SpringDataJPA

Day02（SpringDataJPA入门）

# 课程提要

1. Spring Data JPA概述
2. Spring Data Jpa入门
3. SpringDataJpa内部原理剖析
4. SpringDataJpa查询方式

# Spring Data JPA概述

## 概念

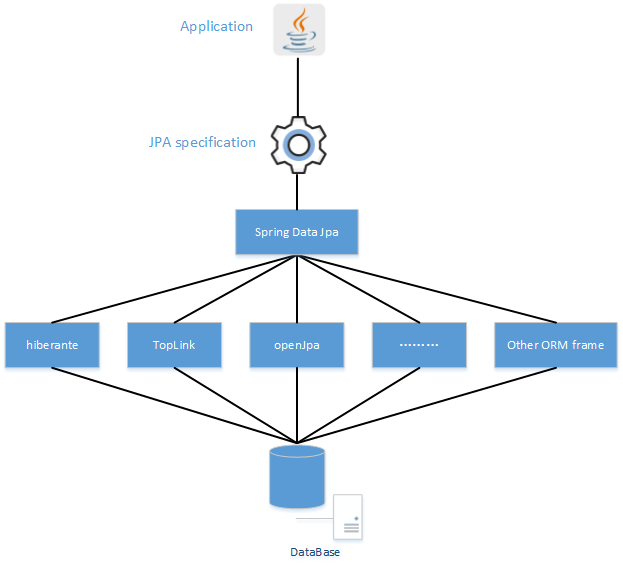
Spring Data JPA 是 Spring 基于 ORM 框架、JPA 规范的基础上封装的一套JPA应用框架，可使开发者用极简的代码即可实现对数据库的访问和操作。它提供了包括增删改查等在内的常用功能，且易于扩展！学习并使用 Spring Data JPA 可以极大提高开发效率！

Spring Data JPA 让我们解脱了DAO层的操作，基本上所有CRUD都可以依赖于它来实现,在实际的工作工程中，推荐使用Spring Data JPA + ORM（如：hibernate）完成操作，这样在切换不同的ORM框架时提供了极大的方便，同时也使数据库层操作更加简单。

## Spring Data JPA 与 JPA和hibernate之间的关系

JPA是一套规范，内部是有接口和抽象类组成的。hibernate是一套成熟的ORM框架，而且Hibernate实现了JPA规范，所以也可以称hibernate为JPA的一种实现方式，我们使用JPA的API编程，意味着站在更高的角度上看待问题（面向接口编程）

Spring Data JPA是Spring提供的一套对JPA操作更加高级的封装，是在JPA规范下的专门用来进行数据持久化的解决方案。



spring data jpa、jpa以及ORM框架之间的关系

# Spring Data JPA的快速入门

## 需求

使用SpringDataJpa实现CRUD操作。

## 环境搭建

创建一个maven工程，然后将以下的依赖添加到工程中。

|  |
| --- |
| <?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 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>   <groupId>com.itheima</groupId>  <artifactId>spring-data-second</artifactId>  <version>1.0-SNAPSHOT</version>   <properties>  <spring.version>4.2.4.RELEASE</spring.version>  <hibernate.version>5.0.7.Final</hibernate.version>  <slf4j.version>1.6.6</slf4j.version>  <log4j.version>1.2.12</log4j.version>  <c3p0.version>0.9.1.2</c3p0.version>  <mysql.version>5.1.6</mysql.version>  <maven.compiler.source>1.9</maven.compiler.source>  <maven.compiler.target>1.9</maven.compiler.target>  </properties>   <dependencies>  <!-- junit单元测试 -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>   <!-- spring beg -->  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.6.8</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>${spring.version}</version>  </dependency>   <!-- spring end -->   <!-- hibernate beg -->  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>  <version>${hibernate.version}</version>  </dependency>  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-entitymanager</artifactId>  <version>${hibernate.version}</version>  </dependency>  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-validator</artifactId>  <version>5.2.1.Final</version>  </dependency>  <!-- hibernate end -->   <!-- c3p0 beg -->  <dependency>  <groupId>c3p0</groupId>  <artifactId>c3p0</artifactId>  <version>${c3p0.version}</version>  </dependency>  <!-- c3p0 end -->   <!-- log end -->  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>   <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>${slf4j.version}</version>  </dependency>   <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>${slf4j.version}</version>  </dependency>  <!-- log end -->    <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>${mysql.version}</version>  </dependency>   <dependency>  <groupId>org.springframework.data</groupId>  <artifactId>spring-data-jpa</artifactId>  <version>1.9.0.RELEASE</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>4.2.4.RELEASE</version>  </dependency>   <!-- el beg 使用spring data jpa 必须引入 -->  <dependency>  <groupId>javax.el</groupId>  <artifactId>javax.el-api</artifactId>  <version>2.2.4</version>  </dependency>   <dependency>  <groupId>org.glassfish.web</groupId>  <artifactId>javax.el</artifactId>  <version>2.2.4</version>  </dependency>  <!-- el end -->  <!--如果是jdk1.8不需要引入此jar包-->  <dependency>  <groupId>javax.xml.bind</groupId>  <artifactId>jaxb-api</artifactId>  <version>2.3.0</version>  </dependency>  </dependencies>    </project> |

## 整合Spring Data JPA与Spring

创建一个applicationContext.xml配置文件，然后在其中配置SpringDataJpa的相关配置。

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:jdbc="http://www.springframework.org/schema/jdbc" xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:jpa="http://www.springframework.org/schema/data/jpa" xmlns:task="http://www.springframework.org/schema/task"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd  http://www.springframework.org/schema/data/jpa  http://www.springframework.org/schema/data/jpa/spring-jpa.xsd"**>   *<!-- 1.dataSource 配置数据库连接池-->* <**bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>  <**property name="driverClass" value="com.mysql.jdbc.Driver"** />  <**property name="jdbcUrl" value="jdbc:mysql://localhost:3306/kaka"** />  <**property name="user" value="root"** />  <**property name="password" value="root"** />  </**bean**>  *<!--2.配置EntityManagerFactory-->* <**bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"**>  <**property name="dataSource" ref="dataSource"**/>  *<!--扫描实体类所在的包-->* <**property name="packagesToScan" value="com.itheima.jpa.entity"**/>  *<!--配置供应商适配器-->* <**property name="jpaVendorAdapter"**>  <**bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"**>  *<!--是否打印sql语句-->* <**property name="showSql" value="true"**/>  *<!--是否自动创建表，如果表存在跳过，如果不存在自动创建-->* <**property name="generateDdl" value="true"**/>  *<!--使用的数据库的类型-->* <**property name="database" value="MYSQL"**/>  </**bean**>  </**property**>  </**bean**>  *<!-- 3.事务配置-->  <!-- 3.1JPA事务管理器 -->* <**bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"**>  <**property name="entityManagerFactory" ref="entityManagerFactory"** />  </**bean**>  *<!-- 3.2.txAdvice-->* <**tx:advice id="txAdvice" transaction-manager="transactionManager"**>  <**tx:attributes**>  <**tx:method name="save\*" propagation="REQUIRED"**/>  <**tx:method name="insert\*" propagation="REQUIRED"**/>  <**tx:method name="update\*" propagation="REQUIRED"**/>  <**tx:method name="delete\*" propagation="REQUIRED"**/>  <**tx:method name="get\*" read-only="true"**/>  <**tx:method name="find\*" read-only="true"**/>  <**tx:method name="\*" propagation="REQUIRED"**/>  </**tx:attributes**>  </**tx:advice**>  *<!-- 3.3.aop-->* <**aop:config**>  <**aop:pointcut id="pointcut" expression="execution(\* com.itheima.jpa.service.\*.\*(..))"** />  <**aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut"** />  </**aop:config**>   *<!-- 4.扫描使用spring data jpa创建的dao，创建代理对象放到spring容器中-->* <**jpa:repositories base-package="com.itheima.jpa.dao"  transaction-manager-ref="transactionManager"  entity-manager-factory-ref="entityManagerFactory"**/>  *<!--5.组装其它 配置文件-->* <**context:component-scan base-package="com.itheima.jpa"**></**context:component-scan**>   </**beans**> |

## 创建Entity实体类

可以直接使用上次课创建的Customer类

|  |
| --- |
| @Entity @Table(name = "cst\_customer") public class Customer {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  //@GeneratedValue(strategy = GenerationType.TABLE)  @Column(name="cust\_id")  private long custId;   @Column(name="cust\_name")  private String custName;   @Column(name="cust\_source")  private String custSource;   @Column(name="cust\_industry")  private String custIndustry;   @Column(name="cust\_level")  private String custLevel;   @Column(name="cust\_address")  private String custAddress;    @Column(name="cust\_phone")  private String custPhone;   public long getCustId() {  return custId;  }   public void setCustId(long custId) {  this.custId = custId;  }   public String getCustName() {  return custName;  }   public void setCustName(String custName) {  this.custName = custName;  }   public String getCustSource() {  return custSource;  }   public void setCustSource(String custSource) {  this.custSource = custSource;  }   public String getCustIndustry() {  return custIndustry;  }   public void setCustIndustry(String custIndustry) {  this.custIndustry = custIndustry;  }   public String getCustLevel() {  return custLevel;  }   public void setCustLevel(String custLevel) {  this.custLevel = custLevel;  }   public String getCustAddress() {  return custAddress;  }   public void setCustAddress(String custAddress) {  this.custAddress = custAddress;  }   public String getCustPhone() {  return custPhone;  }   public void setCustPhone(String custPhone) {  this.custPhone = custPhone;  }  @Override  public String toString() {  return "Customer{" +  "custId=" + custId +  ", custName='" + custName + '\'' +  ", custSource='" + custSource + '\'' +  ", custIndustry='" + custIndustry + '\'' +  ", custLevel='" + custLevel + '\'' +  ", custAddress='" + custAddress + '\'' +  ", custPhone='" + custPhone + '\'' +  '}';  } } |

## 编写Dao

### Spring Data JPA的Dao编写规范

1.创建一个Dao层接口，并实现JpaRepository和JpaSpecificationExecutor

2.提供相应的泛型

### Dao

|  |
| --- |
| public interface CustomerDao extends JpaRepository<Customer, Long>, JpaSpecificationExecutor<Customer> {  } |

## 测试Dao

### 创建测试类

可以使用SpringTest对Dao进行测试。创建一个测试类，并将CustomerDao注入。

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class) @ContextConfiguration("classpath:applicationContext.xml") public class SpringDataJpaTest {   @Autowired  private CustomerDao customerDao;  } |

### 根据id查询

|  |
| --- |
| @Test public void findCustomerById() {  Customer customer = customerDao.findOne(1l);  System.*out*.println(customer); } |

### 添加数据

|  |
| --- |
| @Test public void testSave() {  Customer customer = new Customer();  customer.setCustName("李逵");  customer.setCustAddress("山东");  customer.setCustIndustry("土匪");  customer.setCustLevel("vip");  //保存数据  customerDao.save(customer); } |

### 更新数据

|  |
| --- |
| @Test public void testUpdate() {  Customer customer = customerDao.findOne(1l);  customer.setCustName("武松");  customer.setCustAddress("清河县");  //更新数据  customerDao.save(customer);  } |

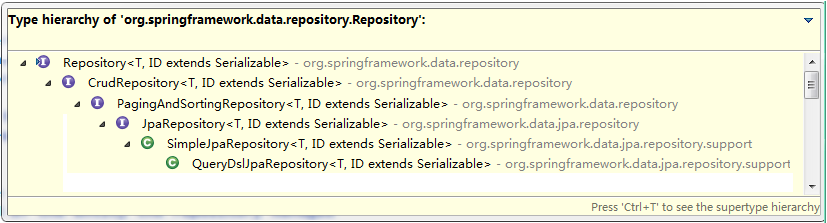
### 删除数据

|  |
| --- |
| @Test public void testDelete() {  customerDao.delete(1l); } |

# Spring Data JPA的内部原理剖析

## Spring Data JPA的常用接口分析

在客户的案例中，我们发现在自定义的CustomerDao中，并没有提供任何方法就可以使用其中的很多方法，那么这些方法究竟是怎么来的呢？答案很简单，对于我们自定义的Dao接口，由于继承了JpaRepository和JpaSpecificationExecutor，所以我们可以使用这两个接口的所有方法。



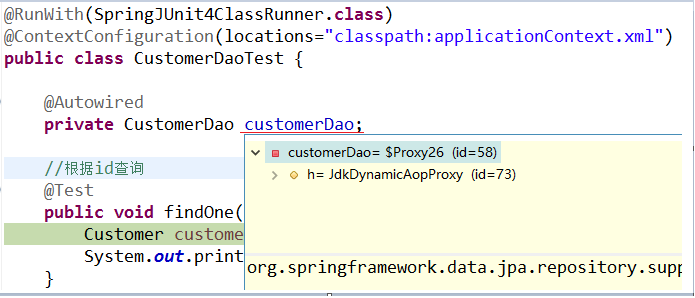
在使用Spring Data JPA时，一般实现JpaRepository和JpaSpecificationExecutor接口，这样就可以使用这些接口中定义的方法，但是这些方法都只是一些声明，没有具体的实现方式，那么在 Spring Data JPA中它又是怎么实现的呢？

## Spring Data JPA的实现过程

通过对客户案例，以debug断点调试的方式，通过分析Spring Data JPA的原来来分析程序的执行过程

我们以findOne方法为例进行分析

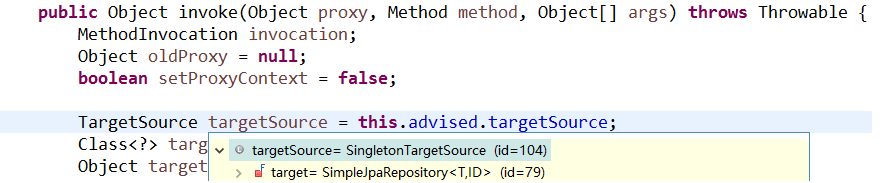
* **代理子类的实现过程**



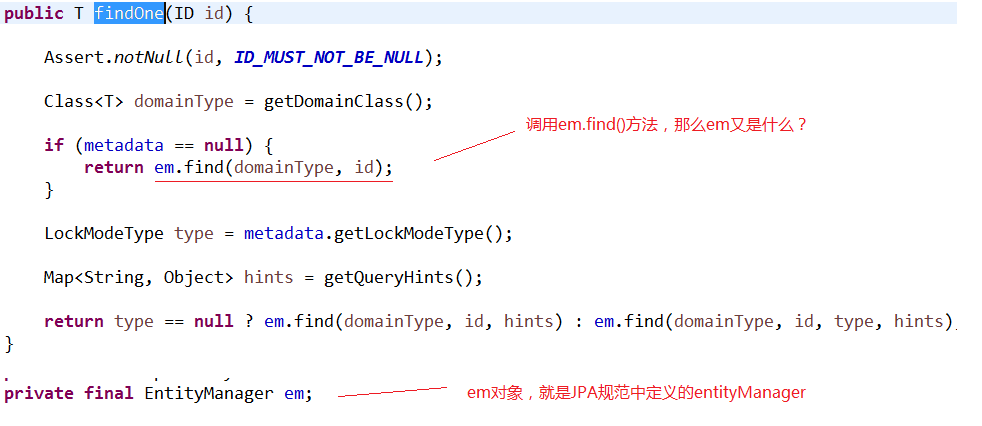
断点执行到方法上时，我们可以发现注入的customerDao对象，本质上是通过JdkDynamicAopProxy生成的一个代理对象

* **代理对象中方法调用的分析**

当程序执行的时候，会通过JdkDynamicAopProxy的invoke方法，对customerDao对象生成动态代理对象。根据对Spring Data JPA介绍而知，要想进行findOne查询方法，最终还是会出现JPA规范的API完成操作，那么这些底层代码存在于何处呢？答案很简单，都隐藏在通过JdkDynamicAopProxy生成的动态代理对象当中，而这个动态代理对象就是SimpleJpaRepository

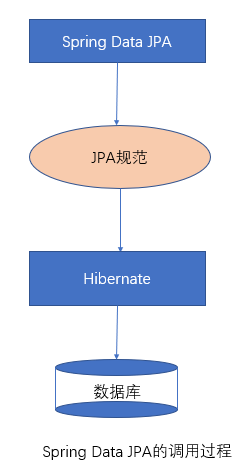


通过SimpleJpaRepository的源码分析，定位到了findOne方法，在此方法中，返回em.find()的返回结果，那么em又是什么呢？



带着问题继续查找em对象，我们发现em就是EntityManager对象，而他是JPA原生的实现方式，所以我们得到结论Spring Data JPA只是对标准JPA操作进行了进一步封装，简化了Dao层代码的开发

## Spring Data JPA完整的调用过程分析

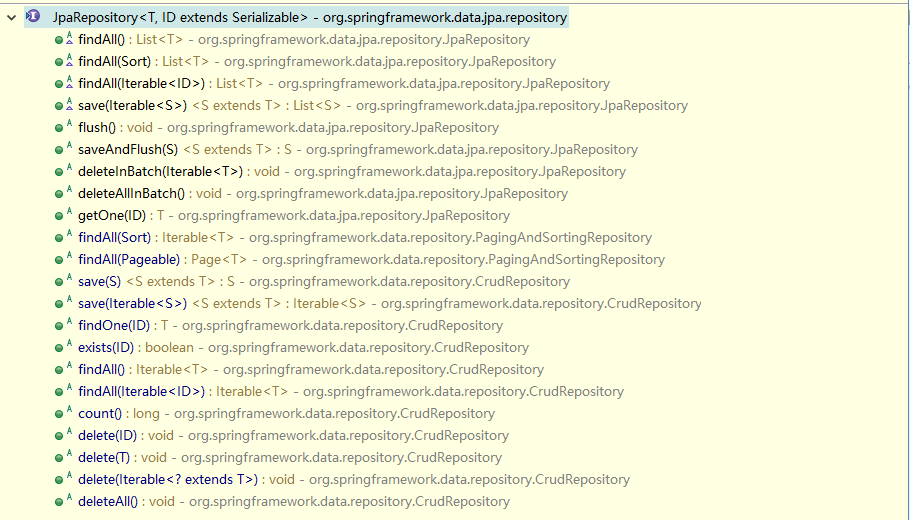


# Spring Data JPA的查询方式

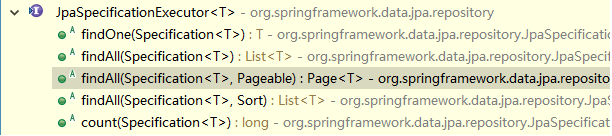
## 使用Spring Data JPA中接口定义的方法进行查询

在继承JpaRepository，和**JpaSpecificationExecutor**接口后,我们就可以使用接口中定义的方法进行查询

* **继承JpaRepository后的方法列表**



* **继承JpaSpecificationExecutor的方法列表**



### 常用方法

|  |
| --- |
| */\*\*  \* 查询所有  \*/* @Test public void testFindAll() {  List<Customer> list = customerDao.findAll();  for(Customer customer : list) {  System.*out*.println(customer);  } }  */\*\*  \* 测试统计查询：查询客户的总数量  \* count:统计总条数  \*/* @Test public void testCount() {  long count = customerDao.count();//查询全部的客户数量  System.*out*.println(count); }  */\*\*  \* 测试：判断id为4的客户是否存在  \* 1. 可以查询以下id为4的客户  \* 如果值为空，代表不存在，如果不为空，代表存在  \* 2. 判断数据库中id为4的客户的数量  \* 如果数量为0，代表不存在，如果大于0，代表存在  \*/* @Test public void testExists() {  boolean exists = customerDao.exists(4l);  System.*out*.println("id为4的客户 是否存在："+exists); }  */\*\*  \* 根据id从数据库查询  \** ***@Transactional*** *: 保证getOne正常运行  \*  \* findOne：  \* em.find() :立即加载  \* getOne：  \* em.getReference :延迟加载  \* \* 返回的是一个客户的动态代理对象  \* \* 什么时候用，什么时候查询  \*/* @Test @Transactional public void testGetOne() {  Customer customer = customerDao.getOne(2l);  System.*out*.println(customer); } |

## 使用JPQL的方式查询

使用Spring Data JPA提供的查询方法已经可以解决大部分的应用场景，但是对于某些业务来说，我们还需要灵活的构造查询条件，这时就可以使用@Query注解，结合JPQL的语句方式完成查询

@Query 注解的使用非常简单，只需在方法上面标注该注解，同时提供一个JPQL查询语句即可

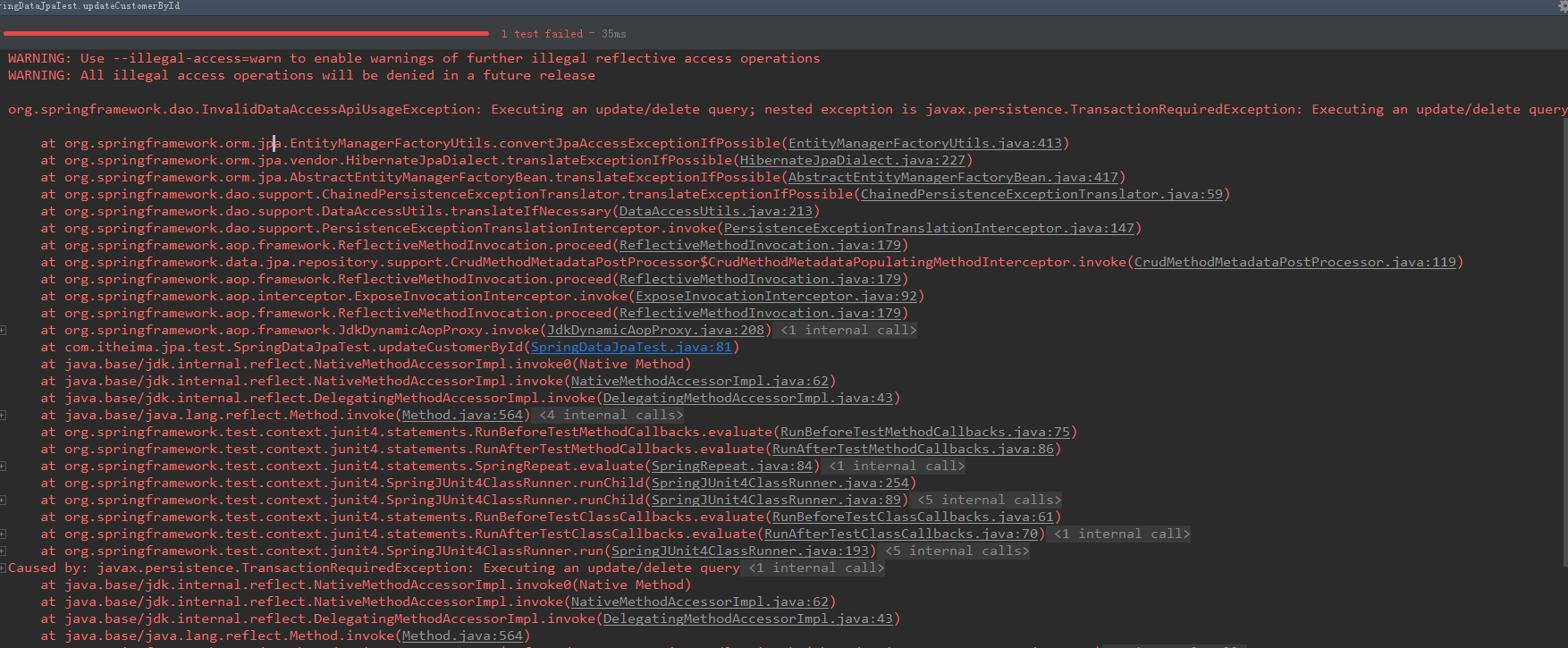
|  |
| --- |
| **public** **interface** CustomerDao **extends** JpaRepository<Customer, Long>,JpaSpecificationExecutor<Customer> {  //@Query 使用jpql的方式查询。  @Query(value="from Customer")  **public** List<Customer> findAllCustomer();    //@Query 使用jpql的方式查询。?1代表参数的占位符，其中1对应方法中的参数索引  @Query(value="from Customer where custName = ?1")  **public** Customer findCustomer(String custName);  } |

### 使用JPQL执行更新操作

也可以通过使用 @Query 来执行一个更新操作，为此，我们需要在使用 @Query 的同时，用 @Modifying 来将该操作标识为修改查询，这样框架最终会生成一个更新的操作，而非查询

|  |
| --- |
| @Query(value="update Customer set custName = ?1 where custId = ?2")  @Modifying  **public** **void** updateCustomer(String custName,Long custId); |

执行测试方法时发生如下异常：



发生此异常的原因是由于更新操作在执行是没有开启事务，所以会出现此问题。可以在此测试方法上开启事务，并且设置执行方法之后提交。

|  |
| --- |
| @Test @Transactional //添加事务 @Commit //执行之后提交 public void updateCustomerById() {   customerDao.updateCustomerNameById( 2l,"李金山"); } |

或者在dao的方法上添加@Transactional注解也可以解决此问题

|  |
| --- |
| @Query("update Customer set custName=?2 where custId=?1") @Modifying @Transactional public void updateCustomerNameById(long id, String name); |

## 使用SQL语句查询

Spring Data JPA同样也支持sql语句的查询，如下：

|  |
| --- |
| /\*\*  \* nativeQuery : 使用本地sql的方式查询  \*/  @Query(value="select \* from cst\_customer",nativeQuery=**true**)  **public** **void** findSql(); |

## 方法命名规则查询

顾名思义，方法命名规则查询就是根据方法的名字，就能创建查询。只需要按照Spring Data JPA提供的方法命名规则定义方法的名称，就可以完成查询工作。Spring Data JPA在程序执行的时候会根据方法名称进行解析，并自动生成查询语句进行查询

按照Spring Data JPA 定义的规则，查询方法以findBy开头，涉及条件查询时，条件的属性用条件关键字连接，要注意的是：条件属性首字母需大写。框架在进行方法名解析时，会先把方法名多余的前缀截取掉，然后对剩下部分进行解析。

|  |
| --- |
| //方法命名方式查询（根据客户名称查询客户）  **public** Customer findByCustName(String custName); |

具体的关键字，使用方法和生产成SQL如下表所示

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Keyword** | **Sample** | **JPQL** |  |  |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |  |  |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |  |  |
| Is,Equals | findByFirstnameIs,  findByFirstnameEquals | … where x.firstname = ?1 |  |  |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |  |  |
| LessThan | findByAgeLessThan | … where x.age < ?1 |  |  |
| LessThanEqual | findByAgeLessThanEqual | … where x.age ⇐ ?1 |  |  |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |  |  |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |  |  |
| After | findByStartDateAfter | … where x.startDate > ?1 |  |  |
| Before | findByStartDateBefore | … where x.startDate < ?1 |  |  |
| IsNull | findByAgeIsNull | … where x.age is null |  |  |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |  |  |
| Like | findByFirstnameLike | … where x.firstname like ?1 |  |  |
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |  |  |
| StartingWith | findByFirstnameStartingWith | … where x.firstname like ?1 (parameter bound with appended %) |  |  |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1 (parameter bound with prepended %) |  |  |
| Containing | findByFirstnameContaining | … where x.firstname like ?1 (parameter bound wrapped in %) |  |  |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |  |  |
| Not | findByLastnameNot | … where x.lastname <> ?1 |  |  |
| In | findByAgeIn(Collection ages) | … where x.age in ?1 |  |  |
| NotIn | findByAgeNotIn(Collection age) | … where x.age not in ?1 |  |  |
| TRUE | findByActiveTrue() | … where x.active = true |  |  |
| FALSE | findByActiveFalse() | … where x.active = false |  |  |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |  |  |

# Specifications条件查询

有时我们在查询某个实体的时候，给定的条件是不固定的，这时就需要动态构建相应的查询语句，在Spring Data JPA中可以通过JpaSpecificationExecutor接口查询。相比JPQL其优势是类型安全、更加的面向对象。当然这也要求Dao接口需要继承JpaSpecificationExecutor接口。

## JpaSpecificationExecutor接口定义

|  |
| --- |
| **import** java.util.List;  **import** org.springframework.data.domain.Page;  **import** org.springframework.data.domain.Pageable;  **import** org.springframework.data.domain.Sort;  **import** org.springframework.data.jpa.domain.Specification;  /\*\*  \* JpaSpecificationExecutor中定义的方法  \*\*/  **public** **interface** JpaSpecificationExecutor<T> {  //根据条件查询一个对象  T findOne(Specification<T> spec);  //根据条件查询集合  List<T> findAll(Specification<T> spec);  //根据条件分页查询  Page<T> findAll(Specification<T> spec, Pageable pageable);  //排序查询查询  List<T> findAll(Specification<T> spec, Sort sort);  //统计查询  **long** count(Specification<T> spec);  } |

对于JpaSpecificationExecutor，这个接口基本是围绕着Specification接口来定义的。我们可以简单的理解为，Specification构造的就是查询条件。

## Specification接口定义

|  |
| --- |
| //构造查询条件  /\*\*  \* root ：Root接口，代表查询的根对象，可以通过root获取实体中的属性  \* query ：代表一个顶层查询对象，用来自定义查询  \* cb ：用来构建查询，此对象里有很多条件方法  \*\*/  **public** Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder cb); |

此接口定义非常简单，其中只有一个方法就是toPredicate，它的作用就是生成Predicate对象，也就是查询条件。

参数说明：

1. CriteriaBuilder：用来构建CritiaQuery的构建器对象
2. CriteriaQuery：代表一个specific的顶层查询对象，它包含着查询的各个部分，比如：select 、from、where、group by、order by等注意：CriteriaQuery对象只对实体类型或嵌入式类型的Criteria查询起作用。通过调用CriteriaBuilder的createQuery或createTupleQuery方法可以获得CriteriaQuery的实例
3. Root：代表Criteria查询的根对象，Criteria查询的查询根定义了实体类型，能为将来导航获得想要的结果，它与SQL查询中的FROM子句类似。可以通过调用CriteriaQuery的from方法可以获得Root实例

返回值：

Predicate：一个简单或复杂的谓词类型，其实就相当于条件或者是条件组合。

可以简单理解为：

CriteriaQuery：sql语句中的各种关键字。例如wherer、order by、group by、having等

Root：设置from

CriteriaBuilder：设置具体查询条件

## 使用Specifications查询

### 测试步骤

1. 创建一个Dao实现JpaSpecificationExecutor接口，可以直接使用上次课创建的dao
2. 编写测试类，在测试类中创建一个测试方法，并将Dao代理对象注入到测试类中
3. 使用dao的findAll方法，此方法需要一个Specification对象
4. 创建一个Specification接口的匿名内部类，在其中创建一个Predicate 对象
5. 执行查询，并打印结果

### Dao

|  |
| --- |
| **public interface** CustomerDao **extends** JpaRepository<Customer, Long>, JpaSpecificationExecutor<Customer> {  } |

### 根据id查询

|  |
| --- |
| @Test **public void** testSpecification() {  Specification specification = **new** Specification() {  @Override  **public** Predicate toPredicate(Root root, CriteriaQuery criteriaQuery, CriteriaBuilder criteriaBuilder) {  **return** criteriaBuilder.equal(root.get(**"custId"**).as(Long.**class**), 2l);  }  };  Customer customer = **customerDao**.findOne(specification);  System.***out***.println(customer);  } |

### 模糊查询

|  |
| --- |
| @Test **public void** testSpecificationLike() {  **customerDao**.findAll((root, query, cb)->cb.like(root.get(**"custName"**).as(String.**class**), **"张%"**))  .stream()  .forEach(c->{  System.***out***.println(c);  }); } |

### 组合条件查询

|  |
| --- |
| @Test **public void** testSpecificationComboCriteria() {  **customerDao**.findAll((root, query, cb)->{  Predicate p1 = cb.gt(root.get(**"custId"**).as(Long.**class**), 3l);  Predicate p2 = cb.like(root.get(**"custName"**).as(String.**class**), **"宋%"**);  **return** cb.and(p1,p2);  }).stream().forEach(c-> System.***out***.println(c)); } |

### 结果排序

|  |
| --- |
| @Test **public void** testSpecificationSort() {  **customerDao**.findAll((root, query, cb)->{  Predicate predicate = cb.like(root.get(**"custName"**).as(String.**class**), **"宋%"**);  **return** query.where(predicate)  .orderBy(cb.desc(root.get(**"custId"**).as(Long.**class**)))  .getRestriction();  }).stream().forEach(c-> System.***out***.println(c)); } |

### 分页查询

|  |
| --- |
| @Test **public void** testSpecificationPage() {  Pageable pageable = **new** PageRequest(0,5);  Page<Customer> result = **customerDao**.findAll((root, query, cb) -> {  **return** cb.like(root.get(**"custName"**).as(String.**class**), **"%李%"**);  }, pageable);   System.***out***.println(**"总页数："** + result.getTotalPages());  System.***out***.println(**"总记录数："** + result.getTotalElements());  result.getContent()  .stream()  .forEach(c->{  System.***out***.println(c);  }); } |

## CriteriaBuilder方法对应关系

|  |  |
| --- | --- |
| 方法名称 | Sql对应关系 |
| equle | filed = value |
| gt（greaterThan ） | filed > value |
| lt（lessThan ） | filed < value |
| ge（greaterThanOrEqualTo ） | filed >= value |
| le（ lessThanOrEqualTo） | filed <= value |
| notEqule | filed != value |
| like | filed like value |
| notLike | filed not like value |
|  |  |