本篇进行Spring-data-jpa的介绍，几乎涵盖该框架的所有方面，在日常的开发当中，基本上能满足所有需求。这里不讲解JPA和Spring-data-jpa单独使用，所有的内容都是在和Spring整合的环境中实现。如果需要了解该框架的入门，百度一下，很多入门的介绍。在这篇文章的接下来一篇，会有一个系列来讲解mybatis，这个系列从mybatis的入门开始，到基本使用，和spring整合，和第三方插件整合，缓存，插件，最后会持续到mybatis的架构，源码解释，重点会介绍几个重要的设计模式，这样一个体系。基本上讲完之后，mybatis在你面前就没有了秘密，你能解决mybatis的几乎所有问题，并且在开发过程中相当的方便，驾轻就熟。

这篇文章由于介绍的类容很全，因此很长，如果你需要，那么可以耐心的看完，本人经历了很长时间的学识，使用，研究的心血浓缩成为这么短短的一篇博客。

大致整理一个提纲：

　　1、Spring-data-jpa的基本介绍；

　　2、和Spring整合；

　　3、基本的使用方式；

　　4、复杂查询，包括多表关联，分页，排序等；

现在开始：

　　1、Spring-data-jpa的基本介绍：JPA诞生的缘由是为了整合第三方ORM框架，建立一种标准的方式，百度百科说是JDK为了实现ORM的天下归一，目前也是在按照这个方向发展，但是还没能完全实现。在ORM框架中，Hibernate是一支很大的部队，使用很广泛，也很方便，能力也很强，同时Hibernate也是和JPA整合的比较良好，我们可以认为JPA是标准，事实上也是，JPA几乎都是接口，实现都是Hibernate在做，宏观上面看，在JPA的统一之下Hibernate很良好的运行。

　　上面阐述了JPA和Hibernate的关系，那么Spring-data-jpa又是个什么东西呢？这地方需要稍微解释一下，我们做Java开发的都知道Spring的强大，到目前为止，企业级应用Spring几乎是无所不能，无所不在，已经是事实上的标准了，企业级应用不使用Spring的几乎没有，这样说没错吧。而Spring整合第三方框架的能力又很强，他要做的不仅仅是个最早的IOC容器这么简单一回事，现在Spring涉及的方面太广，主要是体现在和第三方工具的整合上。而在与第三方整合这方面，Spring做了持久化这一块的工作，我个人的感觉是Spring希望把持久化这块内容也拿下。于是就有了Spring-data-\*\*这一系列包。包括，Spring-data-jpa,Spring-data-template,Spring-data-mongodb,Spring-data-redis，还有个民间产品，mybatis-spring，和前面类似，这是和mybatis整合的第三方包，这些都是干的持久化工具干的事儿。

　　这里介绍Spring-data-jpa，表示与jpa的整合。

　　2、我们都知道，在使用持久化工具的时候，一般都有一个对象来操作数据库，在原生的Hibernate中叫做Session，在JPA中叫做EntityManager，在MyBatis中叫做SqlSession，通过这个对象来操作数据库。我们一般按照三层结构来看的话，Service层做业务逻辑处理，Dao层和数据库打交道，在Dao中，就存在着上面的对象。那么ORM框架本身提供的功能有什么呢？答案是基本的CRUD，所有的基础CRUD框架都提供，我们使用起来感觉很方便，很给力，业务逻辑层面的处理ORM是没有提供的，如果使用原生的框架，业务逻辑代码我们一般会自定义，会自己去写SQL语句，然后执行。在这个时候，Spring-data-jpa的威力就体现出来了，ORM提供的能力他都提供，ORM框架没有提供的业务逻辑功能Spring-data-jpa也提供，全方位的解决用户的需求。使用Spring-data-jpa进行开发的过程中，常用的功能，我们几乎不需要写一条sql语句，至少在我看来，企业级应用基本上可以不用写任何一条sql，当然spring-data-jpa也提供自己写sql的方式，这个就看个人怎么选择，都可以。我觉得都行。

　　2.1与Spring整合我们从spring配置文件开始，为了节省篇幅，这里我只写出配置文件的结构。

复制代码

<?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:p="http://www.springframework.org/schema/p"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mongo="http://www.springframework.org/schema/data/mongo"

xmlns:jpa="http://www.springframework.org/schema/data/jpa"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/data/mongo

http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd

http://www.springframework.org/schema/data/jpa http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

<!-- 数据库连接 -->

<context:property-placeholder location="classpath:your-config.properties" ignore-unresolvable="true" />

<!-- service包 -->

<context:component-scan base-package="your service package" />

<!-- 使用cglib进行动态代理 -->

<aop:aspectj-autoproxy proxy-target-class="true" />

<!-- 支持注解方式声明式事务 -->

<tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true" />

<!-- dao -->

<jpa:repositories base-package="your dao package" repository-impl-postfix="Impl" entity-manager-factory-ref="entityManagerFactory" transaction-manager-ref="transactionManager" />

<!-- 实体管理器 -->

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="packagesToScan" value="your entity package" />

<property name="persistenceProvider">

<bean class="org.hibernate.ejb.HibernatePersistence" />

</property>

<property name="jpaVendorAdapter">

<bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

<property name="generateDdl" value="false" />

<property name="database" value="MYSQL" />

<property name="databasePlatform" value="org.hibernate.dialect.MySQL5InnoDBDialect" />

<!-- <property name="showSql" value="true" /> -->

</bean>

</property>

<property name="jpaDialect">

<bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect" />

</property>

<property name="jpaPropertyMap">

<map>

<entry key="hibernate.query.substitutions" value="true 1, false 0" />

<entry key="hibernate.default\_batch\_fetch\_size" value="16" />

<entry key="hibernate.max\_fetch\_depth" value="2" />

<entry key="hibernate.generate\_statistics" value="true" />

<entry key="hibernate.bytecode.use\_reflection\_optimizer" value="true" />

<entry key="hibernate.cache.use\_second\_level\_cache" value="false" />

<entry key="hibernate.cache.use\_query\_cache" value="false" />

</map>

</property>

</bean>

<!-- 事务管理器 -->

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

<property name="entityManagerFactory" ref="entityManagerFactory"/>

</bean>

<!-- 数据源 -->

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close">

<property name="driverClassName" value="${driver}" />

<property name="url" value="${url}" />

<property name="username" value="${userName}" />

<property name="password" value="${password}" />

<property name="initialSize" value="${druid.initialSize}" />

<property name="maxActive" value="${druid.maxActive}" />

<property name="maxIdle" value="${druid.maxIdle}" />

<property name="minIdle" value="${druid.minIdle}" />

<property name="maxWait" value="${druid.maxWait}" />

<property name="removeAbandoned" value="${druid.removeAbandoned}" />

<property name="removeAbandonedTimeout" value="${druid.removeAbandonedTimeout}" />

<property name="timeBetweenEvictionRunsMillis" value="${druid.timeBetweenEvictionRunsMillis}" />

<property name="minEvictableIdleTimeMillis" value="${druid.minEvictableIdleTimeMillis}" />

<property name="validationQuery" value="${druid.validationQuery}" />

<property name="testWhileIdle" value="${druid.testWhileIdle}" />

<property name="testOnBorrow" value="${druid.testOnBorrow}" />

<property name="testOnReturn" value="${druid.testOnReturn}" />

<property name="poolPreparedStatements" value="${druid.poolPreparedStatements}" />

<property name="maxPoolPreparedStatementPerConnectionSize" value="${druid.maxPoolPreparedStatementPerConnectionSize}" />

<property name="filters" value="${druid.filters}" />

</bean>

<!-- 事务 -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="\*" />

<tx:method name="get\*" read-only="true" />

<tx:method name="find\*" read-only="true" />

<tx:method name="select\*" read-only="true" />

<tx:method name="delete\*" propagation="REQUIRED" />

<tx:method name="update\*" propagation="REQUIRED" />

<tx:method name="add\*" propagation="REQUIRED" />

<tx:method name="insert\*" propagation="REQUIRED" />

</tx:attributes>

</tx:advice>

<!-- 事务入口 -->

<aop:config>

<aop:pointcut id="allServiceMethod" expression="execution(\* your service implements package.\*.\*(..))" />

<aop:advisor pointcut-ref="allServiceMethod" advice-ref="txAdvice" />

</aop:config>

</beans>

复制代码

　　2.2对上面的配置文件进行简单的解释，只对“实体管理器”和“dao”进行解释，其他的配置在任何地方都差不太多。

　　　　1.对“实体管理器”解释：我们知道原生的jpa的配置信息是必须放在META-INF目录下面的，并且名字必须叫做persistence.xml，这个叫做persistence-unit，就叫做持久化单元，放在这下面我们感觉不方便，不好，于是Spring提供了

org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean

这样一个类，可以让你的随心所欲的起这个配置文件的名字，也可以随心所欲的修改这个文件的位置，只需要在这里指向这个位置就行。然而更加方便的做法是，直接把配置信息就写在这里更好，于是就有了这实体管理器这个bean。使用

<property name="packagesToScan" value="your entity package" />

这个属性来加载我们的entity。

　　2.3 解释“dao”这个bean。这里衍生一下，进行一下名词解释，我们知道dao这个层叫做Data Access Object，数据库访问对象，这是一个广泛的词语，在jpa当中，我们还有一个词语叫做Repository，这里我们一般就用Repository结尾来表示这个dao，比如UserDao，这里我们使用UserRepository，当然名字无所谓，随意取，你可以意会一下我的意思，感受一下这里的含义和区别，同理，在mybatis中我们一般也不叫dao，mybatis由于使用xml映射文件（当然也提供注解，但是官方文档上面表示在有些地方，比如多表的复杂查询方面，注解还是无解，只能xml），我们一般使用mapper结尾，比如我们也不叫UserDao，而叫UserMapper。

　　上面拓展了一下关于dao的解释，那么这里的这个配置信息是什么意思呢？首先base-package属性，代表你的Repository接口的位置，repository-impl-postfix属性代表接口的实现类的后缀结尾字符，比如我们的UserRepository，那么他的实现类就叫做UserRepositoryImpl，和我们平时的使用习惯完全一致，于此同时，spring-data-jpa的习惯是接口和实现类都需要放在同一个包里面（不知道有没有其他方式能分开放，这不是重点，放在一起也无所谓，影响不大），再次的，这里我们的UserRepositoryImpl这个类的定义的时候我们不需要去指定实现UserRepository接口，根据spring-data-jpa自动就能判断二者的关系。

　　比如：我们的UserRepository和UserRepositoryImpl这两个类就像下面这样来写。

public interface UserRepository extends JpaRepository<User, Integer>{}

public class UserRepositoryImpl {}

　　那么这里为什么要这么做呢？原因是：spring-data-jpa提供基础的CRUD工作，同时也提供业务逻辑的功能（前面说了，这是该框架的威力所在），所以我们的Repository接口要做两项工作，继承spring-data-jpa提供的基础CRUD功能的接口，比如JpaRepository接口，同时自己还需要在UserRepository这个接口中定义自己的方法，那么导致的结局就是UserRepository这个接口中有很多的方法，那么如果我们的UserRepositoryImpl实现了UserRepository接口，导致的后果就是我们势必需要重写里面的所有方法，这是Java语法的规定，如此一来，悲剧就产生了，UserRepositoryImpl里面我们有很多的@Override方法，这显然是不行的，结论就是，这里我们不用去写implements部分。

　　spring-data-jpa实现了上面的能力，那他是怎么实现的呢？这里我们通过源代码的方式来呈现他的来龙去脉，这个过程中cglib发挥了杰出的作用。

　　在spring-data-jpa内部，有一个类，叫做

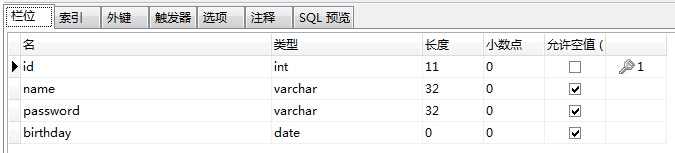
public class SimpleJpaRepository<T, ID extends Serializable> implements JpaRepository<T, ID>,

JpaSpecificationExecutor<T>

我们可以看到这个类是实现了JpaRepository接口的，事实上如果我们按照上面的配置，在同一个包下面有UserRepository，但是没有UserRepositoryImpl这个类的话，在运行时期UserRepository这个接口的实现就是上面的SimpleJpaRepository这个接口。而如果有UserRepositoryImpl这个文件的话，那么UserRepository的实现类就是UserRepositoryImpl，而UserRepositoryImpl这个类又是SimpleJpaRepository的子类，如此一来就很好的解决了上面的这个不用写implements的问题。我们通过阅读这个类的源代码可以发现，里面包装了entityManager，底层的调用关系还是entityManager在进行CRUD。

　　3. 下面我们通过一个完整的项目来基本使用spring-data-jpa，然后我们在介绍他的高级用法。

　　a.数据库建表：user,主键自增



　　b.对应实体：User

复制代码

@Entity

@Table(name = "user")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer id;

private String name;

private String password;

private String birthday;

// getter,setter

}

复制代码

　　c.简历UserRepository接口

public interface UserRepository extends JpaRepository<User, Integer>{}

　　通过上面3步，所有的工作就做完了，User的基础CRUD都能做了，简约而不简单。

　　d.我们的测试类UserRepositoryTest

复制代码

public class UserRepositoryTest {

@Autowired

private UserRepository userRepository;

@Test

public void baseTest() throws Exception {

User user = new User();

user.setName("Jay");

user.setPassword("123456");

user.setBirthday("2008-08-08");

userRepository.save(user);

// userRepository.delete(user);

// userRepository.findOne(1);

}

}

复制代码

　　测试通过。

　　说到这里，和spring已经完成。接下来第三点，基本使用。

4.前面把基础的东西说清楚了，接下来就是spring-data-jpa的正餐了，真正威力的地方。

　　4.1 我们的系统中一般都会有用户登录这个接口，在不使用spring-data-jpa的时候我们怎么做，首先在service层定义一个登录方法。如：

User login(String name, String password);

然后在serviceImpl中写该方法的实现，大致这样：

@Override

public User login(String name, String password) {

return userDao.login(name, password);

}

接下来，UserDao大概是这么个样子：

User getUserByNameAndPassword(String name, String password);

然后在UserDaoImpl中大概是这么个样子：

public User getUserByNameAndPassword(String name, String password) {

Query query = em.createQuery("select \* from User t where t.name = ?1 and t.password = ?2");

query.setParameter(1, name);

query.setParameter(2, password);

return (User) query.getSingleResult();

}

ok，这个代码运行良好，那么这样子大概有十来行代码，我们感觉这个功能实现了，很不错。然而这样子真正简捷么？如果这样子就满足了，那么spring-data-jpa就没有必要存在了，前面提到spring-data-jpa能够帮助你完成业务逻辑代码的处理，那他是怎么处理的呢？这里我们根本不需要UserDaoImpl这个类，只需要在UserRepository接口中定义一个方法

User findByNameAndPassword(String name, String password);

然后在service中调用这个方法就完事了，所有的逻辑只需要这么一行代码，一个没有实现的接口方法。通过debug信息，我们看到输出的sql语句是

select \* from user where name = ? and password = ?

跟上面的传统方式一模一样的结果。这简单到令人发指的程度，那么这一能力是如何实现的呢？原理是：spring-data-jpa会根据方法的名字来自动生成sql语句，我们只需要按照方法定义的规则即可，上面的方法findByNameAndPassword，spring-data-jpa规定，方法都以findBy开头，sql的where部分就是NameAndPassword，被spring-data-jpa翻译之后就编程了下面这种形态：

where name = ? and password = ?

在举个例，如果是其他的操作符呢，比如like，前端模糊查询很多都是以like的方式来查询。比如根据名字查询用户，sql就是

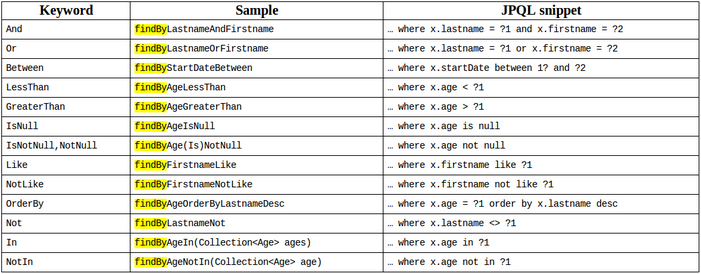
select \* from user where name like = ?

这里spring-data-jpa规定，在属性后面接关键字，比如根据名字查询用户就成了

User findByNameLike(String name);

被翻译之后的sql就是

select \* from user where name like = ?

这也是简单到令人发指，spring-data-jpa所有的语法规定如下图：

通过上面，基本CRUD和基本的业务逻辑操作都得到了解决，我们要做的工作少到仅仅需要在UserRepository接口中定义几个方法，其他所有的工作都由spring-data-jpa来完成。

 接下来：就是比较复杂的操作了，比如动态查询，分页，下面详细介绍spring-data-jpa的第二大杀手锏，强大的动态查询能力。

在上面的介绍中，对于我们传统的企业级应用的基本操作已经能够基本上全部实现，企业级应用一般都会有一个模糊查询的功能，并且是多条的查询，在有查询条件的时候我们需要在where后面接上一个 xxx = yyy 或者 xxx like '% + yyy + %'类似这样的sql。那么我们传统的JDBC的做法是使用很多的if语句根据传过来的查询条件来拼sql，mybatis的做法也类似，由于mybatis有强大的动态xml文件的标签，在处理这种问题的时候显得非常的好，但是二者的原理都一致，那spring-data-jpa的原理也同样很类似，这个道理也就说明了解决多表关联动态查询根儿上也就是这么回事。

　　那么spring-data-jpa的做法是怎么的呢？有两种方式。可以选择其中一种，也可以结合使用，在一般的查询中使用其中一种就够了，就是第二种，但是有一类查询比较棘手，比如报表相关的，报表查询由于涉及的表很多，这些表不一定就是两两之间有关系，比如字典表，就很独立，在这种情况之下，使用拼接sql的方式要容易一些。下面分别介绍这两种方式。

　　a.使用JPQL,和Hibernate的HQL很类似。

　　　前面说道了在UserRepository接口的同一个包下面建立一个普通类UserRepositoryImpl来表示该类的实现类，同时前面也介绍了完全不需要这个类的存在，但是如果使用JPQL的方式就必须要有这个类。如下：

复制代码

public class StudentRepositoryImpl {

@PersistenceContext

private EntityManager em;

@SuppressWarnings("unchecked")

public Page<Student> search(User user) {

String dataSql = "select t from User t where 1 = 1";

String countSql = "select count(t) from User t where 1 = 1";

if(null != user && !StringUtils.isEmpty(user.getName())) {

dataSql += " and t.name = ?1";

countSql += " and t.name = ?1";

}

Query dataQuery = em.createQuery(dataSql);

Query countQuery = em.createQuery(countSql);

if(null != user && !StringUtils.isEmpty(user.getName())) {

dataQuery.setParameter(1, user.getName());

countQuery.setParameter(1, user.getName());

}long totalSize = (long) countQuery.getSingleResult();

Page<User> page = new Page();

page.setTotalSize(totalSize);

List<User> data = dataQuery.getResultList();

page.setData(data);

return page;

}

}

复制代码

通过上面的方法，我们查询并且封装了一个User对象的分页信息。代码能够良好的运行。这种做法也是我们传统的经典做法。那么spring-data-jpa还有另外一种更好的方式，那就是所谓的类型检查的方式，上面我们的sql是字符串，没有进行类型检查，而下面的方式就使用了类型检查的方式。这个道理在mybatis中也有体现，mybatis可以使用字符串sql的方式，也可以使用接口的方式，而mybatis的官方推荐使用接口方式，因为有类型检查，会更安全。

　　b.使用JPA的动态接口，下面的接口我把注释删了，为了节省篇幅，注释也没什么用，看方法名字大概都能猜到是什么意思。

复制代码

public interface JpaSpecificationExecutor<T> {

T findOne(Specification<T> spec);

List<T> findAll(Specification<T> spec);

Page<T> findAll(Specification<T> spec, Pageable pageable);

List<T> findAll(Specification<T> spec, Sort sort);

long count(Specification<T> spec);

}

复制代码

 上面说了，使用这种方式我们压根儿就不需要UserRepositoryImpl这个类，说到这里，仿佛我们就发现了spring-data-jpa为什么把Repository和RepositoryImpl文件放在同一个包下面，因为我们的应用很可能根本就一个Impl文件都不存在，那么在那个包下面就只有一堆接口，即使把Repository和RepositoryImpl都放在同一个包下面，也不会造成这个包下面有正常情况下2倍那么多的文件，根本原因：只有接口而没有实现类。

上面我们的UserRepository类继承了JpaRepository和JpaSpecificationExecutor类，而我们的UserRepository这个对象都会注入到UserService里面，于是如果使用这种方式，我们的逻辑直接就写在service里面了，下面的代码：一个学生Student类，一个班级Clazz类，Student里面有一个对象Clazz，在数据库中是clazz\_id，这是典型的多对一的关系。我们在配置好entity里面的关系之后。就可以在StudentServiceImpl类中做Student的模糊查询，典型的前端grid的模糊查询。代码是这样子的：

复制代码

@Service

public class StudentServiceImpl extends BaseServiceImpl<Student> implements StudentService {

@Autowired

private StudentRepository studentRepository;

@Override

public Student login(Student student) {

return studentRepository.findByNameAndPassword(student.getName(), student.getPassword());

}

@Override

public Page<Student> search(final Student student, PageInfo page) {

return studentRepository.findAll(new Specification<Student>() {

@Override

public Predicate toPredicate(Root<Student> root, CriteriaQuery<?> query, CriteriaBuilder cb) {

Predicate stuNameLike = null;

if(null != student && !StringUtils.isEmpty(student.getName())) {  
　　　　　　　　　　　// 这里也可以root.get("name").as(String.class)这种方式来强转泛型类型

stuNameLike = cb.like(root.<String> get("name"), "%" + student.getName() + "%");

}

Predicate clazzNameLike = null;

if(null != student && null != student.getClazz() && !StringUtils.isEmpty(student.getClazz().getName())) {

clazzNameLike = cb.like(root.<String> get("clazz").<String> get("name"), "%" + student.getClazz().getName() + "%");

}

if(null != stuNameLike) query.where(stuNameLike);

if(null != clazzNameLike) query.where(clazzNameLike);

return null;

}

}, new PageRequest(page.getPage() - 1, page.getLimit(), new Sort(Direction.DESC, page.getSortName())));

}

}

复制代码

先解释下这里的意思，然后我们在结合框架的源码来深入分析。

这里我们是2个表关联查询，查询条件包括Student表和Clazz表，类似的2个以上的表方式差不多，但是正如上面所说，这种做法适合所有的表都是两两能够关联上的，涉及的表太多，或者是有一些字典表，那就使用sql拼接的方式，简单一些。

先简单解释一下代码的含义，然后结合框架源码来详细分析。两个Predicate对象，Predicate按照中文意思是判断，断言的意思，那么放在我们的sql中就是where后面的东西，比如

name like '% + jay + %';

下面的PageRequest代表分页信息，PageRequest里面的Sort对象是排序信息。上面的代码事实上是在动态的组合最终的sql语句，这里使用了一个策略模式，或者callback，就是

studentRepository.findAll(一个接口)

studentRepository接口方法调用的参数是一个接口，而接口的实现类调用这个方法的时候，在内部，参数对象的实现类调用自己的toPredicate这个方法的实现内容，可以体会一下这里的思路，就是传一个接口，然后接口的实现自己来定义，这个思路在nettyJavaScript中体现的特别明显，特别是JavaScript的框架中大量的这种方式，JS框架很多的做法都是上来先闭包，和浏览器的命名空间分开，然后入口方法就是一个回调，比如ExtJS：

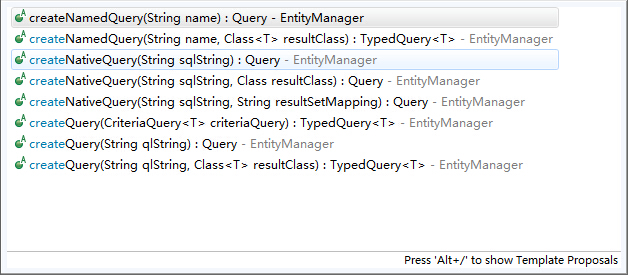
Ext.onReady(function() {

// xxx

});

参数是一个function，其实在框架内部就调用了这个参数，于是这个这个方法执行了。这种模式还有一个JDK的排序集合上面也有体现，我们的netty框架也采用这种方式来实现异步IO的能力。

接下来结合框架源码来详细介绍这种机制，以及这种机制提供给我们的好处。

 这里首先从JPA的动态查询开始说起，在JPA提供的API中，动态查询大概有这么一些方法，

从名字大概可以看出这些方法的意义，跟Hibernate或者一些其他的工具也都差不多，这里我们介绍参数为CriteriaQuery类型的这个方法，如果我们熟悉多种ORM框架的话，不难发现都有一个Criteria类似的东西，中文意思是“条件”的意思，这就是各个框架构建动态查询的主体，Hibernate甚至有两种，在线和离线两种Criteria，mybatis也能从Example中创建Criteria，并且添加查询条件。

那么第一步就需要构建出这个参数CriteriaQuery类型的参数，这里使用建造者模式，

CriteriaBuilder builder = em.getCriteriaBuilder();

CriteriaQuery<Student> query = builder.createQuery(Student.class);

接下来：

Root<Student> root = query.from(Student.class);

在这里，我们看方法名from，意思是获取Student的Root，其实也就是个Student的包装对象，就代表这条sql语句里面的主体。接下来：

Predicate p1 = builder.like(root.<String> get("name"), "%" + student.getName() + "%");

Predicate p2 = builder.equal(root.<String> get("password"), student.getPassword());

Predicate是判断的意思，放在sql语句中就是where后面 xxx = yyy, xxx like yyy这种，也就是查询条件，这里构造了2个查询条件，分别是根据student的name属性进行like查询和根据student的password进行“=”查询，在sql中就是

name like = ? and password = ?

这种形式，接下来

query.where(p1, p2);

这样子一个完整的动态查询就构建完成了，接下来调用getSingleResult或者getResultList返回结果，这里jpa的单个查询如果为空的话会报异常，这点感觉框架设计的不好，如果查询为空直接返回一个null或者一个空的List更好一点。

这是jpa原生的动态查询方式，过程大致就是，创建builder => 创建Query => 构造条件 => 查询。这么4个步骤，这里代码运行良好，如果不使用spring-data-jpa，我们就需要这么来做，但是spring-data-jpa帮我们做得更为彻底，从上面的4个步骤中，我们发现：所有的查询除了第三步不一样，其他几步都是一模一样的，不使用spring-data-jpa的情况下，我们要么4步骤写完，要么自己写个工具类，封装一下，这里spring-data-jpa就是帮我们完成的这样一个动作，那就是在JpaSpecification<T>这个接口中的

Page<T> findAll(Specification<T> spec, Pageable pageable);

这个方法，前面说了，这是个策略模式，参数spec是个接口，前面也说了框架内部对于这个接口有默认的实现类

@Repository

@Transactional(readOnly = true)

public class SimpleJpaRepository<T, ID extends Serializable> implements JpaRepository<T, ID>,

JpaSpecificationExecutor<T> {

}

，我们的Repository接口就是继承这个接口，而通过cglib的RepositoryImpl的代理类也是这个类的子类，默认也就实现了该方法。这个方法的方法体是这样的：

复制代码

/\*

\* (non-Javadoc)

\* @see org.springframework.data.jpa.repository.JpaSpecificationExecutor#findOne(org.springframework.data.jpa.domain.Specification)

\*/

public T findOne(Specification<T> spec) {

try {

return getQuery(spec, (Sort) null).getSingleResult();

} catch (NoResultException e) {

return null;

}

}

复制代码

这里的

getQuery(spec, (Sort) null)

返回类型是

TypedQuery<T>

进入这个getQuery方法：

复制代码

/\*\*

\* Creates a {@link TypedQuery} for the given {@link Specification} and {@link Sort}.

\*

\* @param spec can be {@literal null}.

\* @param sort can be {@literal null}.

\* @return

\*/

protected TypedQuery<T> getQuery(Specification<T> spec, Sort sort) {

CriteriaBuilder builder = em.getCriteriaBuilder();

CriteriaQuery<T> query = builder.createQuery(getDomainClass());

Root<T> root = applySpecificationToCriteria(spec, query);

query.select(root);

if (sort != null) {

query.orderBy(toOrders(sort, root, builder));

}

return applyRepositoryMethodMetadata(em.createQuery(query));

}

复制代码

一切玄机尽收眼底，这个方法的内容和我们前面使用原生jpa的api的过程是一样的，而再进入

Root<T> root = applySpecificationToCriteria(spec, query);

这个方法：

复制代码

/\*\*

\* Applies the given {@link Specification} to the given {@link CriteriaQuery}.

\*

\* @param spec can be {@literal null}.

\* @param query must not be {@literal null}.

\* @return

\*/

private <S> Root<T> applySpecificationToCriteria(Specification<T> spec, CriteriaQuery<S> query) {

Assert.notNull(query);

Root<T> root = query.from(getDomainClass());

if (spec == null) {

return root;

}

CriteriaBuilder builder = em.getCriteriaBuilder();

Predicate predicate = spec.toPredicate(root, query, builder);

if (predicate != null) {

query.where(predicate);

}

return root;

}

复制代码

我们可以发现spec参数调用了toPredicate方法，也就是我们前面service里面匿名内部类的实现。

到这里spring-data-jpa的默认实现已经完全明了。总结一下使用动态查询：前面说的原生api需要4步，而使用spring-data-jpa只需要一步，那就是重写匿名内部类的toPredicate方法。在重复一下上面的Student和Clazz的查询代码，

复制代码

1 @Override

2 public Page<Student> search(final Student student, PageInfo page) {

4 return studentRepository.findAll(new Specification<Student>() {

5 @Override

6 public Predicate toPredicate(Root<Student> root, CriteriaQuery<?> query, CriteriaBuilder cb) {

7

8 Predicate stuNameLike = null;

9 if(null != student && !StringUtils.isEmpty(student.getName())) {

10 stuNameLike = cb.like(root.<String> get("name"), "%" + student.getName() + "%");

11 }

12

13 Predicate clazzNameLike = null;

14 if(null != student && null != student.getClazz() && !StringUtils.isEmpty(student.getClazz().getName())) {

15 clazzNameLike = cb.like(root.<String> get("clazz").<String> get("name"), "%" + student.getClazz().getName() + "%");

16 }

17

18 if(null != stuNameLike) query.where(stuNameLike);

19 if(null != clazzNameLike) query.where(clazzNameLike);

20 return null;

21 }

22 }, new PageRequest(page.getPage() - 1, page.getLimit(), new Sort(Direction.DESC, page.getSortName())));

23 }

复制代码

到这里位置，spring-data-jpa的介绍基本上就完成了，涵盖了该框架使用的方方面面。接下来还有一块比较实用的东西，我们看到上面第15行位置的条件查询，这里使用了一个多级的get，这个是spring-data-jpa支持的，就是嵌套对象的属性，这种做法一般我们叫方法的级联调用，就是调用的时候返回自己本身，这个在处理xml的工具中比较常见，主要是为了代码的美观作用，没什么其他的用途。

最后还有一个小问题，我们上面说了使用动态查询和JPQL两种方式都可以，在我们使用JPQL的时候，他的语法和常规的sql有点不太一样，以Student、Clazz关系为例，比如：

select \* from student t left join clazz tt on t.clazz\_id = tt.id

这是一个很常规的sql，但是JPQL是这么写：

select t from Student t left join t.clazz tt

left join右边直接就是t的属性，并且也没有了on t.clazz\_id == tt.id，然而并不会出现笛卡尔积，这里解释一下为什么没有这个条件，在我们的实体中配置了属性的映射关系，并且ORM框架的最核心的目的就是要让我们以面向对象的方式来操作数据库，显然我们在使用这些框架的时候就不需要关心数据库了，只需要关系对象，而t.clazz\_id = tt.id这个是数据库的字段，由于配置了字段映射，框架内部自己就会去处理，所以不需要on t.clazz\_id = tt.id就是合理的。

　　前面介绍了spring-data-jpa的使用，还有一点忘了，悲观所和乐观锁问题，这里的乐观锁比较简单，jpa有提供注解@Version，加上该注解，自动实现乐观锁，byId修改的时候sql自动变成：update ... set ... where id = ? and version = ?，比较方便。

 in操作的查询：

　　在日常手动写sql的时候有in这种查询是比较多的，比如select \* from user t where t.id in (1, 2, 3)；有人说in的效率不高，要少用，但是其实只要in是主键，或者说是带有索引的，效率是很高的，mysql中如果in是子查询貌似不会走索引，不过我个人经验，在我遇到的实际应用中，in(ids)这种是比较多的，所以一般来说是没有性能问题的。

　　那么，sql里面比较好写，但是如果使用spring-data-jpa的动态查询方式呢，就和前面的稍微有点区别。大致上是这么一个思路：

复制代码

if(!CollectionUtils.isEmpty(ids)) {

In<Long> in = cb.in(root.<Long> get("id"));

for (Long id : parentIds) {

in.value(id);

}

query.where(in);

}

复制代码

　　cb创建一个in的Predicate,然后给这个in赋值，最后把in加到where条件中。

手动配置锁：

　　spring-data-jpa支持注解方式的sql，比如：@Query(xxx)，另外，关于锁的问题，在实体中的某个字段配置@Version是乐观锁，有时候为了使用一个悲观锁，或者手动配置一个乐观锁（如果实体中没有version字段），那么可以使用@Lock这个注解，它能够被解析成为相关的锁。

一对多、多对多查询（查询条件在关联对象中时）：

　　1、在JPA中，一个实体中如果存在多个关联对象，那么不能同时eager获取，只能有一个是eager获取，其他只能lazy；在Hibernate当中有几种独有的解决方法，在JPA当中有2中方法，i.就是前面的改成延时加载；ii.把关联对象的List改成Set（List允许重复，在多层抓去的时候无法完成映射，Hibernate默认抓去4层，在第三层的时候如果是List就无法完成映射）。

　　2、在多对多的查询中，我们可以使用JPQL，也可以使用原生SQL，同时还可以使用动态查询，这里介绍多对多的动态查询，这里有一个条件比较苛刻，那就是查询参数是关联对象的属性，一对多类似，多对一可以利用上面介绍的级联获取属性的方式。这里介绍这种方式的目的是为了更好的利用以面向对象的方式进行动态查询。

　　举例：2张表，分别是Employee(id, name)和Company(id, name)，二者是多对多的关系，那么当查询Employee的时候，条件是更具公司名称。那么做法如下：

复制代码

@Override

public List<Employee> findByCompanyName(final String companyName) {

List<Employee> employeeList = employeeRepository.findAll(new Specification<Employee>() {

public Predicate toPredicate(Root<Employee> root, CriteriaQuery<?> query, CriteriaBuilder cb) {

// ListJoin<Employee, Company> companyJoin = root.join(root.getModel().getList("companyList", Company.class), JoinType.LEFT);

Join<Employee, Company> companyJoin = root.join("companyList", JoinType.LEFT);

return cb.equal(companyJoin.get("name"), companyName);

}

});

return employeeList;

}

复制代码

　　   我们可以使用上面注释掉的方式，也可以使用下面这种比较简单的方式。因为我个人的习惯是尽量不去写DAO的实现类，除非查询特别复杂，万不得已的情况下采用，否则我个人比较偏向于这种方式。

　　上面的情况如果更为极端的话，关联多个对象，可以按照下面的方式：

复制代码

@Override

public List<Employee> findByCompanyName(final String companyName, final String wage) {

List<Employee> employeeList = employeeRepository.findAll(new Specification<Employee>() {

public Predicate toPredicate(Root<Employee> root, CriteriaQuery<?> query, CriteriaBuilder cb) {

// ListJoin<Employee, Company> companyJoin = root.join(root.getModel().getList("companyList", Company.class), JoinType.LEFT);

Join<Employee, Company> companyJoin = root.join("companySet", JoinType.LEFT);

Join<Employee, Wage> wageJoin = root.join("wageSet", JoinType.LEFT);

Predicate p1 = cb.equal(companyJoin.get("name"), companyName);

Predicate p2 = cb.equal(wageJoin.get("name"), wage);

// return cb.and(p1, p2);根据spring-data-jpa的源码，可以返回一个Predicate，框架内部会自动做query.where(p)的操作，也可以直接在这里处理，然后返回null，/// 也就是下面一段源码中的实现

query.where(p1, p2);

return null;

}

});

return employeeList;

}

复制代码

复制代码

/\*\*

\* Applies the given {@link Specification} to the given {@link CriteriaQuery}.

\*

\* @param spec can be {@literal null}.

\* @param query must not be {@literal null}.

\* @return

\*/

private <S> Root<T> applySpecificationToCriteria(Specification<T> spec, CriteriaQuery<S> query) {

Assert.notNull(query);

Root<T> root = query.from(getDomainClass());

if (spec == null) {

return root;

}

CriteriaBuilder builder = em.getCriteriaBuilder();

Predicate predicate = spec.toPredicate(root, query, builder);

　　　　// 这里如果我们重写的toPredicate方法的返回值predicate不为空，那么调用query.where(predicate)

if (predicate != null) {

query.where(predicate);

}

return root;

}

复制代码

说明：虽然说JPA中这种方式查询会存在着多次级联查询的问题，对性能有所影响，但是在一般的企业级应用当中，为了开发的便捷，这种性能牺牲一般来说是可以接受的。

　　特别的：在一对多中或者多对一中，即便是fetch为eager，也会先查询主对象，再查询关联对象，但是在eager的情况下虽然是有多次查询问题，但是没有n+1问题，关联对象不会像n+1那样多查询n次，而仅仅是把关联对象一次性查询出来，因此，在企业级应用当中，访问量不大的情况下一般来说没什么问题。

　　补充一段题外话，关于Hibernate/JPA/Spring-Data-Jpa与MyBatis的区别联系，这种话题很多讨论，对于Hibernate/JPA/Spring-Data-Jpa，我个人而言基本上能够熟练使用，谈不上精通，对于mybatis，由于深入阅读过几次它的源码，对mybatis的设计思想以及细化到具体的方法，属性，参数算是比较熟悉，也开发过一些mybatis的相关插件。对于这两个持久化框架，总体来说的区别是，Hibernate系列的门槛相对较高，配置比较多，相对来说难度要大一些，主要体现在各种关系的问题上，据我所知，很多人的理解其实并不深刻，很多时候甚至配置得有一定的问题，但是优势也很明显，SQL自动生成，改数据库表结构仅仅需要调整几个注解就行了，在熟练使用的基础上相对来说要便捷一点。对于mybatis来说，门槛很低，真的很低，低到分分钟就能入门的程度，我个人最喜欢也是mybatis最吸引人的地方就是灵活，特别的灵活，但是修改数据库表结构之后需要调整的地方比较多，但是利用目前比较优秀的插件，对于单表操作也基本上能够达到和Hibernate差不多的境界（会稍微牺牲一点点性能），多表的情况下就要麻烦一点。性能方面的比较，由于我没做过测试，不太好比较，不过应该mybatis要稍微高一些，毕竟他的查询SQL可控一些（当然Hibernate也支持原生sql，但是对结果集的处理不够友好）。