Spring Data JPA

什么是JPA?

相同处:

1.都跟数据·库操作有关，JPA 是JDBC 的升华，升级版。

2.JDBC和JPA都是一组规范接口

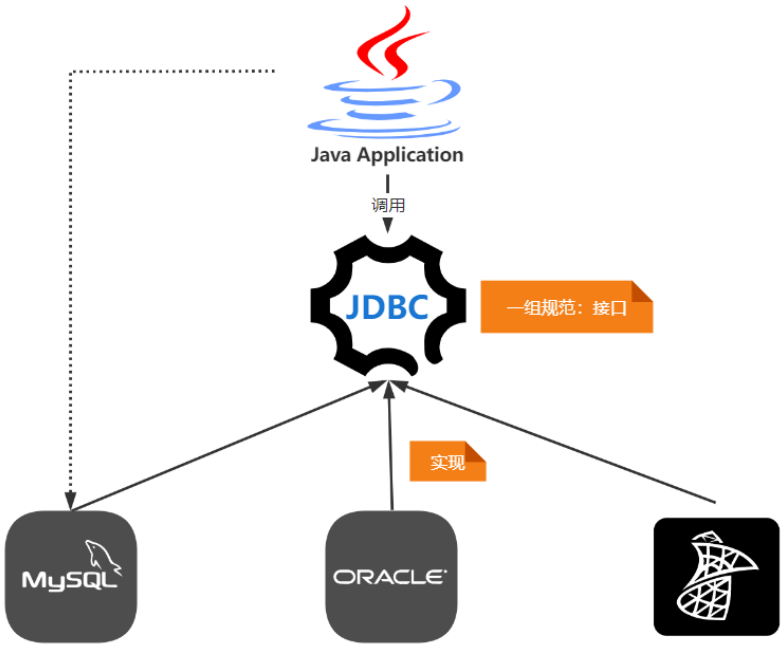
3.都是由SUN官方推出的

不同处：

1.JDBC是由各个关系型数据库实现的， JPA 是由**ORM框架**实现

2.JDBC 使用SQL语句和数据库通信。 JPA用面向对象方式， 通过ORM框架来生成SQL，进行操作。

3.JPA在JDBC之上的， JPA也要依赖JDBC才能操作数据库。



JDBC是我们最熟悉的用来操作数据库的技术，但是随之而来带来了一些问题：

1. 需要面向SQL语句来操作数据库，开发人员学习成本更高。
2. 数据库的移至性不高，不同数据库的SQL语句无法通用。
3. java对象和数据库类型的映射是个麻烦事。

但在Sun在JDK1.5提出了JPA：

JPA全称Java Persistence API（2019年重新命名为 Jakarta Persistence API ），是Sun官方提出的**一种ORM规范。**

O:Object R: Relational M:mapping

作用

1.简化持久化操作的开发工作：让开发者从繁琐的 JDBC 和 SQL 代码中解脱出来，直接面向对象持久化操作。

2.Sun希望持久化技术能够统一，实现天下归一：如果你是基于JPA进行持久化你可以随意切换数据库。

该规范为我们提供了：

1）**ORM映射元数据**：JPA支持XML和注解两种元数据的形式，元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中；

如：`@Entity`、 `@Table`、`@Id` 与 `@Column`等注解。

2）**JPA 的API**：用来操作实体对象，执行CRUD操作，框架在后台替我们完成所有的事情，开发者从繁琐的JDBC和SQL代码中解脱出来。

如：`entityManager.merge(T t);`

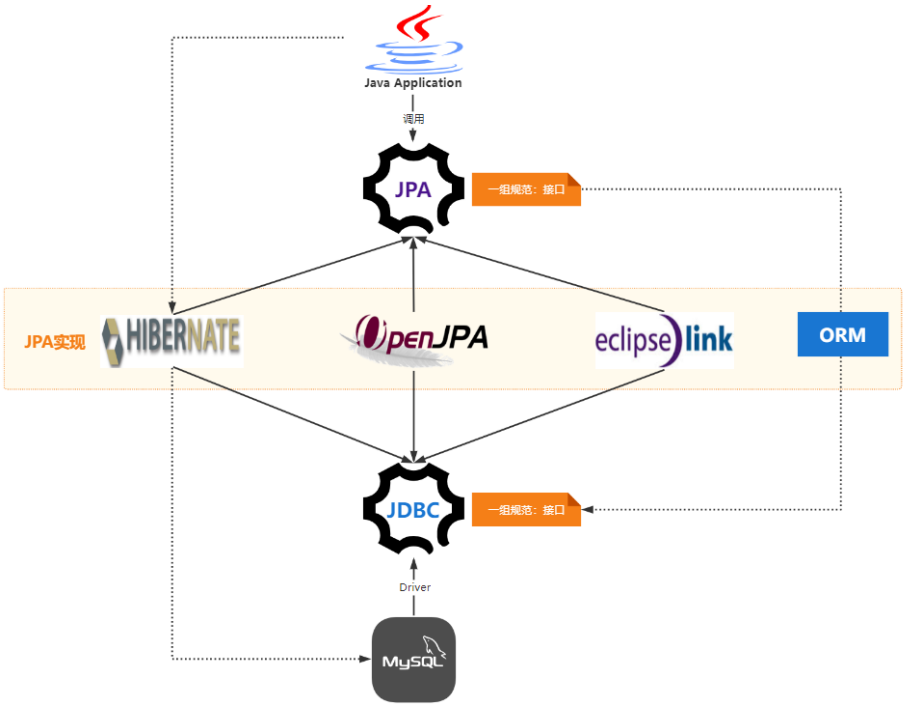
3）**JPQL查询语言**：通过面向对象而非面向数据库的查询语言查询数据，避免程序的SQL语句紧密耦合。

如：`from Student s where s.name = ?`

So: JPA仅仅是一种规范，也就是说JPA仅仅定义了一些接口，而接口是需要实现才能工作的。

Hibernate与JPA:

所以底层需要某种实现，而Hibernate就是实现了JPA接口的ORM框架。
也就是说：
JPA是一套ORM规范，Hibernate实现了JPA规范！



mybatis：小巧、方便？、高效、简单、直接、半自动
半自动的ORM框架，
小巧： mybatis就是jdbc封装
在国内更流行。
场景： 在业务比较复杂系统进行使用，

hibernate：强大、方便、高效、（简单）复杂、绕弯子、全自动
全自动的ORM框架，
强大：根据ORM映射生成不同SQL
在国外更流。
场景： 在业务相对简单的系统进行使用，随着微服务的流行。

Hibernate示例

https://docs.jboss.org/hibernate/orm/5.5/userguide/html_single/Hibernate_User_Guide.html#hql
我们来实现一个Hibernate来示例感受下：

pom.xml

```
<!-- junit4 -->
<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.13</version>
 <scope>test</scope>
</dependency>
<!-- hibernate对jpa的支持包 -->
<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-entitymanager</artifactId>
 <version>5.4.32.Final</version>
</dependency>
<!-- Mysql and MariaDB -->
<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
```

```
 <version>5.1.22</version>
</dependency>
```

实体类

```
package com.xushu.pojo;

import javax.persistence.*;

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@Entity
@Table(name = "cst_customer")
public class Customer {

 /**
 * @Id：声明主键的配置
 * @GeneratedValue:配置主键的生成策略
 * strategy
 * GenerationType.IDENTITY ： 自增，mysql
 * * 底层数据库必须支持自动增长（底层数据库支持的自动增长方式，对id自增）
 * GenerationType.SEQUENCE ： 序列，oracle
 * * 底层数据库必须支持序列
 * GenerationType.TABLE ： jpa提供的一种机制，通过一张数据库表的形式帮助我们完成主键自增
 * GenerationType.AUTO ： 由程序自动的帮助我们选择主键生成策略
 * @Column:配置属性和字段的映射关系
 * name：数据库表中字段的名称
 */
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 @Column(name = "cust_id")
 private Long custId; //客户的主键

 @Column(name = "cust_name")
 private String custName;//客户名称

 @Column(name="cust_address")
 private String custAddress;//客户地址

 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 getCustAddress() {
 return custAddress;
 }

 public void setCustAddress(String custAddress) {
```

```
 this.custAddress = custAddress;
 }

 @Override
 public String toString() {
 return "Customer{" +
 "custId=" + custId +
 ", custName='" + custName + '\'' +
 ", custAddress='" + custAddress + '\'' +
 "}\n";
 }
}

```

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory>
 <!-- 配置数据库连接信息 -->
 <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
 <property name="connection.url">jdbc:mysql://localhost:3306/springdata_jpa?characterEncoding=UTF-8</property>
 <property name="connection.username">root</property>
 <property name="connection.password">123456</property>

 <!-- 允许显示sql语句 -->
 <property name="show_sql">true</property>
 <property name="format_sql">true</property>
 <property name="hbm2ddl.auto">update</property>
 <!-- 配置方言：选择数据库类型 -->
 <property name="dialect">org.hibernate.dialect.MySQL5InnoDBDialect</property>

 <!--映射方式-->
 <mapping class="com.xushu.pojo.Customer"></mapping>
 <mapping class="com.xushu.pojo.User"></mapping>
 <mapping class="com.xushu.pojo.Wife"></mapping>
 <mapping class="com.xushu.pojo.Hobby"></mapping>
 </session-factory>
</hibernate-configuration>
```

测试

```

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
public class HibernateTest {

 private SessionFactory sf;

 @Before
 public void init() {
 StandardServiceRegistry registry = new StandardServiceRegistryBuilder().configure("/hibernate.cfg.xml").build();

 //2. 根据服务注册类创建一个元数据资源集，同时构建元数据并生成应用一般唯一的的session工厂

 sf = new MetadataSources(registry).buildMetadata().buildSessionFactory();
 }

 @Test
```

```java
public void testC() {
// 创建Session
Session sess = sf.openSession();
// 开始事务
Transaction tx = sess.beginTransaction();
// 创建消息实例
Customer customer = new Customer();
customer.setCustName("张三");
// 保存消息
sess.save(customer);
// 提交事务
tx.commit();
// 关闭Session
sess.close();
sf.close();
}


@Test
public void testC_HQL() {
// 创建Session
Session sess = sf.openSession();
// 开始事务
Transaction tx = sess.beginTransaction();
// 保存消息
String sql = "insert into Customer (custName) select custName from Customer where custId=1";

sess.createQuery(sql)
.executeUpdate();
// 提交事务
tx.commit();
// 关闭Session
sess.close();
sf.close();
}


// 延迟查询
@Test
public void testR1() {
// 创建Session
Session sess = sf.openSession();
// 开始事务
Transaction tx = sess.beginTransaction();

Customer customer = sess.load(Customer.class, 1L);
System.out.println("============================");
System.out.println(customer);

// 提交事务
tx.commit();
// 关闭Session
sess.close();
sf.close();
}


@Test
public void testR_HQL_list() {
// 创建Session
Session sess = sf.openSession();
```

```
// 开始事务
Transaction tx = sess.beginTransaction();

List<Customer> list = sess.createQuery("SELECT c FROM Customer c", Customer.class)
.getResultList();

System.out.println(list);
// 提交事务
tx.commit();
// 关闭Session
sess.close();
sf.close();
}


@Test
public void testR_HQL_single() {
// 创建Session
Session sess = sf.openSession();
 // 开始事务
 Transaction tx = sess.beginTransaction();

 Customer customer = sess.createQuery("SELECT c FROM Customer c where c.custId=:id", Customer.class)
 .setParameter("id", 1L)
 .getSingleResult();

 System.out.println(customer);
 // 提交事务
 tx.commit();
 // 关闭Session
 sess.close();
 sf.close();
 }

 @Test
 public void testU() {
 // 创建Session
 Session sess = sf.openSession();
 // 开始事务
 Transaction tx = sess.beginTransaction();

 Customer customer = new Customer();
 customer.setCustId(1L);
 customer.setCustAddress("123456");

 sess.update(customer);

 System.out.println(customer);
 // 提交事务
 tx.commit();
 // 关闭Session
 sess.close();
 sf.close();
 }


 @Test
 public void testU_HQL() {
 // 创建Session
 Session sess = sf.openSession();
 // 开始事务
```

```
Transaction tx = sess.beginTransaction();

String sql = "Update Customer set custName=:custName where custId=:id";
sess.createQuery(sql)
.setParameter("custName", "徐庶")
.setParameter("id", 1L)
.executeUpdate();
// 提交事务
tx.commit();
// 关闭Session
sess.close();
sf.close();
}


@Test
public void testD() {
// 创建Session
Session sess = sf.openSession();
// 开始事务
Transaction tx = sess.beginTransaction();

Customer customer = new Customer();
customer.setCustId(4L);
sess.delete(customer);
// 提交事务
tx.commit();
// 关闭Session
sess.close();
sf.close();
}

@Test
public void testD_HQL() {
// 创建Session
Session sess = sf.openSession();
// 开始事务
Transaction tx = sess.beginTransaction();

String sql = "DELETE FROM Customer WHERE custId=:id";
sess.createQuery(sql)
.setParameter("id", 1L)
.executeUpdate();
// 提交事务
tx.commit();
// 关闭Session
sess.close();
sf.close();
}
```

如果单独使用hibernate的API来进行持久化操作，则不能随意切换其他ORM框架

Jpa示例

1.添加META-INF\persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
 <!--需要配置persistence-unit节点
 持久化单元:
 name:持久化单元名称
 transaction-type:事务管理的方式
 JTA:分布式事务管理
```

```
 RESOURCE_LOCAL：本地事务管理
 -->
 <persistence-unit name="hibernateJPA" transaction-type="RESOURCE_LOCAL">
  <class>xxx</class>
  <!--jpa的实现方式 -->
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

  <!--可选配置：配置jpa实现方的配置信息-->
  <properties>
  <!-- 数据库信息
  用户名，javax.persistence.jdbc.user
  密码， javax.persistence.jdbc.password
  驱动， javax.persistence.jdbc.driver
  数据库地址 javax.persistence.jdbc.url
  -->
  <property name="javax.persistence.jdbc.user" value="root"/>
  <property name="javax.persistence.jdbc.password" value="123456"/>
  <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
  <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/springdata_jpa?
serverTimezone=UTC"/>

  <!--配置jpa实现方(hibernate)的配置信息
  显示sql ： false|true
  自动创建数据库表 ： hibernate.hbm2ddl.auto
  create ：程序运行时创建数据库表（如果有表，先删除表再创建）
  update ：程序运行时创建表（如果有表，不会创建表）
  none ：不会创建表

  -->
  <property name="hibernate.show_sql" value="true" />
  <property name="hibernate.hbm2ddl.auto" value="update" />
  <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />

  </properties>
  </persistence-unit>


</persistence>
```

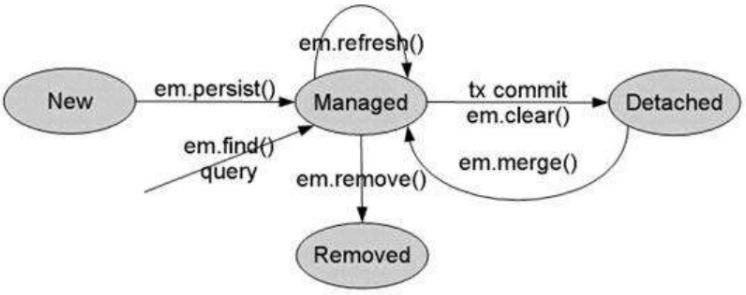
测试

```
package com.tuling.test;

import com.xushu.pojo.Customer;
import org.junit.Before;
import org.junit.Test;

import javax.persistence.*;
import java.util.List;

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
public class JpaTest {

 private EntityManagerFactory factory;
 EntityManager em;

 @Before
 public void init(){
 //1.加载配置文件，创建entityManagerFactory new Configuration.buildSessionFactory();
 factory = Persistence.createEntityManagerFactory("hibernateJPA");
```

```
// 2. 获取entityManager对象 sessionFactory.openSession();
em = factory.createEntityManager();
}
/**
* 查询全部
* jqpl: from cn.itcast.domain.Customer
* sql: SELECT * FROM cst_customer
*/
@Test
public void testR_HQL() { ;
//2.开启事务
EntityTransaction tx = em.getTransaction();
tx.begin();
//3.查询全部
String jpql = "select c from Customer c";
Query query = em.createQuery(jpql);//创建Query查询对象，query对象才是执行jqpl的对象

//发送查询，并封装结果集
List list = query.getResultList();

for (Object obj : list) {
System.out.print(obj);
}

//4.提交事务
tx.commit();
//5.释放资源
em.close();
}

@Test
public void testR() {
//2.开启事务
EntityTransaction tx = em.getTransaction();
tx.begin();

Customer customer = em.getReference(Customer.class, 1L);
System.out.println("============================");
System.out.println(customer);
//4.提交事务
tx.commit();
//5.释放资源
em.close();
}


}
```

jpa的对象4种状态

- 临时状态：刚创建出来，·没有与entityManager发生关系，没有被持久化，不处于entityManager中的对象
- 持久状态：·与entityManager发生关系，已经被持久化，您可以把持久化状态当做实实在在的数据库记录。
- 删除状态：执行remove方法，事物提交之前
- 游离状态：游离状态就是提交到数据库后，事务commit后实体的状态，因为事务已经提交了，此时实体的属性任你如何改变，也不会同步到数据库，因为游离是没人管的孩子，不在持久化上下文中。



- **public void persist(Object entity)**
  - persist方法可以将实例转换为managed(托管)状态。在调用flush()方法或提交事物后，实例将会被插入到数据库中。

对不同状态下的实例A，persist会产生以下操作:

1. 如果A是一个new状态的实体，它将会转为managed状态；
2. 如果A是一个managed状态的实体，它的状态不会发生任何改变。但是系统仍会在数据库执行INSERT操作；
3. 如果A是一个removed(删除)状态的实体，它将会转换为受控状态；
4. 如果A是一个detached(分离)状态的实体，该方法会抛出IllegalArgumentException异常，具体异常根据不同的JPA实现有关。

- **public void merge(Object entity)**
  - merge方法的主要作用是将用户对一个detached状态实体的修改进行归档，归档后将产生一个新的managed状态对象。

对不同状态下的实例A，merge会产生以下操作:

1. 如果A是一个detached状态的实体，该方法会将A的修改提交到数据库，并返回一个新的managed状态的实例A2；
2. 如果A是一个new状态的实体，该方法会产生一个根据A产生的managed状态实体A2;
3. 如果A是一个managed状态的实体，它的状态不会发生任何改变。但是系统仍会在数据库执行UPDATE操作；
4. 如果A是一个removed状态的实体，该方法会抛出IllegalArgumentException异常。

- **public void refresh(Object entity)**
  - refresh方法可以保证当前的实例与数据库中的实例的内容一致。

对不同状态下的实例A，refresh会产生以下操作:

1. 如果A是一个new状态的实例，不会发生任何操作，但有可能会抛出异常，具体情况根据不同JPA实现有关；
2. 如果A是一个managed状态的实例，它的属性将会和数据库中的数据同步；
3. 如果A是一个removed状态的实例，该方法将会抛出异常: Entity not managed
4. 如果A是一个detached状态的实体，该方法将会抛出异常。

- **public void remove(Object entity)**

remove方法可以将实体转换为removed状态，并且在调用flush()方法或提交事物后删除数据库中的数据。

对不同状态下的实例A，remove会产生以下操作:

1. 如果A是一个new状态的实例，A的状态不会发生任何改变，但系统仍会在数据库中执行DELETE语句；
2. 如果A是一个managed状态的实例，它的状态会转换为removed；
3. 如果A是一个removed状态的实例，不会发生任何操作;
4. 如果A是一个detached状态的实体，该方法将会抛出异常。

Spring data JPA

介绍

官网：https://spring.io/projects/spring-data-jpa#overview

> Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies.
> Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that' s actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.
>
> Spring Data JPA 是更大的 Spring Data 系列的一部分，可以轻松实现基于 JPA 的repositories。该模块处理对基于 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 驱动的应用程序变得更加容易。
> 实现应用程序的数据访问层已经很麻烦了。必须编写太多样板代码来执行简单的查询以及执行分页和审计。**Spring Data JPA 旨在改进数据访问层的实现以提升开发效率**。作为开发人员，您编写存储库接口，包括自定义 finder 方法，Spring 将自动提供实现。

人话

spirng data jpa是spring提供的一套**简化JPA开发的框架**，按照约定好的规则进行【方法命名】去写dao层接口，就可以在不写接口实现的情况下，实现对数据库的访问和操作。同时提供了很多除了CRUD之外的功能，如分页、排序、复杂查询等等。

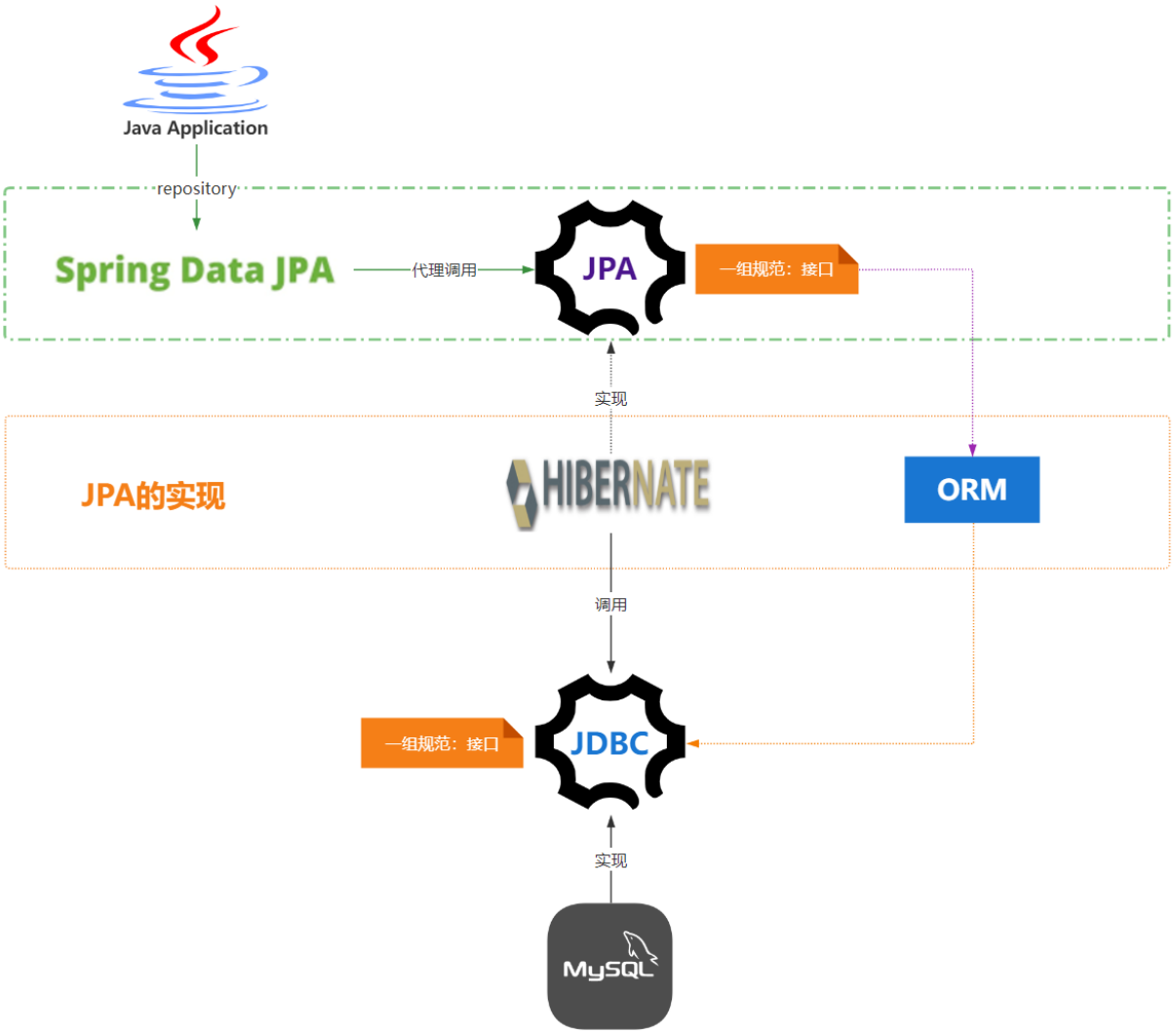
Spring Data JPA 让我们解脱了DAO层的操作，基本上所有CRUD都可以依赖于它来实现,在实际的工作工程中，推荐使用Spring Data JPA + ORM（如：hibernate）完成操作，这样在切换不同的ORM框架时提供了极大的方便，同时也使数据库层操作更加简单，方便解耦

特性

Features

- Sophisticated support to build repositories based on Spring and JPA
- Support for Querydsl predicates and thus type-safe JPA queries
- Transparent auditing of domain class
- Pagination support, dynamic query execution, ability to integrate custom data access code
- Validation of `@Query` annotated queries at bootstrap time
- Support for XML based entity mapping
- JavaConfig based repository configuration by introducing `@EnableJpaRepositories`.

SpringData Jpa 极大简化了数据库访问层代码。 如何简化的呢？ 使用了SpringDataJpa，我们的dao层中只需要写接口，就自动具有了增删改查、分页查询等方法。



Spring Data JPA实例

我们来实现一个基于Spring Data JPA的示例感受一下和之前单独使用的区别：

依赖

1.最好在父maven项目中设置spring data统一版本管理依赖: 因为不同的spring data子项目发布时间版本不一样，你自己维护很麻烦， 这样不同的spring data子项目能保证是统一版本.

```
<dependencyManagement>
 <dependencies>
 <dependency>
 <groupId>org.springframework.data</groupId>
 <artifactId>spring-data-bom</artifactId>
 <version>2020.0.14</version>
 <scope>import</scope>
 <type>pom</type>
 </dependency>
  </dependencies>
</dependencyManagement>
```

2.在子项目中添加：

```
<dependencies>
 <dependency>
 <groupId>org.springframework.data</groupId>
 <artifactId>spring-data-jpa</artifactId>
 </dependency>

 <dependency>
 <groupId>org.springframework.data</groupId>
```

```
 <artifactId>spring-data-jpa</artifactId>
</dependency>

<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-entitymanager</artifactId>
 <version>5.4.32.Final</version>
</dependency>

<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <version>5.1.22</version>
</dependency>


<!--junit4-->
<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.13</version>
 <scope>test</scope>
</dependency>

<dependency>
 <groupId>com.alibaba</groupId>
 <artifactId>druid</artifactId>
 <version>1.2.8</version>
</dependency>

<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-test</artifactId>
 <version>5.3.10</version>
 <scope>test</scope>
</dependency>
<dependencies>


```

JavaConfig

```

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@Configuration
@EnableJpaRepositories("com.tuling.repository")
@EnableTransactionManagement
public class JpaConfig {
 @Bean
 public DataSource dataSource() {
 DruidDataSource dataSource = new DruidDataSource();
 dataSource.setUsername("root");
 dataSource.setPassword("123456");
 dataSource.setDriverClassName("com.mysql.jdbc.Driver");
 dataSource.setUrl("jdbc:mysql://localhost:3306/springdata_jpa");
 return dataSource;
 }

 @Bean
```

```
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
vendorAdapter.setGenerateDdl(true);
vendorAdapter.setShowSql(true);

LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
factory.setJpaVendorAdapter(vendorAdapter);
factory.setPackagesToScan("com.tuling.pojo");
factory.setDataSource(dataSource());
return factory;
}

@Bean
public PlatformTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {

JpaTransactionManager txManager = new JpaTransactionManager();
txManager.setEntityManagerFactory(entityManagerFactory);
return txManager;
}
}
```

xml

```
<?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/jpa" />
        <property name="user" value="root" />
        <property name="password" value="111111" />
    </bean>

    <!-- 2.配置entityManagerFactory -->
    <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="packagesToScan" value="cn.itcast.entity" />
        <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">
                <property name="generateDdl" value="false" />
                <property name="database" value="MYSQL" />
                <property name="databasePlatform" value="org.hibernate.dialect.MySQLDialect" />
                <property name="showSql" value="true" />
```

```
            </bean>
        </property>
    </bean>


    <!-- 整合spring data jpa-->
    <jpa:repositories base-package="cn.itcast.dao"
        transaction-manager-ref="transactionManager"
        entity-manager-factory-ref="entityManagerFactory"></jpa:repositories>


    <!-- 3.事务管理器-->
    <!-- JPA事务管理器  -->
    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

 <!--基于注解方式的事务，开启事务的注解驱动
 如果基于注解的和xml的事务都配置了会以注解的优先
 -->
 <tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-driven>

    <context:component-scan base-package="cn.itcast"></context:component-scan>

    <!--组装其它 配置文件-->

</beans>
```

pojo

```
package com.tuling.pojo;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@Entity // This tells Hibernate to make a table out of this class
public class Customer {
 @Id
 @GeneratedValue(strategy=GenerationType.AUTO)
 private Long id;
 private String firstName;
 private String lastName;

 protected Customer() {}

 public Customer(String firstName, String lastName) {
 this.firstName = firstName;
 this.lastName = lastName;
 }

 @Override
 public String toString() {
 return String.format(
 "Customer[id=%d, firstName='%s', lastName='%s']",
 id, firstName, lastName);
```

```
 }

 public Long getId() {
 return id;
 }

 public String getFirstName() {
 return firstName;
 }

 public String getLastName() {
 return lastName;
 }
}
```

CustomerRepository

```

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
public interface CustomerRepository extends CrudRepository<Customer, Long> {
 }
```

测试

```

// 基于junit4 spring单元测试
//@ContextConfiguration("/spring.xml")
@ContextConfiguration(classes = SpringDataJPAConfig.class)
@RunWith(SpringJUnit4ClassRunner.class)
public class SpringdataJpaTest {

 @Autowired
 CustomerRepository repository;

 @Test
 public void testR(){
 Optional<Customer> byId = repository.findById(1L);

 System.out.println(byId.get());
 }

 @Test
 public void testC(){


 Customer customer = new Customer();
 customer.setCustId(3L);
 customer.setCustName("李四");

 repository.save(customer);
 }

 @Test
 public void testD(){


 Customer customer = new Customer();
 customer.setCustId(3L);
 customer.setCustName("李四");

 repository.delete(customer);
```

```
 }
}
```

使用 Spring Data Repositories

Spring Data repository 抽象的目标是显着减少为各种持久性存储实现数据访问层所需的样板代码量。

`CrudRepository`

```

// 用来插入和修改 有主键就是修改 没有就是新增
// 获得插入后自增id， 获得返回值
<S extends T> S save(S entity);

// 通过集合保存多个实体
<S extends T> Iterable<S> saveAll(Iterable<S> entities);
// 通过主键查询实体
Optional<T> findById(ID id);
 // 通过主键查询是否存在 返回boolean
 boolean existsById(ID id);
 // 查询所有
 Iterable<T> findAll();
 // 通过集合的主键 查询多个实体，， 返回集合
 Iterable<T> findAllById(Iterable<ID> ids);
 // 查询总数量
 long count();
 // 根据id进行删除
 void deleteById(ID id);
 // 根据实体进行删除
 void delete(T entity);
 // 删除多个
 void deleteAllById(Iterable<? extends ID> ids);
 // 删除多个传入集合实体
 void deleteAll(Iterable<? extends T> entities);
 // 删除所有
 void deleteAll();
```

在 之上`CrudRepository`，有一个[PagingAndSortingRepository](#)抽象，它添加了额外的方法来简化对实体的分页访问：

```

@ContextConfiguration(classes = SpringDataJPAConfig.class)
@RunWith(SpringJUnit4ClassRunner.class)
public class SpringDataJpaPagingAndSortTest
{
 // jdk动态代理的实例
 @Autowired
 CustomerRepository repository;

 @Test
 public void testPaging(){
 Page<Customer> all = repository.findAll(PageRequest.of(0, 2));
 System.out.println(all.getTotalPages());
 System.out.println(all.getTotalElements());
 System.out.println(all.getContent());

 }

```

```
@Test
public void testSort(){

Sort sort = Sort.by("custId").descending();

Iterable<Customer> all = repository.findAll(sort);

System.out.println(all);

}


@Test
public void testSortTypeSafe(){

Sort.TypedSort<Customer> sortType = Sort.sort(Customer.class);

Sort sort = sortType.by(Customer::getCustId).descending();


Iterable<Customer> all = repository.findAll(sort);

System.out.println(all);

}
}
```

自定义操作：

jpql（原生SQL）

a. @Query

i. 查询如果返回单个实体 就用pojo接收， 如果是多个需要通过集合

ii. 参数设置方式

1. 索引： ?数字

2. 具名： :参数名 结合@Param注解指定参数名字

iii. 增删改：

1. 要加上事务的支持：

2. 如果是插入方法：一定只能在hibernate下才支持 （Insert into ..select)

```
@Transactional // 通常会放在业务逻辑层上面去声明
@Modifying // 通知springdatajpa 是增删改的操作
```

规定方法名

- 支持的查询方法**主题关键字（前缀）**
  - 决定当前方法作用
  - 只支持查询和删除

表 8. 查询主题关键字

关键词	描述
`find…By` , `read…By` , `get…By` , `query…By` , `search…By` , `stream…By`	通用查询方法通常返回存储库类型、 `Collection` 或 `Streamable` 子类型或结果包装器，例如 `Page` , `GeoResults` 或任何其他特定于商店的结果包装器。可用作 `findBy…` , `findMyDomainTypeBy…` 或与其他关键字结合使用。
`exists…By`	存在投影，通常返回 `boolean` 结果。
`count…By`	计数投影返回数字结果。
`delete…By` , `remove…By`	删除查询方法返回无结果（ `void` ）或删除计数。
`…First<number>…` , `…Top<number>…`	将查询结果限制为第一个 `<number>` 结果。此关键字可以出现在主题的 `find` （和其他关键字）和之间的任何位置 `by` 。
`…Distinct…`	使用不同的查询仅返回唯一的结果。查阅特定于商店的文档是否支持该功能。此关键字可以出现在主题的 `find` （和其他关键字）和之间的任何位置 `by` 。

- 支持的查询方法**谓词关键字和修饰符**
  - 决定查询条件

关键词	样本	JPQL 片段
`Distinct`	`findDistinctByLastnameAndFirstname`	`select distinct … where x.lastname = ?1 and x.firstname = ?2`
`And`	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
`Or`	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`
`Is` , `Equals`	`findByFirstname` , `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` , `Null`	`findByAge(Is)Null`	`… 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` （参数绑定了 append `%` ）
`EndingWith`	`findByFirstnameEndingWith`	`… where x.firstname like ?1` （参数绑定 prepended `%` ）
`Containing`	`findByFirstnameContaining`	`… where x.firstname like ?1` （参数绑定包裹在 `%` ）
`OrderBy`	`findByAgeOrderByLastnameDesc`	`… where x.age = ?1 order by x.lastname desc`
`Not`	`findByLastnameNot`	`… where x.lastname <> ?1`
`In`	`findByAgeIn(Collection<Age> ages)`	`… where x.age in ?1`
`NotIn`	`findByAgeNotIn(Collection<Age> ages)`	`… where x.age not in ?1`
`True`	`findByActiveTrue()`	`… where x.active = true`
`False`	`findByActiveFalse()`	`… where x.active = false`
`IgnoreCase`	`findByFirstnameIgnoreCase`	`… where UPPER(x.firstname) = UPPER(?1)`

Query by Example

b. 只支持查询

i. 不支持嵌套或分组的属性约束，如 firstname = ？ 0 or (firstname = ？ 1 and lastname = ？ 2).

ii. 只支持字符串 start/contains/ends/regex 匹配和其他属性类型的精确匹配。

实现：

1.将Repository继承QueryByExampleExecutor

```
public interface CustomerQBERepository extends
 PagingAndSortingRepository<Customer,Long>
 , QueryByExampleExecutor<Customer> {

}
```

2.测试代码

```
 @Test
public void test01(){
 Customer customer = new Customer();
 customer.setCustName("徐庶");

 Example<Customer> example = Example.of(customer);

 System.out.println(repository.findAll(example));
}

@Test
public void test02(){
 Customer customer = new Customer();
 customer.setCustAddress("beijing");

 // 匹配器， 去设置更多条件匹配
 ExampleMatcher matcher = ExampleMatcher.matching()
 .withIgnoreCase("custAddress");

 Example<Customer> example = Example.of(customer,matcher);

 System.out.println(repository.findAll(example));
}
```

Specifications

- 在之前使用**Query by Example**只能针对字符串进行条件设置，那如果希望对所有类型支持，可以使用Specifications

实现

1.继承接口JpaSpecificationExecutor

```
public interface CustomerRepository extends CrudRepository<Customer, Long>, JpaSpecificationExecutor<Customer> {
 …
}
```

2. 传入Specification的实现： 结合lambda表达式

```
repository.findAll((Specification<Customer>)
 (root, query, criteriaBuilder) ->
 {
 // Todo...
 return null;
 });
}
```

Root：查询哪个表（关联查询） = from

CriteriaQuery：查询哪些字段，排序是什么 =组合(order by . where)

CriteriaBuilder：条件之间是什么关系，如何生成一个查询条件，每一个查询条件都是什么类型（> between in...） = where

Predicate（Expression）： 每一条查询条件的详细描述

```
List<Customer> list = repository.findAll((Specification<Customer>)
 (root, query, criteriaBuilder) ->
 {

 Order weightOrder = criteriaBuilder.desc(root.get("custId"));


 CriteriaBuilder.In<Object> id = criteriaBuilder.in(root.get("custId"));
```

```
 id.value(1).value(7);

 return query.orderBy(weightOrder).where(id).getRestriction();

});
```

限制：

不能分组、聚合函数， 需要自己通过entityManager玩

Querydsl

https://querydsl.com/ QueryDsl文档

QueryDSL是基于ORM框架或SQL平台上的**一个通用查询框架**。借助QueryDSL可以在任何支持的ORM框架或SQL平台**上以通用API方式构建查询。**

JPA是QueryDSL的主要集成技术，是JPQL和Criteria查询的代替方法。目前QueryDSL支持的平台包括JPA,JDO,SQL,Mongodb 等等。。。

Querydsl扩展能让我们以链式方式代码编写查询方法。该扩展需要一个接口QueryDslPredicateExecutor，它定义了很多查询方法。

接口继承了该接口，就可以使用该接口提供的各种方法了

```
public interface QuerydslPredicateExecutor<T> {

 T findOne(Predicate predicate);

 Iterable<T> findAll(Predicate predicate);

 long count(Predicate predicate);

 boolean exists(Predicate predicate);

 // … more functionality omitted.
}

interface UserRepository extends CrudRepository<User, Long>, QuerydslPredicateExecutor<User> {

}
```

引入依赖

```
<querydsl.version>4.4.0</querydsl.version>
<apt.version>1.1.3</apt.version>
```

```
 <!-- querydsl -->
 <dependency>
 <groupId>com.querydsl</groupId>
 <artifactId>querydsl-jpa</artifactId>
 <version>${querydsl.version}</version>
 </dependency>
```

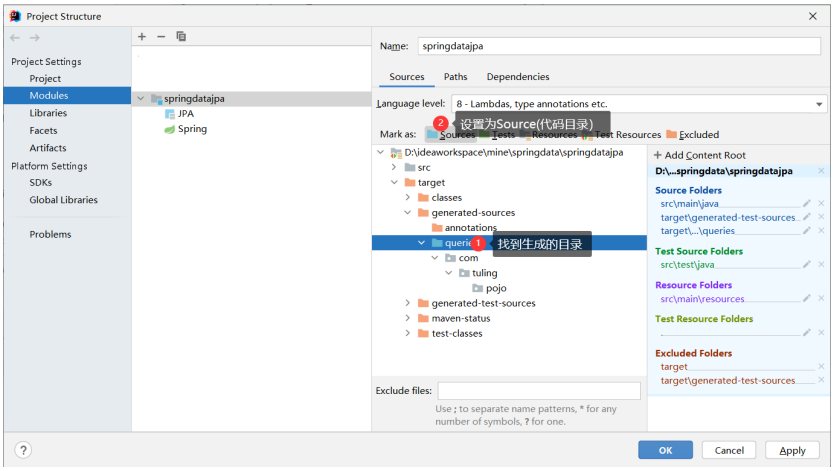
添加maven插件

这个插件是为了让程序自动生成query type(查询实体，命名方式为："Q"+对应实体名)。

```
<build>
 <plugins>
 <plugin>
 <groupId>com.mysema.maven</groupId>
 <artifactId>apt-maven-plugin</artifactId>
 <version>${apt.version}</version>
```

```xml
<dependencies>
<dependency>
<groupId>com.querydsl</groupId>
 <artifactId>querydsl-apt</artifactId>
 <version>${querydsl.version}</version>
 </dependency>
 </dependencies>
 <executions>
 <execution>
 <phase>generate-sources</phase>
 <goals>
 <goal>process</goal>
 </goals>
 <configuration>
 <outputDirectory>target/generated-sources/queries</outputDirectory>
 <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>
 <logOnlyOnError>true</logOnlyOnError>
 </configuration>
 </execution>
 </executions>
 </plugin>
 </plugins>
</build>
```

执行`mvn compile`之后,可以找到该`target/generated-sources/java`,然后IDEA标示为源代码目录即可.



QuerydslPredicateExecutor查询结果:

```java
/*
等于 EQ : equal .eq
不等于 NE : not equal .ne
小于 LT : less than .lt
大于 GT : greater than .gt
小于等于 LE : less than or equal .loe
大于等于 GE : greater than or equal .goe
*/

@Autowired
CustomerDslRepository repository;

@Test
public void test02(){
 QCustomer qCustomer = QCustomer.customer;
 Iterable<Customer> all = repository.findAll(qCustomer.id.in(1L, 5L).and(qCustomer.firstName.in("徐庶", "王五")));
 System.out.println(all);

```

```
}
```

自定义查询结果

```
@Autowired
CustomerDslRepository repository;

@PersistenceContext
EntityManager em;

@Test
public void test01(){
 JPAQueryFactory queryFactory = new JPAQueryFactory(em);
 QCustomer qCustomer = QCustomer.customer;

 QueryResults<Tuple> tupleQueryResults = queryFactory.from(qCustomer)
 .select(qCustomer.id.sum(), qCustomer.id)
 .where(
 qCustomer.id.between(1, 2)
 )
 .orderBy(qCustomer.id.desc())
 .groupBy(qCustomer.id)
 .fetchResults();
 for (Tuple result : tupleQueryResults.getResults()) {

 System.out.println(result.get(qCustomer.id));
 System.out.println(result.get(qCustomer.id.sum()));
 }

}
```

多表关联操作

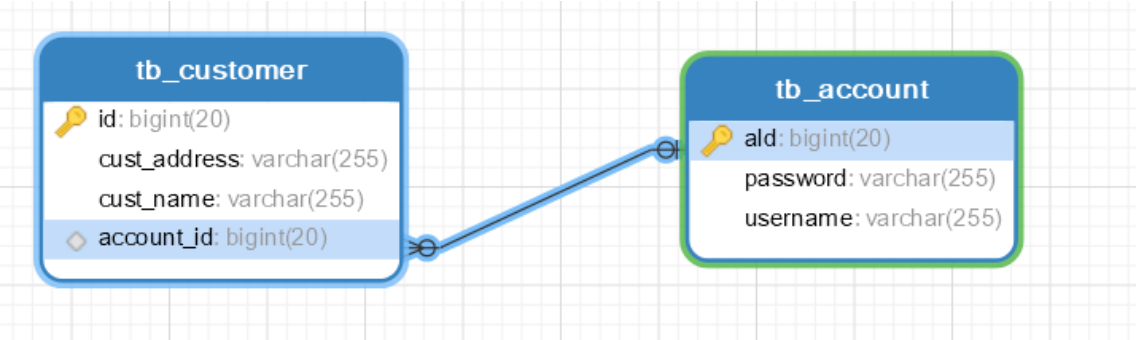
```
<dependency>
 <groupId>org.projectlombok</groupId>
 <artifactId>lombok</artifactId>
 <version>1.18.22</version>
</dependency>
```

一对一

客户表—>老婆表

客户表--->账户表

...



实现：

1. 配置管理关系

@OneToOne

@JoinColumn(name="外键字段名")

```
@OneToOne(mappedBy = "customer",cascade = {CascadeType.PERSIST,CascadeType.REMOVE})
@JoinColumn(name="wife_id")
private Wife wife;

```

Wife

```

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 *
 * 一对一
 * 一个客户对一个账户
 */
@Entity
@Table(name="tb_account")
@Data
/*@Getter // 生成所有属性的get方法
@Setter // 生成所有属性的set方法
@RequiredArgsConstructor // 生成final属性的构造函数， 如果没有final就是无参构造函数
@EqualsAndHashCode*/
public class Account {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String username;
 private String password;
}
```

2. 配置关联操作：

```
// 单向关联 一对一
 /*
 * cascade 设置关联操作
 * ALL，所有持久化操作
 PERSIST 只有插入才会执行关联操作
 MERGE，只有修改才会执行关联操作
 REMOVE，只有删除才会执行关联操作
 fetch 设置是否懒加载
 EAGER 立即加载（默认）
  LAZY 懒加载（ 直到用到对象才会进行查询，因为不是所有的关联对象 都需要用到）
  orphanRemoval 关联移除（通常在修改的时候会用到）
  一旦把关联的数据设置null ，或者修改为其他的关联数据， 如果想删除关联数据， 就可以设置true
  optional 限制关联的对象不能为null
```

```
 true 可以为null(默认 ) false 不能为null
 mappedBy 将外键约束执行另一方维护(通常在双向关联关系中，会放弃一方的外键约束)
 值= 另一方关联属性名
 * */
 @OneToOne(mappedBy = "customer",
 cascade = CascadeType.ALL,fetch = FetchType.LAZY,orphanRemoval=true,optional=false)
 // 设置外键的字段名
 @JoinColumn(name="account_id")
 private Account account;
}
```

测试：

```

@ContextConfiguration(classes = SpringDataJPAConfig.class)
@RunWith(SpringJUnit4ClassRunner.class)
public class OneToOneTest {
 @Autowired
 CustomerRepository repository;

 // 插入
 @Test
 public void testC(){
 // 初始化数据
 Account account = new Account();
 account.setUsername("xushu");

 Customer customer = new Customer();
 customer.setCustName("徐庶");
 customer.setAccount(account);

 account.setCustomer(customer);

 repository.save(customer);
 }


 // 插入
 @Test
 // 为什么懒加载要配置事务 ：
 // 当通过repository调用完查询方法，session就会立即关闭， 一旦session你就不能查询，
 // 加了事务后， 就能让session直到事务方法执行完毕后才会关闭
 @Transactional(readOnly = true)
 public void testR(){
 Optional<Customer> customer = repository.findById(3L); // 只查询出客户， session关闭
 System.out.println("==================");
 System.out.println(customer.get()); // toString
 }


 @Test
 public void testD(){
 repository.deleteById(1L);
 }



 @Test
 public void testU(){

 Customer customer = new Customer();
 customer.setCustId(7L);
 customer.setCustName("徐庶");
```

```
51   customer.setAccount(null);
52   repository.save(customer);
53   }
54 }
```

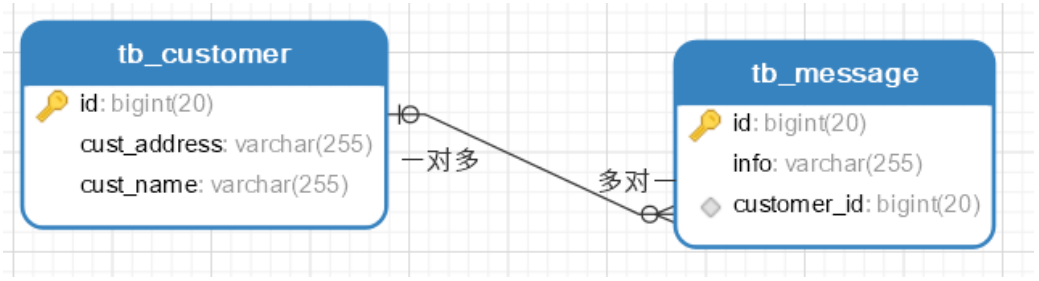
差异:-

这两个设置之间的区别在于对 断开关系.例如，当设置 地址字段设置为null或另一个Address对象.

- 如果指定了 **orphanRemoval = true** ， 则会自动删除断开连接的Address实例.这对于清理很有用 没有一个不应该存在的相关对象(例如地址) 来自所有者对象(例如员工)的引用.
- 如果仅指定 **cascade = CascadeType.REMOVE** ， 则不会执行任何自动操作，因为断开关系不是删除操作

一对多

一个客户 有多条信息



实现：

1. 配置管理关系

@OneToMany

@JoinColumn(name="customer_id")

```
1
2
3 @OneToMany
4 @JoinColumn(name="customer_id")
5 private List<Message> message;
```

2. 配置关联操作：

...

多对一

实现：

1. 配置管理关系

@ManyToOne

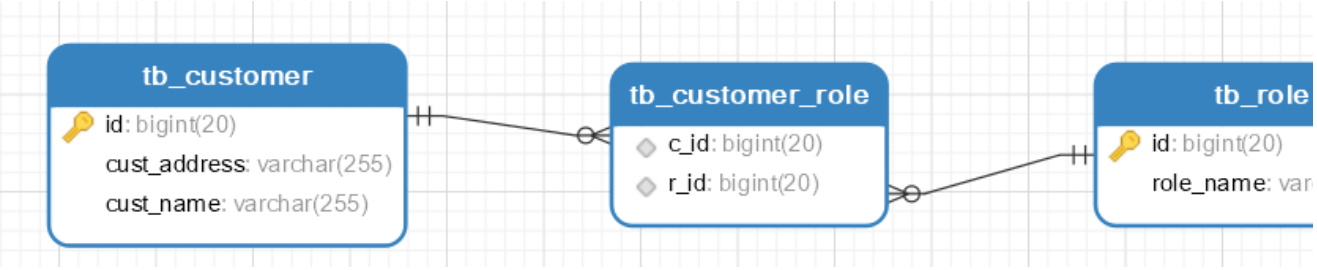
@JoinColumn(name="customer_id")

```
1
2
3 @ManyToOne
4 @JoinColumn(name="customer_id")
5 private List<Message> message;
```

2. 配置关联操作：

...

多对多



1. 配置管理关系

@ManyToMany

@JoinColumn(name="customer_id")

```
@ManyToMany
@JoinTable(name="tb_customer_role",joinColumns={@JoinColumn(name="c_id")},inverseJoinColumns={@JoinColumn(name="r_id")})
private List<Role> roles;
```

2. 配置关联操作:

...

3.测试

```
// 保存
/*
1.如果保存的关联数据 希望使用已有的 ， 就需要从数据库中查出来（持久状态）。否则 提示 游离状态不能持久化
2.如果一个业务方法有多个持久化操作， 记得加上@Transactional ， 否则不能共用一个session
3. 在单元测试中用到了@Transactional ， 如果有增删改的操作一定要加@Commit
4. 单元测试会认为你的事务方法@Transactional， 只是测试而已， 它不会为你提交事务， 需要单独加上 @Commit
 */
@Test
@Transactional
 @Commit
 public void testC() {

  List<Role> roles=new ArrayList<>();
  roles.add(roleRepository.findById(9L).get());
  roles.add(roleRepository.findById(10L).get());

  Customer customer = new Customer();
  customer.setCustName("诸葛");
  customer.setRoles(roles);

  repository.save(customer);
 }


 @Test
 @Transactional(readOnly = true)
 public void testR() {


  System.out.println(repository.findById(14L));

  //repository.save(customer);
 }

```

```
/*
 * 注意加上
 * @Transactional
  @Commit
  多对多其实不适合删除， 因为经常出现数据出现可能除了和当前这端关联还会关联另一端，此时删除就会： ConstraintViolationExceptio
n。
  * 要删除， 要保证没有额外其他另一端数据关联
 * */
@Test
@Transactional
@Commit
public void testD() {


  Optional<Customer> customer = repository.findById(14L);

  repository.delete(customer.get());
}
```

乐观锁

hibernate

防并发修改

```
private @Version Long version;
```

审计

如何使用审计功能

首先申明实体类，需要在类上加上注解@EntityListeners(AuditingEntityListener.class)，其次在application启动类中加上注解EnableJpaAuditing，同时在需要的字段上加上@CreatedDate、@CreatedBy、@LastModifiedDate、@LastModifiedBy等注解。这个时候，在jpa.save方法被调用的时候，时间字段会自动设置并插入数据库，但是CreatedBy和LastModifiedBy并没有赋值，因为需要实现AuditorAware接口来返回你需要插入的值。

1.编写AuditorAware

```
/**
 * 监听
 * @CreatedBy
 * @LastModifiedBy
 * 自动注入用户名
 */
@Configuration
public class UserAuditorAware implements AuditorAware<String> {


  @Override
  public Optional<String> getCurrentAuditor() {
  //TODO: 根据实际情况取真实用户
  return Optional.of("admin");
  }
}
```

2.在实体类中声明@EntityListeners和相应的注解

考虑到所有实体都需要声明，就写在BaseEntityModel 中

```
@MappedSuperclass
```

```java
@EntityListeners(AuditingEntityListener.class)
public class BaseEntityModel implements Serializable {

 /**
 *
 */
 private static final long serialVersionUID = -6163675075289529459L;

 @JsonIgnore
 String entityName = this.getClass().getSimpleName();

 @CreatedBy
 String createdBy;

 @LastModifiedBy
 String modifiedBy;
 /**
 * 实体创建时间
 */
 @Temporal(TemporalType.TIMESTAMP)
 @CreatedDate
 protected Date dateCreated = new Date();

 /**
 * 实体修改时间
 */
 @Temporal(TemporalType.TIMESTAMP)
 @LastModifiedDate
 protected Date dateModified = new Date();

 #省略getter setter
}
```

3.在Application 中启用审计@EnableJpaAuditing

```java

@EnableJpaAuditing

```

```xml
<!--spring-test -->
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-aspects</artifactId>
 <version>5.3.10</version>
 <scope>test</scope>
</dependency>
```

经过测试如果你的实体类上面的多个字段使用了@CreatedBy这样的注解，只会有一个生效，也就是说在一次请求中，只会被调用一次

原理

1.源码（难道）

2.对java高级知识 反射、动态代理..

3.对spring源码有一定了解

4.尽量当做之前没有学习过任何一个源码框架来进行讲解

5手写核心机制的源码

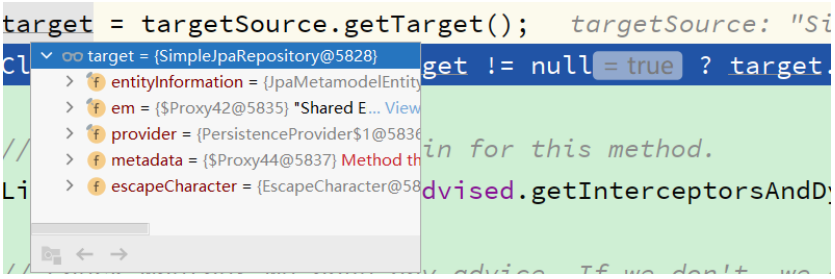
6.总结、画图

7.先关注主线、 细节

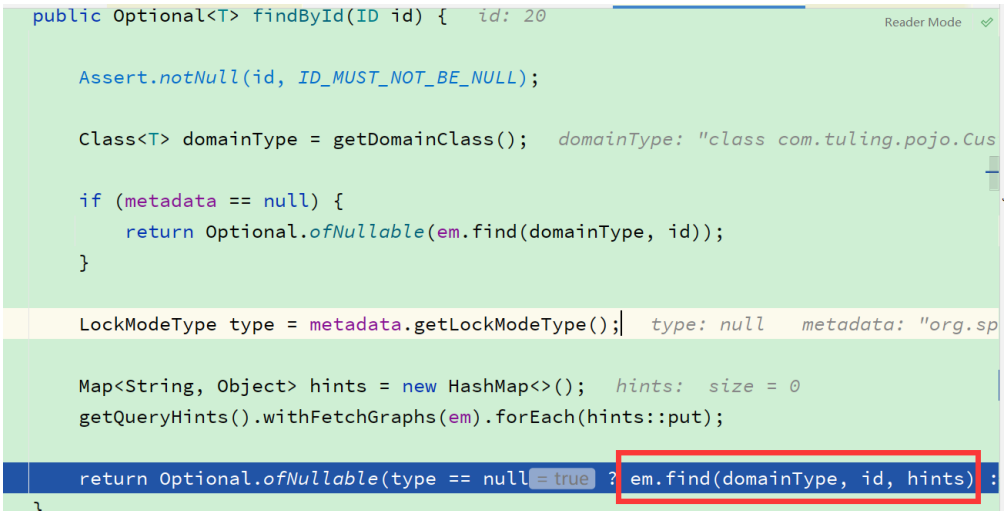
8. 记关键点

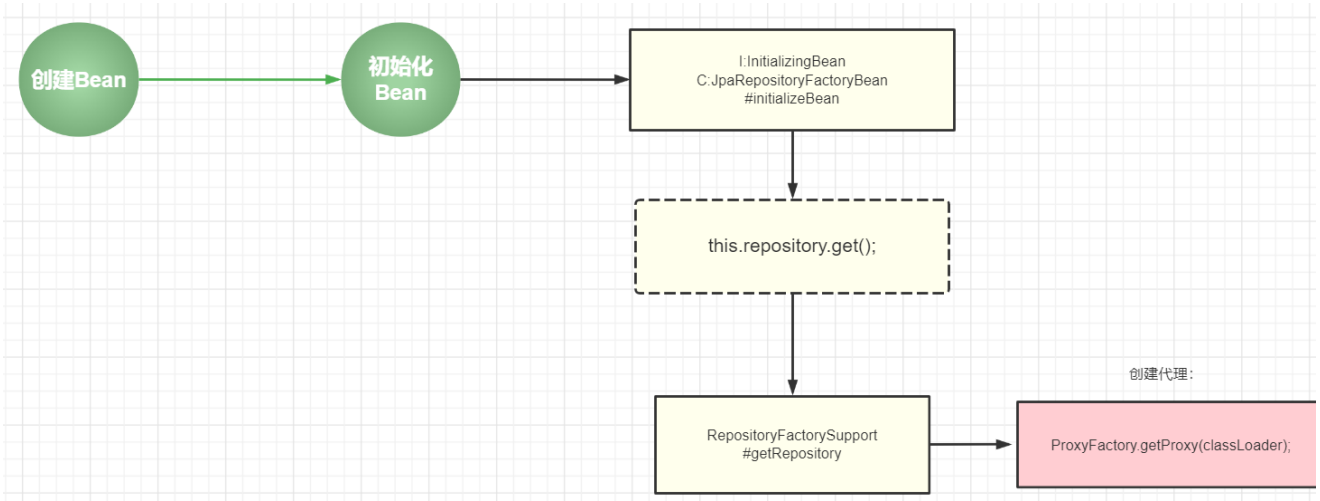
Repository原理

- **核心机制：** **动态代理：**
  - 1. JdkDynamicAopProxy#invoke
- **调用JPA的Repository统一实现**
  - SimpleJpaRepository



- **就是去通过JPA的API完成的持久化操作**





Spring整合jpa原理

手写核心流程

1. Spring怎么管理Repository(怎么创建的Repository的Bean)

1.@EnableJpaRepositories(basePackages="com.tuling.repositories")

2. Spring容器启动的时候ioc 容器加载 根据 "com.tuling.repositories" 去创建Bean

```
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException:
No qualifying bean of type 'com.tuling.repositories.CustomerRepository' available

```

1. 没有找到Bean 解决思路：

应用层面：是不是配置不正确， 配置正确， 排除

底层层面：是不是spring底层扫描 排除。 （Bean --->对象）

1. 同自定义扫描器， 让它将接口包含在内， （必须实现Repository）

1.1 重写isCandidateComponent

```
@Override
protected boolean isCandidateComponent(AnnotatedBeanDefinition beanDefinition) {
 AnnotationMetadata metadata = beanDefinition.getMetadata();
 return metadata.isInterface();
}
```

2. 实现BeanDefinitionRegistryPostProcessor ，动态注册BeanDefinition

1.1 调用自定义扫描器的scan 进行扫描

2. Spring怎么将动态代理创建bean

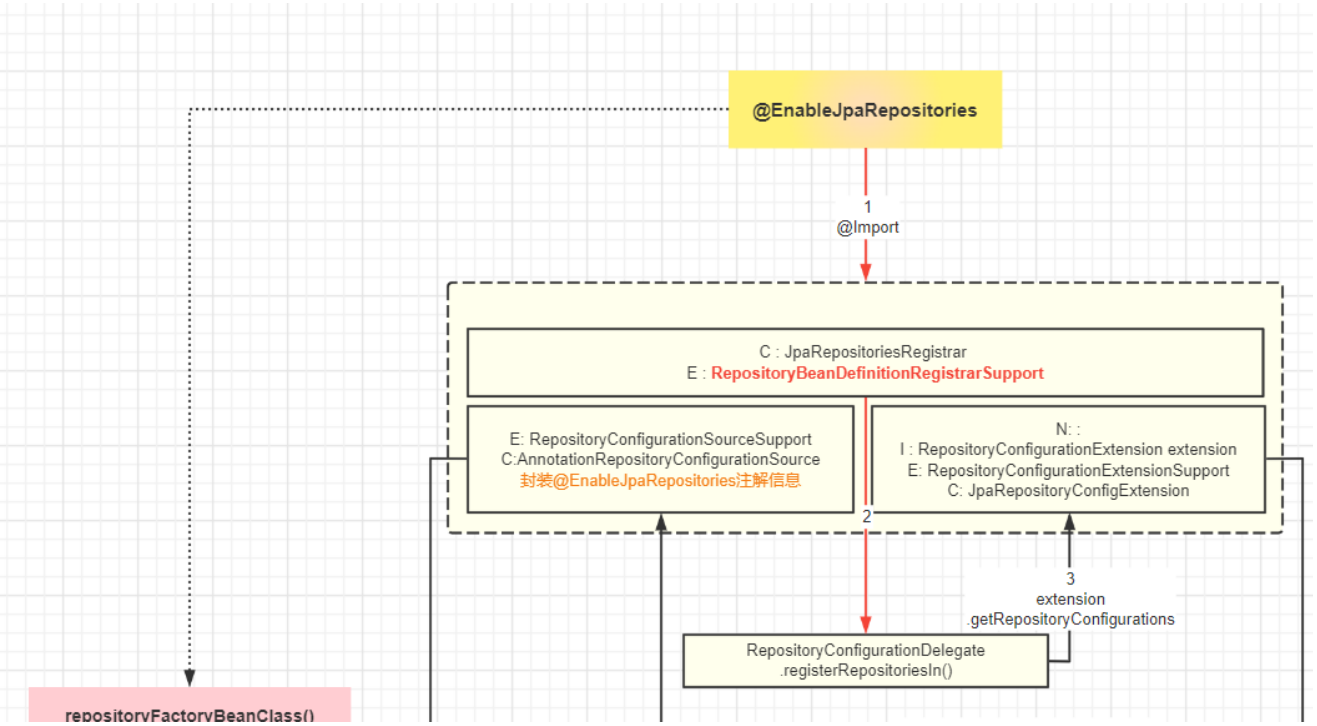
怎么将repository的BeanDefinition和动态代理结合：

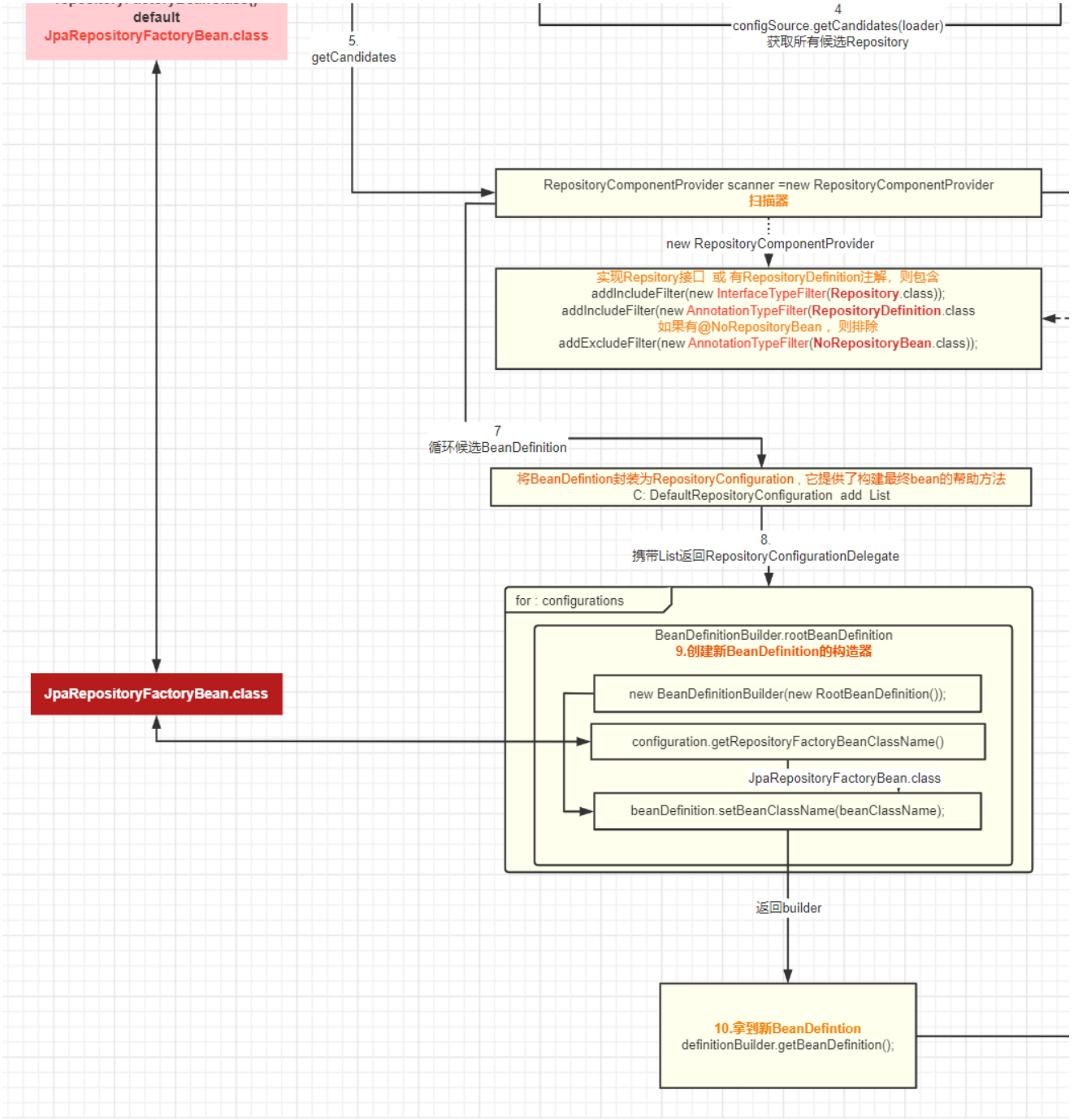
1.有什么方式可以随意去控制bean实例化过程。

@Component

@Bean

FacotryBean动态设置Repository的接口类型， getObject() 自由控制实例化过程 —> 创建动态代理





Spring Data Jpa源码过程：

1. @EnableJpaRepositories(basePackages="com.tuling.repositories")
2. @Import(JpaRepositoriesRegistrar.class)
3. JpaRepositoriesRegistrar实现了ImportBeanDefinitionRegistrar， 就拥有了动态注册BeanDefinition的能力 = BeanDefinitionRegistryPostProcessor的功能
4. 自定义扫描器RepositoryComponentProvider，重写排除接口的方法isCandidateComponent
5. 根据扫描成功候选BeanDefinition重写创建beanDefintion
6. 设置新的beanDefintion为JpaRepositoryFactoryBean
7. JpaRepositoryFactoryBean是一个BeanFactory

8. JpaRepositoryFactoryBean 会创建动态代理

SpringBoot+SpringData Jpa

依赖：

```
<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>
<dependency>
 <groupId>org.projectlombok</groupId>
 <artifactId>lombok</artifactId>
 <optional>true</optional>
</dependency>
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-test</artifactId>
 <scope>test</scope>
</dependency>
```

常用配置：

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/springdata_jpa?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql: true
```

可配置项：

`spring.jpa.database`		要操作的目标数据库，默认 “databasePlatform”属性进
`spring.jpa.database-platform`		要操作的目标数据库的名称 测。也可以使用“数据库”枚
`spring.jpa.generate-ddl`	`false`	是否在启动时初始化架构。
`spring.jpa.hibernate.ddl-auto`		DDL 模式。这实际上是 “hibernate.hbm2ddl.auto 用嵌入式数据库且未检测到 “create-drop”。否则，默认
`spring.jpa.hibernate.naming.implicit-strategy`		隐式命名策略的完全限定名
`spring.jpa.hibernate.naming.physical-strategy`		物理命名策略的完全限定名
`spring.jpa.hibernate.use-new-id-generator-mappings`		是否为 AUTO、TABLE 和 S Hibernate 较新的 Identifie 上是“hibernate.id.new_ge 性的快捷方式。未指定时将
`spring.jpa.mapping-resources`		映射资源（相当于persiste “mapping-file”条目）。
`spring.jpa.open-in-view`	`true`	注册 OpenEntityManager JPA EntityManager 绑定到 处理。
`spring.jpa.properties.*`		要在 JPA 提供程序上设置的
`spring.jpa.show-sql`	`false`	是否启用 SQL 语句的日志记

- ## jpa.generate-ddl和jpa.hibernate.ddl-auto

jpa.generate-ddl和jpa.hibernate.ddl-auto都可以控制是否执行datasource.schema脚本，来初始化数据库结构，只要有一个为可执行状态就会执行，比如jpa.generate-ddl:true或jpa.generate-ddl:update，并没有相互制约上下级的关系。
要想不执行，两者都必须是不可执行状态，比如false和none。
采用implicit-strategy和physical-strategy两个配置项分别控制命名策略

- ## naming.implicit-strategy和naming.physical-strategy

```
spring.jpa.hibernate.naming.implicit-strategy=org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

1、implicit-strategy和physical-strategy的区别

（1）、implicit-strategy负责模型对象层次的处理，将对象模型处理为逻辑名称。physical-strategy负责映射成真实的数据名称的处理，将上述的逻辑名称处理为物理名称。

（2）、当没有使用@Table和@Column注解时，implicit-strategy配置项才会被使用，当对象模型中已经指定时，implicit-strategy并不会起作用。

physical-strategy一定会被应用，与对象模型中是否显式地指定列名或者已经被隐式决定无关。

2、implicit-strategy逻辑名称命名策略

有五个配置值：

不会去改：

```
ImplicitNamingStrategyJpaCompliantImpl：默认的命名策略，兼容JPA 2.0的规范；
ImplicitNamingStrategyLegacyHbmImpl：兼容Hibernate老版本中的命名规范；
ImplicitNamingStrategyLegacyJpaImpl：兼容JPA 1.0规范中的命名规范
ImplicitNamingStrategyComponentPathImpl：大部分与ImplicitNamingStrategyJpaCompliantImpl，但是对于@Embedded等注解标志的组件处理是通过使用attributePath完成的，
因此如果我们在使用@Embedded注解的时候，如果要指定命名规范，可以直接继承这个类来实现；

默认为ImplicitNamingStrategyJpaCompliantImpl，后四者均继承自它。
```

3、physical-strategy物理名称命名策略

有两个配置值：

```
#直接映射，不会做过多的处理
org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
#表名，字段为小写，当有大写字母的时候会添加下划线分隔符号
org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy
 默认为SpringPhysicalNamingStrategy
```

自动配置原理：

通过JpaBaseConfiguration去配置我们以前集成spring data jpa的基本@Bean

JpaRepositoriesAutoConfiguration

```
@Import(JpaRepositoriesImportSelector.class)
JpaRepositoriesImportSelector又会注册一个ImportBeanDefinitionRegistrar 其实就是JpaRepositoriesRegistrar
----JpaRepositoriesRegistrar 跟通过@EnableJpaRepositories 导入进来的组件是同一个
```

就相当于@EnableJpaRepositories