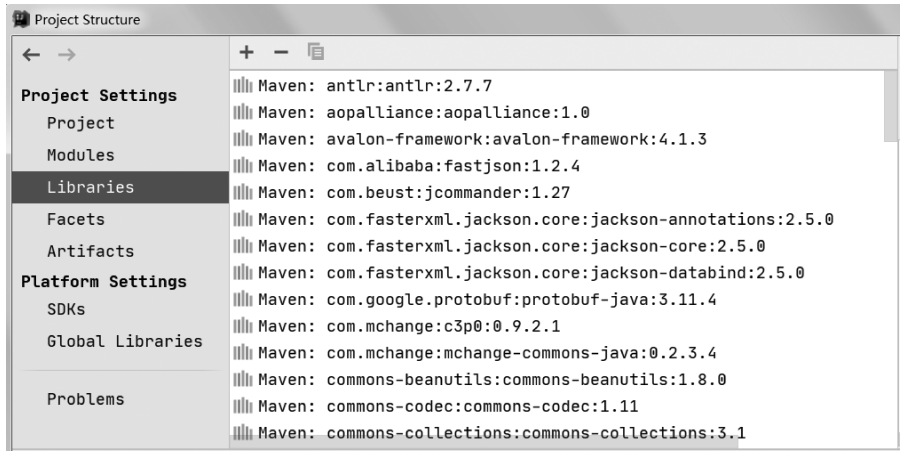

图 3.9　管理依赖

如果当前项目需要添加新的类库,则可单击资源列表上方的 + 按钮或使用快捷键 Alt+Insert,如图 3.10 所示。

用户根据需要选择添加类库的方式,选择 Java 方式可以直接加载外部资源。如果使用了构建管理工具,则可以采用 From Maven 的管理方式,关于 Maven 的使用可参考第 6 章。选择 Java 方式后打开如图 3.11 所示的对话框。

因为 IntelliJ IDEA 采用了模块管理的方式,所以在多模块项目中添加资源时需要指定将资源应用于哪些模块,选择完成后单击 OK 按钮确认。

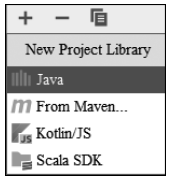

图 3.10 添加依赖

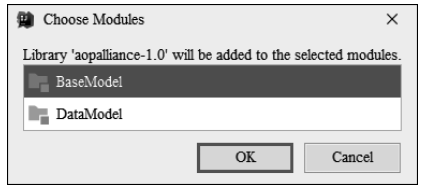

图 3.11 指定模块(第 3 章 HelloWorld 示例)

那么模块如何知道哪些资源是自己的呢？之前提到过,在模块下的同名.iml 文件中存放了模块的配置信息,打开同名.iml 文件查看配置,代码如下：

```
//第 3 章/DataModel.iml
<component name = "NewModuleRootManager" inherit - compiler - output = "true">
  <exclude - output />
  <content URL = "file:// $ MODULE_DIR $ ">
    <sourceFolder URL = "file:// $ MODULE_DIR $ /src" isTestSource = "false" />
    <sourceFolder URL = "file:// $ MODULE_DIR $ /test" isTestSource = "true" />
  </content >
  <orderEntry type = "JDK" JDKName = "1.8" JDKType = "JavaSDK" />
  <orderEntry type = "sourceFolder" forTests = "false" />
  <orderEntry type = "library" name = "antlr - 2.7.7" level = "application" />
  <orderEntry type = "module" module - name = "BaseModel" />
</component >
```

可以看到,模块配置中加载了外部资源并将其指定为 library 类型。level 属性指定了这些资源的应用范围究竟是工程级别还是应用级别。

每次通过手工方式添加外部依赖时,IntelliJ IDEA 都会自动为依赖内容生成一个类库名称,选择的外部资源文件都会自动加入这个类库的管理。通常类库名称会依据所选择的目标文件中第一个文件的名称来命名。如果用户想要更改其名称,则可在右侧窗口上方的名称文本框中进行修改,如图 3.12 所示。

3.1.4 特性

特性(Facets)描述了模块中使用的框架、技术和语言并体现了模块的应用特征,这些 Facets 特性让 IntelliJ IDEA 知道如何对待模块中的内容,并与相应的框架和语言保持一致,如图 3.13 所示。

Facets 分类整理了项目的各种特性,如项目基于 Spring 进行构建同时又是 Web 项目,所以此时会同时展示 Spring 与 Web 选项。

大多数 Facets 可以无冲突地添加到项目中,它是自动且隐蔽的。IntelliJ IDEA 可以很好地识别出项目的特性并进行归类管理,以 Web 特性为例进行说明。

在图 3.13 中,Name 用于定义 Web 特性的名称,也可以使用系统默认提供的名称。

Deployment Descriptors 描述符主要用于管理应用的部署描述。其中 Type 是只读字

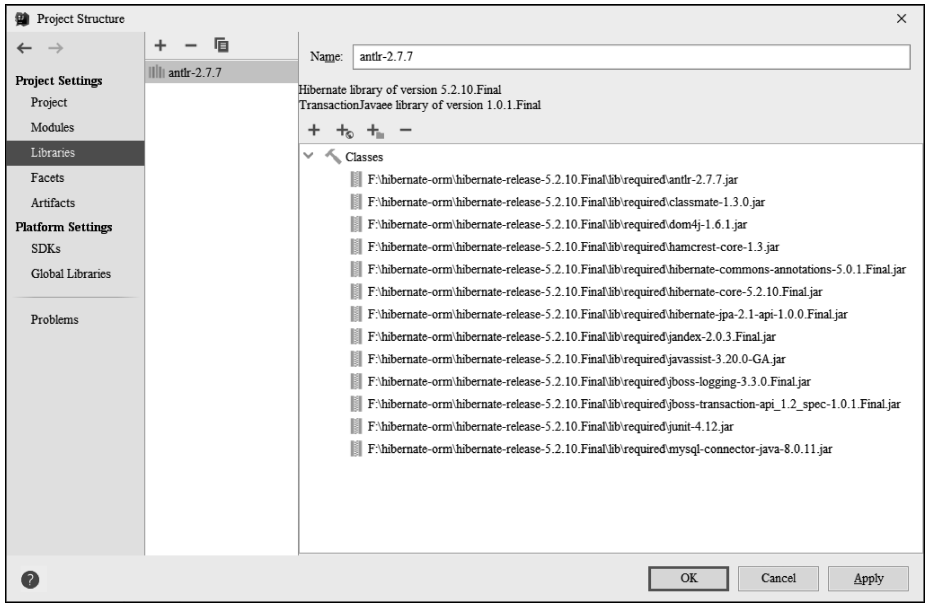

图 3.12　Libraries 类库

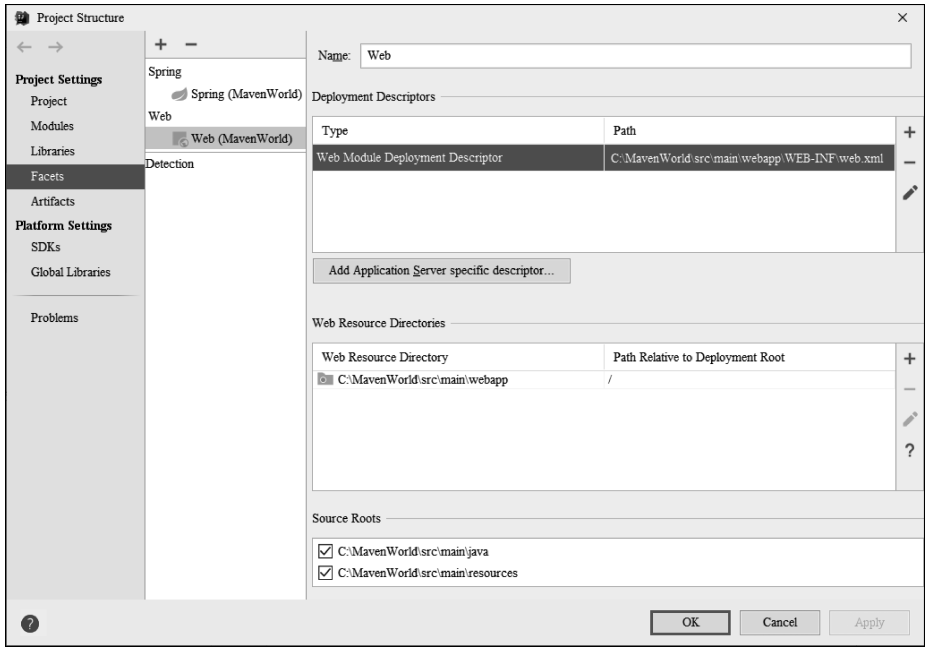

图 3.13　Facets 特性管理

段，用于展示部署描述符类型。

Type 类型值可以是 Web Module Deployment Descriptor、EJB Module Deployment Descriptor 或 Application Module Deployment Descriptor，这主要取决于当前项目的类型。

Path 只读字段用于展示项目的部署配置文件的位置，如 web. xml 或 application. xml 等。

单击 + 按钮或使用快捷键 Alt＋Insert 添加部署描述符，在打开的 Deployment Descriptor Location 对话框中选择部署描述符的位置，如图 3. 14 所示。

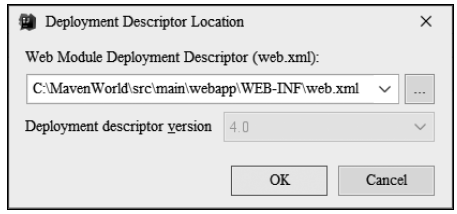

图 3. 14　新建部署描述符

单击 OK 按钮完成添加。要修改部署描述符，单击修改按钮 或使用快捷键 Enter 可以重新配置。单击删除按钮或使用快捷键 Alt＋Delete 从列表中删除选定的描述符。如果希望同时删除磁盘上的描述符，则可以在打开的 Delete Deployment Descriptor 对话框中勾选 Also delete file from disk 选项，如图 3. 15 所示。

在图 3. 13 中，Add Application Server specific descriptor 用于添加一个支持应用服务的部署描述符，常见的应用服务有 JBoss 服务、Glassfish 服务、WebSphere 服务、Tomcat 服务、Jetty 服务和 Weblogic 服务等。如果用户拥有应用服务开发经验，则一定会知道其中的一种或几种，如图 3. 16 所示。

图 3. 15　删除部署描述符

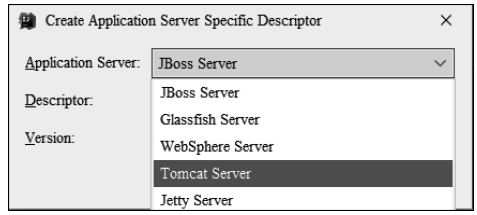

图 3. 16　应用部署描述

Web Resource Directories 选项用于将第三方或未分类资源路径映射到部署根目录，如图 3. 17 所示。

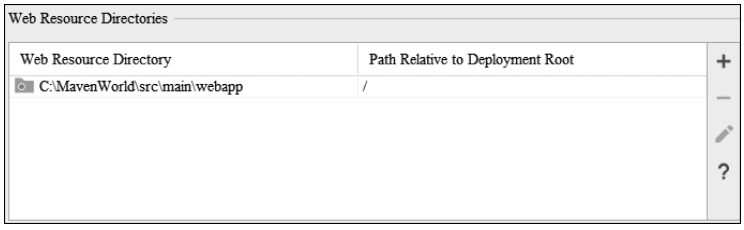

图 3. 17　Web Resource Directories

其中 Web Resource Directory 用于指定 Web Resource 所在的本地目录。在 Web Resource 目录下包含了 Web 开发所需的文件，如 JSP、HTML、XML 等。

Path Relative to Deployment Root 用于展示 Web Resource 相对于 Web 根目录部署的相对路径，如“/”代表 Web 根目录。Web Resource 目录下的内容会被复制到由 Path Relative to Deployment Root 所指定的 Web 模块部署目录中。

单击 + 按钮或使用快捷键 Alt＋Insert 打开 Web Resource Directory Path 对话框，如图 3.18 所示。

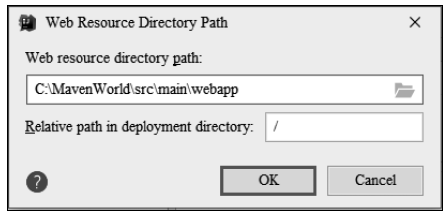

图 3.18　新建 Web Resource 目录映射

最后是 Source Roots 选项，这部分内容展示当前模块中的 source root 列表，如 Web 项目的源码路径及资源路径等。

还有一些 Facet 是继承自其他 Facet 的，要添加这些 Facets 就必须先完成对其父 Facet 的添加，同时这些 Facets 也依赖于 IntelliJ IDEA 的相关插件是否已开启。

Facets 特性主要用于配置工程结构以便应用于 Artifacts 中，也可以这么说：特性的定义是为了发布做准备。

3.1.5　项目生成

项目生成(Artifacts)代表了项目应用的目标，如可进行部署的 War 文件、应用归档文件(Java Archive File)等。对于可运行项目来讲，在 Artifacts 创建完成后就可以将其部署到应用服务器了。

项目的 Artifacts 可能是一个归档文件或者目录。在部署 Tomcat 应用时通常会在 webapps 目录下放置一个项目文件夹，这个文件夹就是标准的 Artifacts 结构。如果项目最终需要生成 JAR 文件供外部使用，那么它也是一种 Artifacts。

那么如何对 Artifacts 进行创建呢？单击 Artifacts 列表上方的 + 按钮或使用快捷键 Alt＋Insert，打开如图 3.19 所示的列表。

根据需要选择合适的生成目标，如果当前项目只是一个普通应用，那么可以选择 JAR 类别，此操作类似于从 Eclipse 中导出 JAR 资源的操作，如图 3.20 所示。

在图 3.20 中，如果要导出带有可运行方法(主方法与测试用例)的 JAR 文件，就要使用 Runnable JAR file 了。

IntelliJ IDEA 中提供了类似的操作，它同样支持生成不同类型的 JAR 文件。用户首先需要选择待导出的模块，或是将全部模块一起导出，如图 3.21 所示。

如果待导出模块中并不包含可运行方法(主方法与测试用例)，则 Main Class 位置可以为空。如果需要包含可运行方法，则用户可以单击右侧的文件夹图标，在弹出的方法列表里选择将要使用的方法或直接输入待运行方法。

在 JAR files from libraries 选项下，如果选择 extract to the target JAR 选项，则会生成

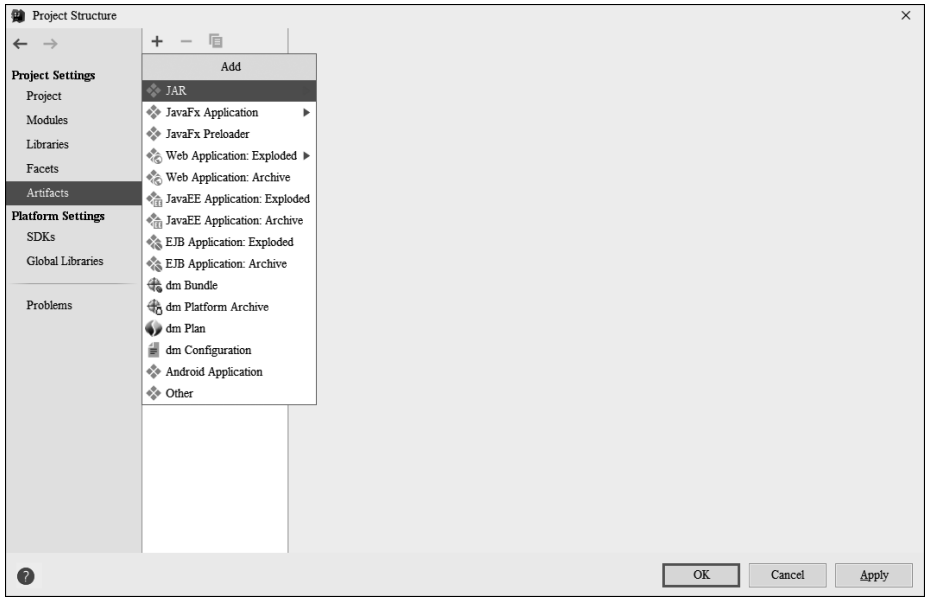

图 3.19　新建 Artifacts

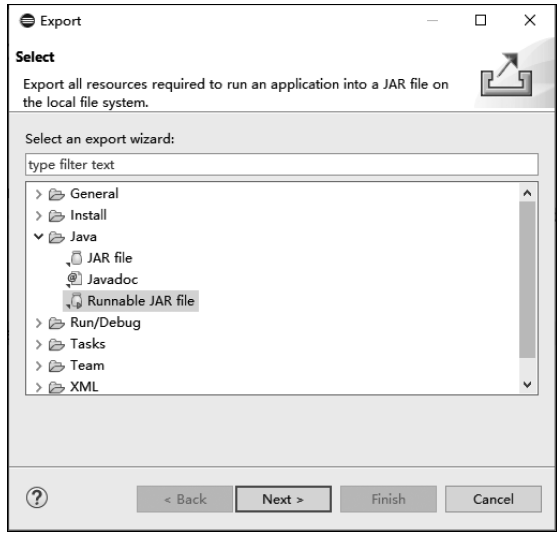

图 3.20　Eclipse 下导出 JAR 文件

对应的 JAR 文件。如果选择 copy to the output directory and link via manifest 选项，则下方的 Directory for META-INF/MANIFEST.MF 会变为可编辑状态，以便为项目生成启动文件 MANIFEST.MF，并放置到项目中用户指定的目录位置。如果 Main Class 中包含内容，则 Directory for META-INF/MANIFEST.MF 一定会处于可编辑状态。

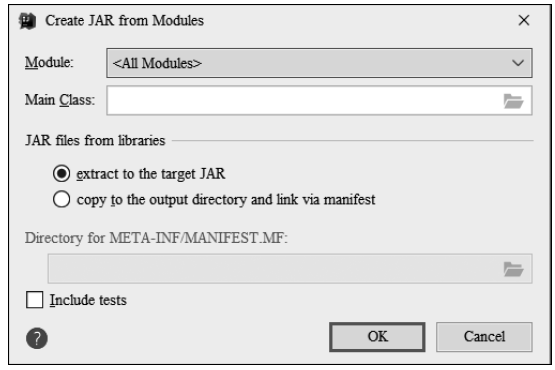

图 3.21　IntelliJ IDEA 导出 JAR File

选择目标 Artifacts 可以对其进行设置，如图 3.22 所示。

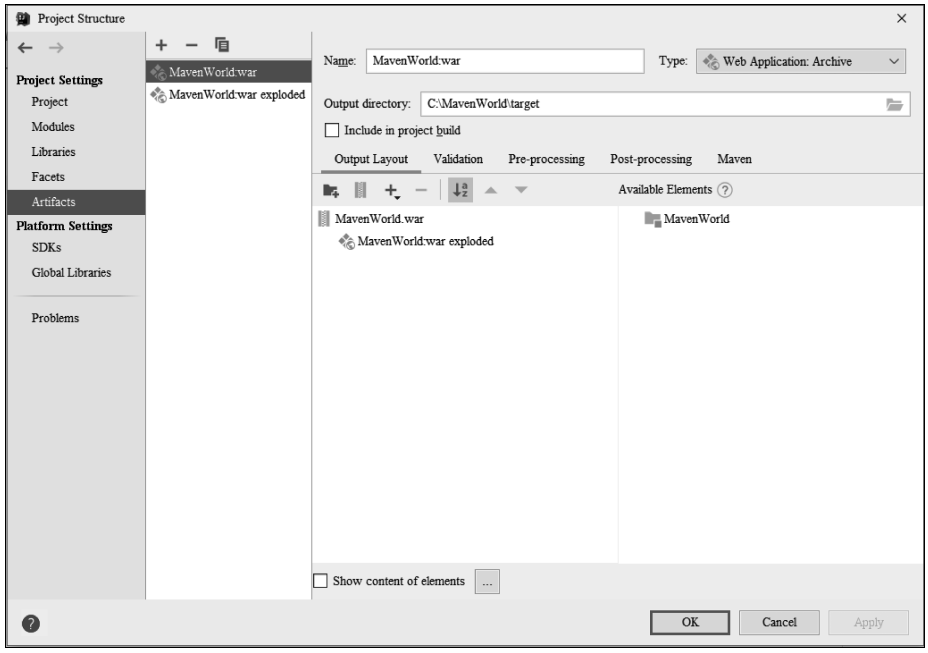

图 3.22　配置 Artifacts

其中 Name 用于指定 Artifacts 配置的名称，用户可以保持默认或自定义。Type 用于指定 Artifacts 的类型。Output directory 用于指定执行构建时 Artifacts 将被放置的目录。

在 Artifacts 配置完成后，执行菜单 Build→Build Artifacts 命令手工执行目标的构建工作，或通过执行 Run/Debug 的方式来构建一个 Artifacts，前提是用户已经建立了 Artifacts 任务，此时在执行 Run/Debug 配置的时候 Artifacts 会自动构建。

基于 Artifacts 进行的构建在部署 Web 项目与运行调试时会经常用到，通常会有 war 和 war exploded 两种方式。它们的区别在于当采用调试模式运行项目时，以 war exploded 形式发布的应用是可以进行热部署与调试的，所以建议采用这种方式进行应用的发布。

项目结构窗口不仅对项目结构进行了管理，还为项目后期的部署运行提供了定义与准备。

3.1.6 开发集成工具

在 SDKs 选项卡中进行全局开发集成工具(SDK)配置，如 Java SDK、Python SDK 等，如图 3.23 所示。

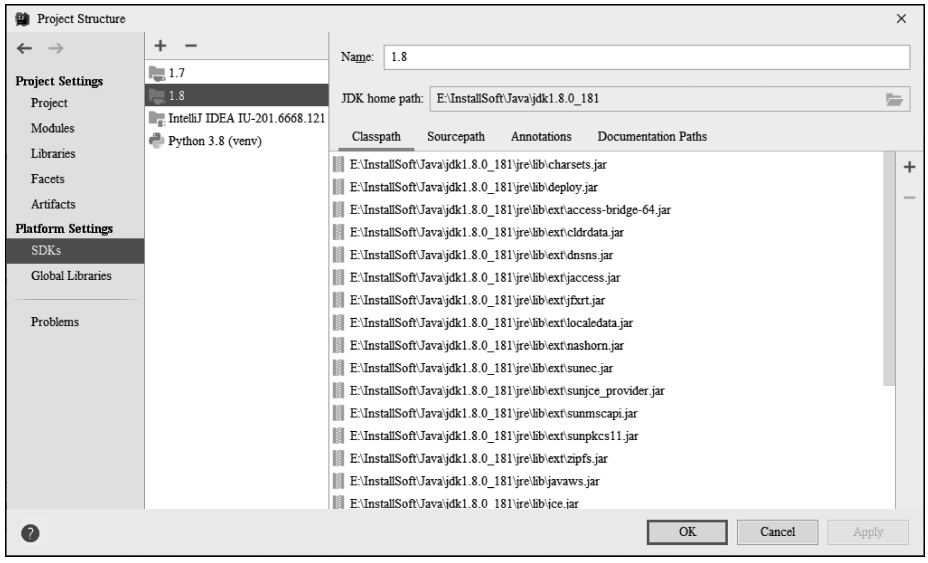

图 3.23 配置全局 SDK

单击 + 按钮或使用快捷键 Alt+Insert 打开新建 SDK 列表，如图 3.24 所示。

单击 Download 按钮 JDK 将会在线下载最新的 JDK 版本，如图 3.25 所示。

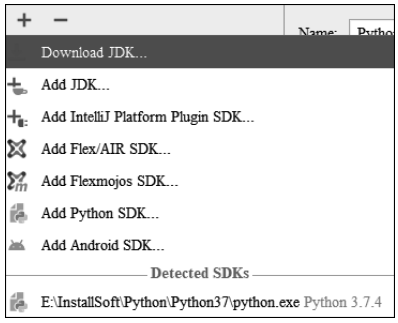

图 3.24 新建 SDK

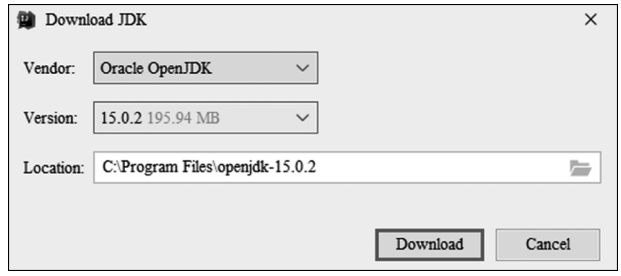

图 3.25 在线下载 JDK

用户也可以手工进行其他 SDK 的安装，第 1 章中我们已经进行过相关操作，因此不再过多描述。

3.1.7　全局类库

全局类库(Global Libraries)中配置了全局可用的类库，如图 3.26 所示。

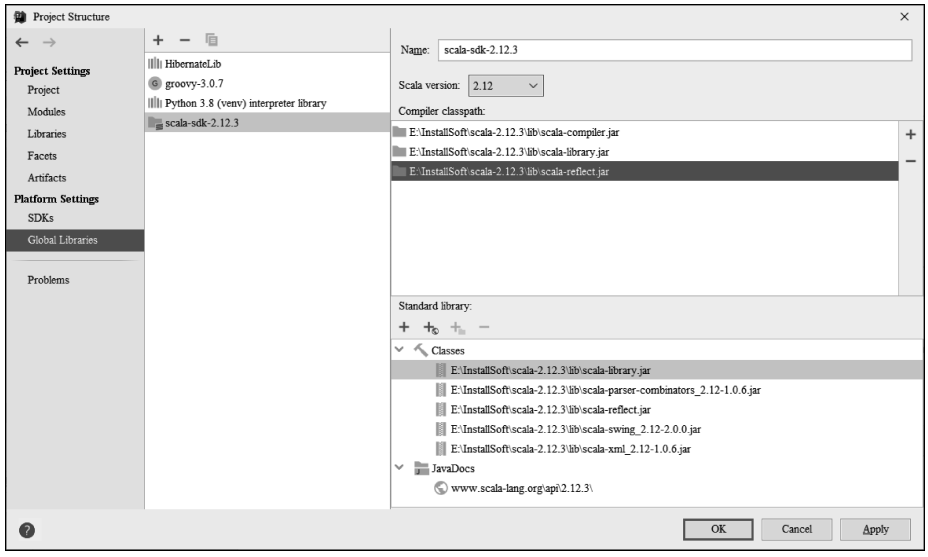

图 3.26　全局类库

全局类库可以被模块共用，用户也可以在项目中配置单独使用的类库。

3.2　模块的创建与使用

在 IntelliJ IDEA 中项目(Project)是由多个模块(Module)组成的，模块既是独立的功能单位，同时彼此之间又存在联系，所以项目中 Project 是规模范围定义的目录，Module 才是项目里面的真正内容。

模块可以独立地编译、运行、测试和调试，它提供了一种降低大型项目复杂性的方法，还可以作为公用部分被多个项目共同使用。在每个模块目录下都存在一个后缀为.iml 的文件，此文件包含了模块的配置信息，如依赖等。

IntelliJ Idea 项目默认为单 Module 的(模块即项目)，开发者既可以为项目创建新的模块，也可以导入已经存在的项目作为模块。

3.2.1　新建模块

要管理项目中的模块，首先需要保证项目存在且处于打开状态。要创建新的模块，执行菜单 File→New→Module 命令打开新建模块窗口，如图 3.27 所示。

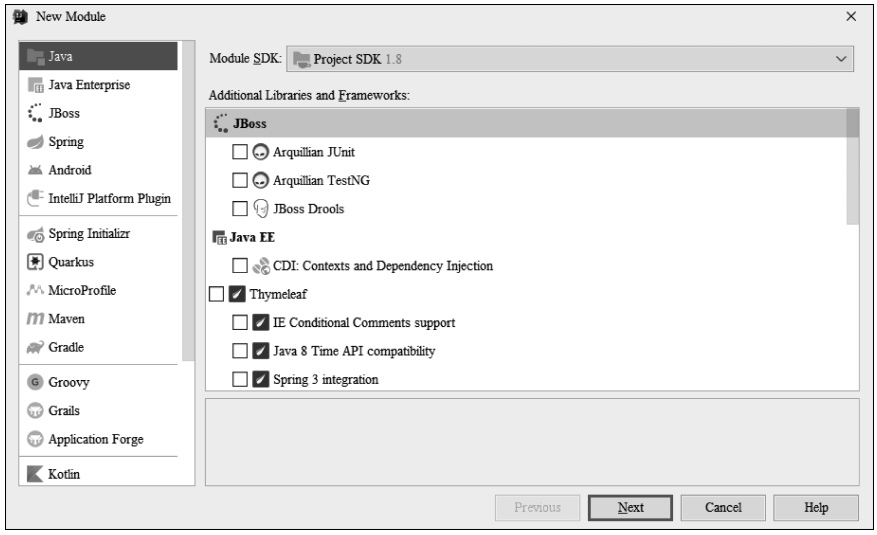

图 3.27　新建模块

图 3.27 中左侧列出了可以创建的模块视图，如 Java 视图代表普通的 Java 程序，它适用于简单结构的项目。Java Enterprise 视图用于开发和部署可移植、健壮、可伸缩且安全的服务器端 Java 应用程序，Web 项目大多在这个视图中进行开发，功能也较为丰富。

注意：社区版(Community)的 IntelliJ IDEA 是没有 Java Enterprise 选项的。

除了上述两种视图外，用户还可以在 Spring 视图下创建标准的 Spring 项目，在 Spring Initializr 下创建基于 Spring Boot 的微服务项目。如果选项卡中没有此项，则旗舰版的用户需要安装对应的 Spring Boot 插件并重启，而社区版用户需要将软件更换为旗舰版。

在新建窗口右侧指定模块的 SDK 版本，同时在 Additional Libraries and Frameworks 列表中勾选需要附加的类库或框架。如要创建带有 Hibernate 持久化功能的模块，则应在功能列表中勾选 Hibernate 选项，如图 3.28 所示。

勾选 Create default hibernate configuration and main class 选项可以为项目模块创建默认的 Hibernate 配置及示例类。

在 Libraries 选项下可以指定使用本地 Hibernate 类库或是在线下载，选择 Use libraray 并单击 Create 按钮选择本地类库文件，如图 3.29 所示。

选择所有文件后单击 OK 按钮完成添加，然后继续向下执行模块创建，如图 3.30 所示。

3.2.2　导入模块

虽然 IntelliJ IDEA 支持多模块管理，但是不推荐将没有任何关联的项目以模块的形式添加到同一项目中。

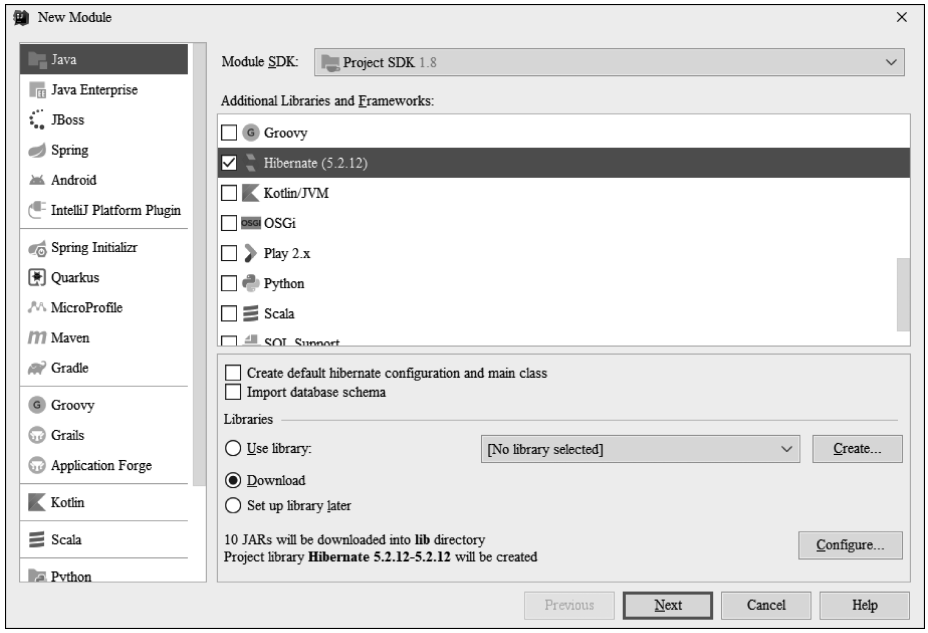

图 3.28　附加功能

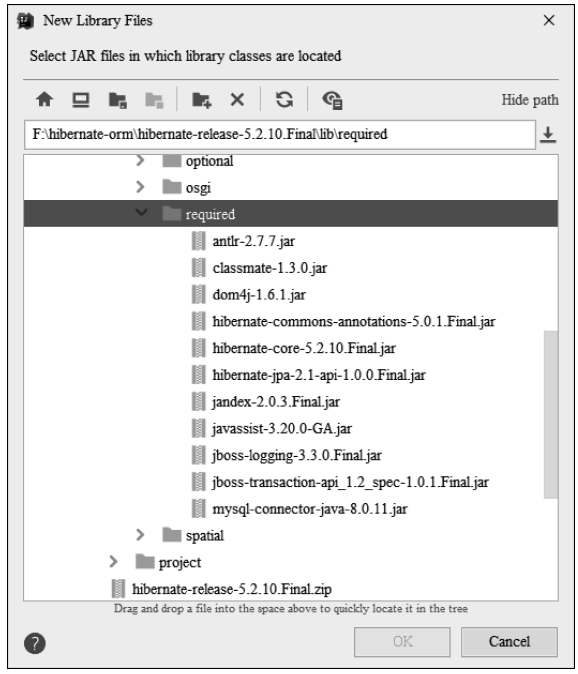

图 3.29　添加本地类库

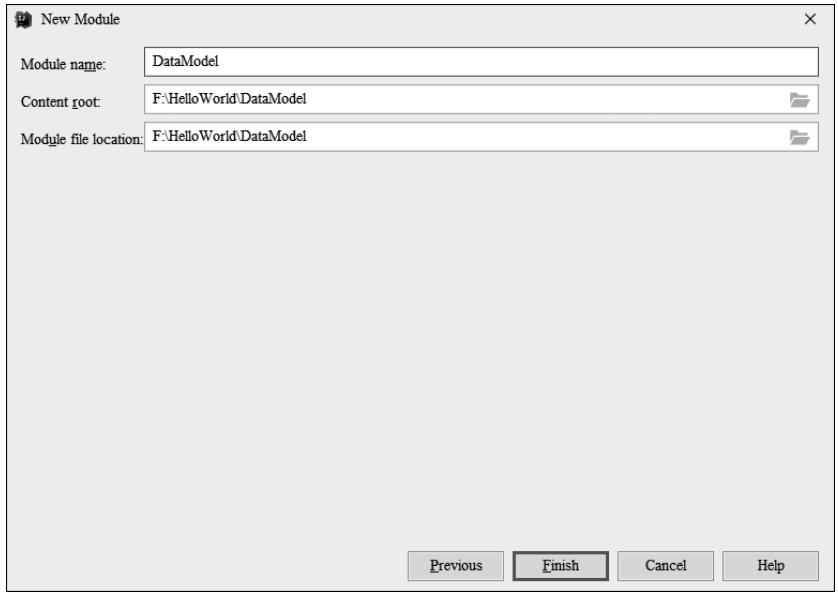

图 3.30　创建 DataModel 模块

要导入已经存在的模块，可在 Project Structure 窗口的 Modules 选项卡下单击 + 按钮或使用快捷键 Alt+Insert，在弹出的下拉列表中选择 Import Module，如图 3.31 所示。

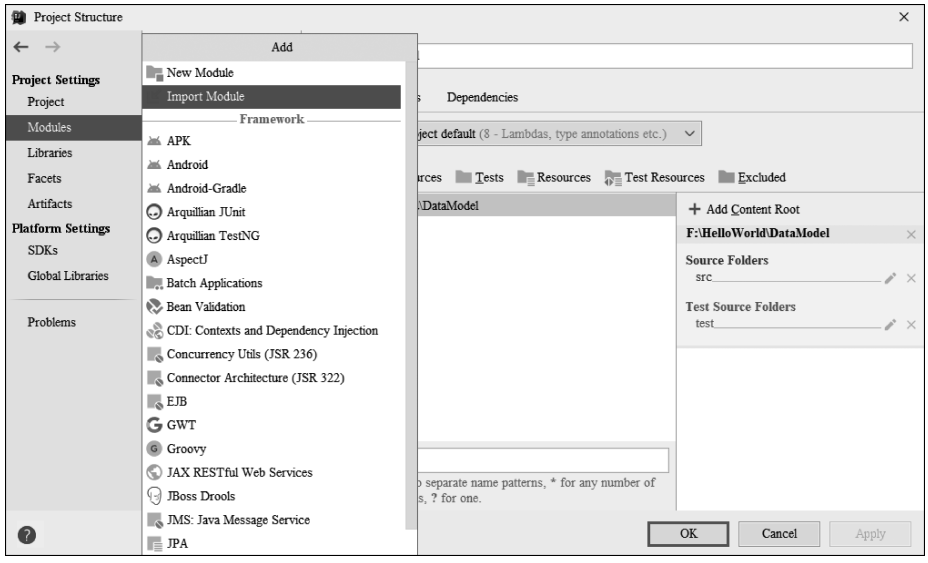

图 3.31　导入模块

选择本地模块项目所在位置，如果项目中存在.iml文件，则可以直接选择，如图3.32所示。

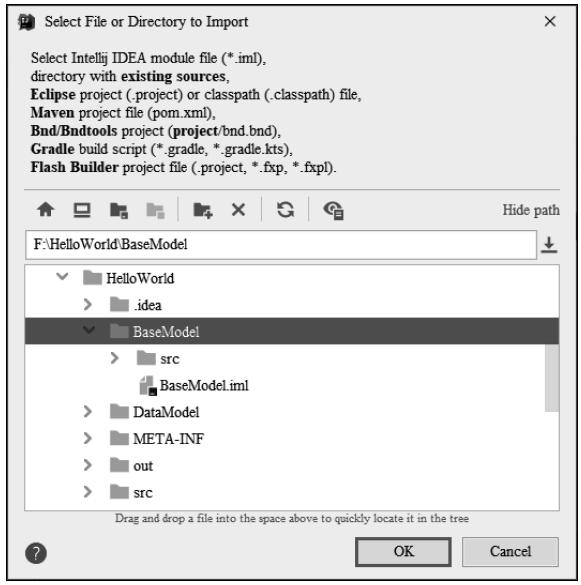

图3.32　导入本地项目作为模块

可以看到，IntelliJ IDAE对于不同类型项目的支持十分丰富，图3.32中列出了引入项目时可以加载的关键配置文件类型。导入时项目类型如图3.33所示。

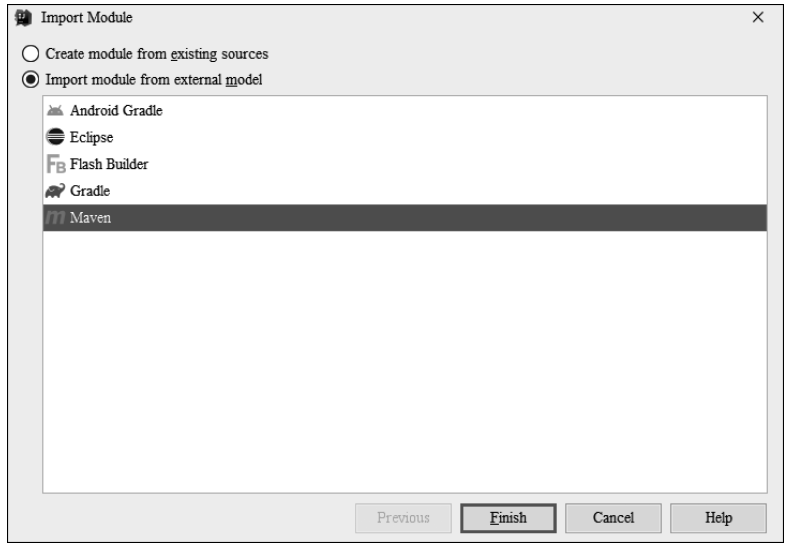

图3.33　选择待引入的模块类型

模块导入完成后的项目结构如图 3.34 所示。

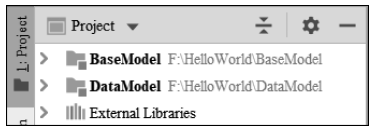

图 3.34　项目模块结构

3.3　本章小结

本章主要介绍了 IntelliJ IDEA 中的项目结构与模块，对于这些概念的理解可以帮助读者更好地组织与管理项目。

在大型项目中通常包含几十个甚至上百个模块，如何维护这些模块之间的结构关系将会是一件具有挑战性的工作。在结合 Maven 等构建工具管理项目时，开发者一定要规划好项目结构并实现最优化管理。