# 01-今日课程安排(了解) 03:58

|  |
| --- |
| * springDataJpa，jpa规范和hibernate之间的关系说明 * Maven构建 springDataJpa项目及配置文件编写 * springDataJPA入门案例增删改查 * 使用方式 * 原始sql语句使用方式 * springDataJpa方法命名规则 |

# 02-springDataJpa介绍 (了解)05:17

|  |
| --- |
| SpringData , SpringDataJPA: 是两个概念大家不要弄混了 |
| SpringData 是Spring访问数据库的所有框架的总称包含如下主要模块:    可以看到,SpringDataJPA 是SpringData 的一个模块 |

|  |
| --- |
|  |
|  |

# 03-springDataJpa，jpa规范和hibernate之间的关系说明(了解)

|  |
| --- |
|  |
|  |

# 04-入门案例和分析步骤的说明(熟悉) 03:48

|  |
| --- |
| 案例：客户的基本CRUD  i.搭建环境  创建工程导入坐标  配置spring的配置文件（配置spring Data jpa的整合）  编写实体类（Customer），使用jpa注解配置映射关系  ii.编写一个符合springDataJpa的dao层接口 |

# 05-入门案例：创建maven工程导入坐标 05:07(掌握)

|  |
| --- |
|  |
| pom.xml maven坐标导入  spring核心  spring AOP  spring IOC  spring ORM  springDataJPA  hibernate  mysql驱动  log4j日志 |
| <properties>  <spring.version>5.0.2.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>  </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>${spring.version}</version>  </dependency>  <!-- spring对orm框架的支持包-->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>${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>${mysql.version}</version>  </dependency>  <!-- spring data jpa 的坐标-->  <dependency>  <groupId>org.springframework.data</groupId>  <artifactId>spring-data-jpa</artifactId>  <version>1.9.0.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>${spring.version}</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 -->  </dependencies> |

# 06-SpringDataJpa配置-1 10:46(掌握)

|  |
| --- |
| <?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"> |
| Spring 约束中我们要注意新导入了一个约束文件 aop |
| *<!-- 1.创建entityManagerFactory对象交给spring容器管理-->* <**bean id="entityManagerFactoty" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"**>  <**property name="dataSource" ref="dataSource"** />  *<!--配置的扫描的包（实体类所在的包） -->* <**property name="packagesToScan" value="cn.itcast.domain"** />  *<!-- jpa的实现厂家 -->* <**property name="persistenceProvider"**>  <**bean class="org.hibernate.jpa.HibernatePersistenceProvider"**/>  </**property**>   *<!--jpa的供应商适配器 -->* <**property name="jpaVendorAdapter"**>  <**bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"**>  *<!--配置是否自动创建数据库表 -->* <**property name="generateDdl" value="false"** />  *<!--指定数据库类型 -->* <**property name="database" value="MYSQL"** />  *<!--数据库方言：支持的特有语法 -->* <**property name="databasePlatform" value="org.hibernate.dialect.MySQLDialect"** />  *<!--是否显示sql -->* <**property name="showSql" value="true"** />  </**bean**>  </**property**> |

# 07-springDataJpa配置-2 09:32(掌握)

|  |
| --- |
| <!—1 jpa的方言 ：高级的特性 -->  <**property name="jpaDialect"**>  <**bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"**></**bean**> </**property**>  <!--2.创建数据库连接池 -->  <**bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>  <**property name="jdbcUrl" value="jdbc:mysql:///jpa"**/>  <**property name="driverClass" value="com.mysql.jdbc.Driver"**/>  <**property name="password" value="root"**/>  <**property name="user" value="root"**/> </**bean**>  *<!--3整合Spring jpa -->* <**jpa:repositories base-package="com.itheima.dao" transaction-manager-ref="transactionManager"  entity-manager-factory-ref="entityManagerFactory"**/> <**bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"**>  <**property name="entityManagerFactory" ref="entityManagerFactory"**/> </**bean**> |
| *<!--4.配置事务管理器 -->* <**bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"**>  <**property name="entityManagerFactory" ref="entityManagerFactoty"**></**property**> </**bean**> |
|  |

# 08-springDataJpa配置-3 02:07(掌握)

|  |
| --- |
| *<!-- 4.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**>  *<!-- 5.aop-->* <**aop:config**>  <**aop:pointcut id="pointcut" expression="execution(\* cn.itcast.service.\*.\*(..))"** />  <**aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut"** /> </**aop:config**>   *<!--5.声明式事务 -->  <!-- 6. 配置包扫描-->* <**context:component-scan base-package="cn.itcast"** ></**context:component-scan**> |

# 09-入门案例：创建客户实体类并配置映射关系 05:43(掌握)

|  |
| --- |
| *\*\*  \* 1.实体类和表的映射关系  \** ***@Eitity*** *\** ***@Table*** *\* 2.类中属性和表中字段的映射关系  \** ***@Id*** *\** ***@GeneratedValue*** *\** ***@Column*** *\*/* @Entity @Table(name=**"cst\_customer"**) **public class** Customer {   @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  @Column(name=**"cust\_id"**)  **private** Long **custId**;  @Column(name=**"cust\_address"**)  **private** String **custAddress**;  @Column(name=**"cust\_industry"**)  **private** String **custIndustry**;  @Column(name=**"cust\_level"**)  **private** String **custLevel**;  @Column(name=**"cust\_name"**)  **private** String **custName**;  @Column(name=**"cust\_phone"**)  **private** String **custPhone**;  @Column(name=**"cust\_source"**)  **private** String **custSource**; |
| *注意: 不要导错包* |

# 10-入门案例：编写符合spring Data JPA规范的dao层接口 07:33(掌握)

|  |
| --- |
| */\*\*  \* 符合SpringDataJpa的dao层接口规范  \* JpaRepository<操作的实体类类型，实体类中主键属性的类型>  \* \* 封装了基本CRUD操作  \* JpaSpecificationExecutor<操作的实体类类型>  \* \* 封装了复杂查询（分页）  \*/* **public interface** CustomerDao **extends** JpaRepository<Customer,Long> ,JpaSpecificationExecutor<Customer> { |
| *回顾知识点: 接口可以多继承* |

# 11-入门案例：完成客户的查询操作 05:07(掌握)

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) *//声明spring提供的单元测试环境* @ContextConfiguration(locations = **"classpath:applicationContext.xml"**)*//指定spring容器的配置信息* **public class** CustomerDaoTest {  @Autowired  **private** CustomerDao **customerDao**;   */\*\*  \* 根据id查询  \*/* @Test  **public void** testFindOne() {  Customer customer = **customerDao**.findOne(4l);  System.***out***.println(customer);  } |

# 12-入门案例：完成客户的添加和修改 08:05(掌握)

|  |
| --- |
| */\*\*  \* save : 保存或者更新  \* 根据传递的对象是否存在主键id，  \* 如果没有id主键属性：保存  \* 存在id主键属性，根据id查询数据，更新数据  \*/* @Test **public void** testSave() {  Customer customer = **new** Customer();  customer.setCustName(**"黑马程序员"**);  customer.setCustLevel(**"vip"**);  customer.setCustIndustry(**"it教育"**);  **customerDao**.save(customer); }  @Test **public void** testUpdate() {  Customer customer = **new** Customer();  customer.setCustId(4l);  customer.setCustName(**"黑马程序员很厉害"**);  **customerDao**.save(customer); } |
| *注意:*  *1更新方法 会按照主键更新*  *2 更新方法会更新对象的所有字段*  *3 实际开发中会先查询后修改要更新的字段,然后再更新 如下:*  @Test **public void** testUpdate() {  *// 1)先查询,* Customer customer = **customerDao**.findOne(4l);  *//2) 修改需要变更的字段更新* customer.setCustName(**"黑马程序员"**);  **customerDao**.save(customer); } |

# 13-入门案例：完成客户的删除 01:52(掌握)

|  |
| --- |
| @Test **public void** testDelete () {  **customerDao**.delete(3l); }  注意:  这里后台是先查询,后删除 |
|  |

# 14-入门案例：查询所有客户 02:41(掌握)

|  |
| --- |
| */\*\*  \* 查询所有  \*/* @Test **public void** testFindAll() {  List<Customer> list = **customerDao**.findAll();  **for**(Customer customer : list) {  System.***out***.println(customer);  } } |

# 15-入门案例：总结 04:33(掌握)

|  |
| --- |
| 🡪findOne（id） ：根据id查询  🡪save(customer):保存或者更新（依据：传递的实体类对象中，是否包含id属性）  🡪delete（id） ：根据id删除  🡪findAll() : 查询全部 |

# 16-执行过程：分析 03:58(理解)

|  |
| --- |
|  |

# 17-执行过程：源码分析 07:54(理解)

|  |
| --- |
| 1)SpringdataJPA地城是创建了Dao 代理对象,  2)SpringdataJPA 底层依旧是封装了EntityManager 对象进行数据库操作 |

# 18-执行过程：执行过程已经内部处理流程的总结 08:02(理解)

|  |
| --- |
| 1.通过JdkDynamicAopProxy的invoke方法创建了一个动态代理对象  2.SimpleJpaRepository当中封装了JPA的操作（借助JPA的api完成数据库的CRUD）  3.通过hibernate完成数据库操作（封装了jdbc）  请结合[03-springDataJpa，jpa规范和hibernate之间的关系说明(了解)](#_03-springDataJpa，jpa规范和hibernate之间的) 理解本节内容 |

# 19-spring Data JPA查询：调用接口方法查询（count，exists） 08:39(掌握)

|  |
| --- |
| */\*\*  \* 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*  exists 方法*使用的是第二种方式  \*/* @Test **public void** testExists() {  **boolean** exists = **customerDao**.exists(4l);  System.***out***.println(**"id为4的客户 是否存在："**+exists); } |

# 20-spring Data JPA查询：调用接口方法查询（findOne和getOne的区别） 05:52(掌握)

|  |
| --- |
| */\*\*  \* 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(4l);  System.***out***.println(customer); }  注意: getOne 方法需要 事务的支持 |
| *问: 这里的方法为什么需要一个事务?*  *Hibernate 需要一个session 的支持,事务可以开启一个session* |
| *<!-- 指定jpa属性，实现厂商配置的属性，此处为hibernate的单独配置属性 -->*  *<property name="jpaPropertyMap">*  *<map>*  *<entry key="hibernate.enable\_lazy\_load\_no\_trans" value="true"/>*  *</map>*  *</property>*  *注意:*  这种情形下，每次初始化一个实体的关联就会创建一个临时的session来加载，每个临时的session都会获取一个临时的数据库连接，开启一个新的事物。这就导致对底层连接池压力很大，而且事物日志也会被每次flush. 设想一下：假如我们查询了一个分页list每次查出1000条，这个实体有三个lazy关联对象,那么，恭喜你，你至少需要创建3000个临时session+connection+transaction. |

# 21-spring Data JPA查询：JPQL查询引入 04:49(理解)

|  |
| --- |
| jpql的查询方式  jpql ： jpa query language （jpq查询语言）  特点：语法或关键字和sql语句类似  查询的是类和类中的属性    需要将JPQL语句配置到接口方法上  1.特有的查询：需要在dao接口上配置方法  2.在新添加的方法上，使用注解的形式配置jpql查询语句  3.注解 ： @Query() |

# 22-jpql：使用jpql完成基本查询 05:03(掌握)

|  |
| --- |
| */\*\*  \* 案例：根据客户名称查询客户  \* 使用jpql的形式查询  \* jpql：from Customer where custName = ?  \*  \* 配置jpql语句，使用的@Query注解  \*/* @Query(value=**"from Customer where custName = ?"**) **public** Customer findJpql(String custName);  @Test **public void** testFindJPQL() {  Customer customer = **customerDao**.findJpql(**"传智播客"**);  System.***out***.println(customer); } |

# 23-jpql：多占位符的赋值 11:24(掌握)

|  |
| --- |
| */\*\*  \* 案例：根据客户名称和客户id查询客户  \* jpql： from Customer where custName = ? and custId = ?  \*  \* 对于多个占位符参数  \* 赋值的时候，默认的情况下，占位符的位置需要和方法参数中的位置保持一致  \*  \* 可以指定占位符参数的位置  \* ? 索引的方式，指定此占位的取值来源  \*/* @Query(value = **"from Customer where custName = ?2 and custId = ?1"**) **public** Customer findCustNameAndId(Long id, String name);  @Query(value = **" update Customer set custName = :name where custId = :id "**) @Modifying @Transactional **void** updateByJpql(@Param(**"id"**) Long id, @Param(**"name"**) String name); |
| @Test **public void** testFindCustNameAndId() {  *// Customer customer = customerDao.findCustNameAndId("传智播客",1l);* Customer customer = **customerDao**.findCustNameAndId(1l,**"传智播客"**);  System.***out***.println(customer); } |

# 24-jpql：使用jpql完成更新操作 10:54(掌握)

注意:

1 Dao方法上需要增加***@Modifying 注解***

2 测试的方法上需要增加事务的支持

3 测试方法上要增加不自动回滚的注解

|  |
| --- |
| */\*\*  \* 使用jpql完成更新操作  \* 案例 ： 根据id更新，客户的名称  \* 更新4号客户的名称，将名称改为“黑马程序员”  \*  \* sql ：update cst\_customer set cust\_name = ? where cust\_id = ?  \* jpql : update Customer set custName = ? where custId = ?  \*  \** ***@Query*** *: 代表的是进行查询  \* \* 声明此方法是用来进行更新操作  \** ***@Modifying*** *\* \* 当前执行的是一个更新操作*  *\*  \*/* @Query(value = **" update Customer set custName = ?2 where custId = ?1 "**) @Modifying **public void** updateCustomer(**long** custId, String custName); |
| */\**  *@Transactional 用在Junit 方法上*  *\* 默认会执行结束之后，回滚事务*  *\* 会覆盖方法上的事务回滚策略*  *\* @Rollback : 设置是否自动回滚*  *\* false | true*  *\* 专门给spring-test 中的测试类方法 提供的*  *\*/* @Test @Transactional *//添加事务的支持* @Rollback(value = **false**) **public void** testUpdateCustomer() {  **customerDao**.updateCustomer(4l,**"黑马程序员"**); } |
|  |

# 25-sql查询：查询全部 07:38(掌握)

|  |
| --- |
| */\*\*  \* 使用sql的形式查询：  \* 查询全部的客户  \* sql ： select \* from cst\_customer;  \* Query : 配置sql查询  \* value ： sql语句  \* nativeQuery ： 查询方式  \* true ： sql查询  \* false：jpql查询  \*  \*/ //@Query(value = " select \* from cst\_customer" ,nativeQuery = true)* @Query(value=**"select** *\** **from cst\_customer "**,nativeQuery = **true**) **public** List<Object [] > findSql(String name); |
| *//测试sql查询* @Test **public void** testFindSql() {  List<Object[]> list = **customerDao**.findSql();  **for**(Object [] obj : list) {  System.***out***.println(Arrays.*toString*(obj));  } } |
| *1)查询返回值是一个数组*  *2)我们可以写成对象的形式*  *2.1) 如果返回只是对象本身,sql 语句是查询字段而不是\* 则需要写全部字段名(没有顺序要求)* |
| *//@Query(value = " select \* from cst\_customer" ,nativeQuery = true)* @Query(value=**"select cust\_name,cust\_id ,cust\_address,cust\_industry,cust\_level,cust\_phone,cust\_source from cst\_customer where cust\_name like ?1"**,nativeQuery = **true**) **public** List<Customer > findSql(String name); |

# 26-sql查询：条件全部 03:29(掌握)

|  |
| --- |
| *//nativeQuery ： 查询方式 // true ： sql查询 // false：jpql查询*  *//@Query(value = " select \* from cst\_customer" ,nativeQuery = true)* @Query(value=**"select** *\** **from cst\_customer where cust\_name like ?1"**,nativeQuery = **true**) **public** List<Object [] > findSql(String name); |
| *//测试sql查询* @Test **public void** testFindSql() {  List<Object[]> list = **customerDao**.findSql(**"传智播客%"**);  **for**(Object [] obj : list) {  System.***out***.println(Arrays.*toString*(obj));  } } |

# 27-方法命名规则查询：基本查询 10:06(掌握)

注:

方法命名规则查询 是对JPQL的进一步封装

*1.findBy + 属性名称(首字母大写）(默认是等于) （根据属性名称进行完成匹配的查询=）*

# 28-方法命名规则查询：模糊匹配 04:55(掌握)

*2.findBy + 属性名称 + “查询方式（Like | isnull）”*

# 29-方法命名规则查询：多条件查询 07:43(掌握)

|  |
| --- |
| */\*\*  \* 方法名的约定：  \* findBy : 查询  \* 对象中的属性名（首字母大写） ： 查询的条件  \* CustName  \* \* 默认情况 ： 使用 等于的方式查询  \* 特殊的查询方式  \*  \* findByCustName -- 根据客户名称查询  \*  \* 再springdataJpa的运行阶段  \* 会根据方法名称进行解析 findBy from xxx(实体类)  \* 属性名称 where custName =  \*  \* 1.findBy + 属性名称 （根据属性名称进行完成匹配的查询=）  \* 2.findBy + 属性名称 + “查询方式（Like | isnull）”  \* findByCustNameLike  \* 3.多条件查询  \* findBy + 属性名 + “查询方式” + “多条件的连接符（and|or）” + 属性名 + “查询方式”  \*/* **public** Customer findByCustName(String custName); **public** List<Customer> findByCustNameLike(String custName); *//使用客户名称模糊匹配和客户所属行业精准匹配的查询* **public** Customer findByCustNameLikeAndCustIndustry(String custName, String custIndustry); |

JPQL

稍微复杂,但是比较灵活

方法命名规则查询

虽然简单,但是复杂查询不支持

补充:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 关键字 | 方法命名 | 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 < ? | | LessThanEquals | findByIdLessThanEquals | where id <= ? | | GreaterThan | findByIdGreaterThan | where id > ? | | GreaterThanEquals | 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 '%?' | |