JPA+SpringData

1. JPA 映射关联关系

1.1. 双向一对多

1.1.1. 多对一

一对多关系中，必须存在一个关系维护端，在 JPA 规范中，要求 many 的一方作为关系的维护端(owner side)，one 的一方作为被维护端(inverse side)。
在 many 方指定 @ManyToOne 注释，并使用@JoinColumn 指定外键名称

```
//多对一
@ManyToOne(targetEntity=Grade.class,fetch=FetchType.LAZY)
@JoinColumn(name="gid")
public Grade getGrade() {
    return grade;
}
```

1.1.2. 一对多

可以在 one 方指定 @OneToMany 注释并设置 mappedBy 属性，以指定它是这一关联中的被维护端，many 为维护端。

```
//mappedBy映射的属性值要是 一方的属性值
@OneToMany(mappedBy="grade")
public List<Student> getLists() {
    return lists;
}
```

1.1.3. 总结：

一对多配置：默认是懒加载
多对一配置：立刻(急)加载

```java
//单向一对多配置
/**
 * fetch抓取策略
 *    FetchType.LAZY   懒加载
 *    FetchType.EAGER 急加载
 */
@OneToMany(mappedBy="grade")   //默认是懒加载
private List<Student> students;
```

```java
@Data
@Entity
@Table
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String sname;
    @Transient
    private String gid;

    @ManyToOne(fetch=FetchType.LAZY) //默认是急加载
    @JoinColumn(name="gid")
    private Grade grade;
```

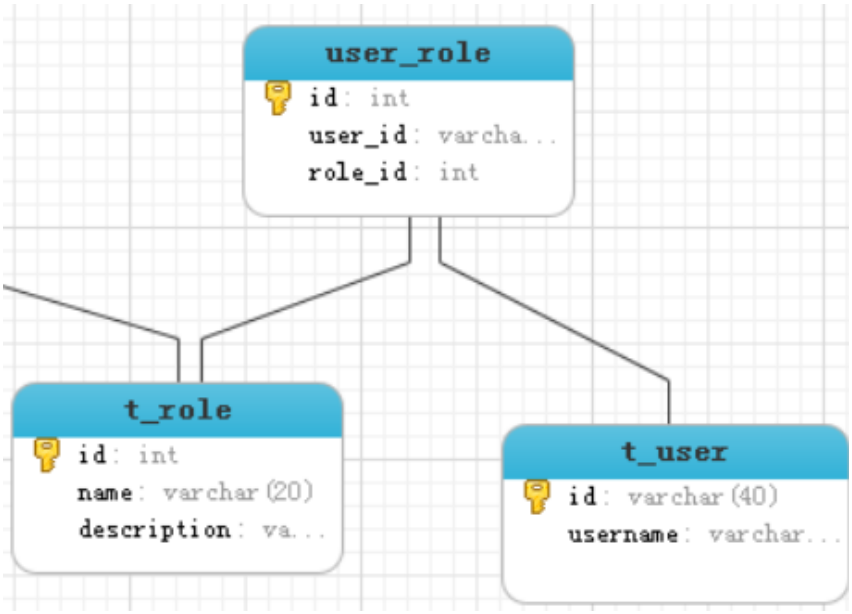
1.2. 双向一对一

基于外键的 1-1 关联关系：在双向的一对一关联中，需要在关系被维护端(inverse side)中的 @OneToOne 注释中指定 mappedBy，以指定是这一关联中的被维护端。同时需要在关系维护端(owner side)建立外键列指向关系被维护端的主键列。

```java
//Teacher中有一个外键GID指向班级表主键ID
//通过JoinColumn指定字段名称，同时指定是一对一unique=true
@JoinColumn(name="gid",unique=true)
@OneToOne(fetch=FetchType.LAZY)
private Grade grade;
```

```java
//对于不维护对方关联关系，无外键方。
//使用OneToOne mappedBy（使用对方的哪个属性维护关联关系）
@OneToOne(mappedBy="grade")
private Teacher teac;
```

1.3. 双向多对多



多对多关系， 我们先确定谁是多对多，关系的维护端，谁是被维护端

多对多关系的维护和被维护端可自行定义即可。此处我们将 Users 定为维护端

Users 实体类中：

```java
// 多对多 角色下的多个用户
@ManyToMany
@JoinTable(name="users_roles",//指定中间表的表名
    //指定当前表在中间表的外键名称和外键所指向的当前表主键 uid为users_roles键，id为Users主键
    joinColumns={@JoinColumn(name="uid",referencedColumnName="id")},
    //rid指定users_roles外键。rid为Roles主键
    inverseJoinColumns={@JoinColumn(name="rid",referencedColumnName="rid")}
)
private Set<Roles> rolesSet;
```

Roles 实体类中：（被维护端）

```
@ManyToMany(mappedBy="rolesSet")
private List<Users> usersSet;
```

2. JPQL

JPQL 语言，即 Java Persistence Query Language 的简称。JPQL 是一种和 SQL 非常类似的中间性和对象化查询语言，它最终会被编译成针对不同底层数据库的 SQL 查询，从而屏蔽不同数据库的差异。

JPQL 语言的语句可以是 select 语句、update 语句或 delete 语句，它们都通过 Query 接口封装执行

2.1. JPQL 是什么

HQL: 对象生成 SQL

HQL: SQL 语句的 表名、字段名—》HQL 中的类名、属性名（代替）严格区分大小写。

HQL:SQL 所有关键字。数据库函数、

HQL→JPQL 没有任何区别。

SpringData—》JPA(规范)→Hiberante(实现)

2.2. Query

javax.persistence.Query

Query 接口封装了执行数据库查询的相关方法。

调用 EntityManager 的 createQuery、createNamedQuery 及 createNativeQuery 方法可以获得查询对象，进而可调用 Query 接口的相关方法来执行查询操作。

Query 接口的主要方法

- int executeUpdate()
  - 用于执行 update 或 delete 语句添加。
- List getResultList()
  - 用于执行 select 语句并返回结果集实体列表。
- Object getSingleResult()
  - 用于执行只返回单个结果实体的 select 语句。

- Query setFirstResult(int startPosition)
  - 用于设置从哪个实体记录开始返回查询结果。
- Query setMaxResults(int maxResult)
  - 用于设置返回结果实体的最大数。与 setFirstResult 结合使用可实现分页查询。
- Query setFlushMode(FlushModeType flushMode)
  - 设置查询对象的 Flush 模式。参数可以取 2 个枚举值：FlushModeType.AUTO 为自动更新数据库记录，FlushModeType.COMMIT 为直到提交事务时才更新数据库记录。

- setHint(String hintName, Object value)
  - **设置与查询对象相关的特定供应商参数或提示信息**。参数名及其取值需要参考特定 JPA 实现库提供商的文档。如果第二个参数无效将抛出 IllegalArgumentException 异常。
- setParameter(int position, Object value)
  - 为查询语句的指定位置参数赋值。Position 指定参数序号，value 为赋给参数的值。
  - setParameter(int position, Date d, TemporalType type)
  - 为查询语句的指定位置参数赋 Date 值。Position 指定参数序号，value 为赋给参数的值，temporalType 取 TemporalType 的枚举常量,包括 DATE、TIME 及 TIMESTAMP 三个,，用于将 Java 的 Date 型值临时转换为数据库支持的日期时间类型（java.sql.Date、java.sql.Time 及 java.sql.Timestamp）。

- setParameter(String name, Object value)
  - 为查询语句的指定名称参数赋值。
- setParameter(String name, Date d, TemporalType type)
  - 为查询语句的指定名称参数赋 Date 值。用法同前。
- setParameter(String name, Calendar c, TemporalType type)
  - 为查询语句的指定名称参数设置 Calendar 值。name 为参数名,其它同前。该方法调用时如果参数位置或参数名不正确，或者所赋的参数值类型不匹配，将抛出 IllegalArgumentException 异常。

2.3. JPQL 支持三种参数方式

1、JPQL 也支持包含参数的查询，（位置参数）例如：

注意：参数名前必须冠以冒号(:)，执行查询前须使用 Query.setParameter(name, value)方法给参数赋值。

其中 ?1 代表第一个参数，?2 代表第一个参数

```java
//根据用户名和密码查询
public void test3(){
    EntityManager en = JpaUtils.get();
    //使用占位符： ?编号
    String jpql="from Users  where uname=?1 and upass=?2";
    Query query=en.createQuery(jpql);
    query.setParameter(1,"guoweixin");
    query.setParameter(2,"2");
    System.out.println(query.getSingleResult());
    JpaUtils.close();
}
```

2、

```java
String jpql = "delete Users where id=? ";
Query query = en.createQuery(jpql);
query.setParameter(0, 6);
int num = query.executeUpdate();
```

3、命名参数方式 :uname

```java
Query q = em.createQuery("SELECT u FROM Userinfo u
WHERE u.usertype = :usertype");
q = q.setParameter("usertype",usertype);
```

2.4. 查询部分属性

如果只须查询实体的部分属性而不需要返回整个实体。执行该查询返回的不再是 Users 实体集合，而是一个对象数组的集合(Object[])，集合的每个成员为一个对象数组，可通过数组元素访问各个属性（投影）

```java
//查询部分属性
public void test7() {
    EntityManager en = JpaUtils.get();
    String jpql = "select new Users(uname,upass) from Users  where id in (7,6,3)";
    Query query = en.createQuery(jpql);
    System.out.println(query.getResultList());
    JpaUtils.close();
}
```

```java
public Users(String uname, String upass) {
    super();
    this.uname = uname;
    this.upass = upass;
}
```

2.5. createNativeQuery

entityManager.createNativeQuery()来执行原生的 SQL 语句

标量原生查询

Query createNativeQuery(String sql)

这将建立一个原生查询返回一个标量结果.它需要一个参数:你的原生 SQL.它执行并且返回结果集的形式,返回标量值

简单的实体原生查询

Query createNativeQuery(String sql, Class entityClass)

第一个参数是 SQL 语句，第二个参数是查询的结果类型，JPA 会把查询结果转换为对象

复杂原生查询

Query createNativeQuery(String sql, String resultSetMapping)

第一个参数是 SQL 语句，第二个参数 把结果封装为自己希望的类型

结果映射可以采用 SqlResultSetMapping 注释，如果有多个，采用 SqlResultSetMappings,每个 SqlResultSetMapping 包含如下信息：

name:映射的名字，与方法的第二个参数 resultSetMapping 相同

entities:映射成多个实体

每个实体使用一个 EntityResult 表示，每个 EntityResult 包含如下部分：

entityClass：指出要映射的实体类

fields:指出要映射哪些属性，每个属性使用一个 FieldResult 表示。

FieldResult 包含：name (实体类属性名字)

column(对应查询结果列的名字）

columns:映射成多个列,每个列的映射使用 ColumnResult 表示,每个 ColumnResult 使用 name 属性指出查询结果对应的列

2.6. Update/Delete 语句

update 语句用于执行数据更新操作。主要用于针对单个实体类的批量更新

delete 语句用于执行数据更新操作。

Int executeUpdate();

3. 整合 Spring+JPA+SpringMVC

通过 maven 完成 Jar 整合
整合：

- JPA 核心配置文件，得到 EntityManagerFactory
- 事务管理

三种整合方式：

- LocalEntityManagerFactoryBean：适用于那些仅使用 JPA 进行数据访问的项目，该 FactoryBean 将根据 JPA PersistenceProvider 自动检测配置文件进行工作，一般从“META-INF/persistence.xml”读取配置信息，这种方式最简单，但**不能设置 Spring 中定义的 DataSource，且不支持 Spring 管理的全局事务**
- 从 JNDI 中获取：**用于从 Java EE 服务器获取指定的 EntityManagerFactory**，这种方式在进行 Spring 事务管理时一般要使用 JTA 事务管理
- LocalContainerEntityManagerFactoryBean：适用于所有环境的 FactoryBean，能全面控制 EntityManagerFactory 配置，如指定 Spring 定义的 DataSource 等等。

3.1. POM.XML

```
<properties>

   <project.build.sourceEncoding>UTF-8</project.build.sourceEn
coding>
      <!-- 自定义版本号 -->
      <spring.version>4.3.8.RELEASE</spring.version>
   </properties>
   <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>3.8.1</version>
         <scope>test</scope>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.8.0</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
```

```
            <version>1.8.0</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>4.3.9.RELEASE</version>
        </dependency>
        <!-- jackson jar -->
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.8.6</version>
        </dependency>
        <!-- SpringMVC文件上传.jar -->
        <dependency>
            <groupId>commons-fileupload</groupId>
            <artifactId>commons-fileupload</artifactId>
            <version>1.3.1</version>
        </dependency>
        <!-- spring 对jpa的支持 -->
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-jpa</artifactId>
            <version>1.11.4.RELEASE</version>
        </dependency>
        <!-- Hibernate jar -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.2.9.Final</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>5.2.10.Final</version>
        </dependency>
        <!-- Hibernate jpa -->
        <dependency>
            <groupId>org.hibernate.javax.persistence</groupId>
            <artifactId>hibernate-jpa-2.1-api</artifactId>
            <version>1.0.0.Final</version>
        </dependency>
        <!-- Mysql -->
```

```
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.42</version>
        </dependency>
    <!-- druid连接池 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.10</version>
        </dependency>
        <!-- lombok.jar -->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.16.18</version>
            <scope>provided</scope>
        </dependency>
        <!--JSTL标签 -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>
        <dependency>
            <groupId>taglibs</groupId>
            <artifactId>standard</artifactId>
            <version>1.1.2</version>
        </dependency>
        <!-- slft4j .jar -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.25</version>
        </dependency>
    </dependencies>
```

3.2. spring_core.XML

```
<!--指定实现JPA的适配器-->
<bean id="jpaVendorAdapter" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"></bean>
<!-- Jpa集成Spring -->
<!-- 配置EntityManagerFactory -->
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <!-- 指定实现JAP的适配器 -->
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter"></property>
    <!-- 待扫描的实体类的包 -->
    <property name="packagesToScan" value="com.qfjy.bean"></property>
    <!-- 设置JPA需要的基本属性信息 -->
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.show_sql" >true</prop>
            <prop key="hibernate.format_sql">true</prop>
        </props>
    </property>
</bean>

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

<!-- Spring配置 声明式注解事务 -->
<tx:annotation-driven transaction-manager="transactionManager" />
```

```
   <context:component-scan   base-package="com.qfjy"
      use-default-filters="true">
      <!-- exclude-filter是针对include-filter里的内容进行排除 -->
      <context:exclude-filter type="annotation"

   expression="org.springframework.stereotype.Controller" />
   </context:component-scan>
   <!-- AOP配置    -->
   <aop:aspectj-autoproxy></aop:aspectj-autoproxy>


   <context:property-placeholder
location="classpath:jdbc.properties" />



   <bean id="dataSource"
class="com.alibaba.druid.pool.DruidDataSource"
      init-method="init" destroy-method="close">
      <property name="driverClassName" value="${jdbc_driver}" />
```

```
        <property name="url" value="${jdbc_url}" />
        <property name="username" value="${jdbc_user}" />
        <property name="password" value="${jdbc_password}" />
        <!-- 配置初始化大小、最小、最大 -->
        <property name="initialSize" value="1" />
        <property name="minIdle" value="1" />
        <property name="maxActive" value="10" />

        <!-- 配置获取连接等待超时的时间 -->
        <property name="maxWait" value="10000" />

        <!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位
是毫秒 -->
        <property name="timeBetweenEvictionRunsMillis"
value="60000" />

        <!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->
        <property name="minEvictableIdleTimeMillis"
value="300000" />

        <property name="testWhileIdle" value="true" />

        <!-- 这里建议配置为TRUE，防止取到的连接不可用 -->
        <property name="testOnBorrow" value="true" />
        <property name="testOnReturn" value="false" />

        <!-- 打开PSCache，并且指定每个连接上PSCache的大小 -->
        <property name="poolPreparedStatements" value="true" />
        <property
name="maxPoolPreparedStatementPerConnectionSize"
            value="20" />
        <!-- 这里配置提交方式，默认就是TRUE，可以不用配置 -->
        <property name="defaultAutoCommit" value="true" />
        <!-- 验证连接有效与否的SQL，不同的数据配置不同 -->
        <property name="validationQuery" value="select 1 " />

        <!-- 配置监控统计拦截的filters -->
        <property name="filters" value="stat" />

    </bean>


    <!--2 Spring JDBC模版 -->
    <bean id="jdbcTemplate"
```

```
class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

     <!--3 spring+ jpa整合 -->
     <bean id="jpaVendorAdapter"
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAd
apter"></bean>
     <bean id="entityManagerFactoryBean"
class="org.springframework.orm.jpa.LocalContainerEntityManager
FactoryBean">
         <!-- 指定数据源 -->
         <property name="dataSource" ref="dataSource"></property>
        <!-- 指定JPA实现产品  -->
        <property name="jpaVendorAdapter"
ref="jpaVendorAdapter"></property>
        <!-- 待扫描的实体类的包 -->
        <property name="packagesToScan"
value="com.qfjy.bean"></property>
        <!-- 设置 自定义配置 hibernate -->
        <property name="jpaProperties">
            <props>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.format_sql">true</prop>
            </props>
        </property>
     </bean>
    <!-- 4 Spring TransctionDataSource
     Jpa事务管理 JpaTranscationManager
     -->
     <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory"
ref="entityManagerFactoryBean"></property>
     </bean>
     <!-- 5 声明式注解事务 -->
    <tx:annotation-driven
transaction-manager="transactionManager"/>
```

3.3. spring_mvc.XML

```
<context:component-scan base-package="com.qfjy.web">
      <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
   </context:component-scan>

   <!-- 处理静态资源 -->
   <!-- 将在 SpringMVC 上下文中定义一个
DefaultServletHttpRequestHandler， 它会对进入 DispatcherServlet
      的 请求进行筛查， 如果发现是没有经过映射的请求，就将该请求交由
WEB 应用服务器默认的 Servlet 处理， 如果不是静态资源的请求，才由
      DispatcherServlet 继续处理 -->
   <!-- 启动注解 是告知Spring，启用注解驱动。然后Spring会自动为我们
注册上面说到多个Bean到工厂中，来处理我们的请求。 主要有两个：
RequestMappingHandlerMapping
      RequestMappingHandlerAdapter      第一个是HandlerMapping的
实现类，它会处理@RequestMapping
      注解，并将其注册到请求映射表中。      第二个是HandlerAdapter
的实现类，它是处理请求的适配器，就是确定调用哪个类的哪个方法，并且构造
方法参数，返回值。
      支持使用 @RequestBody 和 @ResponseBody 注解 -->
   <mvc:default-servlet-handler />
   <mvc:annotation-driven />


   <!-- SpringMVC文件上传 -->
  <!-- Spring pre... jsp  WEB/INF -->
```

3.4. Spring 整合 JPA

```
@Repository
public class UsersDao {
    //如何获取当前的EntityManager?
    //通过该注解来标记变量
    @PersistenceContext
    EntityManager  entityManager;

    public Integer add(Users u){
        entityManager.persist(u);
        System.out.println(u.getId());
        return u.getId();
@Service
public class UsersServiceImpl {
    @Autowired
    UsersDao usersDao;

    @Transactional
    public int add(Users u){
        return usersDao.add(u);
```