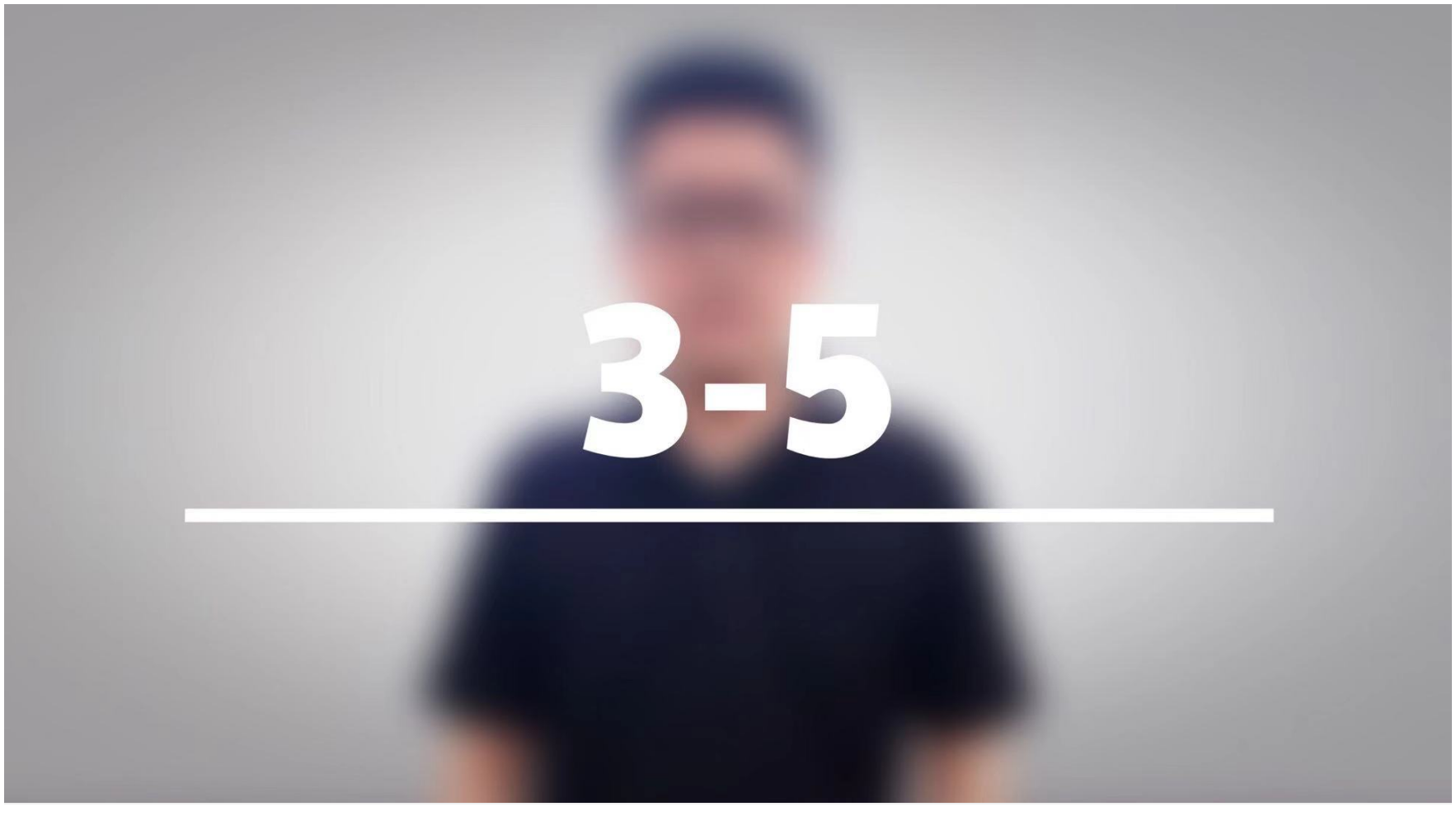

第 3-5 课：Spring Data JPA 的高级用法

上节课介绍了 Spring Data JPA 的使用方式和基本查询，常用的增、删、改、查需求 Spring Data JPA 已经实现了。但对于复杂的数据库场景，动态生成方法不能满足，对此 Spring Data JPA 提供了其他的解决方案，这就是这节课的主要内容。

自定义 SQL 查询

使用 Spring Data 大部分的 SQL 都可以根据方法名定义的方式来实现，但是由于某些原因必须使用自定义的 SQL 来查询，Spring Data 也可以完美支持。

在 SQL 的查询方法上面使用 @Query 注解，在注解内写 Hql 来查询内容。

```
@Query("select u from User u")
Page<User> findALL(Pageable pageable);
```

当然如果感觉使用原生 SQL 更习惯，它也是支持的，需要再添加一个参数 nativeQuery = true。

```
@Query("select * from user u where u.nick_name = ?1", nativeQuery = true)
Page<User> findByNickName(String nickName, Pageable pageable);
```

@Query 上面的 1 代表的是方法参数里面的顺序，如果有多个参数也可以按照这个方式添加 1、2、3....。除了按照这种方式传参外，还可以使用 @Param 来支持。

```
@Query("select u from User u where u.nickName = :nickName")
Page<User> findByNickName(@Param("nickName") String nickName, Pageable pageable);
```

如涉及到删除和修改需要加上 @Modifying，也可以根据需要添加 @Transactional 对事务的支持、操作超时设置等。

```
@Transactional(timeout = 10)
@Modifying
@Query("update User set userName = ?1 where id = ?2")
int modifyById(String  userName, Long id);

@Transactional
@Modifying
@Query("delete from User where id = ?1")
void deleteById(Long id);
```

使用已命名的查询

除了使用 @Query 注解外，还可以预先定义好一些查询，并为其命名，然后再 Repository 中添加相同命名的方法。

定义命名的 Query:

```
@Entity
@NamedQueries({
        @NamedQuery(name = "User.findByPassWord", query = "se-
lect u from User u where u.passWord = ?1"),
        @NamedQuery(name = "User.findByNickName", query = "se-
lect u from User u where u.nickName = ?1"),
})
public class User {
    ……
}
```

通过 @NamedQueries 注解可以定义多个命名 Query，@NamedQuery 的 name 属性定义了 Query 的名称，注意加上 Entity 名称 . 作为前缀，query 属性定义查询语句。

定义对应的方法:

```
List<User> findByPassWord(String passWord);
List<User> findByNickName(String nickName);
```

Query 查找策略

到此，我们有了三种方法来定义 Query：（1）通过方法名自动创建 Query，（2）通过 @Query 注解实现自定义 Query，（3）通过 @NamedQuery 注解来定义 Query。那么，Spring Data JPA 如何来查找这些 Query 呢?

通过配置 @EnableJpaRepositories 的 queryLookupStrategy 属性来配置 Query 查找策略，有如下定义。

- CREATE：尝试从查询方法名构造特定于存储的查询。一般的方法是从方法名中删除一组已知的前缀，并解析方法的其余部分。
- USE_DECLARED_QUERY：尝试查找已声明的查询，如果找不到，则抛出异常。查询可以通过某个地方的注释定义，也可以通过其他方式声明。
- CREATE_IF_NOT_FOUND（默认）：CREATE 和 USE_DECLARED_QUERY 的组合，它首先查找一个已声明的查询，如果没有找到已声明的查询，它将创建一个自定义方法基于名称的查询。它允许通过方法名进行快速查询定义，还可以根据需要引入声明的查询来定制这些查询调优。

一般情况下使用默认配置即可，如果确定项目 Query 的具体定义方式，可以更改上述配置，例如，全部使用 @Query 来定义查询，又或者全部使用命名的查询。

分页查询

Spring Data JPA 已经帮我们内置了分页功能，在查询的方法中，需要传入参数 Pageable，当查询中有多个参数的时候 Pageable 建议作为最后一个参数传入。

```
@Query("select u from User u")
Page<User> findALL(Pageable pageable);

Page<User> findByNickName(String nickName, Pageable pageable);
```

Pageable 是 Spring 封装的分页实现类，使用的时候需要传入页数、每页条数和排序规则，Page 是 Spring 封装的分页对象，封装了总页数、分页数据等。返回对象除使用 Page 外，还可以使用 Slice 作为返回值。

```
Slice<User> findByNickNameAndEmail(String nick-
Name, String email,Pageable pageable);
```

Page 和 Slice 的区别如下。

- Page 接口继承自 Slice 接口，而 Slice 继承自 Iterable 接口。
- Page 接口扩展了 Slice 接口，添加了获取总页数和元素总数量的方法，因此，返回 Page 接口时，必须执行两条 SQL，一条复杂查询分页数据，另一条负责统计数据数量。
- 返回 Slice 结果时，查询的 SQL 只会有查询分页数据这一条，不统计数据数量。
- 用途不一样：Slice 不需要知道总页数、总数据量，只需要知道是否有下一页、上一页，是否是首页、尾页等，比如前端滑动加载一页可用；而 Page 知道总页数、总数据量，可以用于展示具体的页数信息，比如后台分页查询。

```
@Test
public void testPageQuery()  {
    int page=1,size=2;
    Sort sort = new Sort(Sort.Direction.DESC, "id");
    Pageable pageable = PageRequest.of(page, size, sort);
    userRepository.findALL(pageable);
    userRepository.findByNickName("aa", pageable);
}
```

- Sort，控制分页数据的排序，可以选择升序和降序。
- PageRequest，控制分页的辅助类，可以设置页码、每页的数据条数、排序等。

还有一些更简洁的方式来排序和分页查询，如下。

限制查询

有时候我们只需要查询前 N 个元素，或者只取前一个实体。

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable page-
able);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

复杂查询

我们可以通过 AND 或者 OR 等连接词来不断拼接属性来构建多条件查询，但如果参数大于 6 个时，方法名就会变得非常的长，并且还不能解决动态多条件查询的场景。到这里就需要给大家介绍另外一个利器 JpaSpecificationExecutor 了。

JpaSpecificationExecutor 是 JPA 2.0 提供的 Criteria API 的使用封装，可以用于动态生成 Query 来满足我们业务中的各种复杂场景。Spring Data JPA 为我们提供了 JpaSpecificationExecutor 接口，只要简单实现 toPredicate 方法就可以实现复杂的

查询。

我们来看一下 JpaSpecificationExecutor 的源码：

```
public interface JpaSpecificationExecutor<T> {
    //根据 Specifica-
tion 条件查询单个对象，注意的是，如果条件能查出来多个会报错
    T findOne(@Nullable Specification<T> spec);
    //根据 Specification 条件查询 List 结果
    List<T> findAll(@Nullable Specification<T> spec);
    //根据 Specification 条件，分页查询
    Page<T> findAll(@Nullable Specification<T> spec, Pageable page-
able);
    //根据 Specification 条件，带排序的查询结果
    List<T> findAll(@Nullable Specification<T> spec, Sort sort);
    //根据 Specification 条件，查询数量
    long count(@Nullable Specification<T> spec);
}
```

JpaSpecificationExecutor 的源码很简单，根据 Specification 的查询条件返回 List、Page 或者 count 数据。在使用 JpaSpecificationExecutor 构建复杂查询场景之前，我们需要了解几个概念：

- Root <T> root，代表了可以查询和操作的实体对象的根，开一个通过 get("属性名") 来获取对应的值。
- CriteriaQuery query，代表一个 specific 的顶层查询对象，它包含着查询的各个部分，比如 select 、from、where、group by、order by 等。
- CriteriaBuilder cb，来构建 CritiaQuery 的构建器对象，其实就相当于条件或者是条件组合，并以 Predicate 的形式返回。

使用案例

下面的使用案例中会报错这几个对象的使用。

首先定义一个 UserDetail 对象，作为演示的数据模型。

```
@Entity
public class UserDetail {
    @Id
    @GeneratedValue
    private Long id;
    @Column(nullable = false, unique = true)
    private Long userId;
    private Integer age;
    private String realName;
    private String status;
    private String hobby;
    private String introduction;
    private String lastLoginIp;

}
```

创建 UserDetail 对应的 Repository:

```
public interface UserDetailRepository extends JpaSpecificationExe-
cutor<UserDetail>,JpaRepository<UserDetail, Long>  {
}
```

定义一个查询 `Page<UserDetail>` 的接口:

```
public interface UserDetailService {
    public Page<UserDetail> findByCondition(UserDetailParam detail-
Param, Pageable pageable);
}
```

在 UserDetailServiceImpl 中，我们来演示 JpaSpecificationExecutor 的具体使用。

```
@Service
public class UserDetailServiceImpl implements  UserDetailService{

    @Resource
    private UserDetailRepository userDetailRepository;

    @Override
    public Page<UserDetail> findByCondition(UserDetailParam detail-
Param, Pageable pageable){

        return userDetailRepository.findAll((root, query, cb) -> {
            List<Predicate> predicates = new ArrayList<Predi-
cate>();
            //equal 示例
            if (!StringUtils.isNullOrEmpty(detailParam.getIntroduc-
tion())){
                predicates.add(cb.equal(root.get("introduction"),de
tailParam.getIntroduction()));
            }
            //like 示例
            if (!StringUtils.isNullOrEmpty(detailParam.getReal-
Name())){
                predicates.add(cb.like(root.get("realName"),"%"+det
ailParam.getRealName()+"%"));
            }
            //between 示例
            if (detailParam.getMinAge()!=null && detailParam.get-
MaxAge()!=null) {
                Predicate agePredicate = cb.between(root.get("age")
, detailParam.getMinAge(), detailParam.getMaxAge());
                predicates.add(agePredicate);
            }
            //greaterThan 大于等于示例
            if (detailParam.getMinAge()!=null){
                predicates.add(cb.greaterThan(root.get("age"),de-
tailParam.getMinAge()));
            }
            return query.where(predicates.toArray(new Predi-
cate[predicates.size()])).getRestriction();
        }, pageable);

    }
}
```

上面的示例是根据不同条件来动态查询 UserDetail 分页数据，UserDetailParam 是参数的封装，示例中使用了常用的大于、like、等于等示例，根据这个思路我们可以不断扩展完成更复杂的动态 SQL 查询。

使用时只需要将 UserDetailService 注入调用相关方法即可：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class JpaSpecificationTests {

    @Resource
    private UserDetailService userDetailService;

    @Test
    public void testFindByCondition()  {
        int page=0,size=10;
        Sort sort = new Sort(Sort.Direction.DESC, "id");
        Pageable pageable = PageRequest.of(page, size, sort);
        UserDetailParam param=new UserDetailParam();
        param.setIntroduction("程序员");
        param.setMinAge(10);
        param.setMaxAge(30);
        Page<UserDetail> page1=userDetailService.findByCondi-
tion(param,pageable);
        for (UserDetail userDetail:page1){
            System.out.println("userDetail: "+userDetail.to-
String());
        }
    }

}
```

多表查询

多表查询在 Spring Data JPA 中有两种实现方式，第一种是利用 Hibernate 的级联查询来实现，第二种是创建一个结果集的接口来接收连表查询后的结果，这里主要介绍第二种方式。

我们还是使用上面的 UserDetail 作为数据模型来使用，定义一个结果集的接口类，接口类的内容来自于用户表和用户详情表。

```
public interface UserInfo {
    String getUserName();
    String getEmail();
    String getAddress();
    String getHobby();
}
```

> 在运行中 Spring 会给接口（UserInfo）自动生产一个代理类来接收返回的结果，代码中使用 getXX 的形式来获取。

在 UserDetailRepository 中添加查询的方法，返回类型设置为 UserInfo:

```
@Query("select u.userName as userName, u.email as email, d.introd-
uction as introduction , d.hobby as hobby from User u , UserDe-
tail d " +
            "where u.id=d.userId  and  d.hobby = ?1 ")
List<UserInfo> findUserInfo(String hobby);
```

特别注意这里的 SQL 是 HQL，需要写类的名和属性，这块很容易出错。

测试验证:

```
@Test
public void testUserInfo()  {
    List<UserInfo> userInfos=userDetailRepository.findUserInfo("钓
鱼");
    for (UserInfo userInfo:userInfos){
        System.out.println("userInfo: "+userInfo.getUserName()+"-"+
userInfo.getEmail()+"-"+userInfo.getHobby()+"-"+userInfo.getIntro-
duction());
    }
}
```

运行测试方法后返回:

```
userInfo: aa-aa@126.com-钓鱼-程序员
```

证明关联查询成功，最后的返回结果来自于两个表，按照这个思路可以进行三个或者更多表的关联查询。

总结

Spring Data JPA 使用动态注入的原理，根据方法名动态生成方法的实现，因此根据方法名实现数据查询，即可满足日常绝大部分使用场景。除了这种查询方式之外，Spring Data JPA 还支持多种自定义查询来满足更多复杂场景的使用，两种方式相结合可以灵活满足项目对 Orm 层的需求。

通过学习 Spring Data JPA 也可以看出 Spring Boot 的设计思想，80% 的需求通过默认、简单的方式实现，满足大部分使用场景，对于另外 20% 复杂的场景，提供另外的技术手段来解决。Spring Data JPA 中根据方法名动态实现 SQL，组件环境自动配置等细节，都是将 Spring Boot **约定优于配置**的思想体现的淋淋尽致。

> 点击这里下载源码 (https://github.com/ityouknow/spring-boot-leaning/tree/gitbook_column2.0)。

(/gitchat/column/5b86228ce15aa17d68b5b55a/topic/5bf22499b36fd43be4715f15)(/gitcha