第 3 章

Spring Data JPA

Spring Data 是 Spring 的一个子项目，致力于简化数据库访问，支持 NoSQL（非关系型数据库）和关系型数据库存储方式。Spring Data 项目所支持的 NoSQL 存储技术包括 MongoDB（文档数据库）、Neo4j（图形数据库）、Redis（键/值存储）、HBase（列族数据库）。Spring Data 项目所支持的关系数据存储技术包括 JDBC 和 JPA。Spring Data JPA 遵从 JPA 规范，是依靠 Hibernate 实现的一种 ORM（对象关系模型）框架。

3.1 Spring Data JDBC 技术

Spring Data JDBC 提供了 JDBCTemplates 类，它是 Spring 对 JDBC 的封装，由 Spring 来完成 JDBC 底层的工作，并提供大量的 API 供用户直接使用，实现对数据库的增、删、改、查操作。Spring Boot 通过其自动配置机制，自动配置了 JDBCTemplate，只需在 pom.xml 文件中导入 spring-boot-starter-jdbc 依赖和数据库连接 Java 的依赖，并在 application.properties 文件中设置好数据库连接信息，Spring Boot 就会自动向容器注入 JDBCTemplate 实例，这样开发者就可使用@Autowired 注解依赖注入 JDBCTemplate，再调用 JDBCTemplate 的 API 方法对数据库进行增、删、改、查操作。由于 JDBCTemplate 并非主流，这里只提供一个简单的增、删、改、查的案例供参考。

【例 3-1】 使用 Spring Boot 整合 JDBCTemplate 操作 studentdb 中的 student 表。

（1）数据库准备。在 MySQL 数据库 studentdb 中创建表 student，插入若干数据，见图 3-1。

	id	studentname	gender	age
□	1	王维	男	18
□	2	李白	男	20
□	3	杜甫	男	19
□	4	李清照	女	17

图 3-1 创建表 student

（2）创建项目导入依赖。在 IDEA 新建 Spring Boot 项目 JDBCtemplate1，导入 spring-boot-starter-jdbc 依赖、mysql-connector-java 依赖等，pom.xml 文件的关键代码如下：

```
//第 3 章/jdbc1/pom.xml
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.25</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
```

(3) 配置数据库连接信息。在 application.properties 文件中配置数据库连接信息,代码如下:

```
//第 3 章/jdbc1/application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/studentdb?useUnicode=
true&characterEncoding=UTF-8&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

(4) 创建实体类。在 com.seehope.entity 包下创建实体类 Student,关键代码如下:

```
//第 3 章/jdbc1/Student.java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Student {
   private int id;
   private String studentname;
   private String gender;
   private int age;
}
```

(5) 创建数据访问层。在 com.seehope.dao 包下创建 StudentRepository 类,用@Respository 进行注解,表示是数据访问层,在类中创建 JDBCTemplate 类型的属性 JDBCTemplate,并使用@Autowired 注解,表示从容器中注入 JDBCTemplate 实例,在类中创建有关添加、删除、修改、查询学生的若干方法,这些方法在内部分别调用 JDBCTemplate

提供的相应方法进行增、删、改、查操作，但需要提供相应的 SQL 语句作为参数，关键代码如下：

```
//第 3 章/jdbc1/StudentRepository.java
@Repository
public class StudentRepository {
   @Autowired
private JDBCTemplate JDBCTemplate;

   //添加学生
   public int addStudent(Student student) {
      return JDBCTemplate.update("insert into
student(studentname,gender,age) values(?,?,?)",
            student.getStudentname(), student.getGender(), student.getAge());
}
   //删除学生
   public int deleteStudent(int id) {
      return JDBCTemplate.update("delete from student where id = ?", id);
}
   //修改学生
   public int updateStudent(Student student) {
      return JDBCTemplate.update("update student set studentname = ?,gender = ?,age = ? where id = ?",
            student.getStudentname(), student.getGender(), student.getAge(), student.getId());
   }
   //查询所有学生
   public List < Student > findAllStudents() {
       return JDBCTemplate.query("select * from student", new BeanPropertyRowMapper < >
(Student.class));
}
   //根据 id 号查询单个学生
   public Student findStudentById(int id) {
      return JDBCTemplate.queryForObject("select * from student where id = ?",
            new BeanPropertyRowMapper <>(Student.class), id);
   }
}
```

在上述代码中 JDBCTemplate 的 update 方法用于执行增、删、改 SQL 语句，SQL 语句中的“?”号表示占位符，有几个占位符 update 方法就需提供几个实参。query 方法用于执行查询语句，返回 List < T >泛型集合，其中 BeanPropertyRowMapper 类型的参数用于将查询结果封装为指定的对象类型，即指定泛型 T 的实际类型。queryForObject 方法用于查询单一返回值的结果，同样封装为对象类型。

（6）创建业务逻辑层。在 com.seehope.service 包下创建 StudentService 类，关键代码如下：

```
//第 3 章/jdbc1/StudentService.java
@Service
public class StudentService {
   @Autowired
   private StudentRepository studentRepository;
```

```
    public int addStudent(Student student) {
        return studentRepository.addStudent(student);
    }

    public int deleteStudent(int id) {
        return studentRepository.deleteStudent(id);
    }

    public int updateStudent(Student student) {
        return studentRepository.updateStudent(student);
    }

    public List<Student> findAllStudents() {
        return studentRepository.findAllStudents();
    }

    public Student findStudentById(int id) {
        return studentRepository.findStudentById(id);
    }
}
```

（7）创建控制器类 StudentController，代码如下：

```
//第 3 章/jdbc1/StudentController.java
@Controller
public class StudentController {
    @Autowired
    private StudentService studentService;
    @GetMapping("/stus")
    public ModelAndView findAllStudents(){
        List<Student> stus =  studentService.findAllStudents();
        ModelAndView mv = new ModelAndView();
        mv.addObject("stus",stus);
        mv.setViewName("stus");
        return mv;
    }
    @GetMapping("/stu/{id}")
    public ModelAndView findStudentById(@PathVariable("id") int id){
        Student student = studentService.findStudentById(id);
        ModelAndView mv = new ModelAndView();
        mv.addObject("student",student);
        mv.setViewName("student");
        return mv;
    }
    @PostMapping("/addStudent")
    public ModelAndView addStudent(Student student){
        studentService.addStudent(student);
        ModelAndView mv = new ModelAndView();
    //添加成功后,跳转到查找所有学生的控制器
        mv.setViewName("redirect:/stus");
        return mv;
```

```
    }
    @GetMapping("/addStudent")
    public String addStudent(){
        return "addStudent";
    }
    @GetMapping("/deleteStudent/{id}")
    public ModelAndView deleteStudent(@PathVariable("id") int id){
        studentService.deleteStudent(id);
        ModelAndView mv = new ModelAndView();
    //删除成功后,跳转到查找所有学生的控制器
        mv.setViewName("redirect:/stus");
        return mv;
    }
    @GetMapping("/updateStudent/{id}")
    public ModelAndView toUpdateStudent(@PathVariable("id") int id){
        Student student = studentService.findStudentById(id);
        ModelAndView mv = new ModelAndView();
        mv.addObject("student",student);
        mv.setViewName("updateStudent");
        return mv;
    }
    @PostMapping("/updateStudent")
    public ModelAndView UpdateStudent(Student student){
        studentService.updateStudent(student);
        System.out.println(student);
        ModelAndView mv = new ModelAndView();
    //修改成功后,跳转到查找所有学生的控制器
        mv.setViewName("redirect:/stus");
        return mv;
    }
}
```

(8) 创建视图。创建 stus.html、stu.html、addStudent.html、updateStudent.html 页面(详见教材配套电子资源)。

(9) 运行代码进行测试。这里仅提供部分测试结果,更多的结果可自行测试。

在浏览器网址栏输入 http://localhost:8080/stus,结果如图 3-2 所示。

在浏览器网址栏输入 http://localhost:8080/stu/1,结果如图 3-3 所示。

图 3-2 查找所有学生

图 3-3 查找一个学生

单击图 3-2 中的添加学生超链接，并填写数据，如图 3-4 所示。单击“添加”按钮，结果如图 3-5 所示。

图 3-4 添加学生

图 3-5 添加后的结果

3.2 Spring Data JPA 技术

Spring Data JPA 是基于 Hibernate 的一款持久层框架，它通过注解或 XML 文件描述 Java 对象与数据库表之间的映射关系，将内存中的对象持久化到数据库中。Spring Data JPA 提供了多种基于 JPA 规范的接口，内置了大量的常用查询方法，由 Spring 生成接口的代理类，注入 Spring 容器中进行管理，开发者在大多数情况下无须自行编写 SQL（或 JPQL）即可实现对数据库的访问与操作。项目中在使用 Spring Data JPA 时需引入下述依赖，代码如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Spring Data JPA 提供的接口有 Repository、JpaRepository、PagingAndSortingRepository、CrudRepository 及 QueryByExampleExecutor，其中 CrudRepository 接口定义了基本的增、删、改、查 CRUD 操作，PagingAndSortingRepository 定义了分页和排序的查询操作，QueryByExampleExecutor 接口定义了简单的动态查询操作。

JpaRepository 接口继承了 PagingAndSortingRepository、CrudRepository 和 QueryByExampleExecutor 接口，继承了它们的所有操作并提供了其他一些常用操作，PagingAndSortingRepository 和 CrudRepository 接口又继承自 Repository，Repository 只是一个空接口，具有标记作用，各个接口之间的关系如图 3-6 所示。

所以开发者只要自定义一个接口继承 JpaRepository 接口即可获得上述所有接口的功能。开发者自定义的接口还可以进行方法命名查询，方法名称和参数只要符合 JPA 的命名规范，Spring Data JPA 就会根据方法名字来确定需要实现什么样的逻辑，无须开发者编写 SQL 语句。自定义的接口中的方法也可配套使用 JPQL 语言或原生 SQL 语句进行更加灵活的查询，此外 Spring Data JPA 还可方便地处理一对一，一对多，以及多对多表关联查询。

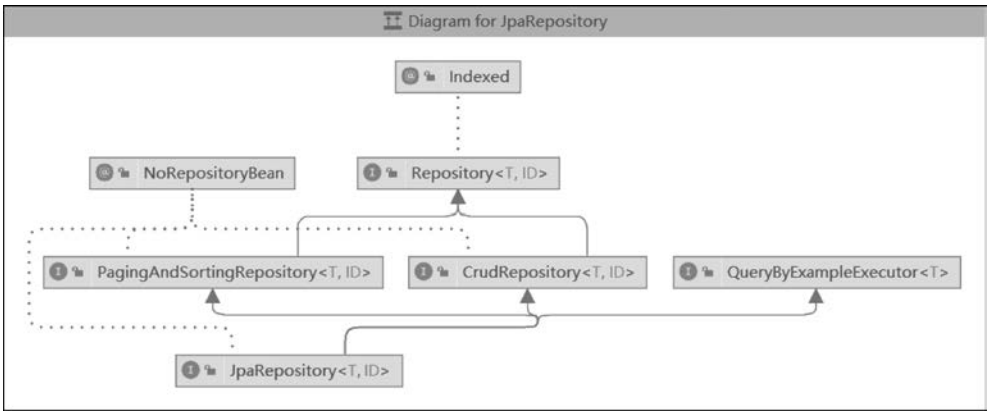

图 3-6　接口之间的关系

3.2.1　JpaRepository 接口

JpaRepository 接口除继承了父接口的方法外还提供了其他方法，见表 3-1。

表 3-1　JpaRepository 接口方法

方 法 名 称	功 能 描 述
List＜T＞ findAll()	查询所有对象数据
List＜T＞ findAll(Sort var1)	查询所有对象数据并进行排序
List＜T＞ findAllById(Iterable＜ID＞ var1)	根据提供的多个 ID 号，将有关对象都查询出来
＜S extends T＞ List＜S＞ saveAll(Iterable＜S＞ var1)	将集合数据保存到数据库
void flush()	将缓存中的对象更新到数据库
＜S extends T＞ S saveAndFlush(S var1)	保存对象并更新到数据库
void deleteInBatch(Iterable＜T＞ var1)	批量删除集合中的对象
void deleteAllInBatch()	批量删除所有对象
T getOne(ID var1)	根据 ID 号查询对象
＜S extends T＞ List＜S＞ findAll(Example＜S＞ var1)	根据提供的 Example 实例查询对象
＜S extends T＞ List＜S＞ findAll(Example＜S＞ var1, Sort var2)	根据提供的 Example 实例查询对象并排序

3.2.2　PagingAndSortingRepository 接口

PagingAndSortingRepository 接口主要提供了用于分页查询和排序的方法，见表 3-2。

表 3-2　PagingAndSortingRepository 接口方法

方 法 名 称	功 能 描 述
Iterable＜T＞ findAll(Sort var1)	查询所有对象并根据 Sort 对象中的规则进行排序
Page＜T＞ findAll(Pageable var1)	查询所有对象并根据 Pageable 对象中的规则进行分页处理，封装为一个 Page 对象。注意：Pageable 中又可包含排序

Page 是 Spring Data 提供的一个封装了分页信息的接口，Pageable 是构造分页查询的接口，用于指定页码(从 0 算起)，每页显示记录数，以及是否排序等，具体由 PageRequest 类的 of 静态方法进行实现。

【例 3-2】 假设用户在项目中已经创建好了继承 JpaRepository 的 Dao 层接口 StudentRepository，在控制器 StudentController 中使用 Page < T > findAll(Pageable var1) 方法进行分页与排序查询，代码如下：

```
//第 3 章/jpa1/StudentController.java
@Autowired
Private StudentRepository studentRepository;
@GetMapping("/findAllPage")
    public ModelAndView findAll(@RequestParam(value = "start",defaultValue = "0") Integer
start,
    @RequestParam(value = "size",defaultValue = "3") Integer size) {
        ModelAndView mv = new ModelAndView();
          //构造 Pageable,默认查询第 1 页,每页显示 3 条记录,根据属性 id 进行降序排序
          Pageable pageable =  PageRequest.of(start,size,Sort.by(Sort.Direction.DESC,"id"));
            //也可以不排序
          //Pageable pageable =  PageRequest.of(start,size);
            //根据 pageable 进行查询,结果封装为 Page 对象
          Page < Student > page = studentRepository.findAll(pageable);
          mv.addObject("page",page);
          mv.setViewName("stusPage");
          return mv;
      }
```

PageRequest.of 方法的第 3 个参数用到了 Sort 类，用于制定排序规则。可以使用 Sort 类的带参构造方法创建 Sort 对象，第 1 个参数用于指定升序还是降序，值为 Sort.Direction.DESC 或 Sort.Direction.ASC，第 2 个参数用于指定排序属性(字段)。

【例 3-3】 Sort sort=new Sort(Sort.Direction.DESC, "id")；表示创建一个根据属性 id 进行降序排序的 Sort 对象。也可以使用 Sort 类的静态方法 by 创建 Sort 对象，语法可参照上述示例代码。

3.2.3 CrudRepository 接口

CrudRepository 接口提供了常用的增、删、改、查方法，见表 3-3。

表 3-3 CrudRepository 接口方法

方法名称	功能描述
< S extends T > S save(S var1)	保存对象，如果对象包括主键 ID，则进行更新操作
< S extends T > Iterable < S > saveAll(Iterable < S > var1)	保存对象集合
Optional < T > findById(ID var1)	根据主键 ID 查询对象，结果可以为空
boolean existsById(ID var1)	根据主键 ID 查询对象是否存在
Iterable < T > findAll()	查询所有对象

续表

方法名称	功能描述
Iterable<T> findAllById(Iterable<ID> var1)	根据给定的多个 ID 号查询对象集合
long count()	查询对象的数量
void deleteById(ID var1)	根据 ID 号删除对象，没有返回值，如果要判断是否成功，则要结合 try…catch…语句
void delete(T var1)	删除对象
void deleteAll(Iterable<? extends T> var1)	删除给定的对象
void deleteAll()	删除所有对象

用户自定义的接口只要实现 JpaRepository 接口，调用上述接口提供的方法，无须编写 SQL 或 JPQL 语句就可操作数据库。

3.2.4 基本增、删、改、查方法

【例 3-4】 Spring Data JPA 的基本增、删、改、查及分页操作。

(1) 数据库准备。在 studentdb 数据库中创建表 course，主键 courseid 自增长，添加若干数据，如图 3-7 所示。也可以不创建表，由程序自动创建，见第(3)步中有关 update 的说明。

	courseid	coursename	coursescore	coursetime	coursetype
□	1	Java程序设计基础	3	64	必修
□	2	Python程序设计基础	4	64	必修
□	3	C程序设计基础	4	64	必修
□	4	大学英语	3	60	必修
□	5	大学物理	3	50	选修
□	6	大学化学	3	60	选修
□	7	大学语文	2	30	选修
□	8	计算机基础	2	40	选修
□	9	MySQL数据库	4	40	必修

图 3-7 数据库表 course

(2) 创建 Spring Boot 项目。在 IDEA 中创建 Spring Boot 项目，命名为 pom.xml 并在 pom.xml 文件中导入 JPA 相关依赖，关键代码如下：

```
//第 3 章/jpa1/pom.xml
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
```

```
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
```

(3) 配置 application.properties。需要配置数据库连接信息和 JPA 属性,配置如下:

```
//第3章/jpa1/application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/studentdb?useUnicode=true&characterEncoding=
utf8&serverTimezone=UTC&useSSL=true
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.properties.hibernate.hbm2ddl.auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.show-sql=true
```

其中 spring.jpa.properties.hibernate.hbm2ddl.auto 属性用于设置自动创建、更新或验证数据库表结构。一般取以下 4 个值之一。

create:每次加载 Hibernate 时都会删除数据库表,然后根据 Model 类重新创建新表,这样会导致数据库数据丢失,应谨慎使用。

create-drop:每次加载 Hibernate 时都会根据 Model 类创建新表,一旦 SessionFactory 关闭就自动删除表。

update:首次加载 Hibernate 时若没有 Model 类对应的数据库表就会根据 Model 类创建数据表,以后加载时,不会创建新表,若 Model 类被修改过,则会更新数据库表结构,但数据不会被删除。

validate:每次加载 Hibernate 时,会验证数据库的表结构而不会创建新表,但会插入新值。

spring.jpa.properties.hibernate.dialect:对特定的关系数据库生成优化的 SQL。

spring.jpa.show-sql=true 属性用于设置控制台输出 SQL 语句。

(4) 创建实体类 Course,关联数据库表 course,代码如下:

```
//第3章/jpa1/Course.java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity(name="course")
public class Course {
    //课程编号
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private int courseid;
     //课程名称
    @Column(name = "coursename",nullable = false,length = 30)
    private String coursename;
     //课程学分
    @Column(name = "coursescore")
    private int coursescore;
     //课时
    @Column(name = "coursetime")
    private int coursetime;
     //选修或必修课
    @Column(name = "coursetype")
    private String coursetype;
     //备注,数据库中无对应的列
    @Transient
    private String description;
}
```

@Entity 注解表示该实体类是一个与数据库表映射的实体类，用 name 属性指定该实体类映射的数据库表名，也可以省略 name 属性，默认数据库表名与类名一致。

@Id 注解表示该属性映射为数据库表的主键，并用@GeneratedValue 注解来指定主键生成策略，若设置为 strategy = GenerationType.IDENTITY，则表示主键由数据库自动增长，若设置为 strategy = GenerationType.AUTO，则表示主键由程序生成。

@Column 注解用来描述该属性对应的数据库表中的字段的详细定义，name 属性表示所映射的数据库表中的字段，默认字段名与属性名一致，nullable 属性表示该字段是否允许为空，length 属性表示该字段的长度，仅对 String 类型有效，还可以有 unique 属性，表示该字段是否是唯一标识。

@Transient 注解用于标注数据库表中无对应列的属性，ORM 框架将忽略该属性。

(5) 创建数据访问层。创建 CourseDao 接口，继承 JpaRepository 接口，关键代码如下：

```
//第 3 章/jpa1/CourseDao.java
public interface CourseDao extends JpaRepository<Course,Integer>{
}
```

这里暂时不用写任何自定义方法(父类接口已经有很多直接可用的方法)。

(6) 创建业务逻辑层。创建 CourseService 类，关键代码如下：

```
//第 3 章/jpa1/CourseService.java
@Service
public class CourseService {

    @Autowired
    private CourseDao courseDao;

     //使用 JpaRepository 的 API 查询,DAO 层无须手动编写 SQL 或 JPQL 语句
     //查找所有课程,不分页,相当于 SQL: select * from course
    public List<Course> findAll(){
```

```
        return courseDao.findAll();
    }

     //查找所有课程,有排序,相当于 SQL: select * from course order by xxx DESC|ASC
    public List<Course> findAll(Sort sort){
        return courseDao.findAll(sort);
    }

     //查找所有课程,有分页,相当于 SQL: select * from course limit x,y
    public Page<Course> findAll(Pageable pageable){
        return courseDao.findAll(pageable);
    }

     //查询某个编号的课程,相当于 SQL: select * from course where courseid = ?1
     //也可以用方法名 findByIdIs findByIdEquals
    public Course findById(int courseid){
        return courseDao.findById(courseid).orElse(new Course());
    }

    public void updateCourse(Course course) {
    //若参数的主键存在,则进行更新操作
        courseDao.save(course);
    }

    public void addCourse(Course course) {
    //添加课程,若参数的主键不存在,则进行添加操作
        courseDao.save(course);
    }

    public void deleteCourse(int courseid) {
    //删除课程
        courseDao.deleteById(courseid);
    }
}
```

(7) 创建控制器。创建 CourseController,关键代码如下:

```
//第 3 章/jpa1/CourseController.java
@Controller
public class CourseController {

    @Autowired
    private CourseService courseService;
     //获取旧数据并传递到视图进行展示以便修改
@GetMapping("/updateCourse/{id}")
    public ModelAndView toupdateCourse(@PathVariable("id") int courseid){
        ModelAndView mv = new ModelAndView();
        Course course = courseService.findById(courseid);
        mv.addObject("course",course);
        mv.setViewName("updateCourse");
        return mv;
    }
```

```
    //获取新数据,修改数据库
    @PostMapping("/updateCourse")
    public ModelAndView updateCourse(Course course){
        ModelAndView mv = new ModelAndView();
        courseService.updateCourse(course);
    //更新完毕后转发到查询所有课程的控制器
        mv.setViewName("redirect:findAll1");
        return mv;
    }
    //将课程添加到数据库
    @PostMapping("/addCourse")
    public ModelAndView addCourse(Course course){
        ModelAndView mv = new ModelAndView();
        courseService.addCourse(course);
        mv.setViewName("redirect:findAll1");
        return mv;
    }
    //用于超链接,呈现添加课程的视图
    @GetMapping("/addCourse")
    public String addCourse(){
        return "addCourse";
    }
    //删除课程
@GetMapping("/deleteCourse/{id}")
    public ModelAndView deleteCourse(@PathVariable("id") int courseid){
        ModelAndView mv = new ModelAndView();
        courseService.deleteCourse(courseid);
        mv.setViewName("redirect:findAll1");
        return mv;
}
    //查询所有课程,无排序
    @GetMapping("/findAll1")
    public ModelAndView findAll(){
        ModelAndView mv = new ModelAndView();
        List<Course> courselist = courseService.findAll();
        mv.addObject("courses",courselist);
        mv.setViewName("courses");
        return mv;
}
    //查找所有课程,有排序
    @GetMapping("/findAll2")
    public ModelAndView findAll2(){
        ModelAndView mv = new ModelAndView();
    //按课程学分降序排序
        List < Course > courselist = courseService.findAll(Sort.by(Sort.Direction.DESC,
"coursescore"));
        mv.addObject("courses",courselist);
        mv.setViewName("courses");
        return mv;
}
    //查找所有课程,有分页与排序
    @GetMapping("/findAll3")
    public ModelAndView findAll(@RequestParam(value = "start",defaultValue = "0") Integer start,
```

```
    @RequestParam(value = "size",defaultValue = "3") Integer size) {
        ModelAndView mv = new ModelAndView();
    //默认查询第 1 页,1 页显示 3 条记录
        Pageable pageable = PageRequest.of(start, size, Sort.by(Sort.Direction.DESC,
"courseid"));
    //也可以不排序
        //Pageable pageable = PageRequest.of(start,size);
        Page<Course> page = courseService.findAll(pageable);
        mv.addObject("page",page);
        mv.setViewName("coursePage");
        return mv;
    }

    //查询某个编号的课程
    @GetMapping("/findById/{id}")
public ModelAndView findById(@PathVariable("id") int courseid){
        ModelAndView mv = new ModelAndView();
        Course course = courseService.findById(courseid);
        mv.addObject("course",course);
        mv.setViewName("course");
        return mv;
    }
}
```

(8) 创建视图。在 resource/templates 目录下创建 courses.html,用于以列表的形式展示所有课程,course.html 用于展示课程详情,addCourse.html 用于添加课程,coursePage.html 用于分页展示所有课程。具体代码参见随书配置电子资源,其中分页展示是重点,关键代码如下:

```
//第 3 章/jpa1/coursePage.html
<body>
<a th:href = "@{/addCourse}">添加学生</a><br/>
<table border = "1" align = "center">
    <tr>
        <td>课程编号</td>
        <td>课程名称</td>
        <td>课程学分</td>
        <td>课程学时</td>
        <td>课程类型</td>
        <td>修改</td>
        <td>删除</td>
    </tr>
<tr th:each = "course: ${page.getContent()}">
    <td th:text = "${course.courseid}"></td>
    <td th:text = "${course.coursename}"></td>
    <td th:text = "${course.coursescore}"></td>
    <td th:text = "${course.coursetime}"></td>
    <td th:text = "${course.coursetype}"></td>
<td><a th:href = "@{/updateCourse/} + ${course.courseid}">修改</a></td>
    <td><a th:href = "@{/deleteCourse/} + ${course.courseid}">删除</a></td>
</tr>
```

```
</table>
<div>
    <a th:if="${not page.isFirst()}" th:href="@{/findAll3(start=${page.number-1})}">上一页</a>
    |总页数:<span th:text="${page.getTotalPages()}"></span> 
    |当前页:<span th:text="${page.getNumber()+1}"></span>
    |总记录数:<span th:text="${page.getTotalElements()}"></span> \
    |当前页记录数:<span th:text="${page.getNumberOfElements()}"></span>
    <a th:if="${not page.isLast()}" th:href="@{/findAll3(start=${page.number+1})}">下一页</a>
</div>
</body>
```

(9) 运行代码进行测试。这里仅提供查询全部和分页查询的结果,其余结果可自行下载源码进行测试。在浏览器地址栏输入 http://localhost:8080/findAll1,结果如图 3-8 所示。

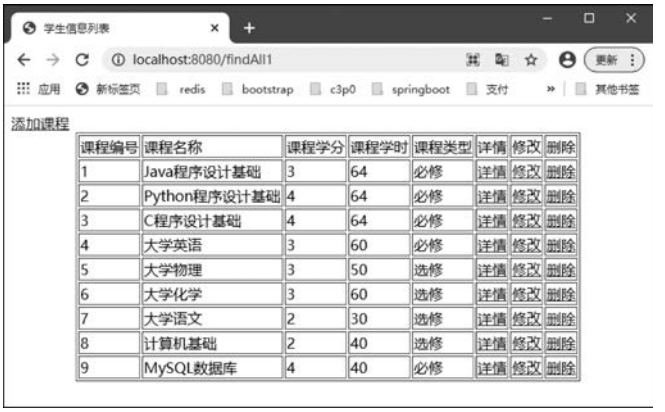

图 3-8　查询所有图书列表展示

在浏览器输入 http://localhost:8080/findAll3,结果如图 3-9 所示。

图 3-9　分页查询结果

3.2.5　方法命名查询

在用户自定义的继承了 JpaRepository 的接口中,用户可以自定义一些方法,只要这些

方法符合 JPA 的约定命名规范，Spring Data JPA 就能分析出开发者的意图，自动调用 SQL 实现特定的查询，而无须手动编写 SQL 语句。以自定义方法名称为 findByName(String username)为例，这里的 findBy 可以认为是关键字，而 Name 是可变的，可以是任意一个实体类的名称，框架会将该方法解释为查询条件 where name=?，如果方法的返回值类型为 List<User>，则完整的 SQL 语句为 select * from user where name=?。如果有多个条件，则方法名称中还可以出现 And 或 Or 进行连接。符合命名规范的方法名见表 3-4。

表 3-4 符合命名规范的方法名示例

关 键 词	方法命名示例	对应的 SQL
And	findByGenderAndAge	where gender=? And age=?
Or	findByGenderOrAge	where gender=? And age=?
=	findByName	where name=?
	findByNameIs	
	findByNameEquals	
Between	findByAgeBetween	where age between ? and ?
<	findByAgeLessThan	where age <?
<=	findByAgeLessThanEqual	where age <=?
>	findByAgeGreaterThan	where age >?
>=	findByAgeGreaterThanEqual	where age>=?
<>	findByAgeNot	where age <> ?
IsNull	findByNameIsNull	where name is null
NotNull	findByNameNotNull	where name is not null
In	findByAgeIn(Collection<Age> ages)	where age in (?)
NotIn	findByAgeNotIn(Collection<Age> ages)	where age not in(?)
like	findByNameLike	where name like ?
NotLike	findByNameNotLike	where name not like ?
StartingWith	findByNameStartingWith	where name like '?%'
EndingWith	findByNameEndingWith	where name like '%?'
ContainingWith	findByNameContainingWith	where name like '%?%'
OrderBy	findByGenderOrderByAgeDesc	Where gender=? Order by age desc
True	findByGenderTrue	where gender=true
False	findByGenderFalse	where gender=false

此外约定方法名还可用 Top 或 First 关键词来限制查询结果的数量，如 findTop10ByGender，指显示查询结果的前 10 条，相当于 SQL 语句：where gender=? limit 0,10。findFirst20ByAgeGreaterThan，指显示查询结果的前 20 条，相当于 SQL 语句：where age >? limit 0,20。

【例 3-5】 根据约定命名规范查询。

(1) 创建数据访问层。接着上述案例项目 jpa1，在 CourseDao 中创建方法，代码如下：

```
//第 3 章/jpa1/CourseDao.java
     //使用 JpaRepository 的约定命名规范查询,DAO 层无须手动编写 SQL 或 JPQL 语句
     //查询学分为 xx 的选修(或必修)课,按照命名规范,无须手动编写 SQL 语句
     //相当于 SQL: select * from course where //coursescore = ?1 and coursetype = ?2
     public List<Course> findByCoursescoreAndCoursetype(int coursescore,String
//coursetype);

     //查询学分为 xx 或者选修(或必修)的课程,按照命名规范,无须手动编写 SQL 语句
     //相当于 SQL: select * from course where coursescore = ?1 or coursetype = ?2
     public List<Course> findByCoursescoreOrCoursetype(int coursescore,String coursetype);

     //查询课时在 xx 到 xx 的课程
   public List<Course> findByCoursetimeBetween(int start,int end);

    //查询课时小于 xx 的课程
   public List<Course> findByCoursetimeLessThan(int coursetime);

    //查询课程名称包含 xx 字的课程
   public List<Course> findByCoursenameContains(String coursename);
```

(2) 创建业务逻辑层。在 CourseService 中添加方法,代码如下:

```
//第 3 章/jpa1/CourseService.java
     //查询学分为 xx 的选修(或必修)课
    public List<Course> findByCoursescoreAndCoursetype(int coursescore,String coursetype){
       return
courseDao.findByCoursescoreAndCoursetype(coursescore,coursetype);
    }

     //查询学分为 xx 或者选修(或必修)的课程
    public List<Course> findByCoursescoreOrCoursetype(int coursescore,String coursetype){
       return
courseDao.findByCoursescoreOrCoursetype(coursescore,coursetype);
    }

     //查询课时在 xx 到 xx 的课程
    public List<Course> findByCoursetimeBetween(int start,int end){
       return courseDao.findByCoursetimeBetween(start,end);
    }

     //查询课时小于 xx 的课程
    public List<Course> findByCoursetimeLessThan(int coursetime){
       return courseDao.findByCoursetimeLessThan(coursetime);
    }

     //查询课程名称包含 xx 字的课程
    public List<Course> findByCoursenameContains(String coursename){
       return courseDao.findByCoursenameContains(coursename);
    }
```

(3) 创建控制器。在 CourseController 中添加的代码如下:

```
//第 3 章/jpa1/CourseController.java
    //查询学分为 xx 的选修(或必修)课
@GetMapping("/findByCoursescoreAndCoursetype/{coursescore}/{coursetype}")
public ModelAndView findByCoursescoreAndCoursetype(@PathVariable("coursescore") int
coursescore,@PathVariable("coursetype") String coursetype){
    ModelAndView mv = new ModelAndView();
     List<Course> courses = courseService.findByCoursescoreAndCoursetype(coursescore,
coursetype);
    mv.addObject("courses",courses);
    mv.setViewName("courses");
    return mv;
}

    //查询学分为 xx 或者选修(或必修)的课程
@GetMapping("/findByCoursescoreOrCoursetype/{coursescore}/{coursetype}")
public ModelAndView findByCoursescoreOrCoursetype(@PathVariable("coursescore") int
coursescore,@PathVariable("coursetype") String coursetype){
    ModelAndView mv = new ModelAndView();
     List<Course> courses = courseService.findByCoursescoreOrCoursetype(coursescore,
coursetype);
    mv.addObject("courses",courses);
    mv.setViewName("courses");
    return mv;
}

    //查询课时在 xx 到 xx 的课程
@GetMapping("/findByCoursetimeBetween/{start}/{end}")
public ModelAndView findByCoursetimeBetween(@PathVariable("start") int start, @
PathVariable("end") int end){
    ModelAndView mv = new ModelAndView();
    List<Course> courses = courseService.findByCoursetimeBetween(start,end);
    mv.addObject("courses",courses);
    mv.setViewName("courses");
    return mv;
}

    //查询课时小于 xx 的课程
@GetMapping("/findByCoursetimeLessThan/{coursetime}")
public ModelAndView findByCoursetimeLessThan(@PathVariable("coursetime") int coursetime){
    ModelAndView mv = new ModelAndView();
    List<Course> courses = courseService.findByCoursetimeLessThan(coursetime);
    mv.addObject("courses",courses);
    mv.setViewName("courses");
    return mv;
}

    //查询课程名称包含 xx 字的课程
@GetMapping("/findByCoursenameContains/{coursename}")
public ModelAndView findByCoursenameContains(@PathVariable("coursename") String
coursename){
    ModelAndView mv = new ModelAndView();
    List<Course> courses = courseService.findByCoursenameContains(coursename);
```

```
        mv.addObject("courses",courses);
        mv.setViewName("courses");
        return mv;
    }
```

(4) 运行代码进行测试。在浏览器中输入：http://localhost:8080/findByCoursescoreAndCoursetype/3/选修,结果如图 3-10 所示。其他结果可自行进行测试。

图 3-10　查询结果

3.2.6　使用 JPQL 或原生 SQL 查询

除了可使用 JPA 接口或约定命名规范进行查询,还可自定义方法,使用 JPQL 或原生 SQL 进行更加灵活的查询。

JPQL 全称为 Java Persistence Query Language,是一种功能类似 SQL,但不是结构化而是对象化的查询语言,查询的是对象,而不是表,查询时使用的不是字段而是对象的属性,它是一种适用不同关系数据库的通用查询语言,但最终会被编译成不同底层数据库的 SQL 语句。要使用 JPQL,需要在 DAO 层接口的方法上添加@Query 注解。

JPQL 不支持 INSERT,此外在执行 UPDATE 和 DELETE 操作时需要添加@Transactional 和@Modifying 注解。由于 JPQL 语法类似 SQL,只是用对象取代数据库表,用属性取代字段,无须特意去学习 JPQL 语法,只需通过下面的一些案例熟悉一下 JPQL 的风格。

【例 3-6】 SQL 与 JPQL 的对比。

```
//第 3 章/jpa1/CourseDao.java
  //SQL 语句,查询所有课程的所有列
select * from course c
  //JPQL 语句,查询所有课程的所有属性
select c from course c
  //JPQL 语句查询所有属性的简化格式
from course c

  //SQL 查询部分列
select c.coursename from course c
  //JPQL 查询部分属性
select c.coursename from course c
```

在 DAO 层方法上的@Query 注解中添加一个 nativeQuery 参数,将值设置为 true 即可

使用原生 SQL 语句进行查询。如果不指明 nativeQuery 参数，则默认为 JPQL 查询。

在 JPQL 或原生 SQL 中，若用到参数，语法如下：？＋参数的序号，如？1 表示第 1 个参数，？2 表示第 2 个参数。也可用命名参数，语法是“:＋参数名称”，在这种情况下方法的参数要用@Param 注解进行定义，并且名称要相同，具体用法见下面的案例步骤(1)。@Query 注解中若进行增、删、改操作，则需要再加上@Transaction 注解和@Modifying 注解。

【例 3-7】 使用 JPQL 及原生 SQL 查询。

(1) 数据访问层添加方法。接着上述案例项目 jpa1，在 CourseDao 中添加方法，代码如下：

```
//第 3 章/jpa1/CourseDao.java
     //自定义查询,DAO 层需手动编写 SQL 或 JPQL 语句
     //使用 JPQL,默认
    @Query("select c from course c where c.coursescore>?1")
    public List<Course> selectCourses1(int coursescore);
     //使用 JPQL,默认,多个参数
    @Query("select c from course c where c.coursescore>?1 and c.coursetime>?2")
    public List<Course> selectCourses2(int coursescore,int coursetime);
     //使用原始 SQL
    @Query(value="select * from course where coursescore>?1",nativeQuery=true)
    public List<Course> selectCourses3(int coursescore);
     //使用命名参数
    @Query("select c from course c where c.coursescore>:coursescore")
    public List<Course> selectCourses4(@Param("coursescore")int coursescore);
     //模糊查询
    @Query("select c from course c where c.coursename like %?1%")
    public List<Course> selectCourses5(String coursename);
     //自定义的增、删、改、查还要加上@Transactional 和@Modifying 注解
    @Transactional
    @Modifying
    @Query("update course set coursename=?1,coursescore=?2,coursetime=?3,coursetype=?4
where courseid=?5")
     public void updateCourse(String coursename,int coursescore,int coursetime,String
coursetype,int courseid);
```

(2) 在业务逻辑层添加方法。在业务逻辑层 CourseService 中添加上述 Dao 层方法对应的方法。由于代码几乎一样，这里不再提供。

(3) 控制器添加方法。在 CourseController 中添加如下方法，代码如下：

```
//第 3 章/jpa1/CourseController.java
     //自定义查询,DAO 层需手动编写 SQL 或 JPQL 语句
     //查询学分大于某个值的课程信息,DAO 层使用 JPQL
@GetMapping("/selectCourses1/{coursescore}")
public ModelAndView selectCourses1(@PathVariable("coursescore") int coursescore) {
    ModelAndView mv=new ModelAndView();
    List<Course> courses=courseService.selectCourses1(coursescore);
    mv.addObject("courses",courses);
    mv.setViewName("courses");
```

```
        return mv;
    }

        //查询学分大于某个值,并且课时大于某个值的课程,DAO层使用JPQL,多个参数
    @GetMapping("/selectCourses2/{coursescore}/{coursetime}")
    public ModelAndView selectCourses2(@PathVariable("coursescore") int coursescore,@PathVariable
    ("coursetime") int coursetime){
        ModelAndView mv = new ModelAndView();
        List<Course> courses = courseService.selectCourses2(coursescore,coursetime);
        mv.addObject("courses",courses);
        mv.setViewName("courses");
        return mv;
    }

        //查询学分大于某个值的课程信息,DAO层使用原生SQL
    @GetMapping("/selectCourses3/{coursescore}")
    public ModelAndView selectCourses3(@PathVariable("coursescore") int coursescore) {
        ModelAndView mv = new ModelAndView();
        List<Course> courses = courseService.selectCourses3(coursescore);
        mv.addObject("courses",courses);
        mv.setViewName("courses");
        return mv;
    }

        //查询学分大于某个值的课程信息,使用命名参数
    @GetMapping("/selectCourses4/{coursescore}")
    public ModelAndView selectCourses4(@PathVariable("coursescore") int coursescore){
        ModelAndView mv = new ModelAndView();
        List<Course> courses = courseService.selectCourses4(coursescore);
        mv.addObject("courses",courses);
        mv.setViewName("courses");
        return mv;
    }

        //模糊查询
    @GetMapping("/selectCourses5/{coursename}")
    public ModelAndView selectCourses5(@PathVariable("coursename") String coursename){
        ModelAndView mv = new ModelAndView();
        List<Course> courses = courseService.selectCourses5(coursename);
        mv.addObject("courses",courses);
        mv.setViewName("courses");
        return mv;
    }
```

(4) 运行代码进行测试。这里仅测试控制器的第1种方法,其余的方法读者可自行测试。在浏览器输入http://localhost:8080/selectCourses1/3,结果如图3-11所示。

3.2.7 一对一关联查询

对象关系映射模型ORM中有一对一、一对多、多对多等关联映射。所谓一对一映射是指每个实体有唯一的另一个实体与之关联,如人与身份证,每个人只能对应唯一的一张身份证,又如学生与借书证,每个学生只能对应唯一的一张借书证。

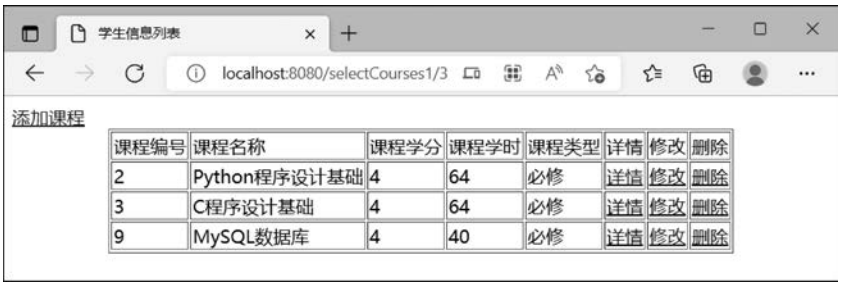

图 3-11 查询结果

ORM 的映射方向分为两种，一种是单向关联，另一种是双向关联。单向关联是指在其中一个实体中设置了另一个实体类型的属性，称为关联属性，从而可以引用（导航到）另一个实体，但另一个实体中并没有设置该实体类型的关联属性。双向关联是指实体双方都设置了对方类型的关联属性，从而任意一个实体均可引用（导航到）对方实体。一对一映射通常只做单向关联即可，一对多映射可以只做单向关联，但如果同时要多对一映射，则需要做双向关联，多对多映射则都需要双向关联。

在 Spring Data JPA 中，在一个实体上设置代表另一个实体的关联属性，在该属性上面添加@OneToOne 注解和@JoinColumn 注解关联到另一个实体的主键，并在注解中将外键设置为唯一约束即可实现一对一映射。这样查询一个实体的结果将包含另一个实体的数据。

【例 3-8】 实现学生 Student 与借书证 LibraryCard 之间的一对一关联。

（1）数据库准备。在 studentdb 数据库中创建表 librarycard，主键为 id，自增长，另一个字段为 creditnum，int 类型，表示可借本数，填入若干测试数据。在 student 表中添加一个字段 librarycardid，并添加若干数据，需参照 librarycard 表中的 id 值。

（2）创建实体类 LibraryCard，关键代码如下：

```
//第 3 章/jpa1/LibraryCard.java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity(name = "librarycard")
public class LibraryCard {
     //借书证号
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
     //可借本数
    @Column(name = "creditnum")
    private int creditNum;
}
```

（3）创建实体类 Student，通过注解实现一对一关联，关键代码如下：

```
//第 3 章/jpa1/Student.java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity(name = "student")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "studentname")
    private String studentname;

    @Column(name = "gender")
    private String gender;

    @Column(name = "age")
     private int age;
     //一对一关联
     //表示一对一关联
     @OneToOne(
     //级联操作
           cascade = CascadeType.ALL,
     //延迟加载
           fetch = FetchType.LAZY,
     //关联的另一方实体
           targetEntity = LibraryCard.class
     )
     @JoinColumn(
     //外键列
           name = "librarycardid",
     //将外键列设置为唯一约束
           unique = true,
     //主表的主键
           referencedColumnName = "id"
)
     //关联属性
    private LibraryCard libraryCard;

    public Student(String studentname, String gender, int age) {
       this.studentname = studentname;
       this.gender = gender;
       this.age = age;
}
}
```

(4) 创建数据访问层。创建 StudentDao 接口,继承 JpaRepository 接口,代码如下:

```
//第 3 章/jpa1/StudentDao.java
public interface StudentDao extends JpaRepository<Student,Integer>{

}
```

(5) 创建业务逻辑层。创建 StudentService,关键代码如下:

```
//第 3 章/jpa1/StudentService.java
@Service
public class StudentService {
    @Autowired
    private StudentDao studentDao;
    public List < Student > findAllStudents(){
        return studentDao.findAll();
    }
    public Student findStudentById(int id){
        return studentDao.findById(id).orElse(new Student());
    }
}
```

（6）创建控制器。创建 StudentController 类，关键代码如下：

```
//第 3 章/jpa1/StudentController.java
@Controller
public class StudentController {
    @Autowired
    private StudentService studentService;

@GetMapping("/findStudentById/{id}")
    public ModelAndView findStudentById(@PathVariable("id") int id){
        ModelAndView mv = new ModelAndView();
        Student student = studentService.findStudentById(id);
        mv.addObject("student",student);
        mv.setViewName("student");
        return mv;
    }
}
```

（7）创建视图。创建 student.html，关键代码如下：

```
//第 3 章/jpa1/student.html
< body >
< h1 >学生详情</h1 >
< table border = "1" align = "center">
    < tr >< td >学号</td >< td th:text = " $ {student.id}"></td ></tr >
    < tr >< td >姓名</td >< td th:text = " $ {student.studentname}"></td ></tr >
    < tr >< td >性别</td >< td th:text = " $ {student.gender}"></td ></tr >
    < tr >< td >年龄</td >< td th:text = " $ {student.age}"></td ></tr >
    < tr >< td >借书证信息</td >< td >
        < table >
            < tr >< td >借书证号</td >< td th:text = " $ {student.libraryCard.id}"></td ></tr >
            < tr >< td >可借本数</td >< td th:text = " $ {student.libraryCard.creditNum}"></td >
</tr >
        </table >
    </td ></tr >
</table >
</body >
```

（8）运行代码进行测试。在浏览器中输入：http://localhost:8080/findStudentById/1，结果如图 3-12 所示。

图 3-12　查询结果

3.2.8　一对多与多对一关联查询

对象关系映射模型 ORM 中的一对多映射是指一个实体有若干个其他实体与之对应，如一个部门有若干名员工，部门为“一方”，员工为“多方”，一个班级有若干名学生，班级为“一方”，学生为“多方”。多对一则与一对多正好相反。在 Spring Data JPA 中，在代表“一方”的实体类上设置代表“多方”实体集合的关联属性，在该属性上面添加@OneToMany 注解，在代表“多方”的实体类上设置代表“一方”的关联属性，在该属性上面添加@ManyToOne 注解和@JointColumn 注解即可同时实现一对多和多对一映射。这样查询“一方”实体的结果将包含“多方”实体集合的数据，查询“多方”实体的结果将包含“一方”实体的数据。

【例 3-9】 实现班级 Classes 对象与学生 Student 的一对多与多对一映射。

(1) 数据库准备。创建表 classes，主键 id 自增长，添加若干数据，如图 3-13 所示。修改 student 表，添加 classes_id 列，添加若干数据，如图 3-14 所示。

id	address	classname
1	1号楼101教室	计算机科学与技术1班
2	1号楼102教室	计算机科学与技术2班
3	1号楼103教室	计算机科学与技术3班

图 3-13　班级表

id	studentname	gender	age	librarycardid	classes_id
1	李白	男	18	1	1
2	白居易	男	19	2	2
3	张飞	男	20	3	3
4	李清照	女	21	4	1
5	王维	男	22	5	2
6	苏东坡	男	23	6	3
7	苏小妹	女	17	7	1
8	李牧	男	25	8	2
9	杜甫	男	28	9	3
10	林冲	男	26	10	1
11	刘备	男	25	11	2
12	周瑜	男	25	12	3
13	曹操	男	35	13	1

图 3-14　学生表

（2）创建实体类 Classes 作为“一方”。Classes 类的关键代码如下：

```
//第 3 章/jpa1/Classes.java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity(name = "classes")
public class Classes {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "classname")
    private String classname;

    @Column(name = "address")
     //教室
    private String address;

    @OneToMany(
     //指定关联的另一方实体
           targetEntity = Student.class,
     //级联操作
           cascade = CascadeType.ALL,
     //延迟加载
           fetch = FetchType.LAZY
)
     //指定多方表的外键
    @JoinColumn(name = "classes_id")
       //关联属性,多方的集合
private List < Student > students;
}
```

（3）修改实体类 Student。添加的关键属性及注解如下：

```
@ManyToOne
     //多对一关联
    private Classes classes;
```

这表示多对一关联映射，如果只想做一对多，不做多对一，则 Student 实体类可以不要 classes 属性。

（4）创建 Dao 层。创建 ClassDao，代码如下：

```
public interface ClassesDao extends JpaRepository < Classes, Integer > {
}
```

（5）创建业务层。在包 com.seehope.service 下创建 ClassesService 类，关键代码如下：

```
//第 3 章/jpa1/ClassesService.java
@Service
public class ClassesService {
    @Autowired
```

```
    private ClassesDao classesDao;
    public List<Classes> findAllClasses(){
        return classesDao.findAll();
    }
}
```

（6）创建控制器。在 com.seehope.controller 包下创建 ClassesController 类，关键代码如下：

```
//第 3 章/jpa1/ClassesController.java
@Controller
public class ClassesController {
    @Autowired
    private ClassesService classService;

    @GetMapping("/findAllClasses")
    public ModelAndView findAllClasses(){
        ModelAndView mv = new ModelAndView();
        List<Classes> classesList = classService.findAllClasses();
        mv.addObject("classesList",classesList);
        mv.setViewName("classesList");
        return mv;
    }
}
```

在 StudentController 中添加方法，代码如下：

```
//第 3 章/jpa1/StudentController.java
@GetMapping("/findStudentClasses/{id}")
public ModelAndView findStudentClasses(@PathVariable("id") int id){
    ModelAndView mv = new ModelAndView();
    Student student = studentService.findStudentById(id);
    mv.addObject("student",student);
    mv.setViewName("student2");
    return mv;
}
```

（7）创建视图。创建 classesList.html，复制 student.html 并保存为 student2.html，适当修改，以便可以显示班级信息。详细代码见随书配套资源。

（8）运行代码进行测试。在浏览器中输入 http://localhost:8080/findAllClasses，结果如图 3-15 所示，体现了一对多（每个班级对应多名学生）。在浏览器中输入 http://localhost:8080/findStudentClasses/1，结果如图 3-16 所示，体现了多对一（每名学生对应一个班级）。

3.2.9 多对多关联查询

多对多是指双方的一个实体有多个对方类型的实体与之对应，如学生与课程，一名学生可以选多门课程，一门课程可以被多选。多对多关联只能通过中间表进行映射。在 Spring Data JPA 中，通过在实体类中进行双向关联，设置对方类型的集合属性，并在该属性上面添加@ManyToMany 注解，以及使用@JointTable 指定中间表及外键即可实现多对多关联。

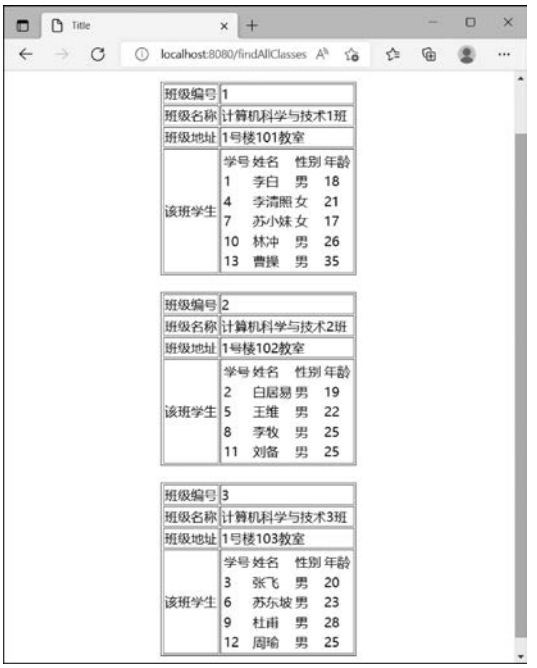

图 3-15 一对多查询

图 3-16 多对一查询

【例 3-10】 实现学生 Student 与课程 Course 之间的多对多映射。

(1) 数据库准备。创建一个中间表 student_course,主键自增长。另有两个字段 student_id 和 course_id,分别作为 student 表和 course 表的外键。

(2) 修改实体类 Student,添加的关键代码如下:

```
//第 3 章/jpa1/Student.java
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(name = "student_course", joinColumns = {@JoinColumn(name = "student_id")},
inverseJoinColumns = {@JoinColumn(name = "course_id")})
    //该学生选修的课程集合,多对多关联
    private List<Course> courseList;
```

这里@JointTable 用于指定中间表,joinColumns 属性用于指定外键列,inverseJoinColumns 用于指定关联方的外键列。

(3) 修改实体类 Course,添加的关键代码如下:

```
//第 3 章/jpa1/Course.java
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(name = "student_course",joinColumns = {@JoinColumn(name = "course_id")},
inverseJoinColumns = {@JoinColumn(name = "student_id")})
    //选修该课程的学生集合,多对多关联
private List<Student> studentList;
```

(4) 创建测试类,关键代码如下:

```
//第 3 章/jpa1/Test1.java
@Spring BootTest
public class Test1 {
    @Autowired
    private StudentDao studentDao;

    @Autowired
    private CourseDao courseDao;

    @Test
    public void addStudentAndCourse(){
        List < Student > students = new ArrayList <>();
        List < Course > courses = new ArrayList <>();
        Course course1 = new Course("数据结构", 2, 40, "选修", "aaa") ;
        Course course2 = new Course("操作系统", 4, 40, "必修", "aaa") ;
        courses.add(course1);
        courses.add(course2);
        courseDao.save(course1);
        courseDao.save(course2);
        Student student1 = new Student("Smith","男",20);
        Student student2 = new Student("Alice","女",19);
        students.add(student1);
        students.add(student2);
        course1.setStudentList(students);       //一门课程关联多名学生
        course2.setStudentList(students);       //一门课程关联多名学生
        student1.setCourseList(courses);        //一名学生关联多门课程
        student2.setCourseList(courses);        //一名学生关联多门课程
        studentDao.save(student1);
        studentDao.save(student2);
        System.out.println("保存成功!");
    }
}
```

运行测试类，查看数据库表 student_course 可发现多了若干记录。

(5) 创建控制器。在 CourseController 中添加的方法如下：

```
//第 3 章/jpa1/CourseController.java
    @GetMapping("/findCourseStudent")
      //查询某个编号的课程
    public ModelAndView findCourseStudent(int id){
        ModelAndView mv = new ModelAndView();
        Course course = courseService.findById(id);
        mv.addObject("course",course);
        mv.setViewName("course2");
        return mv;
    }
```

在 StudentController 中添加的方法如下：

```
//第 3 章/jpa1/StudentController.java
    @GetMapping("/findStudentCourse")
    //查询某个编号的学生
```

```
public ModelAndView findStudentCourse(int id){
        ModelAndView mv = new ModelAndView();
        Student student = studentService.findStudentById(id);
        mv.addObject("student",student);
        mv.setViewName("student3");
        return mv;
    }
```

(6) 创建视图。创建 course2.html 和 student3.html 页面,详见配套资源。

(7) 运行代码进行测试。在浏览器输入 http://localhost:8080/findStudentCourse?id=8,结果如图 3-17 所示。

在浏览器输入 http://localhost:8080/findCourseStudent?id=12,结果如图 3-18 所示。

图 3-17 一名学生对应多门课程

图 3-18 一门课程对应多名学生

3.2.10 多条件动态查询

对于查询条件不确定的情况,要根据用户的不同选择生成动态的 SQL 语句。JPA 提供了 JpaSpecificationExecutor 接口,该接口提供了 List<T> findAll(@Nullable Specification<T> spec)等带有 Specification 对象的方法,这个 Specification 接口的 toPredicate 方法专门用于实现多条件动态查询。详情参见下面的案例。

【例 3-11】 在上面项目的基础上,根据学生姓名和性别等条件动态查询学生信息。

(1) Dao 接口继承 JpaSpecificationExecutor 接口。在原有继承的基础上,增加继承 JpaSpecificationExecutor 接口,代码如下:

```
public interface StudentDao extends JpaRepository<Student,Integer>, JpaSpecificationExecutor {

}
```

(2) 在业务层添加方法。在业务层 StudentService 添加的代码如下:

```
//第 3 章/jpa1/StudentService.java
    //查找学生
```

```
public List<Student> searchStudents(Student student){
     //封装查询对象 Specification,这是个自带的动态条件查询
    Specification<Student> spec = new Specification<Student>() {
       @Override
       public Predicate toPredicate(Root<Student> root, CriteriaQuery<?> query, CriteriaBuilder
criteriaBuilder) {
     //定义集合来肯定 Predicate[] 的长度,由于 CriteriaBuilder 的 or 方法需要传入的是断言数组
          List<Predicate> predicates = new ArrayList<Predicate>();
     //对客户端查询条件进行判断并封装 Predicate 断言对象
          if(null != student.getStudentname()&&""!= student.getStudentname()){
predicates.add(criteriaBuilder.like(root.get("studentname").as(String.class), "%" +
student.getStudentname() + "%"));
          }
          if(null != student.getGender()&&""!= student.getGender()){
predicates.add(criteriaBuilder.equal(root.get("gender").as(String.class), student.
getGender()));
          }
          return criteriaBuilder.and(predicates.toArray(new Predicate[predicates.size()]));
       }
     };
     //查询出所有数据源列表
     return studentDao.findAll(spec);
}
```

多条件动态查询的关键技术就是上述代码。

(3) 在控制器添加方法,代码如下:

```
//第 3 章/jpa1/StudentController.java
@GetMapping("/searchStudents")
public ModelAndView searchStudents(Student student){
   ModelAndView mv = new ModelAndView();
   List<Student> students = studentService.searchStudents(student);
   mv.addObject("students",students);
   mv.setViewName("students");
   return mv;
}
```

(4) 前端页面。在 students.html 页面添加的代码如下:

```
//第 3 章/jpa1/students.html
<div>
    <form action="/searchStudents" method="get">
        姓名:<input type="text" name="studentname" /><br/>
        性别:<input type="text" name="gender"/><br/>
        <input type="submit" value="搜索"/>
    </form>
</div>
<br/>
```

(5) 运行代码进行测试。通过浏览器访问 http://localhost:8080/findAllStudents,结果如图 3-19 所示。在搜索框中输入李,结果如图 3-20 所示。在两个搜索框分别输入李和

男，结果如图 3-21 所示。可见不同条件都可以动态地查出来。

图 3-19 全部学生信息

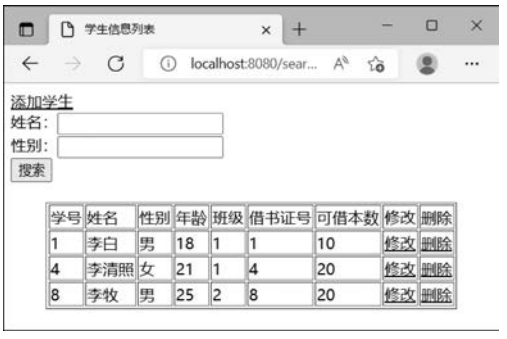

图 3-20 姓李的学生

图 3-21 姓李的男生

本章小结

本章学习了 Spring Boot JDBC 操作数据库的技术，Spring Data JPA 操作数据库技术包括简单查询、命名查询、JPQL 或原生 SQL 查询、一对多、多对一、多对多等复杂查询与动态查询。