Part I. Spring Data JPA 简介

- 什么是Spring Data JPA?

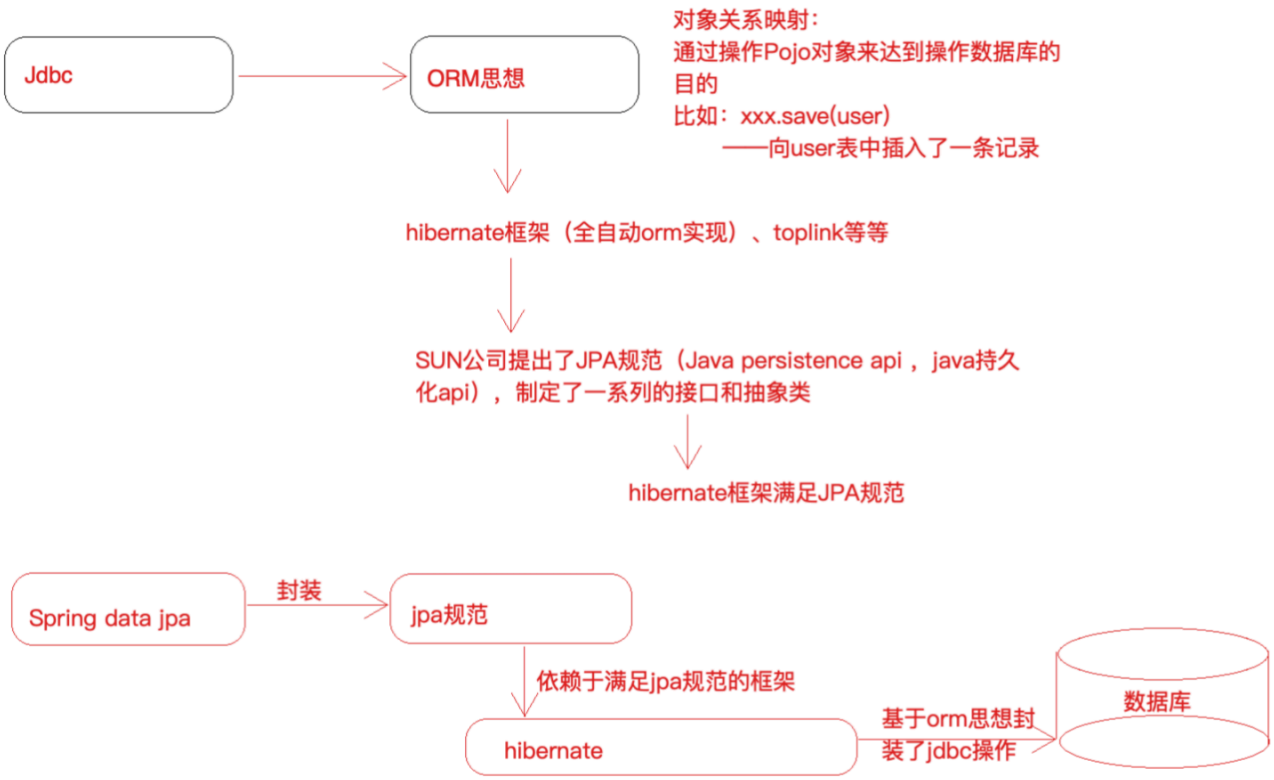
  Spring Data JPA 是 Spring 基于**JPA** 规范的基础上封装的一套 JPA 应用框架，可使开发者用极简的代码（不需要额外手动开发自己的sql语句）即可实现对数据库的访问和操作。Spring Data JPA 是 Spring Data 家族的一员。
- General作用：和Mybatis框架一样作用于Dao层，但是在使用方式和底层机制是有所不同的。
- 最大特点：Spring Data JPA 极大简化了数据访问层代码，Dao层中只需要写接口，不需要写实现类，就自动具有了增删改查、分页查询等方法。

Part II. JPA 规范、Spring Data JPA 和 Hibernate 之间的关系

ORM (Object-Relation Mapping，对象关系映射) 思想是把对表的操作转换成对对象（Pojo）的操作，更便利于Java面向对象的开发形式。

JPA 是一套规范，内部是由接口和抽象类组成的，Hiberanate 是一套成熟的 ORM 框架并实现了 JPA 规范，是 JPA 的一种实现方式，我们使用 JPA 的 API（基于Hibernate） 编程，意味着站在更高的角度去看待问题(面向接口编程)。

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



Part III. Spring Data JPA 应用

1. 需求、表单

- 需求：使用 Spring Data JPA 完成对 tb_resume 表(简历表)的Dao 层操作(增删改查，排序，分页等)
- 表单结构：

名	类型	长度	小数点	不是 null	虚拟	键	注释
id	bigint	20	0	☑	☐	🔑	
address	varchar	255	0	☐	☐		
name	varchar	255	0	☐	☐		
phone	varchar	255	0	☐	☐		

- 创表与初始化SQL：

```
CREATE TABLE `xiaoyudb`.`tb_resume` (
  `id` BIGINT(20) NOT NULL,
  `address` VARCHAR(255) NULL,
  `name` VARCHAR(255) NULL,
  `phone` VARCHAR(45) NULL,
```

```
  PRIMARY KEY (`id`));

INSERT INTO `xiaoyudb`.`tb_resume` (`id`, `address`, `name`, `phone`) VALUES
('1', '北京', 'Philip', '131000');
INSERT INTO `xiaoyudb`.`tb_resume` (`id`, `address`, `name`, `phone`) VALUES
('2', '上海', 'Jenny', '158000');
INSERT INTO `xiaoyudb`.`tb_resume` (`id`, `address`, `name`, `phone`) VALUES
('3', '杭州', 'Hancle', '153000');

-- 让id自增
ALTER TABLE `xiaoyudb`.`tb_resume`
CHANGE COLUMN `id` `id` BIGINT NOT NULL AUTO_INCREMENT ;
```

2. 开发步骤

构建工程：

- 创建project (不需要用web框架了，简单的maven工程即可)，引入jar包
- 配置Spring的配置文件，编写Spring Data JPA 细节
- 编写实体类Resume，使用JPA注解配置映射关系
- 编写符合Spring Data JPA的Dao层接口 ResumeDao Interface
- 操作ResumeDao Interface 完成Dao层开发

3. 开发实现

Step 1: Create a basic maven project and import dependencies in pom.xml

Step 2: 在applicationContext.xml中，对Spring和SpringDataJPA进行配置

1. 创建DB连接池
2. 配置JPA重要对象 entityManagerFactory
   entityManagerFactory 类似于 Mybatis 中的 SQLSessionFactory
   entityManager 类似于 Mybatis 中的 SQLSession，提供了 CRUD 方法

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <!-- 1. 数据源 -->
    <property name="dataSource" ref="dataSource"/>

    <!-- 2. 扫描Pojo类所在包 -->
    <property name="packagesToScan" value="com.lagou.edu.pojo"/>

    <!-- 3. 指定JPA的具体实现，即Hibernate -->
    <property name="persistenceProvider">
      <bean class="org.hibernate.jpa.HibernatePersistenceProvider"/>
    </property>

    <!-- 4. 指定JPA的dialect(方言)
      不同的JPA实现，在具体方法实现（e.g. beginTransaction 等）上有所不同，
      所以传入jpaDialect Interface的具体的实现类，即这里的 HibernateJpaDialect
    -->
    <property name="jpaDialect">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
    </property>

    <!-- 5. 配置具体provider 和 Hibernate框架的执行细节
      传入 jpaVendorAdapter Interface的具体的实现类，即这里的
HibernateJpaVendorAdapter
    -->
    <property name="jpaVendorAdapter">
      <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

      <!-- 5.1 配置数据表是否自动创建
        程序启动时，若数据表还未创建，是否需要程序通过已建好的ORM关系来建表。这里不需要自动
创建。
      -->
        <property name="generateDdl" value="false"/>

      <!-- 5.2 指定数据库类型
        Hibernate是Dao层框架，可以支持多找数据库类型。
      -->
        <property name="database" value="MYSQL"/>

      <!-- 5.3 指定数据库方言
        Hibernate可实现sql语句拼接，但不同数据库sql语法不同，这里需要指明dialect实现
类。
      -->
        <property name="databasePlatform"
value="org.hibernate.dialect.MySQLDialect"/>
```

```
      <!-- 5.4 是否显示sql
        执行数据库操作时，是否打印sql
      -->
        <property name="showSql" value="true"/>
      </bean>
    </property>

  </bean>
```

3. 引用上面的 entityManagerFactory

```
<beans
     xmlns:jpa="http://www.springframework.org/schema/data/jpa"
     xsi:schemaLocation="
       http://www.springframework.org/schema/data/jpa
       http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
">

  <!-- 3. 引用上面的 entityManagerFactory
  jpa:repositories  配置JPA的Dao层细节
  base-package  指定Dao层 Interface 所在pkg
  entity-manager-factory-ref  指定谁来实现 base-pkg 中那些 Interface
-->
  <jpa:repositories base-package="com.lagou.edu.dao" entity-manager-factory-
ref="entityManagerFactory"
            transaction-manager-ref="transactionManager" />

</beans>
```

4. 事务管理器配置

```
<!-- 4. 事务管理器配置
  jdbcTemplate / Mybatis 使用的是 DataSourceTransactionManager
  JPA 规范使用的是 JpaTransactionManager
-->
  <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>
```

5. 声明式事务配置 （与Mybatis类似）

```
<!-- 5. 声明式事务配置 （与Mybatis类似）
  要使用的话，还需引用 beans 约束和相关jar包
 -->
  <!-- <tx:annotation-driven/> -->
```

6. 配置Spring包扫描

```
<context:component-scan base-package="com.lagou.edu"/>
```

Step 3: 编写实体类Resume，使用JPA注解配置映射关系

```
/*
* 简历实体类，使用注解建立：
* 1. 实体类本身 与 数据表 之间的mapping
*   @Entity
*   @Table
*  2. 属性 与 column 之间的mapping  （Pojo中用的camel，DB中用的underscore）
*   @Id  标识主键
*   @GeneratedValue  标识主键的生成策略，常用的有2种
*                       - 依赖DB自带的自增功能（MySQL）：GenerationType.IDENTITY
*                       - 依靠序列来产生主键（Oracle）：GenerationType.SEQUENCE
*   @Column  属性 mapping to column name
* */
@Entity
@Table(name = "tb_resume")
public class Resume {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "address")
    private String address;
```

```
    @Column(name = "phone")
    private String phone;

// setters, getters, toString()
}
```

Step 4: 编写符合Spring Data JPA的Dao层接口 ResumeDao Interface

```
/*
* 一个符号SpringDataJPA要求的Dao层Interface是需要继承 JpaRepository 和
JpaSpecificationExecutor Interface
*
* JpaRepository<操作的实体类类型，主键类型>
        封装了基本的CRUD操作
*
* JpaSpecificationExecutor<操作的实体类类型>
        封装了复杂的查询操作（如分页、排序等）
* */
public interface IResumeDao extends JpaRepository<Resume, Long>,
JpaSpecificationExecutor<Resume> {
}
```

Step 5: 操作ResumeDao Interface 完成Dao层开发

Part I - Test Basic CRUD operations

Part II - 专门针对查询来测试（因为查询通常占比更高）

方式一：即Part I

方式二：在自己的接口中自定义方法，并引入 jpql 语句

方式三：在自己的接口中自定义方法，引入原生sql语句

如果SpringDataJPA框架提供的查询调用起来太麻烦，推荐用原生SQL（其实调用的SpringDataJPA的底层实现也是用的原生SQL）

方式四：命名规则查询

在自己的接口中自定义方法，但不必引入jpql或sql，由于自定义的方法的方法名字是按照一定规则形成的，那么SpringDataJPA框架就能够读懂并自动执行相应语句（https://www.jianshu.com/p/1d6f27f675bb）。

方式五：动态查询

当Service层对Dao层发起Query请求时，可能表中有很多column，Query的花样也会很多，这样如果每一条Query都在Dao层Interface中写一个单独的语句，Dao层Interface就会变很冗杂。所以，将Service层发起Query请求时传过来的东西（即条件们）封装成一个对象（即Specification），然后把对象传给Dao层。那么Dao层Interface就只用handle这种传入参数为Specification来Query的请求即可。例如 IResumeDao extends了下面这个Interface：

```
public interface JpaSpecificationExecutor<T> {
    // 根据不同条件，查单个
    Optional<T> findOne(@Nullable Specification<T> var1);
    // 根据不同条件，查多个
    List<T> findAll(@Nullable Specification<T> var1);
    // 根据不同条件，查询并分页
    Page<T> findAll(@Nullable Specification<T> var1, Pageable var2);
    // 根据不同条件，查询并进行排序
    List<T> findAll(@Nullable Specification<T> var1, Sort var2);
    // 根据不同条件，来统计个数
    long count(@Nullable Specification<T> var1);
}
```

来看看作为唯一参数的 Specification：

```
public interface Specification<T> extends Serializable {
    long serialVersionUID = 1L;

    static <T> Specification<T> not(Specification<T> spec) {
        return Specifications.negated(spec);
    }

    static <T> Specification<T> where(Specification<T> spec) {
        return Specifications.where(spec);
    }

    default Specification<T> and(Specification<T> other) {
        return Specifications.composed(this, other, CompositionType.AND);
    }
```

```
    default Specification<T> or(Specification<T> other) {
        return Specifications.composed(this, other, CompositionType.OR);
    }

    /*
    核心方法，用来拼接 Specification 中的各种条件。
    Root：根属性，所有属性都可从根对象中获取
    CriteriaQuery：自定义查询方式（一般用不上）
    CriteriaBuilder： 查询构造器，封装了很多的查询条件（例如 like，= 等），用来拼接、组合属
性
    */
    @Nullable
    Predicate toPredicate(Root<T> var1, CriteriaQuery<?> var2, CriteriaBuilder
var3);
}
```

所以，要使用动态查询，就需要利用 toPredicate 这个方法。

- 动态查询Given单个条件

```
 @Test
    public void testFindOneGivenSpecification() {
        /*
        * 封装动态条件 -- 利用匿名内部类
        * toPredicate 方法作用：动态组装查询条件
        *
        * 需求：根据 name（指定为"Jenny"）来查询其 Resume
        * SQL = select * from tb_resume where name = 'Jenny'
        * */
        Specification<Resume> specification = new Specification<Resume>() {
            @Override
            public Predicate toPredicate(Root<Resume> root, CriteriaQuery<?>
criteriaQuery, CriteriaBuilder criteriaBuilder) {
                // 获得属性名称 name
                Path<Object> attribute = root.get("name");

                // 使用CriteriaBuilder针对name属性构建条件（这里是精准查询）
                Predicate predicate = criteriaBuilder.equal(attribute,
"Jenny");

                return predicate;
            }
        };
```

```
        Optional<Resume> optional = resumeDao.findOne(specification);
        Resume result = optional.get();
        System.out.println(result);
        // 底层依然是Hibernate，打印结果： Resume{id=2, name='Jenny', address='上海', phone='158000'}
    }
```

- 动态查询Given多个条件及模糊匹配

```
 @Test
    public void testFindOneGivenMultiConditions() {
        /*
         * 封装动态条件 -- 利用匿名内部类
         * toPredicate 方法作用：动态组装查询条件
         *
         * 需求：根据 name (指定为"Jenny") 并且 address以"海"结尾 来查询记录
         * SQL = select * from tb_resume where name = 'Jenny' and address = '%海'
         * */
        Specification<Resume> specification = new Specification<Resume>() {
            @Override
            public Predicate toPredicate(Root<Resume> root, CriteriaQuery<?> criteriaQuery, CriteriaBuilder criteriaBuilder) {
                // 获得属性名称 name 和 address
                Path<Object> attribute1 = root.get("name");
                Path<Object> attribute2 = root.get("address");

                // 使用CriteriaBuilder针对name属性构建条件（一个精准查询 + 一个模糊查询）
                Predicate predicate1 = criteriaBuilder.equal(attribute1, "Jenny");
                // attribute2 可能是个Integer，所以这里要做个类型转换，转成String类型
                Predicate predicate2 = criteriaBuilder.like(attribute2.as(String.class), "%海");

                // 组合：result = predicate1 and predicate2
                Predicate result = criteriaBuilder.and(predicate1, predicate2);
                return result;
            }
        };

        Optional<Resume> optional = resumeDao.findOne(specification);
```

```
        Resume result = optional.get();
        System.out.println(result);
        // 底层依然是Hibernate，打印结果：Resume{id=2, name='Jenny', address='上
海', phone='158000'}
    }
```

4. 关于排序和分页

4.1 排序

```
    /*
    * Sort（(1) 指明direction, (2) 按照实体类哪个属性）
    * Based on id, 倒序打印
    * */
    @Test
    public void testFindAllReversedOrder() {
        Sort order = new Sort(Sort.Direction.DESC, "id");
        List<Resume> result = resumeDao.findAll(order);

        for (int i = 0; i < result.size(); i++) {
            System.out.println(result.get(i));
        }
    }
```

Result:

```
Hibernate:
 select resume0_.id as id1_0_, resume0_.address as address2_0_, resume0_.name as
name3_0_, resume0_.phone as phone4_0_
   from tb_resume resume0_
       order by resume0_.id desc

Resume{id=5, name='Jenifer', address='上海', phone='188000'}
Resume{id=3, name='Hancle', address='杭州', phone='153000'}
Resume{id=2, name='Jenny', address='上海', phone='158000'}
Resume{id=1, name='Philip', address='北京', phone='131000'}
```

4.2 分页

```
    /*
     * Pageable 分页参数 (1)当前页数，从0开始  (2) 每页显示的数量
     * */
    @Test
    public void testFindAllPaging() {
        Pageable pageable = PageRequest.of(0, 2);
        Page<Resume> all = resumeDao.findAll(pageable);
        System.out.println(all);
    }
```

Result:

```
Hibernate:
    select resume0_.id as id1_0_, resume0_.address as address2_0_, resume0_.name
as name3_0_, resume0_.phone as phone4_0_
        from tb_resume resume0_ limit ?
Hibernate:
    select count(resume0_.id) as col_0_0_ from tb_resume resume0_

Page 1 of 2 containing com.lagou.edu.pojo.Resume instances
```

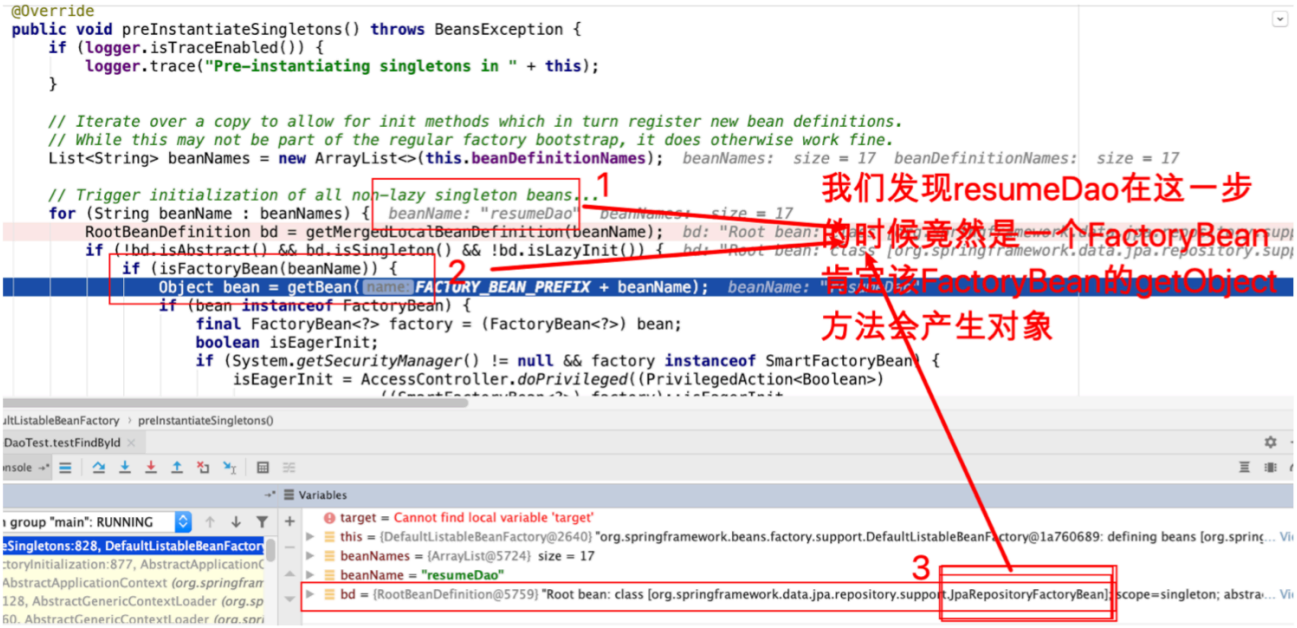
Part IV. Spring Data JPA 源码分析

> 这里使用的的代码是 `spring-data-jpa` 中的测试 testFindById，而不是 spring source code

开发Dao Interface，不需要手动写实现类，是因为Dao Interface的实现肯定是由动态代理来完成的。resumeDao是一个代理对象，这个代理对象的类型是SimpleJapRepository。

1. 这个SimpleJapRepository代理对象是怎么产生，过程怎样?

以往，在AbstractApplicationContext中的refresh方法中的 `finishBeanFactoryInitialization(beanFactory)`，在BeanPostProcessor的after方法中，会给一个对象产生代理对象。



1.1 为啥指定resumeDao为一个 JpaRepositoryFactoryBean?

FactoryBean有 getObject 方法，可以返回具体的对象。

代理对象固定为了SimpleJpaRepository。在ProxyFactory产生代理对象的时候，选择使用了JDK代理对象。

所以，JdkDynamicAopProxy会生成一个代理对象类型为SimpleJpaRespository。

1.2 什么时候指定的?

是BeanDefinitionBuilder 在进行 beandefinition 注册的时候，<jpa:repository basePackage 扫描到接口，将 resumeDao 的class set成了JpaRepositoryFacotryBean。

2. 这个代理对象类型SimpleJapRepository有何特别?

SimpleJpaRepository类实现了JpaRepository接口和JpaSpecificationExecutor接口，所以它的方法会调用jpa原本的API。