

# SpringBoot Data Jpa简介

# 课程目标

- SpringBoot Data Jpa简介
- Jpa 常用注解
- Spring Data JPA核心接口

# SpringBoot Data Jpa简介

# 什么是JPA

- JPA(Java Persistence API)意即Java持久化API，是Sun官方在JDK5.0后提出的Java持久化规范（JSR 338，这些接口所在包为javax.persistence）。
- JPA的出现主要是为了简化持久层开发以及整合ORM技术，结束Hibernate、TopLink、JDO等ORM框架各自为营的局面。JPA是在吸收现有ORM框架的基础上发展而来，易于使用，伸缩性强。

# 什么是Spring Data JPA

- Spring Data JPA是Spring Data家族的一部分，可以轻松实现基于JPA的存储库。
- 此模块处理对基于JPA的数据访问层的增强支持。
- 它使构建使用数据访问技术的Spring驱动应用程序变得更加容易。

- ❑在相当长的一段时间内，实现应用程序的数据访问层一直很麻烦，必须编写太多样板代码来执行简单查询以及执行分页和审计。
- ❑Spring Data JPA旨在通过减少实际需要的工作量来显著改善数据访问层的实现。作为开发人员，只需编写 repository 接口，包括自定义查找器方法，Spring 将自动提供实现。

# JPA和Hibernate的关系

- Hibernate是之前很流行的ORM框架，也是JPA的一个实现，其它还有Toplink之类的ROM框架。
- JPA和Hibernate之间的关系，可以简单的理解为JPA是标准接口，Hibernate是实现。

# 基本配置

## □POM.XML

*<!--引入hibernate实体管理-->*

**<dependency>**

**<groupId>**org.hibernate**</groupId>**

**<artifactId>**hibernate-entitymanager**</artifactId>**

**</dependency>**

*<!--引入spring data jpa-->*

**<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-data-jpa**</artifactId>**

**</dependency>**



```
<!-- 5/ 入 spring jdbc -->
```

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-jdbc</artifactId>
```

```
</dependency>
```

```
<!-- 5/ 入 lombok -->
```

```
<dependency>
```

```
  <groupId>org.projectlombok</groupId>
```

```
  <artifactId>lombok</artifactId>
```

```
  <scope>provided</scope>
```

```
</dependency>
```

<!--引入数据库驱动-->

<dependency>

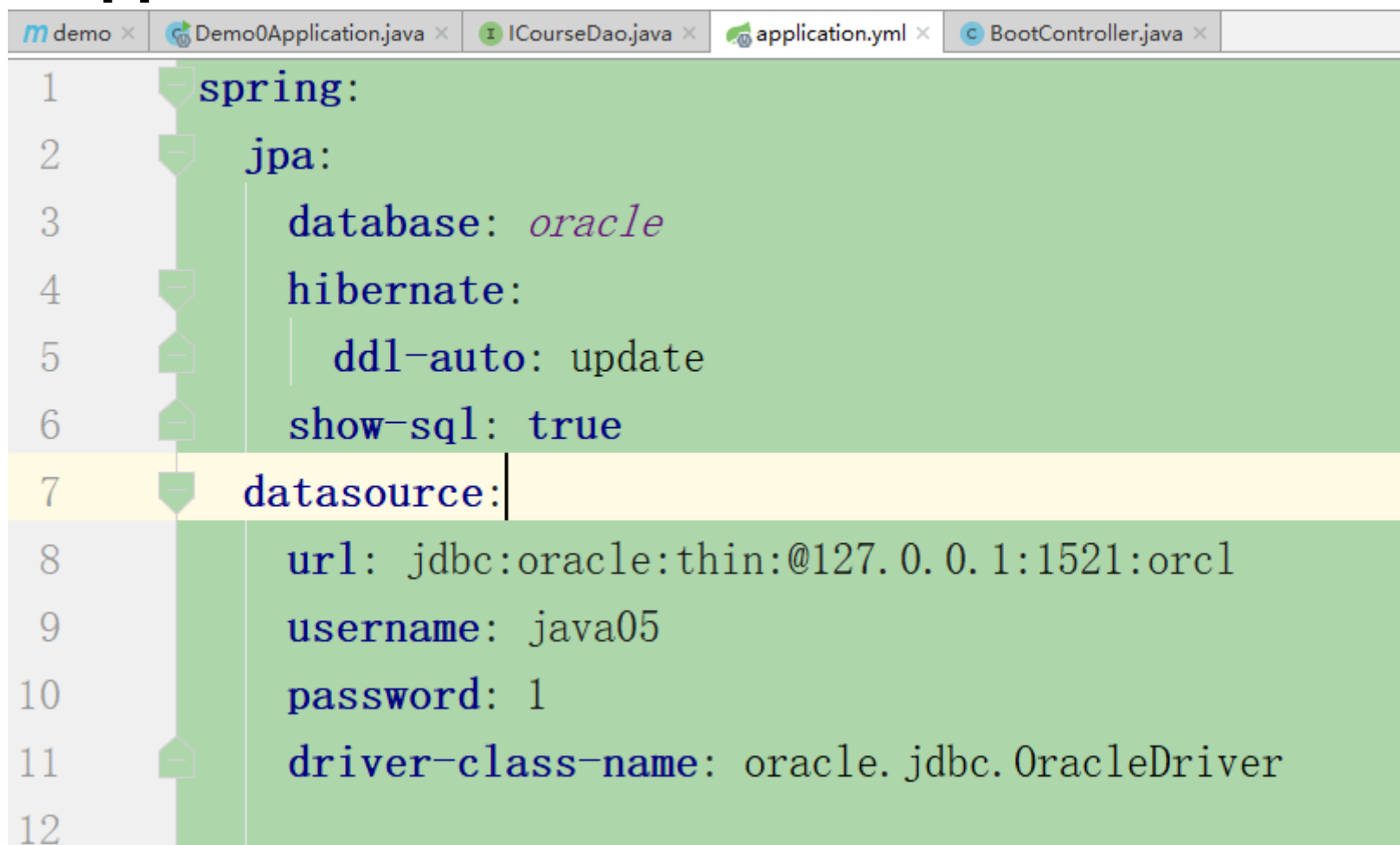
<groupId>com.oracle.database.jdbc</groupId>

<artifactId>ojdbc8</artifactId>

<scope>runtime</scope>

</dependency>

## □Application.YML配置



```
1  spring:
2    jpa:
3      database: oracle
4      hibernate:
5        ddl-auto: update
6      show-sql: true
7  datasource:
8    url: jdbc:oracle:thin:@127.0.0.1:1521:orcl
9    username: java05
10   password: 1
11   driver-class-name: oracle.jdbc.OracleDriver
12
```

# Jpa 常用注解

# 常用注解-@Entity

## □ @Entity:用于修饰类

- @Entity说明这个class是实体类，并且使用默认的orm规则，即class名对应数据库表中表名，class字段名即表中的字段名。
- 如果想改变这种默认的orm规则，就要使用@Table来改变class名与数据库中表名的映射规则，@Column来改变class中字段名与db中表的字段名的映射规则

# @Table

## □ @Table:用于修饰类

➤ @Table用来定义当前entity对应主表的name, catalog, schema等属性。

➤ 主要属性说明：

**name**:指定表的名称(默认为实体名)

**catalog**:指定数据库名称

**schema**:指定数据库的用户名

## □ @Column:用于修饰成员属性

➤ @Column定义了映射到数据库的列的所有属性：列名，是否唯一，是否允许为空，是否允许更新等。

➤ 主要属性说明：

name:指定列名(默认与当前属性名相同)。

unique: 是否唯一

nullable: 是否允许为空

insertable: 是否允许插入

updatable: 是否允许更新

## □ @Id:用于修饰成员属性

- 映射到数据库表的主键的属性,一个实体只能有一个属性被映射为主键.



# @GeneratedValue

## □ @GeneratedValue:用于修饰成员属性

➤ 为一个实体生成一个唯一标识的主键,并提供了主键的生成策略.

➤ 主要属性说明:

generator:值是一个字符串,默认为"",其声明了主键生成器的名称.

strategy: 主键生成策略,提供以下四种值.

## □主键生成策略

- -AUTO主键由程序控制, 是默认选项, 不设置就是这个
- -IDENTITY 主键由数据库生成, 采用数据库自增长, Oracle不支持这种方式
- -SEQUENCE 通过数据库的序列产生主键, MYSQL 不支持
- -Table 提供特定的数据库产生主键, 该方式更有利于数据库的移植

# @GeneratedValue

□**注意**：默认SpringBoot的@GeneratedValue 是不需要加参数的,但是如果数据库控制主键自增(auto\_increment), 不加参数就会报错

# @Data

## □ @Data:用于修饰类

- 为实体类提供读写功能，从而不用写get、set方法。
- 为类提供 equals()、hashCode()、toString() 方法。

□ **注意:**此注解由**lombok**插件提供,需要在pom.xml文件中添加依赖

```
<dependency>  
  <groupId>org.projectlombok</groupId>  
  <artifactId>lombok</artifactId>  
  <scope>provided</scope>  
</dependency>
```

# 上述注解实例

```
@Data
@Entity
@Table(name="COURSE") //设置数据库中表名字
public class Course {
    @Id
    @Column(name="COURSE_ID") //数据库实际列名为COURSE_ID
    private Integer courseId;

    @Column(name="COURSE_NAME")
    private String courseName;

    @Column(name="TEACHER_ID")
    private Integer teacherId;
```

# @Query

- ❑ @Query注解查询适用于所查询的数据无法通过关键字查询得到结果的查询。用于修饰方法
- ❑ 主要属性说明:
  - value:简单的说就是方法对应的sql语句
  - nativeQuery:标识使用原生sql语句还是jpql语句,一般使用"nativeQuery=true"使用原生sql语句.

## □对sql语句中的参数进行赋值有以下方法:

- 索引参数:索引值从1开始, 查询中"**?X**"个数需要与方法定义的参数个数相一致, 并且顺序也要一致。
- 命名参数(推荐使用此方式): 可以定义好参数名, 赋值时使用@Param("参数名"),而不用管顺序。

*//索引参数使用'?x'与方法参数对应*

```
@Query(value = "select * from Course where COURSE_TIME = ?1 and COURSE_LOC=?2",  
        nativeQuery = true)  
List<Course> queryCoursesByCourseTimeAndCourseLoc(int courseTime,String courseLoc);
```

*//命名参数使用':'与@Param对应*

```
@Query(value = "select * from Course where COURSE_NAME = :courseName",  
        nativeQuery = true)  
List<Course> queryCoursesByCourseName(@Param("courseName") String courseName);
```

# @Modifying

## □ @Modifying: 用于修饰方法

- 在@Query注解中编写JPQL实现DELETE和UPDATE操作的时候必须加上@modifying注解,以通知Spring Data这是一个DELETE或UPDATE操作。
- UPDATE或者DELETE操作需要使用事务,此时需要定义Service层,在Service层的方法上添加事务操作(@Transactional)。
- 注意JPQL不支持INSERT操作。



# @Transactional

## □ @Transactional

- 标识该方法需要在事务中执行,用于调用增删改功能.
- 可以定义事务的传播行为和隔离级别等属性.
- 此注解用于Service层,修饰需要开启事务的方法.

## 其他注解

□ Jpa不仅仅提供了以上常用注解,还有很多非常有用的注解,如关系映射注解,验证注解等.

# Spring Data JPA核心接口

## □ Spring Data JPA 提供如下核心接口

- Repository接口
- CrudRepository接口
- PagingAndSortingRepository接口、
- JpaRepository接口
- JPASpecificationExecutor接口

## □ Repository接口

### ➤ 提供了方法名称命名查询方式

- 简单的说就是可以解析方法名生成出对应的sql语句

### ➤ 提供了基于@Query注解查询与更新

## ❑ CrudRepository接口

- 继承Repository接口并提供一组预设的CURD操作方法,实现简单查询.

```
CrudRepository<T, ID extends Serializable>  
    ^ save(S) <S extends T> : S  
    ^ save(Iterable<S>) <S extends T> : Iterable<S>  
    ^ findOne(ID) : T  
    ^ exists(ID) : boolean  
    ^ findAll() : Iterable<T>  
    ^ findAll(Iterable<ID>) : Iterable<T>  
    ^ count() : long  
    ^ delete(ID) : void  
    ^ delete(T) : void  
    ^ delete(Iterable<? extends T>) : void  
    ^ deleteAll() : void
```

## ❑ PagingAndSortingRepository接口

- 该接口提供了分页与排序的操作。注意：该接口集成了CrudRepository接口

```
▲ 1 PagingAndSortingRepository<T, ID extends Serializable>  
    ● ^ findAll(Sort) : Iterable<T>  
    ● ^ findAll(Pageable) : Page<T>
```

## □ JpaRepository接口

- 该接口继承了PagingAndSortingRepository接口。
- 对继承的父接口中的方法的返回值进行适配。(不用对返回值去转换类型了!)



## □ JPASpecificationExecutor接口

- 该接口主要是提供了多条件查询的支持，并且可以在查询中添加分页与排序。

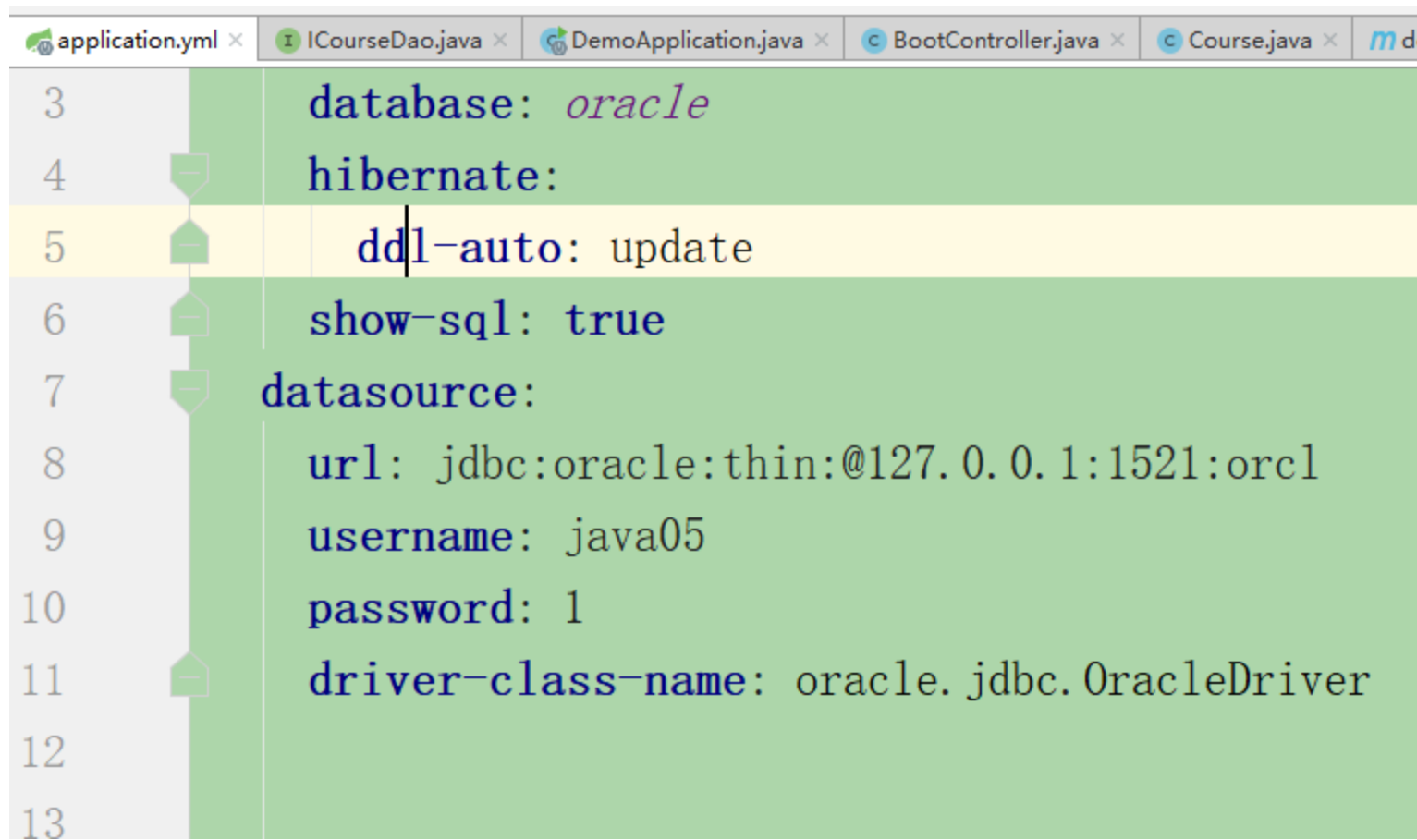
注意：JPASpecificationExecutor是单独存在--完全独立。

# 实例

## □1.添加依赖

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

## □2.在application文件设置全局参数



The screenshot shows an IDE with several tabs open: application.yml, ICourseDao.java, DemoApplication.java, BootController.java, Course.java, and m d. The application.yml file is selected and displays the following configuration:

```
3 database: oracle
4 hibernate:
5   ddl-auto: update
6 show-sql: true
7 datasource:
8   url: jdbc:oracle:thin:@127.0.0.1:1521:orcl
9   username: java05
10  password: 1
11  driver-class-name: oracle.jdbc.OracleDriver
12
13
```

## □ Dao层接口ICourseDao

@Repository

public interface ICourseDao extends JpaRepository<Course, Integer> {

//索引参数使用'?x'与方法参数对应

@Query(value = "select \* from Course where COURSE\_TIME = ?1 and COURSE\_LOC=?2",  
nativeQuery = true)

List<Course> queryCoursesByCourseTimeAndCourseLoc(int courseTime, String courseLoc);

//命名参数使用':'与@Param对应

@Query(value = "select \* from Course where COURSE\_NAME = :courseName",  
nativeQuery = true)

List<Course> queryCoursesByCourseName(@Param("courseName") String courseName);

实体类类型

表主键的java类型

//参数为实体类对象

@Modifying

```
@Query(value = "update Course set COURSE_NAME = :#{#course.courseName} where  
COURSE_ID = :#{#course.courseId}", nativeQuery=true)  
void updateCourse(@Param("course") Course course);
```

//方法名解析规则如下:

//find+全局修饰+By+实体属性名称+限定词+连接词+

// (其他实体属性)+OrderBy+排序属性+排序方向

Course findCourseById(int courseId);

□简单的CURD方法直接调用即可,如插入一条数据可直接使用save(entity)的方法实现.

```
@Service
public class CourseService {
    @Autowired
    ICourseDao id;

    @Transactional
    public void addCourse(Course c) {
        id.save(c);
    }
}
```