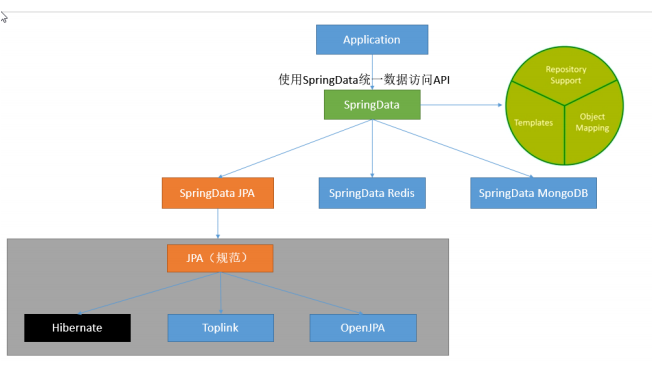
# SpringBoot整合Spring Data JPA技术1.Spring Data JPA 概述

SpringData JPA是Spring基于ORM框架、JPA规范的基础上封装的一套JPA应用框架，可以使开发者使用极简的代码实现对数据库的访问和操作。它提供了包括增删改查等在内的基本功能，且易于扩展。

ORM:（Object-Relational Mapping） 表示对象关系映射；只要有一套程序能够做到建立对象与数据库的关联，操作对象就可以直接操作数据库数据。

JPA规范:（Java Persistence API） Java 持久化 API。是SUN 公司推出的一套基于 ORM规范的，内部是由一系列的接口和抽象类构成。JPA规范本质上就是一种ORM规范，注意不是ORM框架——因为JPA并未提供ORM实现，它只是制订了一些规范，提供了一些编程的 API 接口，但具体实现则由服务厂商来提供实现。

Hibernate：是一个开放源代码的对象关系映射框架，它对JDBC进行了非常轻量级的对象封装，它将POJO与数据库表建立映射关系，是一个全自动的 orm 框架，hibernate 可以自动生成 SQL 语句，自动执行，使得 Java程序员可以随心所欲的使用对象编程思维来操纵数据库。



通俗来讲Spring Data JPA是对JPA规范的一层封装，Hibernate实现了JPA规范。

java代码----->springdata jpa ------>jpa规范(接口)------>hibernate（实现类）------>jdbc ----->mysql数据库

## 2.环境搭建

### 2.1加入Spring Data JPA的依赖

|  |
| --- |
| <!-- 数据库 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <scope>runtime</scope>  </dependency> |

## 3.入门案例

### 2.2 添加配置信息

在application.properties或application.yml中加入如下配置信息

|  |
| --- |
| **#连线数据库配置**  spring.datasource.driverClassName=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql://localhost:3306/ssm  spring.datasource.username=root  spring.datasource.password=hancong  **#**JPA  spring.jpa.database=mysql  spring.jpa.show-sql=true  spring.jpa.hibernate.ddl-auto=update  表的生成策略：  spring.jpa.hibernate.ddl-auto=create-drop  可选参数  create 启动时删数据库中的表，然后创建，退出时不删除数据表  create-drop 启动时删数据库中的表，然后创建，退出时删除数据表 如果表不存在报错  update 如果启动时表格式不一致则更新表，原有数据保留  validate 项目启动表结构进行校验 如果不一致则报错  #取消驼峰转下划线  spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl |

### 3.1添加实体

|  |
| --- |
| //使用JPA注解配置映射关系  @Entity //告诉JPA这是一个实体类（和数据表映射的类）  @Table(name = "tbl\_user") //@Table来指定和哪个数据表对应;如果省略默认表名就是user；  public class User {  @Id //这是一个主键  @GeneratedValue(strategy = GenerationType.IDENTITY)//自增主键  private Integer id;  @Column(name = "last\_name",length= 50) //这是和数据表对应的一个列  private String lastName;  @Column //省略默认列名就是属性名  private String email; |

常用注解的说明

@Entity

作用：指定当前类是实体类。

@Table

作用：指定实体类和表之间的对应关系。

属性：

name：指定数据库表的名称

@Id

作用：指定当前字段是主键。

@GeneratedValue

作用：指定主键的生成方式。

属性：

strategy ：指定主键生成策略。

@Column

作用：指定实体类属性和数据库表之间的对应关系

属性：

name：指定数据库表的列名称。

unique：是否唯一

nullable：是否可以为空

inserttable：是否可以插入

updateable：是否可以更新

columnDefinition: 定义建表时创建此列的 DDL

secondaryTable: 从表名。如果此列不建在主表上（默认建在主表），该属性定义该列所在从表的名字

### 3.2添加Dao接口

#### 案例

1. 创建一个 Dao 层接口，并实现JpaRepository接口
2. 提供相应的泛型

|  |
| --- |
| /\*\*  \* JpaRepository<实体类类型，主键类型>：底层封装好了基本的CRUD操作的方法  \*/  **public** **interface** UserDao **extends** JpaRepository<User, Integer>{  } |

#### 核心接口

1. **Repository：**

方式一： 提供了findBy + 属性方法 ：方法的名称必须遵循驼峰式名称规则

|  |  |  |
| --- | --- | --- |
| **查询方式** | **举例** | **sql where字句** |
| **And** | findByNameAndPwd | where name= ? and pwd =? |
| **Or** | findByNameOrSex | where name= ? or sex=? |
| **Is,Equals** | findById,findByIdEquals | where id= ? |
| **Between** | findByIdBetween | where id between ? and ? |
| **LessThan** | findByIdLessThan | where id < ? |
| **LessThanEqual** | findByIdLessThanEquals | where id <= ? |
| **GreaterThan** | findByIdGreaterThan | where id > ? |
| **GreaterThanEqual** | findByIdGreaterThanEquals | where id > = ? |
| **After** | findByIdAfter | where id > ? |
| **Before** | findByIdBefore | where id < ? |
| **IsNull** | findByNameIsNull | where name is null |
| **IsNotNull,NotNull** | findByNameNotNull | where name is not null |
| **Like** | findByNameLike | where name like ? |
| **NotLike** | findByNameNotLike | where name not like ? |
| **StartingWith** | findByNameStartingWith | where name like '?%' |
| **EndingWith** | findByNameEndingWith | where name like '%?' |
| **Containing** | findByNameContaining | where name like '%?%' |
| **OrderBy** | findByIdOrderByXDesc | where id=? order by x desc |
| **Not** | findByNameNot | where name <> ? |
| **In** | findByIdIn(Collection<?> c) | where id in (?) |
| **NotIn** | findByIdNotIn(Collection<?> c) | where id not in (?) |
| **TRUE** | findByAaaTue | where aaa = true |
| **FALSE** | findByAaaFalse | where aaa = false |
| **IgnoreCase** | findByNameIgnoreCase | where UPPER(name)=UPPER(?) |

举例：

|  |
| --- |
| public **interface** UserDao **extends** Repository<SysUsers, Integer> {  // 单条件  List<SysUsers> findByName(String name);  } |

方式二： @Query注解

|  |
| --- |
| public **interface** SysUsersRepositoryQueryAnnotation **extends** Repository<SysUsers, Integer> {  // 这种写法语句中 SysUsers 必须是和实体类名称一样 不能是数据里的表名称（sys\_users）  // 底层会对**HQL**语句就行转换,这种方法nativeQuery默认为false  @Query("from SysUsers where name = ?")  List<SysUsers> QueryByNameHQL(String name);    // nativeQuery= true 说明这的语句就是正常的**SQL**语句，底层不会对改语句进行转换  @Query(value = "select \* from sys\_users where name = ?1", nativeQuery = true)  List<SysUsers> QueryByNameSQL(String name);  @Query("update SysUsers set name = ? where id =?")  @Modifying // 需要加上@Modifying  void UpdateSysUsersNameById(String name, Integer id);  } |

1. **CurdRepository**

继承了Repository 主要提供了对数据的增删改查

1. **PagingAndSortRepository**

继承了CrudRepository 提供了对数据的分页和排序，

缺点是只能对所有的数据进行分页或者排序，不能做条件判断

1. **JpaRepository**

继承了PagingAndSortRepository；开发中经常使用的接口

1. **JpaSpecificationExecutor**

提供多条件查询，复杂查询

|  |
| --- |
| String stuName = null;  Integer status = null;  List<Student> list = stuDao.findAll(new Specification<Student>() {  /\*\*  \* Root：查询哪个表  CriteriaQuery：查询哪些字段  CriteriaBuilder：字段如何生成一个查询条件，每一个查询条件都是什么方式  Predicate（Expression）：单独每一条查询条件的详细描述  \*/  @Override  public Predicate toPredicate(  Root<Student> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {    List<Predicate> conditions = new ArrayList<>();  //姓名  if(stuName!=null && !"".equals(stuName)) {  **Predicate stuN= criteriaBuilder.like(root.get("stuName"), "%"+stuName+"%");**  **conditions.add(stuN);**  }  //状态  if(status!=null) {  **Predicate statusP = criteriaBuilder.equal(root.get("status"), status);**  **conditions.add(statusP);**  }  **Predicate[] preArr = conditions.toArray(new Predicate[conditions.size()]);**  **query.where(preArr);**  return null;  }  }); |

### 3.3单元测试

#### 增删改查

省略

#### 排序

|  |
| --- |
| //Order 定义排序规则  Order order = Order.asc("id");  //Sort对象封装了排序规则  Sort sort = Sort.by(order);  List<Users> list = (List<Users>)userDao.findAll(sort);  for (Users users : list) {  System.out.println(users);  } |

#### 分页

|  |
| --- |
| //Pageable:封装了分页的参数，当前页，每页显示的条数。注意：他的当前页是从0开始。  //PageRequest(page,size) page:当前页。size:每页显示的条数  Pageable pageable = PageRequest.of(1, 2);  Page<Users> page = usersDao.findAll(pageable);  System.out.println("总条数："+page.getTotalElements());  System.out.println("总页数"+page.getTotalPages());  List<Users> list = page.getContent();  for (Users users : list) {  System.out.println(users);  } |

#### 排序+分页

|  |
| --- |
| Sort sort = Sort.by(Orde.ascr( "id"));  Pageable pageable = PageRequest.of(1, 2, **sort**);  例如：PageRequest.of(0, 2, Sort.by(Order.desc("id"))  ... ... |

## 多表关联

### 3.1一对多，多对一

@OneToMany：一对多 一端使用

@ManyToOne：多对一 多端使用

角色： 1端

|  |
| --- |
| @Entity  public class Role{    //拥有mappedBy注解的实体类为关系被维护端  //mappedBy="author"中的author是Article中的author属性  @OneToMany(mappedBy="roles")  private Set<Users> users = new HashSet<>();  } |

用户： 多端

|  |
| --- |
| @Entity  @Table(name="t\_users")  public class Users {  @ManyToOne(cascade=CascadeType.PERSIST,fetch=FetchType.LAZY)  @JoinColumn(name="roles\_id") //@JoinColumn:维护外键  private Roles roles;  } |
| **cascade属性：**  CascadeType.PRESIST 级联持久化（保存）操作        CascadeType.REMOVE 级联删除操作        CascadeType.MERGE 级联更新操作        CascadeType.REFRESH 级联刷新操作  CascadeType.ALL 包含以上全部级联操作  fetch属性：  FetchType.LAZY：懒加载 FetchType.EAGER：立刻加载 |

#### 模糊分页

Dao接口定义方法：

|  |
| --- |
| Page<User> findBynameContaining (String name,Pageable page); |

### 3.2多对多

@ManyToMany：多对多

权限

|  |
| --- |
| @Entity  @Table(name="permission")  public class Permission{  //由角色方作为维护方  @ManyToMany(mappedBy="permissions",fetch = FetchType.LAZY)  private Set<Role> roles;  } |

角色:

|  |
| --- |
| @Entity  @Table(name="role")  public class Role{  @ManyToMany(cascade = CascadeType.MERGE,fetch = FetchType.LAZY)  @JoinTable(name="permission\_role",  joinColumns=@JoinColumn(name="role\_id"),  inverseJoinColumns=@JoinColumn(name="permission\_id"))  private Set<Permission> permissions = new HashSet<Permission>();  } |