项目配置

1. `application.yml`配置文件：

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydatabase?
useUnicode=true&characterEncoding=utf-8&useSSL=false
    username: admin
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
mybatis:
  mapper-locations: classpath:mapper/*.xml
  type-aliases-package: com.example.demo.entity
```

在`application.yml`文件中进行了数据库连接的配置，其中：

- `spring.datasource.url`：指定`MySQL`数据库的连接地址和相关参数，需要替换为实际的数据库名称。
- `spring.datasource.username`和`spring.datasource.password`：分别指定数据库访问的用户名和密码。
- `spring.datasource.driver-class-name`：指定`MySQL`数据库的驱动程序名。

此外，在上面的例子中还配置了`MyBatis`的mapper映射文件路径和实体类包路径。

2. `pom.xml`依赖包导入：

```
<dependencies>
    <!-- Spring Boot starters -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- MySQL Connector driver -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.26</version>
    </dependency>

    <!-- MyBatis dependencies -->
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>2.2.0</version>
    </dependency>
</dependencies>
```

在`pom.xml`文件中添加了`MySQL`连接驱动和`MyBatis`的`Spring Boot Starter`依赖。

- `mysql-connector-java`：`MySQL`的`JDBC`驱动程序。

- `mybatis-spring-boot-starter`：包含了`MyBatis`的核心依赖和与Spring Boot集成所需的依赖。

以上配置项中，`application.yml`文件中的配置内容用于指定数据库访问的相关信息，而`pom.xml`文件中的依赖包则用于实现与`MySQL`数据库和`MyBatis`框架的交互。两者之间没有直接的关系，但是它们共同实现了项目中数据层的功能。

项目结构设计及调用过程

`Spring Boot+Mybatis`项目结构设计的目的是为了将不同的业务逻辑分隔开，保持单一职责原则，并降低代码耦合度，提高代码可读性和维护性。具体分析如下：

1. /main/entity/*：用于存放实体类，与数据库表对应。在`Mybatis`中，通过配置文件（`/resources/mapping/xxxMapper.xml`）来映射实体类和数据库表之间的关系。
2. `/resources/mapping/xxxMapper.xml`：存储`Mybatis`映射器（Mapper），用于描述`SQL`语句和Java方法之间的关系。该文件中定义了`SQL`语句及其参数映射关系，以及返回结果集的映射方式。
3. `/main/mapper/xxxMapper.java`：存储Java接口，通过`Mybatis`的动态代理技术自动生成代理对象，实现接口调用时自动执行相应的`SQL`语句，并将结果转换为Java对象返回给调用方。
4. `/main/service/xxxService.java`：服务层，用于处理业务逻辑，调用Mapper层的方法实现数据操作，并将操作结果进行封装后返回给调用方。
5. `/main/controller/xxxController.java`：控制层，用于接收外部请求，调用服务层的方法完成业务逻辑处理，并将处理结果返回给前端。

执行过程

外部请求→→Controller层→→Service层→→Mapper层→→数据库操作

- 调用过程：
  - Controller层调用Service层
  - Service层调用Mapper层
  - Mapper层与数据库交互完成数据操作。

示例展示

假设有一个学生信息管理系统，需要实现查询、新增、修改和删除学生信息的功能。其中，学生信息对应的表名为student，包含id、name、age三个字段。

1. 创建实体类`Student.java`：

```
public class Student {
    private int id;
    private String name;
    private int age;

    //getter和setter
}
```

2. 在/resources/mapping/目录下创建`StudentMapper.xml`，定义SQL语句：

```
<mapper namespace="com.example.mapper.StudentMapper">
    <select id="getStudentById" parameterType="int"
resultType="com.example.entity.Student">
```

```
        SELECT * FROM student WHERE id=#{id}
    </select>

    <insert id="addStudent" parameterType="com.example.entity.Student">
        INSERT INTO student(name, age) VALUES(#{name}, #{age})
    </insert>

    <update id="updateStudent" parameterType="com.example.entity.Student">
        UPDATE student SET name=#{name}, age=#{age} WHERE id=#{id}
    </update>

    <delete id="deleteStudentById" parameterType="int">
        DELETE FROM student WHERE id=#{id}
    </delete>
</mapper>
```

3. 在/main/mapper/目录下创建`StudentMapper.java`，定义接口方法：

```
@Mapper
@Repository
public interface StudentMapper {
    Student getStudentById(int id);

    void addStudent(Student student);

    void updateStudent(Student student);

    void deleteStudentById(int id);
}
```

4. 在/main/service/目录下创建`StudentService.java`，实现业务逻辑：

```
@Service
public class StudentService {
    @Autowired
    private StudentMapper studentMapper;

    public Student getStudentById(int id) {
        return studentMapper.getStudentById(id);
    }

    public void addStudent(Student student) {
        studentMapper.addStudent(student);
    }

    public void updateStudent(Student student) {
        studentMapper.updateStudent(student);
    }

    public void deleteStudentById(int id) {
        studentMapper.deleteStudentById(id);
    }
}
```

5. 在/main/controller/目录下创建`StudentController.java`，实现接口方法：

```
@RestController
@RequestMapping("/student")
public class StudentController {
    @Autowired
    private StudentService studentService;

    @GetMapping("/{id}")
    public ResponseEntity<Student> getStudentById(@PathVariable int id) {
        Student student = studentService.getStudentById(id);
        if (student == null) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        } else {
            return new ResponseEntity<>(student, HttpStatus.OK);
        }
    }

    @PostMapping("")
    public ResponseEntity<Void> addStudent(@RequestBody Student student) {
        studentService.addStudent(student);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Void> updateStudent(@PathVariable int id, @RequestBody
Student student) {
        student.setId(id);
        studentService.updateStudent(student);
        return new ResponseEntity<>(HttpStatus.OK);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteStudentById(@PathVariable int id) {
        studentService.deleteStudentById(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
}
```

分析过程和结果

通过上述代码示例可以看出，在该项目结构中，各层之间的职责清晰、相互独立，同时又能够很好地协同工作。Controller层接收请求并验证参数格式，Service层进行业务逻辑处理，Mapper层完成数据操作，将结果返回给Service层，Service层将结果封装后返回给Controller层，最终Controller层将结果响应给前端。

这种项目结构设计具有良好的扩展性和可维护性，便于团队协作开发。同时，由于Spring Boot框架提供了自动化配置功能，可以大幅减少开发者在配置方面的工作量，提高开发效率，加快项目迭代速度。