第 5 章 构建父子模块及配置文件

本项目的架构采用父子模块模型，其中父模块与多个子模块共同构建整体系统。子模块被用于实现不同代码功能，实现任务分工明确。例如，公共枚举类、配置、方法等可以在子模块中集成，从而为系统提供共享资源。这种模块化的划分使各子模块能够承担特定的职责，有利于代码库的组织、管理和维护。

5.1 构建子模块

在 4.1 节中，已经构建了一个名为 library 的项目。当前要将其演化为一个父模块，随后在其父模块的基础上创建多个子模块。接下来将创建一个名为 library-admin 的子模块。主要职责在于充当整个系统的启动项目，并承担项目所有配置文件的配置。

5.1.1 创建 library-admin 子模块

1. 使用 IDEA 工具创建 library-admin

打开 IDEA 工具，新建一个项目的主项目：library-admin，因为是系统的主入口，所以要保留启动类。右击 library 项目后选择 New→Module 选项，并新建一个 Module 模块，如图 5-1 所示。

选择左侧的 Spring Initializr 选项，与之前 Web 页面创建 library 项目基本一致，然后填写项目名称、选择语言、JDK 版本和打包方式等信息，填写完成后，单击 Next 按钮，如图 5-2 所示。

选择 Spring Boot 3.1.3 版本，其余的保持默认，单击 Create 按钮，如图 5-3 所示。

等待 library-admin 项目创建完成，当前项目的结构目录如图 5-4 所示。

2. 父模块依赖修改

在 Maven 项目中，父级依赖项可以被子模块继承，从而实现共享的依赖关系、插件配置等信息。这种机制让子模块能够利用父模块中定义的设置，从而达到统一管理、简化构建过程的效果。通过这样的方式，多个相关模块之间的依赖管理和构建过程将变得更加简单和高效。

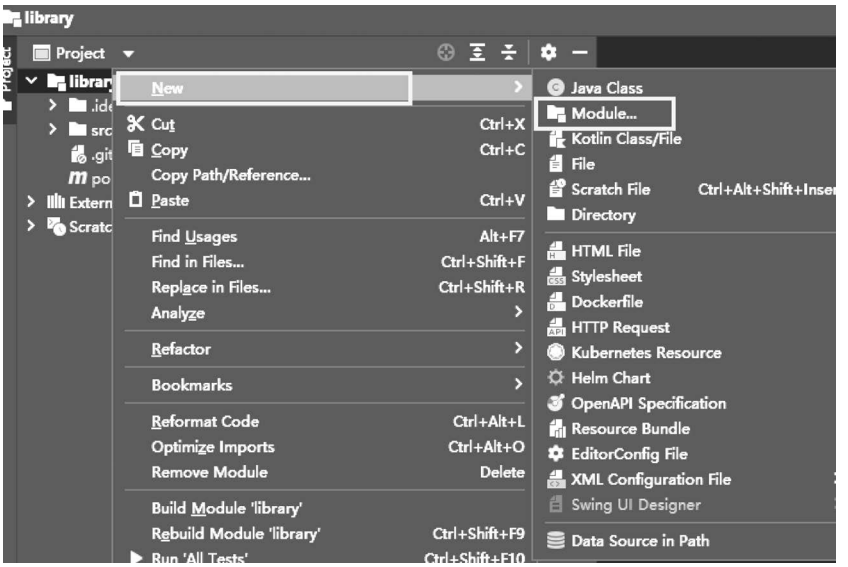

图 5-1　打开创建项目模块页面

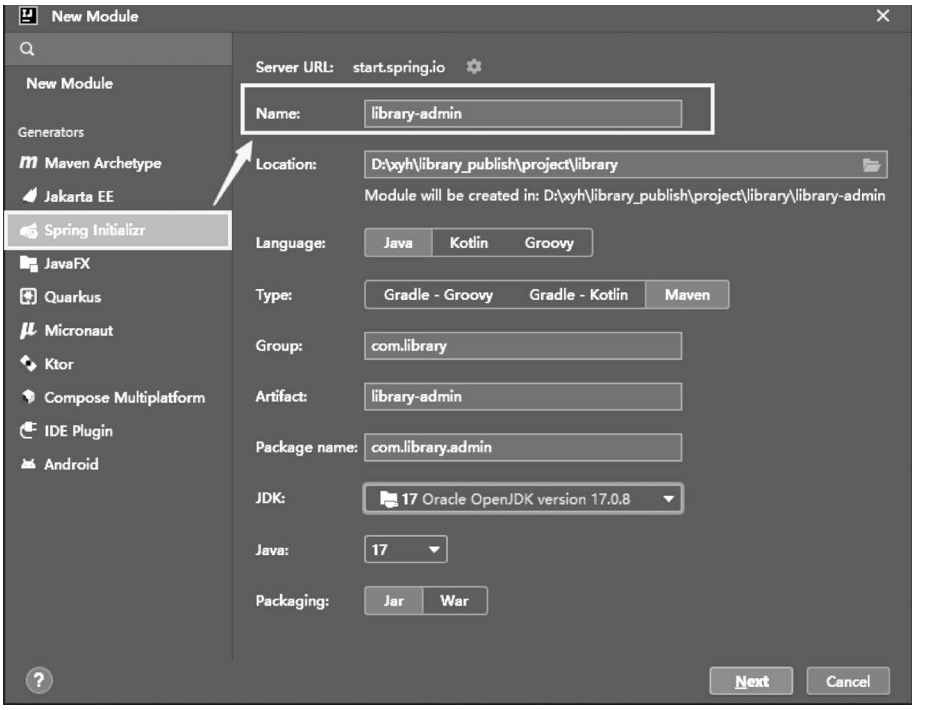

图 5-2　填写 library-admin 模块信息

父模块不包含任何业务代码，可以删除 src 文件夹和 target 文件夹，并修改父模块下的 pom.xml 文件，由于父模块不包含业务代码，仅用于管理子模块，所以选择将 packaging 标

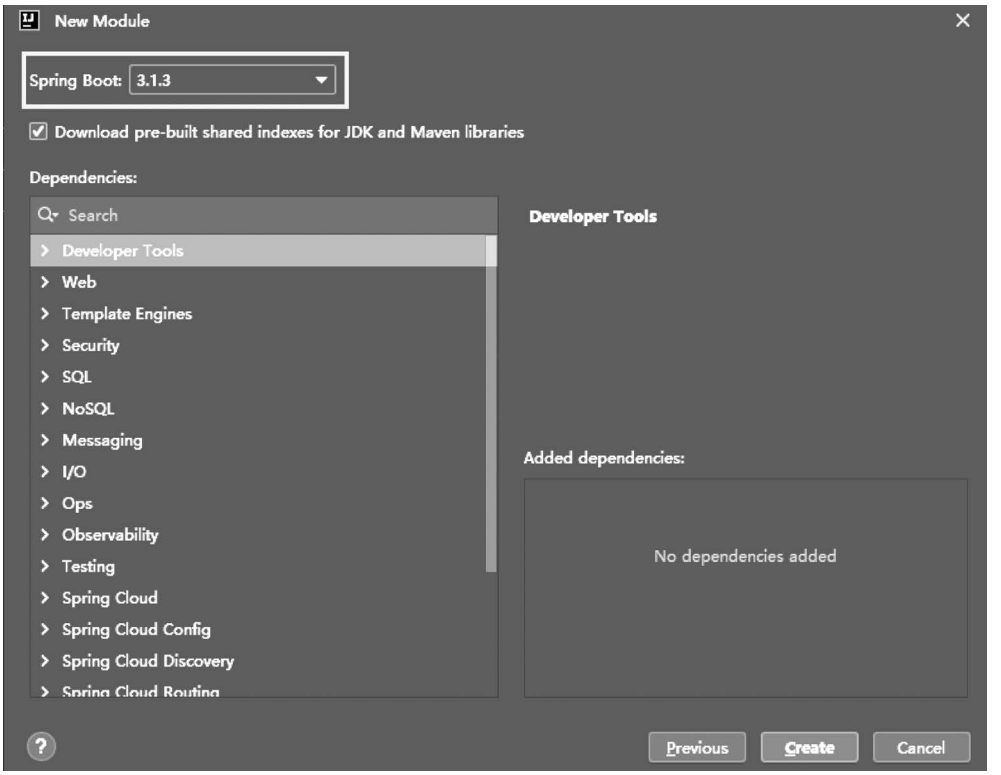

图 5-3　选择 Spring Boot 版本

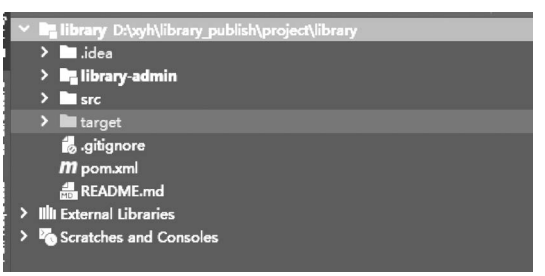

图 5-4　目录结构

签的值设置为 pom 是非常合适的。这会告诉 Maven 该项目仅用于管理其他子模块，不会生成实际的构建产物，代码如下：

```
//第 5 章/library/pom.xml
<groupId>com.library</groupId>
<artifactId>library</artifactId>
<version>0.0.1-SNAPSHOT</version>

<!--父模块的打包类型为 pom-->
<packaging>pom</packaging>
<name>library</name>
```

```
<description>父模块</description>
<properties>
    <java.version>17</java.version>
</properties>
```

packaging 标签在 Maven 项目的 pom.xml 文件中，用于指定项目的打包方式。它可以包含多个值，常见的有 jar、war 和 pom，开发项目一般选择的是 JAR 包的方式。

(1) jar 表示项目将以 JAR(Java Archive)包的形式打包。适用于纯 Java 项目，将编译后的类文件打包成一个 JAR 文件，供其他项目或模块使用。

(2) war 表示项目将以 WAR(Web Application Archive)包的形式打包。适用于 Web 应用项目，将 Web 应用的代码、资源和配置打包成一个 WAR 文件，可以部署到 Servlet 容器中。

(3) pom 表示项目本身不会生成任何产物，仅作为父模块或聚合项目，用于管理子模块的构建和依赖。

使用 module 标签将 library-admin 模块加入父模块中，并声明依赖的版本号，代码如下：

```
//第 5 章/library/pom.xml
<!-- 子模块依赖 -->
<modules>
        <module>library-admin</module>
</modules>

<dependencyManagement>
        <dependencies>
        <dependency>
     <groupId>com.library</groupId>
     <artifactId>library-admin</artifactId>
     <version>${version}</version>
        </dependency>
</dependencies>
</dependencyManagement>
```

3. 子模块依赖修改

修改 library-admin 项目依赖，打开 pom.xml 文件修改 parent 标签，引入父级依赖，命名为 library 项目。使当前项目可以继承父模块的相关配置，将 pom 的 groupId 设置为 com.library、将 artifactId 设置为 library、将版本号设置为 0.0.1-SNAPSHOT，代码如下：

```
<parent>
    <groupId>com.library</groupId>
    <artifactId>library</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</parent>
```

然后删除 Spring Boot Starter 依赖和测试依赖，使项目可以直接继承父模块的依赖和配置，代码如下：

```
//第 5 章/library/library - admin/pom.xml
<artifactId>library - admin</artifactId>
<version>0.0.1 - SNAPSHOT</version>
<packaging>jar</packaging>
<name>library - admin</name>
<description>主系统</description>

<dependencies>
</dependencies>
```

pom.xml 文件修改完成后，将 library-admin 的配置文件修改为 yml 格式，然后添加项目名和端口，可参考 4.1.2 节，添加完成并启动项目，如果没有启动成功，则可查看控制台提示的错误信息进行相关修改。

4. 依赖版本管理

在父模块的 pom.xml 文件中，使用 dependencyManagement 标签声明依赖项的版本信息，父模块充当了一个中央配置的角色，为所有子模块提供了集中管理依赖版本的机制。这种方式使子模块无须指定版本号，而是从父模块中自动继承版本信息，确保了所有子模块都使用统一的依赖版本。

通过这种统一的依赖版本管理方式，项目的构建和开发过程变得更加规范和可控。同时，它也确保了依赖版本的一致性，避免了不同子模块因版本不匹配而可能引发的问题和错误，用于版本声明的代码如下：

```
//第 5 章/library/pom.xml
<properties>
   <project.build.sourceEncoding>UTF - 8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF - 8</project.reporting.
        utputEncoding>
    <java.version>17</java.version>
    <!-- 全局配置项目版本号 -->
    <version>0.0.1 - SNAPSHOT</version>
    <!-- 表示打包时跳过 mvn test -->
    <maven.test.skip>true</maven.test.skip>
        <fastjson.version>1.2.83</fastjson.version>
</properties>

<!-- 依赖声明 -->
<dependencyManagement>
    <dependencies>
       <!-- 子模块依赖 -->
       <dependency>
          <groupId>com.library</groupId>
          <artifactId>library - admin</artifactId>
          <version>${version}</version>
       </dependency>
       <!-- fastjson -->
       <dependency>
          <groupId>com.alibaba</groupId>
          <artifactId>fastjson</artifactId>
```

```
            <version>${fastjson.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

5. 项目 Maven 配置

在搭建 Maven 环境时，在 Maven 的 settings.xml 文件中添加了一个新的 mirror 节点，配置了阿里云镜像网址，那个是全局的配置方式。接下来要在项目中配置阿里云镜像网址，只能在当前项目中生效。

修改父模块的 pom.xml 文件，在 repositories 节点下加入 repository 节点，并配置阿里云镜像网址，代码如下：

```
//第5章/library/pom.xml
<repositories>
        <repository>
                <id>public</id>
                <name>aliyun nexus</name>
        <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
                <releases>
                <enabled>true</enabled>
                </releases>
        </repository>
</repositories>
<pluginRepositories>
        <pluginRepository>
                <id>public</id>
                <name>aliyun nexus</name>
        <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
                <releases>
                <enabled>true</enabled>
                </releases>
                <snapshots>
                <enabled>false</enabled>
                </snapshots>
        </pluginRepository>
</pluginRepositories>
```

5.1.2 创建 library-common 子模块

library-common 子模块旨在集中管理项目中的公共资源和实用工具。该模块将涵盖一些公共类的功能，包括 Redis 实用工具类、各种枚举类型及常用的错误码类等。

通过将这些公共资源和功能模块整合到一个统一的子模块中，总结以下几个优点。

(1) 通过将常见的功能集中在一个位置，项目中的不同部分可以轻松地共享和重用这些代码片段，从而减少重复编写代码的工作量。

(2) 所有公共资源都集中在一个模块中，更容易对其进行维护和更新。当需要修复漏洞或添加新特性时，只需在一个地方进行修改，从而降低维护成本。

（3）通过在一个模块中定义共享的枚举类型和工具函数，可以确保在整个项目中使用一致的标准，有助于提高代码的可读性和一致性。

在 library 项目下，新建一个 library-common 子模块，创建方式和创建 library-admin 子模块的方式基本一致，只要修改项目名称即可，如图 5-5 所示。

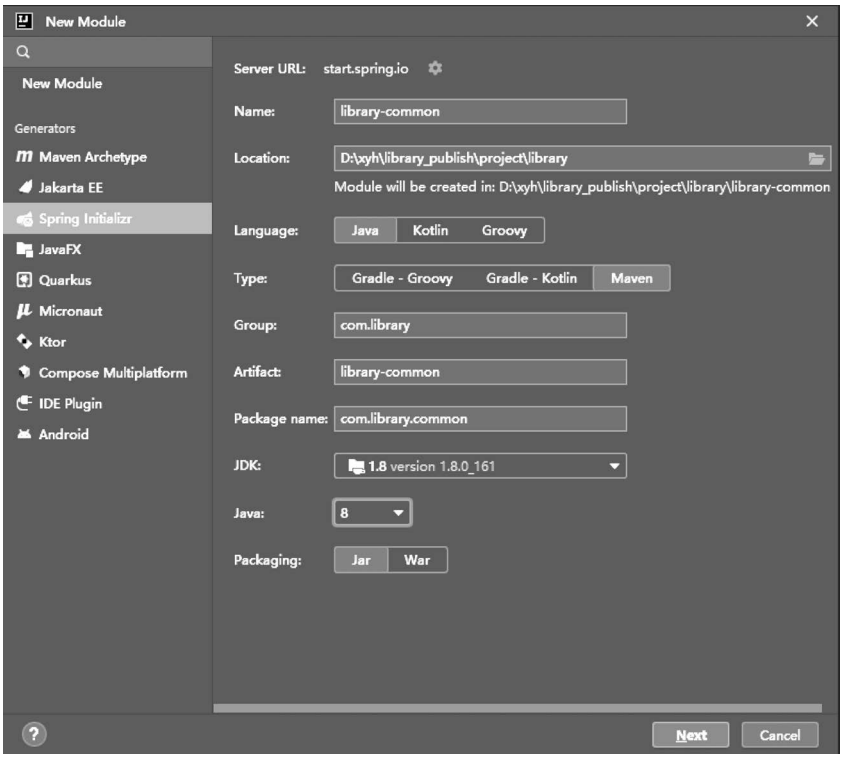

图 5-5　创建 library-common 子模块

创建项目完成后，删除不必要的代码文件，具体的目录文件如图 5-6 所示。

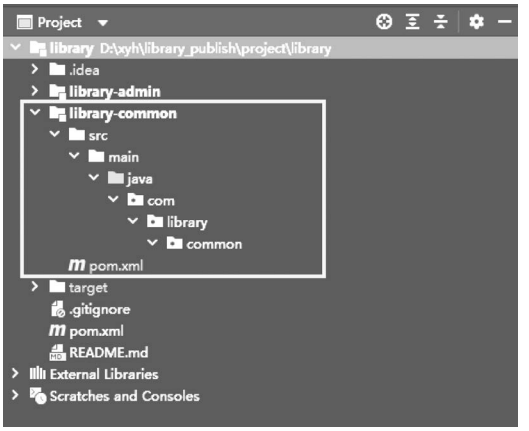

图 5-6　library-common 目录结构

修改 pom.xml 文件，添加父模块依赖，代码如下：

```
//第 5 章/library/library-common/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 https://maven.apache.org/
xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.library</groupId>
        <artifactId>library</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

    <artifactId>library-common</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>library-common</name>
    <description>工具模块</description>

    <dependencies>
    </dependencies>
</project>
```

将 library-common 子模块的依赖添加到父模块的 pom.xml 文件中，并声明依赖的版本信息，代码如下：

```
//第 5 章/library/pom.xml
<modules>
        <module>library-admin</module>
        <module>library-common</module>
</modules>

<!-- 依赖声明 -->
<dependencyManagement>
    <dependencies>
       <!-- 子模块依赖 -->
        <dependency>
          <groupId>com.library</groupId>
          <artifactId>library-common</artifactId>
          <version>${version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

5.1.3 添加项目配置文件

在开发和管理项目时，配置文件管理是非常重要的，尤其是在不同环境中部署和运行项目时。为了解决本地开发和测试环境与线上环境之间的配置不一致问题，这里要引入 3 个独立的配置文件，分别用于本地开发环境、测试环境和线上环境。这种方法不仅使项目配置

更加规范和清晰，还解决了在不同环境中可能出现的问题。

1. application.yml 配置文件

在 library-admin 项目中，将重新配置 application.yml 文件，以满足项目的需求。对项目命名及项目使用不同环境的配置进行切换，代码如下：

```
spring:
  application:
    name: library
  profiles:
    active: dev
```

配置上传文件大小的限制，如果这里不限制，则在用到文件上传时会出现报错，代码如下：

```
#上传文件大小配置
  servlet:
    multipart:
      enabled: true
      max-file-size: 50MB
      max-request-size: 50MB
```

在 Spring Boot 中默认只有 GET 和 POST 两种请求，但是可以使用隐藏请求去替换默认请求，例如，删除使用 DELETE 请求、修改使用 PUT 请求等，代码如下：

```
mvc:
    hiddenmethod:
      filter:
        enabled: true
```

配置项目的上下文路径，也可以称为项目路径，这是构成请求接口地址的一部分，例如，在项目中有个接口为/library/list，项目端口为 8080，然后访问这个接口的 URL：localhost：8080/library/list，然后在配置文件中加上了 context-path 为/api/library 之后，再去访问这个接口的 URL，就要改成 localhost：8080/api/library/library/list 才可以正常访问，代码如下：

```
server:
  servlet:
    context-path: '/api/library'
```

2. application-dev.yml 配置文件

开发项目时，在本地机器上运行项目是一个常见的场景。为了确保项目在机器上能够正常运行，先创建 application-dev.yml 配置文件。该文件包含适用于开发环境的配置选项。

(1) 连接本地开发数据库，以便可以使用本地数据库进行开发和测试。

(2) 配置连接本地的 Redis 缓存。

(3) 配置本地项目启动的端口号，这个可以和测试环境及线上的端口号不一致。

如何配置这些信息呢？在接下来的项目中会涉及，这里先配置本地项目启动的端口号，代码如下：

```
server:
  port: 8081
```

其余的两个配置文件 application-test.yml 和 application-prod.yml 先创建好，里面的具体内容在后面部署的章节中再添加。

3. 修改项目启动配置

环境的配置文件创建完成后，还需要配置不同环境的 profiles 才能在不同的环境下切换不同的配置文件。profiles 标签用于定义不同环境下的配置，每个 profile 元素表示一个不同的环境（本地开发环境、测试环境、生产环境），并包含了一些特定环境的配置。

在 library-admin 子模块的 pom.xml 文件中添加相关配置，代码如下：

```
//第 5 章/library/library-admin/pom.xml
<!-- 环境 -->
<profiles>
    <!-- profile 元素包含了特定环境的配置.每个 profile 都有一个唯一的标识符 id 标签,用于
标识不同环境 -->
    <profile>
        <id>dev</id>
        <properties>
                <!-- 标识当前环境 -->
            <package.environment>dev</package.environment>
        </properties>
        <activation>
            <!-- 默认激活配置 -->
            <activeByDefault>true</activeByDefault>
        </activation>
    </profile>
    <!-- 测试环境 -->
    <profile>
        <id>test</id>
        <properties>
            <package.environment>test</package.environment>
        </properties>
    </profile>
    <!-- 生产环境 -->
    <profile>
        <id>prod</id>
        <properties>
            <package.environment>prod</package.environment>
        </properties>
    </profile>
</profiles>
```

配置 Spring Boot Maven 插件，使其能够按照指定的方式重新打包项目，将生成的 JAR 文件存放在 ci 目录下并设置适当的文件名，代码如下：

```
//第 5 章/library/library - admin/pom.xml
<build>
    <finalName>library - admin - pro</finalName>
    <plugins>
        <!-- Spring Boot Maven 插件配置 -->
        <plugin>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring - boot - maven - plugin</artifactId>
              <configuration>
                 <!-- 构建生成的最终文件名,例如,本项目的文件名是 library - admin - pro -->
<finalName>${project.build.finalName} - ${project.version}</finalName>
              </configuration>
              <executions>
                  <execution>
                      <id>repackage</id>
                      <goals>
                         <goal>repackage</goal>
                      </goals>
                      <!-- 配置输出目录 -->
                      <configuration>
                         <!-- 打包可运行的 JAR 并存放至 ci 文件下 -->
                         <outputDirectory>ci</outputDirectory>
                         <executable>true</executable>
                      </configuration>
                  </execution>
               </executions>
          </plugin>
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven - surefire - plugin</artifactId>
              <configuration>
                  <skipTests>true</skipTests>
              </configuration>
          </plugin>
     </plugins>
 </build>
```

到这里项目的基础配置文件已经完成,现在启动项目,如果项目启动成功,则表明配置没有问题,可以继续进行后续的开发。

5.2　整合项目日志

为了实现在项目异常报错、出现问题时可以迅速定位及更精准地监控项目的运行状态,可以采用日志文件记录监控异常的错误信息。使用 Log4j2 作为日志框架,旨在项目启动时即刻开始记录关键信息,有助于团队形成记录日志的习惯,提升问题追踪和排查的效率。

5.2.1　日志级别

在项目中常用的日志级别有 debug、info、warn、error、trace、fatal、off 等级别,每个级别

的使用方式和概念如下。

(1) debug 主要用于开发过程中记录关键逻辑的运行时数据,以支持调试和开发活动。

(2) info 用于记录关键信息,可用于问题排查,包括出入参、缓存加载成功等,有助于有效地监控应用状态。

(3) warn 用于警告信息,指示一般性错误,对业务影响较小,但仍需减少此类警告以保持稳定性。

(4) error 用于记录错误信息,对业务会产生影响。需配置日志监控以追踪异常情况。

(5) trace 提供最详细的日志信息,可用于深入分析和跟踪应用的内部流程。

(6) fatal 表示严重错误,通常与应用无法继续执行相关,用于标识致命问题。

(7) off 关闭日志输出,用于特定情况下暂停日志记录。

合理地选择不同级别的日志记录,可实现对应用状态的全面监控、异常排查和问题追踪,为开发团队提供高效的日常运维支持。

5.2.2 日志使用技巧和建议

在编写代码的过程中,日志记录是不可或缺的,而在项目中日志记录的技巧则显得尤为重要,以下是一些日志使用的技巧和建议。

(1) 在多个 if-else 条件判断时,每个分支首行尽量打印日志信息。

(2) 为了提高日志的可读性,建议使用参数占位符{},避免使用字符串拼接“+”。

(3) 在使用异常处理块 try-catch 时,避免直接调用 e.printStackTrace()和 System.out.println()输出日志,正确写法的代码如下:

```
try {
    //TODO 处理业务代码
} catch (Exception e) {
    log.error("处理业务 id:{}", id, e);
}
```

(4) 异常的日志应完整地输出错误信息,代码如下:

```
log.error("业务处理出错 id: {}", id, e);
```

(5) 建议进行日志文件分离。针对不同的日志级别,将日志输到不同的文件中,例如,使用 debug.log、info.log、warn.log、error.log 等文件进行分类记录。

(6) 处理错误时,避免在捕获异常后再次抛出新异常,以防重复记录和混淆错误追踪。

5.2.3 添加日志依赖

Log4j2 是一个基于 Java 的日志框架,提供了一种灵活高效的方式来记录应用程序中的日志消息。它是原始 Log4j 库的进化版本,旨在更加稳健、高性能和功能丰富。Log4j2 提供了各种日志功能,包括将日志记录到多个输出、配置日志级别、定义自定义日志格式等,可以更好地控制和管理应用程序的日志记录。

1. 添加 Log4j2 依赖

在父模块的 pom.xml 文件中添加 Log4j2 依赖，先在 properties 中声明依赖的版本号，代码如下：

```
<!-- 定义依赖版本号 -->
<log4j2.version>2.20.0</log4j2.version>
<disruptor.version>3.4.2</disruptor.version>
```

然后添加日志和 Log4j2 相关依赖，代码如下：

```
//第5章/library/pom.xml
<!-- 引入 Log4j2 依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
<!-- 日志 -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>${log4j2.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>${log4j2.version}</version>
</dependency>
<dependency>
    <groupId>com.lmax</groupId>
    <artifactId>disruptor</artifactId>
<version>${disruptor.version}</version>
</dependency>
```

添加完成后，需要将 Spring Boot 自带的 LogBack 去掉，在父模块的 pom.xml 文件中修改 spring-boot-starter 依赖，使用 Excelusions 排除 log，代码如下：

```
//第5章/library/pom.xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <Excelusions>
        <Excelusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </Excelusion>
        <Excelusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-autoconfigure</artifactId>
        </Excelusion>
    </Excelusions>
</dependency>
```

2. 添加 Log4j2 配置文件

在子模块 library-admin 的 resource 资源目录中，新建一个名为 log4j2.xml 的配置文件，用于定义日志框架的配置参数、相关信息及存储路径等，代码如下：

```
//第 5 章/library/library-admin/src/main/resources/log4j2.xml
<Properties>
        <!-- 日志字符集为 UTF-8 -->
    <property name="log_charset">UTF-8</property>
<!-- 日志文件的基本名称 -->
    <Property name="filename">library</Property>
<!-- 日志文件存储路径 -->
    <Property name="log_path">/library/logs</Property>
<!-- 日志文件的编码格式为 UTF-8 -->
    <Property name="library_log_encoding">UTF-8</Property>
<!-- 单个日志文件的最大大小为 200MB -->
    <Property name="library_log_size">200MB</Property>
<!-- 日志记录的最低级别为 INFO -->
    <property name="data_level">INFO</property>
<!-- 日志文件的最长保留时间为 5 天，超过这段时间的日志文件将被自动删除 -->
    <Property name="library_log_time">5d</Property>
<!-- 日志输出的格式模式 -->
    <property name="log_pattern" value="%-d{yyyy-MM-dd HH:mm:ss} %-5r [%t] [%-
5p] %c %x - %m%n" />
</Properties>
```

3. 配置日志输出器

在日志配置文件中有一个 Appenders 输出器，共要配置两个输出，一个 Console 在控制台输出；另一个 RollingRandomAccessFile 设置为文件格式的输出，代码如下：

```
//第 5 章/library/library-admin/src/main/resources/log4j2.xml
<Appenders>
    <Console name="Console" target="SYSTEM_OUT">
       <!-- 输出日志的格式 -->
                <PatternLayout pattern="${log_pattern}"/>
                <!-- 控制台只输出 level 及其以上级别的信息(onMatch),其他的信息直接拒
                绝(onMismatch) -->
        <ThresholdFilter level="info" onMatch="ACCEPT" onMismatch="DENY" />
    </Console>

    <RollingRandomAccessFile name="LIBRARY_FILE"
                             fileName="${log_path}/${filename}.log" filePattern=
"${log_path}/${filename}_%d{yyyy-MM-dd}_%i.log.gz">
        <PatternLayout pattern="[%style{%d{yyyy-MM-dd HH:mm:ss.SSS}}{bright,green}]
[%-5p][%t][%c{1}] %m%n"/>
        <Policies>
            <SizeBasedTriggeringPolicy size="${library_log_size}"/>
        </Policies>
    </RollingRandomAccessFile>
```

```
    <RollingFile name="RollingFileError" fileName="${log_path}/error.log" filePattern=
"${log_path}/${filename}-ERROR-%d{yyyy-MM-dd}_%i.log.gz">
        <ThresholdFilter level="error" onMatch="ACCEPT" onMismatch="DENY"/>
        <PatternLayout pattern="${log_pattern}"/>
        <Policies>
        <!-- interval 属性用来指定多久滚动一次,默认为 1 hour -->
            <TimeBasedTriggeringPolicy interval="1"/>
            <SizeBasedTriggeringPolicy size="10MB"/>
        </Policies>
        <!-- 如不设置,则默认为最多同一文件夹下 7 个文件开始覆盖 -->
        <DefaultRolloverStrategy max="15"/>
    </RollingFile>
</Appenders>
```

4. 配置日志记录

根据不同的业务功能和应用程序,其日志记录也要使用不同的输出源,可使用 Loggers 来配置日志的输出,代码如下:

```
//第 5 章/library/library-admin/src/main/resources/log4j2.xml
<Loggers>
        <!—监控系统信息,如果 additivity 为 false,则子 Logger 只会在自己的 appender 里输出,
而不会在父 Logger 的 appender 里输出 -->
   <AsyncLogger name="org.springframework" level="info" additivity="false">
        <AppenderRef ref="Console"/>
   </AsyncLogger>
<!-- 过滤 Spring 和 MyBatis 的一些无用的 Debug 信息 -->
        <AsyncLogger name="org.mybatis" level="info" additivity="false">
        <AppenderRef ref="Console"/>
    </AsyncLogger>
    <AsyncLogger additivity="false" name="com.library.admin" level="INFO">
        <AppenderRef ref="Console" level="INFO"/>
        <AppenderRef ref="LIBRARY_FILE"/>
        <AppenderRef ref="RollingFileError"/>
    </AsyncLogger>
        <!-- 系统相关日志 -->
    <AsyncRoot level="info">
        <AppenderRef ref="Console"/>
        <AppenderRef ref="LIBRARY_FILE"/>
        <AppenderRef ref="RollingFileError"/>
    </AsyncRoot>
</Loggers>
```

5. 查看日志文件

在本节中,已经成功地将 Log4j2 日志框架整合到了项目中。现在启动项目并检查是否可以启动成功,如果项目能够正常启动,则查看项目根目录下的硬盘是否已生成了/library/logs 目录。在该目录下会有两个日志文件:error.log 和 library.log。打开 library.log 日志文件,就能够看到项目启动的相关日志已经被记录在其中,如图 5-7 所示。

```
[2023-09-08 10:59:18.538][INFO ][main][LibraryAdminApplication] Starting
LibraryAdminApplication using Java 17.0.8 with PID 21828 (
D:\xyh\library_publish\project\library\library-admin\target\classes started by admin
in D:\xyh\library_publish\project\library)
[2023-09-08 10:59:18.544][INFO ][main][LibraryAdminApplication] The following 1
profile is active: "dev"
[2023-09-08 10:59:19.169][INFO ][main][Http11NioProtocol] Initializing ProtocolHandler
["http-nio-8081"]
[2023-09-08 10:59:19.170][INFO ][main][StandardService] Starting service [Tomcat]
[2023-09-08 10:59:19.170][INFO ][main][StandardEngine] Starting Servlet engine: [
Apache Tomcat/10.1.12]
[2023-09-08 10:59:19.253][INFO ][main][[/api/library]] Initializing Spring embedded
WebApplicationContext
[2023-09-08 10:59:19.507][INFO ][main][Http11NioProtocol] Starting ProtocolHandler
["http-nio-8081"]
[2023-09-08 10:59:19.529][INFO ][main][LibraryAdminApplication] Started
LibraryAdminApplication in 1.353 seconds (process running for 2.288)
```

图 5-7　项目启动日志

5.3　Spring Boot 整合 MyBatis-Plus

MyBatis-Plus(简称 MP)是一个基于 MyBatis 的增强工具,能够帮助开发者简化开发过程、提升开发效率。对于单表的 CRUD 操作,MyBatis-Plus 提供了丰富便捷的 API,让开发者可以轻松地实现各种数据操作。此外,MyBatis-Plus 还支持多种查询方式和分页功能,并且不用编写烦琐的 XML 配置,从而大大降低了开发难度,让开发者能够更加专注于业务代码的编写。总而言之,MyBatis-Plus 的出现使 MyBatis 的使用变得更加简单和高效。

5.3.1　为什么选择 MyBatis-Plus

本项目使用 MyBatis-Plus 来简化持久层的操作,使开发更加高效、便捷、易于维护。项目使用 MyBatis-Plus 主要有以下优势。

(1) 强大的 CRUD 操作,内置通用 Mapper、通用 Service,仅仅通过少量配置即可实现单表大部分 CRUD 操作,更有强大的条件构造器,满足各类使用需求。

(2) MyBatis-Plus 是基于 MyBatis 的增强工具,与 MyBatis 完美兼容。如果已经使用过 MyBatis 开发项目,则可以直接将项目升级为 MyBatis-Plus,无须做太多改动,这样可以节省迁移成本。

(3) MyBatis-Plus 官方提供了详细清晰的文档,对于使用和配置都有详细的说明,方便开发者快速上手。此外,MyBatis-Plus 拥有庞大的社区支持,开发者可以通过社区获得帮助、分享经验和获取更多的资料。

(4) 支持 Lambda 形式调用,可以方便地编写各类查询条件,无须再担心字段写错等问题。

5.3.2　整合 MyBatis-Plus

MyBatis-Plus 3.0 以上版本要求 JDK 8 及以上版本,现在项目使用的是 JDK 17,基本

满足官方提出的要求，所以本项目使用 MP3.0＋的版本。

1. 导入依赖

在父模块的 pom.xml 文件中添加两个依赖，一个是 mybatis-plus-boot-starter 依赖，此依赖提供了 MyBatis-Plus 在 Spring Boot 中的自动配置功能，它会自动集成 MyBatis-Plus 和 Spring Boot，简化了配置过程；另一个是 mybatis-plus-extension 依赖，此依赖是 MyBatis-Plus 的扩展模块，提供了额外的功能和工具来增强 MyBatis-Plus 的能力。例如，分页插件、逻辑删除插件、自动填充插件、动态表名插件等提升开发效率和灵活性，代码如下：

```
//第 5 章/library/pom.xml
<!-- 版本号 -->
<mybatis-plus.version>3.5.3.2</mybatis-plus.version>

<!-- mybatis-plus -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>${mybatis-plus.version}</version>
</dependency>
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-extension</artifactId>
    <version>${mybatis-plus.version}</version>
</dependency>
```

2. 编写配置文件

在 library-admin 模块的 application.yml 配置文件中添加 mybatis-plus 的相关配置，代码如下：

```
//第 5 章/library/library-admin/src/main/resources/application.yml
#MyBatis-Plus 相关配置
mybatis-plus:
  global-config:
        #关闭自带的横幅广告的设置
    banner: false
    db-config:
        #主键生成策略为自动增长
      id-type: auto
        #开启数据库表名的下画线命名规则
      table-underline: true
#指定 Mapper XML 文件的路径，项目生成的 XML 文件全部放在 resource 目录下的 mapper 文件中
  mapper-locations: classpath:mapper/*.xml
  configuration:
    use-generated-keys: true
        #日志输出到控制台，数据库查询、删除等执行日志都会输出到控制台
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
    call-setters-on-nulls: true
```

本章小结

在本章中构建了两个子模块，并成功地将它们与父依赖的配置关联了起来。同时还集成了 Log4j2 作为项目的日志框架，用于记录项目运行中的执行日志信息，并深入学习了如何在项目中高效地应用日志记录等功能。此外，项目还顺利地整合了持久层的框架 MyBatis-Plus，以简化对数据层的操作。

总之，本章的内容涵盖了多个关键领域，包括项目构建、依赖管理、日志框架的集成及持久层的配置，这些步骤的顺利完成，为接下来的项目开发奠定了良好的基础。