Spring Boot 之JPA实战

> 作者：李仁密

1、开始之前

先说明几个问题:

1、该课程主要包括哪些内容?

- 核心概念解释

- JPA的基本使用
    - 基本使用步骤
    - 实体类
    - Repository接口
    - 方法名创建查询
    - JPQL语句
    - 分页与排序
    - 事务处理
- JPA中的实体关系
    - 一对一关系
    - 一对多和多对一关系
    - 多对多关系

2、课程有什么特点?

- 课程内容重点：Spring Boot中JPA规则的实际应用
- 思维重点：是什么？怎么用?
- 实例代码演示规则的使用

3、你能从课程中得到什么?

- 学会Spring Boot中JPA规则的实际运用

4、关于环境与工具

-
-
-
- 操作系统：macOS

2、Spring Boot和JPA

2.1、JPA是什么?

- JPA（Java Persistent API）翻译：Java持久化API，Sun官方提出的一种Java持久化的规范。
- JPA统一已有的ORM框架，给开发者提供了统一、相对简单的持久化的工具，降低了程序和ORM产品之间的耦合度，提供程序的可移植性。
- JPA本身不可以直接在程序中使用，需要依赖实现了JPA规范的JPA产品，比如：Hibernate。
- JPA产品，主要包括：
    - ORM映射元数据
    - Java持久化API（CRUD）
    - 查询语言JPQL

2.2、Spring Boot 中的JPA是什么?

Spring Boot中JPA从技术层面上来讲，其实是一个 `spring-boot-starter-data-jpa` 模块, 包括：

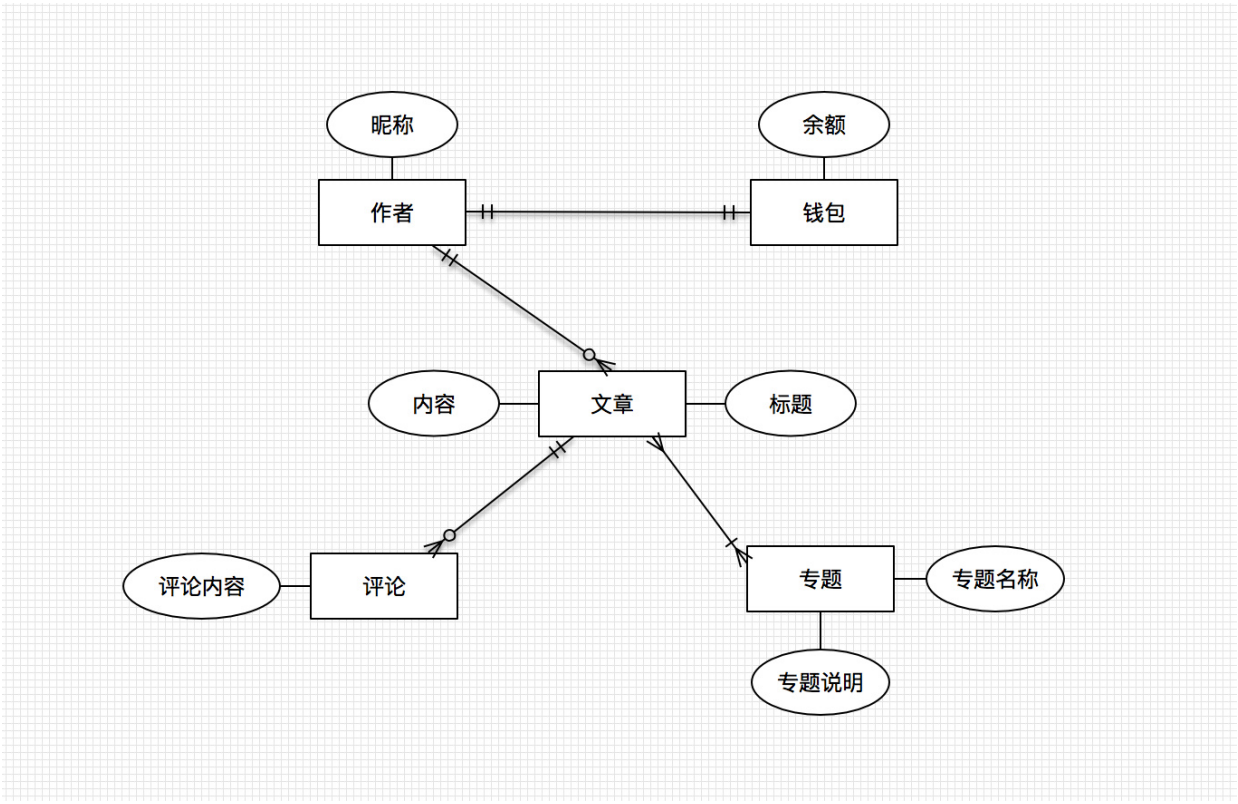
- Hibernate 实现了JPA规范一个流行JPA产品
- **Spring Data JPA** 基于JPA进一步简化了数据访问层的实现，提供了一宗类似于声明编程方

式，开发者访问数据层不再需要重复的模板代码，只需要编写Repository接口，它就可以根据方法名自动生成实现。
- Spring ORMs 是Spring Framework对ORM的核心支撑。

Spring Boot中的JPA提供了更友好的数据持久化工具。开发者只需关注数据业务操作。无需处理繁琐的配置和重复的模板代码。

3、JPA基本使用步骤

3.1、 关于课程实例

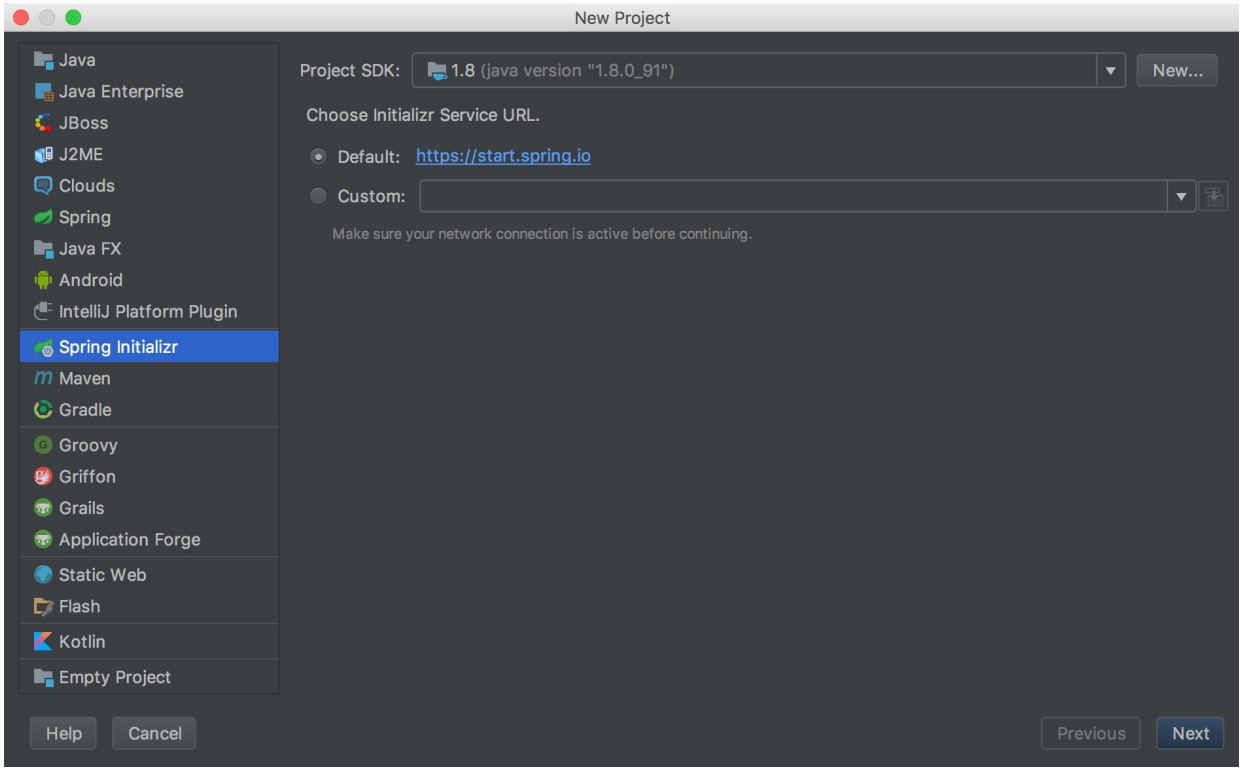


- 作者注册、更新、删除
- 文章新增、更新、删除
- 评论文章、删除文章
- 专题新增、更新、删除
- 专题收录文章、专题取消收录文章

3.2、 JPA使用步骤

1、Spring Boot 构建

使用IDEA中Spring Boot框架的构建工具



2、引入JPA模块

在maven的pom.xml文件引入 `spring-boot-starter-data-jpa` :

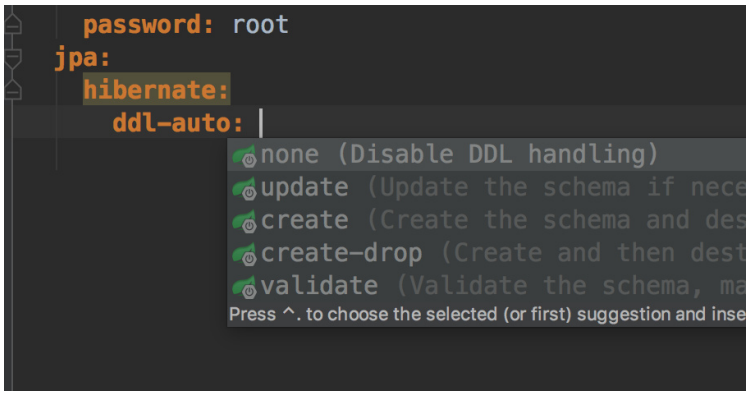
```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

3、在Spring Boot中配置JPA

application.yml:

```
spring:
#数据库连接信息配置
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/spring-boot-jpa-demo?
useUnicode=true&characterEncoding=utf-8
    username: root
    password: root
  # JPA配置
  jpa:
    hibernate:
      ddl-auto: update  #数据库schema的DDL策略
    show-sql: true
```

jpa.hibernate.ddl-auto:



update: 第一次加载Hibernate，会根据我们的model类创建新表，以后再加载Hibernate时，会根据model类自动更新表结构。注意：即使model类结构发生了变化，表结构不会删除以前的列或者行，只会新增一个列。

create：加载Hibernate时，根据model类创建新的表。注意：表数据的丢失，**慎用!**

create-drop： 加载Hibernate时，根据model类创建一个新的表，sessionFactory销毁的时，把对应表给删除掉。

validate: 加载Hibernate时，根据model类来验证我的表结构，不会创建新的表，或者更新表。

> jpa.hibernate.ddl-auto：慎用，避免数据的丢失

4、定义实体类

Author.java

```
package com.lrm.domain;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.util.Date;

@Entity
public class Author {
    @Id
    @GeneratedValue
    private Long id;
    private String nickName;
    private String phone;
    private Date signDate;

    public Author() {
    }

    //省略getter和setter方法
}
```

5、创建Repository接口

```
package com.lrm.domain;
import org.springframework.data.jpa.repository.JpaRepository;
public interface AuthorRepostiory extends JpaRepository<Author, Long> {
}
```

4、定义实体类

通常在企业开发中有两种做法：

1. 先创建数据库表，然后再根据表来创建实体类。这是传统的**数据库建模思想**。
2. 先编写实体类，然后再生成数据库表。这种采用**领域建模思想**，更加OOP。建议采用。

4.1、关于注解

主要通过注解来定义实体类：

@Entity

应用于实体类，表明该实体类被JPA管理，将映射到指定的数据库表

@Table

应用于实体类，通过`name`属性指定对应该实体类映射表的名字

@Id

应用于实体类的属性或者属性对应的getter方法，表示该属性映射为数据库表的主键

@GerneratedValue

于@Id一同使用，表示主键的生成策略，通过`strategy`属性指定。

JPA提供的生成策略有：

- ***AUTO*** — JPA自动选择合适的策略，是默认选项
- *IDENTITY* — 采用数据库ID自增长的方式来生成主键值，Oracle不支持这种方式；
- *SEQUENCE* — 通过序列产生主键，通过@SequenceGenerator注解指定序列名，MySql不支持这种方式；
- *TABLE* — 采用表生成方式来生成主键值，这种方式比较通用，但是效率低

@Basic

应用于属性，表示该属性映射到数据库表，@Entity标注的实体类的所有属性，默认即为@Basic，@Baisc有两个属性：

1. `fetch`：属性的读取策略，有`EAGER`和`LAZY`两种取值，分别表示主动抓取和延迟抓取，默认为`EAGER`
2. `optional`：表示该属性是否允许为`null`，默认值为`true`。

> @Basic(fetch =FetchType.LAZY)标注某属性时，表示只有调用Hibernate对象的该属性的get方法时，才会从数据库表中查找对应该属性的字段值

@Column

应用于实体类的属性，可以指定数据库表字段的名字和其他属性。其属性包括：

- `name`：表示数据库表中该字段的名称，默认情形属性名称一致。
- `nullable`：表示该字段是否允许为`null`，默认为`true`。
- `unique`：表示该字段是否是唯一标识，默认为`false`。
- `length`：表示该字段的大小，仅对`String`类型的字段有效。
- `insertable`：表示在ORM框架执行插入操作时，该字段是否应出现INSETRT语句中，默认为`true`。
- `updateable`：表示在ORM框架执行更新操作时，该字段是否应该出现在UPDATE语句中，默认为`true`。对于一经创建就不能更改的字段，该属性非常有用，比如`email`属性。
- `columnDefinition`：表示该字段在数据库中的实际类型。通常ORM框架可以根据属性类型自动判断数据库中字段的类型，但是依然有些例外：
  - `Date`类型无法确定数据库中字段类型究竟是`DATE`、`TIME`还是`TIMESTAMP`
  - `String`的默认映射类型为`VARCHAR`，如果希望将`String`类型映射到特定数据库的`BLOB`或`TEXT`字段类型，则需要进行设置

@Transient

应用在实体类属性上，表示该属性不映射到数据库表，JPA会忽略该属性。

@Temporal

应用到实体类属性上，表明该属性映射到数据库是一个时间类型，具体定义：

- `@Temporal(TemporalType.DATE)`映射为日期 // date （只有日期）
- `@Temporal(TemporalType.TIME)`映射为日期 // time （是有时间）
- `@Temporal(TemporalType.TIMESTAMP)`映射为日期 // date time （日期+时间）

@Lob

应用到实体类属性上，表示将属性映射成数据库支持的大对象类型，Clob或者Blog。其中：

- Clob（Character Large Ojects）类型是长字符串类型，`java.sql.Clob`、`Character[]`、`char[]`和`String`将被映射为 Clob 类型。
- Blob（Binary Large Objects）类型是字节类型，`java.sql.Blob`、`Byte[]`、`byte[]`和实现了`Serializable`接口的类型将被映射为 Blob 类型。

因为这两种类型的数据一般占用的内存空间比较大，所以通常使用延迟加载的方式，与`@Basic`标注同时使用，设置加载方式为`FetchType.LAZY`。

4.2、关于构造函数

JPA中对象是由Hibernate为我们创建的，当我们通过ID来获取某个实体的时候，这个实体给我们返回了这个对象的创建是由Hibernate内部通过**反射技术**来创建的，反射的时候用到了默认的构造函数，所以这时候必须给它提供一个public的无参构造函数。

5、Repository接口

5.1、接口层次与方法

Spring Data JPA简化了持久层的操作，开发者只需声明持久层接口，而不需要实现该接口。Spring Data JPA内部会根据不同的接口方法，采用不同的策略自动生成实现。

而开发者声明持久层接口，需要直接或者间接的方式继承Repository接口，从而使自定义的持久层拥有了持久层的操作能力。

Repository接口：

- **Repository** 是Spring Data的核心接口，最顶层接口，不包括任何方法，他的目的是为了统一所有Repository的类型，且让组件扫描的时候自动识别。

扩展的Repository接口：

- **CrudRepository** 继承了Repository，提供了曾删改查方法，可以直接调用。
- **PagingAndSortingRepository** 继承了CrudRepository，提供了分页和排序两个方法。
- **JpaRepository** 继承了PagingAndSortingRepository，针对于JPA技术的接口，提供了flush(), saveFlush(), deleteInBatch()等方法

`CrudRepository` 完整定义如下

```
public interface CrudRepository<T, ID extends Serializable> extends
Repository<T, ID> {

    <S extends T> S save(S var1); //将一个对象持久化到数据库中

    <S extends T> Iterable<S> save(Iterable<S> var1); //将一组对象持久化到数据
库中

    T findOne(ID var1); //根据id查找并返回一个对象

    boolean exists(ID var1); //判断某个id是否存在

    Iterable<T> findAll(); //返回所有对象

    Iterable<T> findAll(Iterable<ID> var1); //根据一组id返回对应的对象

    long count(); //返回共有多少条数据

    void delete(ID var1); //根据id删除某个对象

    void delete(T var1); //删除某个对象

    void delete(Iterable<? extends T> var1); //删除某一组对象

    void deleteAll(); //删除所有对象
}
```

`PagingAndSortingRepository`完整定义：

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
extends CrudRepository<T, ID> {
    Iterable<T> findAll(Sort var1); //根据某个排序获取所有数据

    Page<T> findAll(Pageable var1); //根据分页信息获取某一页的数据
}
```

`JpaRepository`的完整定义：

```
public interface JpaRepository<T, ID extends Serializable> extends
PagingAndSortingRepository<T, ID> {
    List<T> findAll(); //获取所有数据，以List的方式返回

    List<T> findAll(Sort var1); //根据某个排序获取所有数据，以List的方式返回

    List<T> findAll(Iterable<ID> var1); //根据一组id返回对应的对象，以List的方式
返回

    <S extends T> List<S> save(Iterable<S> var1); //将一组对象持久化到数据库中，
以List的方式返回

    void flush(); //将修改更新到数据库

    <S extends T> S saveAndFlush(S var1); //保存数据并将修改更新到数据库

    void deleteInBatch(Iterable<T> var1); //批量删除数据

    void deleteAllInBatch(); //批量删除所有数据

    T getOne(ID var1); //根据id查找并返回一个对象
}
```

5.2、使用步骤

1、声明持久层接口

```
public interface AuthorRepository extends JpaRepository<Author, Long> {
}
```

2、在持久层接口中声明业务方法

```
public interface AuthorRepository extends JpaRepository<Author, Long> {
    List<Author> findByPhoneAndNickName(String phone,String nickName);
}
```

3、获取Repository实例并使用

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

    @Autowired
    private AuthorRepository authorRepostiory;

    @Test
    public void saveAuthorTest() {
        Author author = new Author();
        author.setNickName("Arvin");
        author.setPhone("192782342348");
        author.setSignDate(new Date());
        authorRepostiory.save(author);
    }

    @Test
    public void findAuthorTest() {
        List<Author> authors =
authorRepostiory.findByPhoneAndNickName("192782342348","Arvin");
        System.out.println(JSON.toJSONString(authors,true));
    }
}
```

6、方法名创建查询

Spring Data JPA中除了可以直接使用Repository扩展接口声明的方法以外，我们可以根据规则声明方法名自定义查询，也可以直接使用，方法名声明规则：

```
find + 全局修饰 + By + 实体的属性名称 + 限定词 + 连接词 + ... （其他实体属性）+
OrderBy + 排序属性 + 排序方向
```

例如： findDistinctByNickNameIgnoreCaseAndPhoneOrderBySignDateDesc(String nickname,String phone)

- `Distinct` 是全局修饰 非必须
- `NickName` 和 `Phone` 是实体的属性名
- `And` 是连接词
- `IgnoreCase` 是限定词
- `SignDate` 是排序属性
- `Desc` 是排序方向

关键字分类：

- 全局修饰： `Distinct` , `Top` , `First`
- 限定词：`IsNull`,`IsNotNull`,`Like`,`NotLike`,`Containing`,`In`,`NotIn`,`IgnoreCase`,`Between`,`Equals`,`LessThan`,`GreaterThan`,`After`,`Before`.....

- 排序方向：`Asc`, `Desc`
- 连接词：`And`, `Or`

关键字	例子	JPQL语句
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	findByAgeIsNull	… 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 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection age)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

7、JPQL语句

7.1、JPQL查询

```
public interface AuthorRepository extends JpaRepository<Author, Long> {

    @Query("select a from Author a where a.phone like %:phone%")
    List<Author> findByPhone(@Param("phone") String phone);

    @Query("select a.nickName,length(a.nickName) from Author a where
a.nickName like %?1%")
    List<Object[]> findArry(String nickName);

    @Query("select a from Author a where a.nickName like %?1%")
    List<Author> findByNickName(String nickName, Sort sort);

    @Query(value = "select * from author where nick_name like %?1%",
nativeQuery = true)
    List<Author> findbySql(String nickName);
}
```

7.2、使用SQL查询

```
public interface AuthorRepository extends JpaRepository<Author, Long> {

    @Query(value = "select * from author where nick_name like %?1%",
nativeQuery = true)
    List<Author> findbySql(String nickName);

}
```

`nativeQuery = true` 开启SQL语句查询

7.3、JPQL更新

```
public interface AuthorRepository extends JpaRepository<Author, Long> {

    @Transactional //开启事务
    @Modifying
    @Query("update Author a set a.nickName = ?1 where a.phone = ?2")
    int setNickName(String nickName,String phone);

}
```

> update和delete必须在事务下执行
>
> update和delete必须使用@Modifying注解
>
> JPQL不支持INSERT

8、分页与排序和事务处理

8.1、分页与排序

自定义的Repository接口

```
public interface AuthorRepository extends JpaRepository<Author, Long> {...}
```

调用分页方法：

```
  @Test
    public void findAuthorForPageTest() {

        Sort sort = new Sort(Sort.Direction.DESC,"id");//按照id倒叙排序
        Pageable pageable = new PageRequest(1,5,sort);//第二页，每页5条数据
        Page<Author> page = authorRepostiory.findAll(pageable);

        System.out.println(JSON.toJSONString(page,true));
    }
```

返回结果：

```
{
    "content":[//数据
        {
            "id":2,
            "nickName":"Jode",
            "phone":"18278234238",
            "signDate":1504972800000
        },
        {
            "id":1,
            "nickName":"Arvin",
            "phone":"18676572283",
            "signDate":1504886400000
        }
    ],
    "first":false,//是否第一页
    "last":true,//是否最后一页
    "number":1,//当前页，0是第一页，1是第二页
    "numberOfElements":2,//总条数中当前页数据条数
    "size":5,//每页数据条数
    "sort": [//排序
        {
            "direction": "DESC",
            "property": "id",
            "ignoreCase": false,
            "nullHandling": "NATIVE",
            "ascending": false,
            "descending": true
        }
    ],
    "totalElements":7,//总条数
    "totalPages":2//总页数
}
```

8.2、事务处理

Service：

```java
@Service
public class AuthorServiceImpl implements AuthorService {

    @Autowired
    private AuthorRepository authorRepostiory;

    @Transactional
    @Override
    public Author updateAuthor() {
        Author author = new Author();
        author.setPhone("9999999999");
        author.setNickName("自由自在1");
        author.setSignDate(new Date());
        Author author1 = authorRepostiory.save(author);

        author1.setPhone("111111111");
        Author author2 = authorRepostiory.save(author1);

        int i = 8/0;//抛出异常，事务回滚，以上save方法会操作数据库
        return author2;
    }
}
```

使用`@Transactional`注解开启事务，一个事务中的所有持久层操作保持一致，只要所有持久层操作都执行成功，且事务方法里面不报异常，才会操作数据库。

9、一对一实体关系

9.1、@OneToOne

通过@OneToOne注解设置实体之间的一对一关系：

Author实体类：

```
@Entity
public class Author {
    @Id
    @GeneratedValue
    private Long id;
    private String nickName;
    private String phone;
    @Temporal(TemporalType.DATE)
    private Date signDate;

    @OneToOne
    private Wallet wallet;//设置与Wallet之间的一对一关系

    public Author() {
    }

    //... setter,getter省略
}
```

Wallet实体类：

```
@Entity
public class Wallet {
    @Id
    @GeneratedValue
    private Long id;
    private BigDecimal balance;
    public Wallet() {
    }
  //... setter,getter省略
}
```

Author实体类是关系维护方，其对应的author表会生成外键和关联字段，Wallet是关系被维护方，其对应数据库表字段的id主键值作为author表外键关联值。Wallet不能维护关系字段，Author维护关系字段。

9.2、@OneToOne属性

Author实体类：

```
@Entity
public class Author {
    @Id
    @GeneratedValue
    private Long id;
    private String nickName;
    private String phone;
    @Temporal(TemporalType.DATE)
    private Date signDate;

    @OneToOne(cascade =
{CascadeType.PERSIST,CascadeType.MERGE,CascadeType.REMOVE},optional =
false, fetch = FetchType.EAGER)
   @JoinColumn(name = "author_wallet_id" )
    private Wallet wallet;//设置与Wallet之间的一对一关系

    public Author() {
    }

    //... setter,getter省略
}
```

`cascade`属性表示级联操作策略：

1. 不定义,则对关系表不会产生任何影响
2. `CascadeType.PERSIST`:级联新建
3. `CascadeType.REMOVE`:级联删除
4. `CascadeType.REFRESH`: 级联刷新，即重新同步到数据库中状态，会覆盖掉已经修改但是还没保存的实体类属性
5. `CascadeType.MERGE`: 级联更新
6. `CascadeType.ALL`:表示选择全部四项

`fetch`属性表示实体的加载方式，有`LAZY`和`EAGER`两种取值，默认值为`EAGER`

`optional`属性表示关联的实体是否能够存在`null`值，默认为`true`，表示可以存在`null`值

`@JoinColumn(name = "author_wallet_id" )` 指定外键字段名字为author_wallet_id

`@JoinColumn(name="author_wallet_id", referencedColumnName="balance")` 中 的referencedColumnName指定外键对应到外联表的取值字段为balance，默认是从id取值。

9.3、双向关联

Wallet实体类：

```
@Entity
public class Wallet {
    @Id
    @GeneratedValue
    private Long id;
    private BigDecimal balance;

    @OneToOne(mappedBy = "wallet")//指定通过Author的wallet属性一对一关联
Author，Wallet实体类为关系被维护方法
    private Author author;
    public Wallet() {
    }
  //... setter,getter省略
}
```

10、一对多和多对一实体关系

10.1、关系设置

Article和Comment实体之间的关系为一对多的关系。Article为1，Comment为多端。JPA规定多的一端为关系维护端。通过@ManyToOne 和 @OneToMany注解设置实体一对多和多对一关系。

Article中包含多个Comment

```
@Entity
public class Article {

    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private String content;

    @OneToMany(mappedBy = "article", cascade = {CascadeType.PERSIST,
CascadeType.REMOVE}, fetch = FetchType.EAGER)
    private List<Comment> comments = new ArrayList<>();

    public void addComment(Comment comment) {
        comment.setArticle(this);
        comments.add(comment);
    }

    //setter 和 getter省略

}
```

> Article为关系被维护端，没有权限维护关系字段，若要维护需要显示的设置好关系

一个Comment对应一个Article：

```
@Entity
public class Comment {

    @Id
    @GeneratedValue
    private Long id;
    private String content;

    @ManyToOne
    private Article article;

    public void clearComment() {
        this.getArticle().getComments().remove(this);
    }

    public Comment() {
    }
    //省略setter和getter
}
```

> Comment为关系维护端，可以维护关系字段

10.2、属性设置

@OneToMany，@ManyToOne的属性：

`cascade` 属性表示级联操作策略：

1. 不定义,则对关系表不会产生任何影响
2. `CascadeType.PERSIST` :级联新建
3. `CascadeType.REMOVE` :级联删除
4. `CascadeType.REFRESH` : 级联刷新，即重新同步到数据库中状态，会覆盖掉已经修改但是还没保存的实体类属性
5. `CascadeType.MERGE` : 级联更新
6. `CascadeType.ALL` :表示选择全部四项

`fetch` 属性表示实体的加载方式，有 `LAZY` 和 `EAGER` 两种取值，默认值为 `EAGER`

`optional` 属性表示关联的实体是否能够存在 `null` 值，默认为 `true` ，表示可以存在 `null` 值

`@JoinColumn(name = "author_wallet_id" )` 指定外键字段名字为author_wallet_id

`@JoinColumn(name="author_wallet_id", referencedColumnName="balance")` 中 的referencedColumnName指定外键对应到外联表的取值字段为balance，默认是从id取值。

慎用CascadeType.ALL与CascadeType.REMOVE，避免数据丢失

10.3、延迟加载

默认情况下Article中的comments属性的 `fetch=FetchType.LAZY` ，即延迟加载，获取article对象的时候，其中的comments不会从数据中获取到值，只有显示的调用comments的getter方法后，才会从数据库中获取comments的数据。

凡是被@xxToMany注解(比如@OneToMany、@ManyToMany)标识的属性，默认情况下 `fetch=FetchType.LAZY`

@xxToOne注解（例如：@OneToOne、@ManyToOne）标识的属性，默认情况下 `fetch=FetchType.EAGER`

11、多对多实体关系

11.1、@ManyToMany

实体与实体之间的多对多的关系通过 `@ManyToMany` 注解去设置

Topic实体类：

```
@Entity
public class Topic {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @ManyToMany
    private List<Article> articles = new ArrayList<>();

    public Topic() {
    }

   //setter和getter省略
}
```

Article实体类：

```java
@Entity
public class Article {

    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private String content;

    @OneToMany(mappedBy = "article", cascade = {CascadeType.PERSIST,
CascadeType.REMOVE}, fetch = FetchType.EAGER)
    private List<Comment> comments = new ArrayList<>();


    /**
    *通过Topic实体类中的articles属性与Topic建立多对多关系
    *mppedBy意思是声明Article实体类为关系被维护方
    */
    @ManyToMany(mappedBy = "articles")
    private List<Topic> topics = new ArrayList<>();

    public void addComment(Comment comment) {
        comment.setArticle(this);
        comments.add(comment);
    }


    public Article() {
    }

    //省略setter和getter

}
```

- 虽然多对多关系中两个实体类都使用 `@ManyToMany` 标注关系，但需要通过mappedBy指定关系被维护方，则另一方就是关系维护方。关系维护方的指定根据实际业务需求来确定。
- 多对多关系指定后，JPA会根据实体及其关系自动生成实体类对应的表和两个实体之间的关系表。中间关系表：

  表名称：topics_articles

字段	类型	说明
topics_id	bigint	topics表的外联字段

articles_id	bigint	articles表的外联字段

11.2、@JoinTable属性

```
@Entity
public class Topic {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(
            name = "t_topic_article",
            joinColumns = @JoinColumn(name = "topic_id",
referencedColumnName = "id"),
            inverseJoinColumns = @JoinColumn(name = "article_id")
    )
    private List<Article> articles = new ArrayList<>();

    public Topic() {
    }

   //setter和getter省略
}
```

@JoinTable 注解用来控制，多对多的关系中生成的中间关系表。可以自定义一些设置：

name：代表中间关系表的名字

joinColumns：关系维护端的设置

- @JoinColumn：配置外联字段
- name：外联字段的名字
- referencedColumnName：外联字段的取值来源字段名字

inverseJoinColumns：关系被维护端设置

补充

实体之间的多对多关系，除了使用 `@ManyToMany` 设置，还可以拆分成两个一对多的关系，比如上面的 `Article` 和 `Topic` 之间的多对多关系可以拆分成

- `Article` 和 `TopicArticle` 的一对多
- `Topic` 和 `TopicArticle` 的一对多

具体步骤：

1. 创建一个中间关系实体类`TopicArticle`

```
@Entity
public class TopicArticle {

    @ManyToOne
    private Topic topic;

    @ManyToOne
    private Article article;

    private Date createTime;

    public TopicArticle() {
    }
    //setter和getter省略

}
```

1. 设置`Topic`实体类与`TopicArticle`一对多关系

```
@Entity
public class Topic {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @OneToMany(mappedBy = "topic")
    private List<TopicArticle> topicArticle = new ArrayList<>();

    public Topic() {
    }

   //setter和getter省略
}
```

1. 设置`Article`实体类与`TopicArticle`一对多关系

```java
@Entity
public class Article {

    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private String content;

    @OneToMany(mappedBy = "article")
    private List<TopicArticle> topicArticle = new ArrayList<>();

    public Article() {
    }

    //省略setter和getter

}
```