SpringBoot其他框架

通过了解其他的SpringBoot框架，我们就可以在我们自己的Web服务器上实现更多更高级的功能。

邮件发送：Mail

我们在注册很多的网站时，都会遇到邮件或是手机号验证，也就是通过你的邮箱或是手机短信去接受网站发给你的注册验证信息，填写验证码之后，就可以完成注册了，同时，网站也会绑定你的手机号或是邮箱。

那么，像这样的功能，我们如何实现呢？SpringBoot已经给我们提供了封装好的邮件模块使用：

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

邮件发送

在学习邮件发送之前，我们需要先了解一下什么是电子邮件。

电子邮件也是一种通信方式，是互联网应用最广的服务。通过网络的电子邮件系统，用户可以以非常低廉的价格（不管发送到哪里，都只需负担网费，实际上就是把信息发送到对方服务器而已）、非常快速的方式，与世界上任何一个地方的电子邮箱用户联系。

虽说方便倒是方便，虽然是曾经的霸主，不过现在这个时代，QQ微信横行，手机短信和电子邮箱貌似就只剩收验证码这一个功能了。

要在Internet上提供电子邮件功能，必须有专门的电子邮件服务器。例如现在Internet很多提供邮件服务的厂商：新浪、搜狐、163、QQ邮箱等，他们都有自己的邮件服务器。这些服务器类似于现实生活中的邮局，它主要负责接收用户投递过来的邮件，并把邮件投递到邮件接收者的电子邮箱中。

所有的用户都可以在电子邮件服务器上申请一个账号用于邮件发送和接收，那么邮件是以什么样的格式发送的呢？实际上和Http一样，邮件发送也有自己的协议，也就是约定邮件数据长啥样以及如何通信。

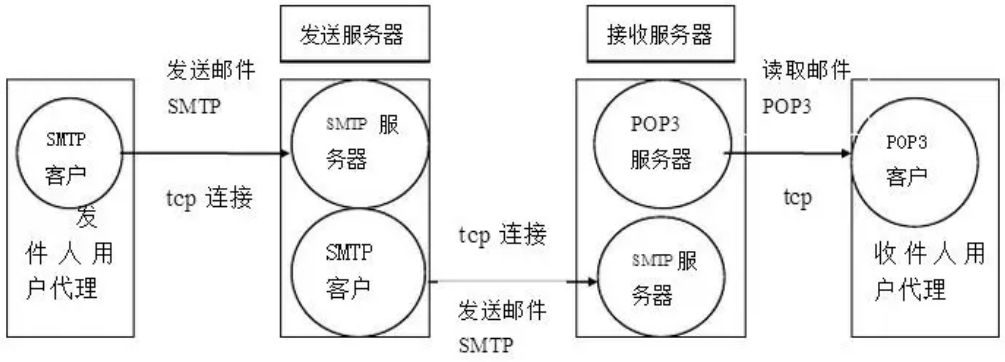
协议名称	协议类型	端口号
smtp	tcp	25
pop3	tcp	110
smtps	tcp	465
pop3s	tcp	995
imap	tcp	143
imaps	tcp	993

比较常用的协议有两种：

1. SMTP协议（主要用于发送邮件 Simple Mail Transfer Protocol）

2. POP3协议（主要用于接收邮件 Post Office Protocol 3）

整个发送/接收流程大致如下：



实际上每个邮箱服务器都有一个smtp发送服务器和pop3接收服务器，比如要从QQ邮箱发送邮件到163邮箱，那么我们只需要通过QQ邮箱客户端告知QQ邮箱的smtp服务器我们需要发送邮件，以及邮件的相关信息，然后QQ邮箱的smtp服务器就会帮助我们发送到163邮箱的pop3服务器上，163邮箱会通过163邮箱客户端告知对应用户收到一封新邮件。

而我们如果想要实现给别人发送邮件，那么就需要连接到对应电子邮箱的smtp服务器上，并告知其我们要发送邮件。而SpringBoot已经帮助我们将最基本的底层通信全部实现了，我们只需要关心smtp服务器的地址以及我们要发送的邮件长啥样即可。

这里以163邮箱 https://mail.163.com 为例，我们需要在配置文件中告诉SpringBootMail我们的smtp服务器的地址以及你的邮箱账号和密码，首先我们要去设置中开启smtp/pop3服务才可以，开启后会得到一个随机生成的密钥，这个就是我们的密码。

```yaml
spring:
  mail:
    # 163邮箱的地址为smtp.163.com，直接填写即可
    host: smtp.163.com
    # 你申请的163邮箱
    username: javastudy111@163.com
    # 注意密码是在开启smtp/pop3时自动生成的，记得保存一下，不然就找不到了
    password: AZJTOAWZESLMHTNI
```

配置完成后，接着我们来进行一下测试：

```java
@SpringBootTest
class SpringBootTestApplicationTests {

    //JavaMailSender是专门用于发送邮件的对象，自动配置类已经提供了Bean
    @Autowired
    JavaMailSender sender;

    @Test
    void contextLoads() {
        //SimpleMailMessage是一个比较简易的邮件封装，支持设置一些比较简单内容
        SimpleMailMessage message = new SimpleMailMessage();
        //设置邮件标题
        message.setSubject("【电子科技大学教务处】关于近期学校对您的处分决定");
        //设置邮件内容
```

```
        message.setText("XXX同学您好，经监控和教务巡查发现，您近期存在旷课、迟到、早退、上课刷抖音行为，" +
                "现已通知相关辅导员，请手写5000字书面检讨，并在2022年4月1日17点前交到辅导员办公室。");
        //设置邮件发送给谁，可以多个，这里就发给你的QQ邮箱
        message.setTo("你的QQ号@qq.com");
        //邮件发送者，这里要与配置文件中的保持一致
        message.setFrom("javastudy111@163.com");
        //OK，万事俱备只欠发送
        sender.send(message);
    }

}
```

如果需要添加附件等更多功能，可以使用MimeMessageHelper来帮助我们完成：

```
@Test
void contextLoads() throws MessagingException {
    //创建一个MimeMessage
    MimeMessage message = sender.createMimeMessage();
    //使用MimeMessageHelper来帮我们修改MimeMessage中的信息
    MimeMessageHelper helper = new MimeMessageHelper(message, true);
    helper.setSubject("Test");
    helper.setText("lbwnb");
    helper.setTo("你的QQ号@qq.com");
    helper.setFrom("javastudy111@163.com");
    //发送修改好的MimeMessage
    sender.send(message);
}
```

邮件注册

既然我们已经了解了邮件发送，那么我们接着来看如何在我们的项目中实现邮件验证。

首先明确验证流程：请求验证码 -> 生成验证码（临时有效，注意设定过期时间） -> 用户输入验证码并填写注册信息 -> 验证通过注册成功！

持久层框架：JPA

- 用了Mybatis之后，你看那个JDBC，真是太逊了。
- 这么说，你的项目很勇哦?
- 开玩笑，我的写代码超勇的好不好。
- 阿伟，你可曾幻想过有一天你的项目里不再有SQL语句?
- 不再有SQL语句？那我怎么和数据库交互啊?
- 我看你是完全不懂哦
- 懂，懂什么啊?
- 你想懂？来，到我项目里来，我给你看点好康的。
- 好康？是什么新框架哦?
- 什么新框架，比新框架还刺激，还可以让你的项目登duang郎哦。
- 哇，杰哥，你项目里面都没SQL语句诶，这是用的什么框架啊?

在我们之前编写的项目中，我们不难发现，实际上大部分的数据库交互操作，到最后都只会做一个事情，那就是把数据库中的数据映射为Java中的对象。比如我们要通过用户名去查找对应的用户，或是通过ID查找对应的学生信息，在使用Mybatis时，我们只需要编写正确的SQL语句就可以直接将获取的数据映射为对应的Java对象，通过调用Mapper中的方法就能直接获得实体类，这样就方便我们在Java中数据库表中的相关信息了。

但是以上这些操作都有一个共性，那就是它们都是通过某种条件去进行查询，而最后的查询结果，都是一个实体类，所以你会发现你写的很多SQL语句都是一个套路`select * from xxx where xxx=xxx`，那么能否有一种框架，帮我们把这些相同的套路给封装起来，直接把这类相似的SQL语句给屏蔽掉，不再由我们编写，而是让框架自己去组合拼接。

认识SpringDataJPA

首先我们来看一个国外的统计：

不对吧，为什么Mybatis这么好用，这么强大，却只有10%的人喜欢呢？然而事实就是，在国外JPA几乎占据了主导地位，而Mybatis并不像国内那样受待见，所以你会发现，JPA都有SpringBoot的官方直接提供的starter，而Mybatis没有。

至于为啥SSM阶段不讲这个，而是放到现在来讲也是因为，在微服务场景下它的优势才能更多的发挥出来。

那么，什么是JPA?

JPA（Java Persistence API）和JDBC类似，也是官方定义的一组接口，但是它相比传统的JDBC，它是为了实现ORM而生的，即Object-Relationl Mapping，它的作用是在关系型数据库和对象之间形成一个映射，这样，我们在具体的操作数据库的时候，就不需要再去和复杂的SQL语句打交道，只要像平时操作对象一样操作它就可以了。

在之前，我们使用JDBC或是Mybatis来操作数据，通过直接编写对应的SQL语句来实现数据访问，但是我们发现实际上我们在Java中大部分操作数据库的情况都是读取数据并封装为一个实体类，因此，为什么不直接将实体类直接对应到一个数据库表呢？也就是说，一张表里面有什么属性，那么我们的对象就有什么属性，所有属性跟数据库里面的字段一一对应，而读取数据时，只需要读取一行的数据并封装为我们定义好的实体类既可以，而具体的SQL语句执行，完全可以交给框架根据我们定义的映射关系去生成，不再由我们去编写，因为这些SQL实际上都是千篇一律的。

而实现JPA规范的框架一般最常用的就是`Hibernate`，它是一个重量级框架，学习难度相比Mybatis也更高一些，而SpringDataJPA也是采用Hibernate框架作为底层实现，并对其加以封装。

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

使用JPA

同样的，我们只需要导入stater依赖即可：

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

接着我们可以直接创建一个类，比如账户类，我们只需要把一个账号对应的属性全部定义好即可：

```
@Data
public class Account {
    int id;
    String username;
    String password;
}
```

接着，我们可以通过注解形式，在属性上添加数据库映射关系，这样就能够让JPA知道我们的实体类对应的数据库表长啥样。

```
@Data
@Entity   //表示这个类是一个实体类
@Table(name = "users")    //对应的数据库中表名称
public class Account {

    @GeneratedValue(strategy = GenerationType.IDENTITY)   //生成策略，这里配置为自增
    @Column(name = "id")    //对应表中id这一列
    @Id     //此属性为主键
    int id;

    @Column(name = "username")   //对应表中username这一列
```

```
    String username;

    @Column(name = "password")    //对应表中password这一列
    String password;
}
```

接着我们来修改一下配置文件：

```
spring:
  jpa:
        #开启SQL语句执行日志信息
    show-sql: true
    hibernate:
        #配置为自动创建
      ddl-auto: create
```

`ddl-auto`属性用于设置自动表定义，可以实现自动在数据库中为我们创建一个表，表的结构会根据我们定义的实体类决定，它有4种

- create 启动时删数据库中的表，然后创建，退出时不删除数据表
- create-drop 启动时删数据库中的表，然后创建，退出时删除数据表 如果表不存在报错
- update 如果启动时表格式不一致则更新表，原有数据保留
- validate 项目启动表结构进行校验 如果不一致则报错

我们可以在日志中发现，在启动时执行了如下SQL语句：

```
Hibernate: create table users (id integer not null auto_increment, password
varchar(255), username varchar(255), primary key (id)) engine=InnoDB
```

而我们的数据库中对应的表已经创建好了。

我们接着来看如何访问我们的表，我们需要创建一个Repository实现类：

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Integer> {

}
```

注意JpaRepository有两个泛型，前者是具体操作的对象实体，也就是对应的表，后者是ID的类型，接口中已经定义了比较常用的数据库操作。编写接口继承即可，我们可以直接注入此接口获得实现类：

```
@SpringBootTest
class JpaTestApplicationTests {

    @Resource
    AccountRepository repository;

    @Test
    void contextLoads() {
        //直接根据ID查找
        repository.findById(1).ifPresent(System.out::println);
    }

}
```

运行后，成功得到查询结果。我们接着来测试增删操作：

```
@Test
void addAccount(){
    Account account = new Account();
    account.setUsername("Admin");
    account.setPassword("123456");
    account = repository.save(account);  //返回的结果会包含自动生成的主键值
    System.out.println("插入时，自动生成的主键ID为："+account.getId());
}
```

```
@Test
void deleteAccount(){
    repository.deleteById(2);   //根据ID删除对应记录
}
```

```
@Test
void pageAccount() {
    repository.findAll(PageRequest.of(0, 1)).forEach(System.out::println);  //直接分页
}
```

我们发现，使用了JPA之后，整个项目的代码中没有出现任何的SQL语句，可以说是非常方便了，JPA依靠我们提供的注解信息自动完成了所有信息的映射和关联。

相比Mybatis，JPA几乎就是一个全自动的ORM框架，而Mybatis则顶多算是半自动ORM框架。

方法名称拼接自定义SQL

虽然接口预置的方法使用起来非常方便，但是如果我们需要进行条件查询等操作或是一些判断，就需要自定义一些方法来实现，同样的，我们不需要编写SQL语句，而是通过方法名称的拼接来实现条件判断，这里列出了所有支持的条件判断名称：

`Distinct`	`findDistinctByLastnameAndFirstname`	`select distinct … where x.lastname = ?1 and x.firstname = ?2`
`And`	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`

`Distinct`	`findDistinctByLastnameAndFirstname`	`select distinct … 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`（参数与附加%绑定）
`EndingWith`	`findByFirstnameEndingWith`	`… where x.firstname like ?1`（参数与前缀%绑定）
`Containing`	`findByFirstnameContaining`	`… where x.firstname like ?1`（参数绑定以%包装）
`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)`

比如我们想要实现根据用户名模糊匹配查找用户：

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Integer> {
        //按照表中的规则进行名称拼接，不用刻意去记，IDEA会有提示
    List<Account> findAllByUsernameLike(String str);
}
```

```
@Test
void test() {
    repository.findAllByUsernameLike("%T%").forEach(System.out::println);
}
```

又比如我们想同时根据用户名和ID一起查询：

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Integer> {

    Account findByIdAndUsername(int id, String username);
  //可以使用Optional类进行包装，Optional<Account> findByIdAndUsername(int id, String
username);

    List<Account> findAllByUsernameLike(String str);
}
```

```
@Test
void test() {
    System.out.println(repository.findByIdAndUsername(1, "Test"));
}
```

比如我们想判断数据库中是否存在某个ID的用户：

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Integer> {

    Account findByIdAndUsername(int id, String username);

    List<Account> findAllByUsernameLike(String str);

    boolean existsAccountById(int id);
}
```

```
@Test
void test() {
    System.out.println(repository.existsAccountByUsername("Test"));
}
```

注意自定义条件操作的方法名称一定要遵循规则，不然会出现异常：

```
Caused by: org.springframework.data.repository.query.QueryCreationException:
Could not create query for public abstract  ...
```

关联查询

在实际开发中，比较常见的场景还有关联查询，也就是我们会在表中添加一个外键字段，而此外键字段又指向了另一个表中的数据，当我们查询数据时，可能会需要将关联数据也一并获取，比如我们想要查询某个用户的详细信息，一般用户简略信息会单独存放一个表，而用户详细信息会单独存放在另一个表中。当然，除了用户详细信息之外，可能在某些电商平台还会有用户的购买记录、用户的购物车，交流社区中的用户帖子、用户评论等，这些都是需要根据用户信息进行关联查询的内容。

我们知道，在JPA中，每张表实际上就是一个实体类的映射，而表之间的关联关系，也可以看作对象之间的依赖关系，比如用户表中包含了用户详细信息的ID字段作为外键，那么实际上就是用户表实体中包括了用户详细信息实体对象：

```
@Data
@Entity
@Table(name = "users_detail")
public class AccountDetail {

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

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

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

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

    @Column(name = "real_name")
    String realName;
}
```

而用户信息和用户详细信息之间形成了一对一的关系，那么这时我们就可以直接在类中指定这种关系：

```
@Data
```

```java
@Entity
@Table(name = "users")
public class Account {

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

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

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

    @JoinColumn(name = "detail_id")    //指定存储外键的字段名称
    @OneToOne    //声明为一对一关系
    AccountDetail detail;
}
```

在修改实体类信息后，我们发现在启动时也进行了更新，日志如下：

```
Hibernate: alter table users add column detail_id integer
Hibernate: create table users_detail (id integer not null auto_increment, address
varchar(255), email varchar(255), phone varchar(255), real_name varchar(255),
primary key (id)) engine=InnoDB
Hibernate: alter table users add constraint FK7gbO21edkxf3mdv5bs75ni6jd foreign
key (detail_id) references users_detail (id)
```

是不是感觉非常方便！都懒得去手动改表结构了。

接着我们往用户详细信息中添加一些数据，一会我们可以直接进行查询：

```java
@Test
void pageAccount() {
    repository.findById(1).ifPresent(System.out::println);
}
```

查询后，可以发现，得到如下结果：

```
Hibernate: select account0_.id as id1_0_0_, account0_.detail_id as
detail_i4_0_0_, account0_.password as password2_0_0_, account0_.username as
username3_0_0_, accountdet1_.id as id1_1_1_, accountdet1_.address as
address2_1_1_, accountdet1_.email as email3_1_1_, accountdet1_.phone as
phone4_1_1_, accountdet1_.real_name as real_nam5_1_1_ from users account0_ left
outer join users_detail accountdet1_ on account0_.detail_id=accountdet1_.id where
account0_.id=?
Account(id=1, username=Test, password=123456, detail=AccountDetail(id=1, address=
四川省成都市青羊区, email=8371289@qq.com, phone=1234567890, realName=本伟))
```

也就是，在建立关系之后，我们查询Account对象时，会自动将关联数据的结果也一并进行查询。

那要是我们只想要Account的数据，不想要用户详细信息数据怎么办呢？我希望在我要用的时候再获取详细信息，这样可以节省一些网络开销，我们可以设置懒加载，这样只有在需要时才会向数据库获取：

```
@JoinColumn(name = "detail_id")
@OneToOne(fetch = FetchType.LAZY)     //将获取类型改为LAZY
AccountDetail detail;
```

接着我们测试一下：

```
@Transactional   //懒加载属性需要在事务环境下获取，因为repository方法调用完后Session会立即关闭
@Test
void pageAccount() {
    repository.findById(1).ifPresent(account -> {
        System.out.println(account.getUsername());   //获取用户名
        System.out.println(account.getDetail());  //获取详细信息（懒加载）
    });
}
```

接着我们来看看控制台输出了什么：

```
Hibernate: select account0_.id as id1_0_0_, account0_.detail_id as
detail_i4_0_0_, account0_.password as password2_0_0_, account0_.username as
username3_0_0_ from users account0_ where account0_.id=?
Test
Hibernate: select accountdet0_.id as id1_1_0_, accountdet0_.address as
address2_1_0_, accountdet0_.email as email3_1_0_, accountdet0_.phone as
phone4_1_0_, accountdet0_.real_name as real_nam5_1_0_ from users_detail
accountdet0_ where accountdet0_.id=?
AccountDetail(id=1, address=四川省成都市青羊区, email=8371289@qq.com,
phone=1234567890, realName=卢本)
```

可以看到，获取用户名之前，并没有去查询用户的详细信息，而是当我们获取详细信息时才进行查询并返回AccountDetail对象。

那么我们是否也可以在添加数据时，利用实体类之间的关联信息，一次性添加两张表的数据呢？可以，但是我们需要稍微修改一下级联关联操作设定：

```
@JoinColumn(name = "detail_id")
@OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL) //设置关联操作为ALL
AccountDetail detail;
```

- ALL：所有操作都进行关联操作
- PERSIST：插入操作时才进行关联操作
- REMOVE：删除操作时才进行关联操作
- MERGE：修改操作时才进行关联操作

可以多个并存，接着我们来进行一下测试：

```
@Test
void addAccount(){
    Account account = new Account();
    account.setUsername("Nike");
    account.setPassword("123456");
    AccountDetail detail = new AccountDetail();
    detail.setAddress("重庆市渝中区解放碑");
```

```
    detail.setPhone("1234567890");
    detail.setEmail("73281937@qq.com");
    detail.setRealName("张三");
    account.setDetail(detail);
    account = repository.save(account);
    System.out.println("插入时，自动生成的主键ID为："+account.getId()+"，外键ID
为："+account.getDetail().getId());
}
```

可以看到日志结果：

```
Hibernate: insert into users_detail (address, email, phone, real_name) values (?,
?, ?, ?)
Hibernate: insert into users (detail_id, password, username) values (?, ?, ?)
插入时，自动生成的主键ID为：6，外键ID为：3
```

结束后会发现数据库中两张表都同时存在数据。

接着我们来看一对多关联，比如每个用户的成绩信息：

```
@JoinColumn(name = "uid")  //注意这里的name指的是Score表中的uid字段对应的就是当前的主键，
会将uid外键设置为当前的主键
@OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.REMOVE)   //在移除
Account时，一并移除所有的成绩信息，依然使用懒加载
List<Score> scoreList;
```

```
@Data
@Entity
@Table(name = "users_score")   //成绩表，注意只存成绩，不存学科信息，学科信息id做外键
public class Score {

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

    @OneToOne   //一对一对应到学科上
    @JoinColumn(name = "cid")
    Subject subject;

    @Column(name = "socre")
    double score;

    @Column(name = "uid")
    int uid;
}
```

```
@Data
@Entity
@Table(name = "subjects")   //学科信息表
public class Subject {

    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "cid")
    @Id
    int cid;

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

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

    @Column(name = "time")
    int time;
}
```

在数据库中填写相应数据，接着我们就可以查询用户的成绩信息了：

```
@Transactional
@Test
void test() {
    repository.findById(1).ifPresent(account -> {
        account.getScoreList().forEach(System.out::println);
    });
}
```

成功得到用户所有的成绩信息，包括得分和学科信息。

同样的，我们还可以将对应成绩中的教师信息单独分出一张表存储，并建立多对一的关系，因为多门课程可能由同一个老师教授（千万别搞晕了，一定要理清楚关联关系，同时也是考验你的基础扎不扎实）：

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "tid")   //存储教师ID的字段，和一对一是一样的，也会当前表中创个外键
Teacher teacher;
```

接着就是教师实体类了：

```
@Data
@Entity
@Table(name = "teachers")
public class Teacher {

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

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

    @Column(name = "sex")
    String sex;
}
```

最后我们再进行一下测试：

```
@Transactional
@Test
void test() {
    repository.findById(3).ifPresent(account -> {
        account.getScoreList().forEach(score -> {
            System.out.println("课程名称："+score.getSubject().getName());
            System.out.println("得分："+score.getScore());
            System.out.println("任课教
师："+score.getSubject().getTeacher().getName());
        });
    });
}
```

成功得到多对一的教师信息。

最后我们再来看最复杂的情况，现在我们一门课程可以由多个老师教授，而一个老师也可以教授多个课程，那么这种情况就是很明显的多对多场景，现在又该如何定义呢？我们可以像之前一样，插入一张中间表表示教授关系，这个表中专门存储哪个老师教哪个科目：

```
@ManyToMany(fetch = FetchType.LAZY)   //多对多场景
@JoinTable(name = "teach_relation",     //多对多中间关联表
        joinColumns = @JoinColumn(name = "cid"),    //当前实体主键在关联表中的字段名称
        inverseJoinColumns = @JoinColumn(name = "tid")   //教师实体主键在关联表中的字
段名称
)
List<Teacher> teacher;
```

接着，JPA会自动创建一张中间表，并自动设置外键，我们就可以将多对多关联信息编写在其中了。

JPQL自定义SQL语句

虽然SpringDataJPA能够简化大部分数据获取场景，但是难免会有一些特殊的场景，需要使用复杂查询才能够去完成，这时你又会发现，如果要实现，只能用回Mybatis了，因为我们需要自己手动编写SQL语句，过度依赖SpringDataJPA会使得SQL语句不可控。

使用JPA，我们也可以像Mybatis那样，直接编写SQL语句，不过它是JPQL语言，与原生SQL语句很类似，但是它是面向对象的，当然我们也可以编写原生SQL语句。

比如我们要更新用户表中指定ID用户的密码：

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Integer> {

    @Transactional     //DML操作需要事务环境，可以不在这里声明，但是调用时一定要处于事务环境
下
    @Modifying      //表示这是一个DML操作
    @Query("update Account set password = ?2 where id = ?1") //这里操作的是一个实体
类对应的表，参数使用?代表，后面接第n个参数
    int updatePasswordById(int id, String newPassword);
}
```

```
@Test
void updateAccount(){
    repository.updatePasswordById(1, "654321");
}
```

现在我想使用原生SQL来实现根据用户名称修改密码：

```
@Transactional
@Modifying
@Query(value = "update users set password = :pwd where username = :name",
nativeQuery = true) //使用原生SQL，和Mybatis一样，这里使用 :名称 表示参数，当然也可以继续用上面那种方式。
int updatePasswordByUsername(@Param("name") String username,   //我们可以使用@Param指定名称
                             @Param("pwd") String newPassword);
```

```
@Test
void updateAccount(){
    repository.updatePasswordByUsername("Admin", "654321");
}
```

通过编写原生SQL，在一定程度上弥补了SQL不可控的问题。

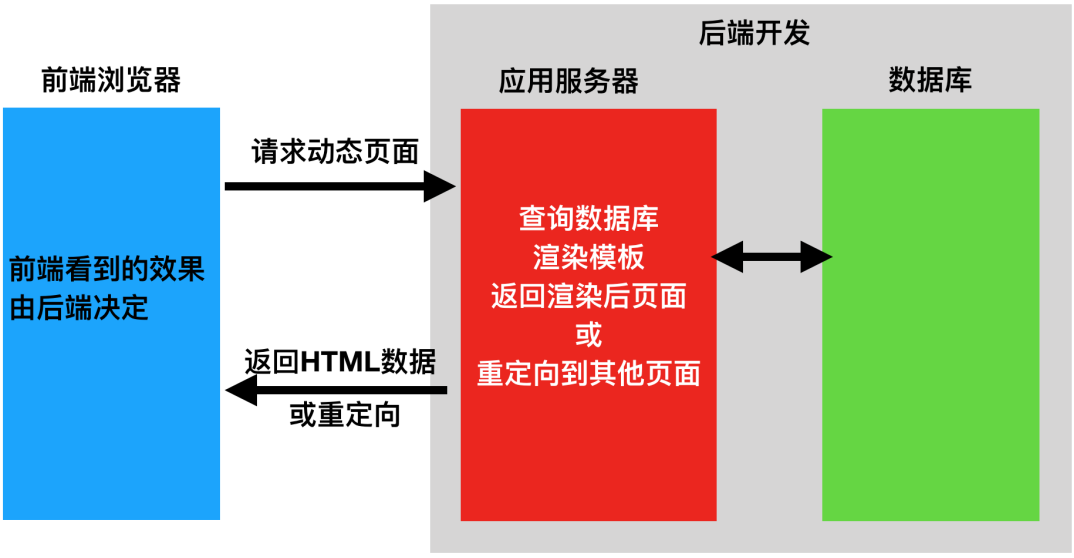
虽然JPA能够为我们带来非常便捷的开发体验，但是正式因为太便捷了，保姆级的体验有时也会适得其反，可能项目开发到后期特别庞大时，就只能从底层SQL语句开始进行优化，而由于JPA尽可能地在屏蔽我们对SQL语句的编写，所以后期优化是个大问题，并且Hibernate相对于Mybatis来说，更加重量级。不过，在微服务的时代，单体项目一般不会太大，而JPA的劣势并没有太明显地体现出来。

有关Mybatis和JPA的对比，可以参考：https://blog.csdn.net/u010253246/article/details/105731204

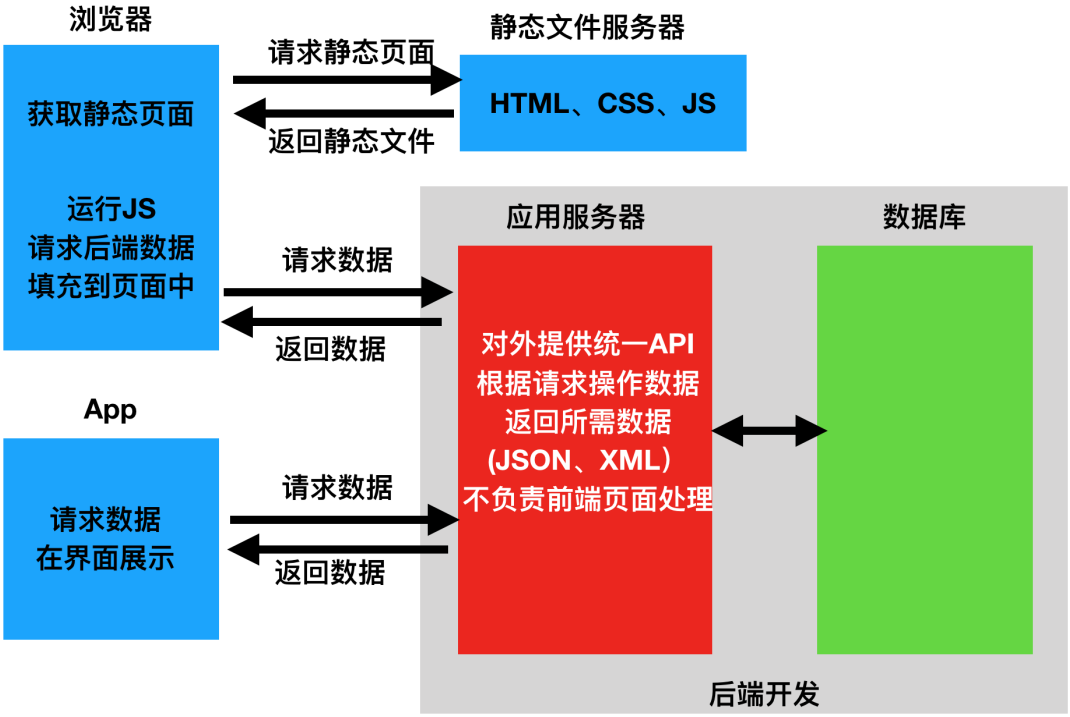
Extra. 前后端分离跨域处理

我们的项目已经处于前后端分离状态了，那么前后端分离状态和我们之前的状态有什么区别的呢？

- **不分离**：前端页面看到的都是由后端控制，由后端渲染页面或重定向，后端需要控制前端的展示，前端与后端的耦合度很高。比如我们之前都是使用后端来执行重定向操作或是使用Thymeleaf来填充数据，而最终返回的是整个渲染好的页。



- **分离：**后端仅返回前端所需的数据，不再渲染HTML页面，不再控制前端的效果。至于前端用户看到什么效果，从后端请求的数据如何加载到前端中，都由前端通过JS等进行动态数据填充和渲染。这样后端只返回JSON数据，前端处理JSON数据并展示，这样前后端的职责就非常明确了。



实现前后端分离有两种方案，一种是直接放入SpringBoot的资源文件夹下，但是这样实际上还是在依靠SpringBoot内嵌的Tomcat服务器进行页面和静态资源的发送，我们现在就是这种方案。

另一种方案就是直接将所有的页面和静态资源单独放到代理服务器上（如Nginx），这样我们后端服务器就不必再处理静态资源和页面了，专心返回数据即可，而前端页面就需要访问另一个服务器来获取，虽然逻辑和明确，但是这样会出现跨域问题，实际上就是我们之前所说的跨站请求伪造，为了防止这种不安全的行为发生，所以对异步请求会进行一定的限制。

这里，我们将前端页面和后端页面直接分离进行测试，在登陆时得到如下错误：

```
Access to XMLHttpRequest at 'http://localhost:8080/api/auth/login' from origin
'http://localhost:63342' has been blocked by CORS policy: No 'Access-Control-
Allow-Origin' header is present on the requested resource.
```

可以很清楚地看到，在Ajax发送异步请求时，我们的请求被阻止，原因是在响应头中没有包含`Access-Control-Allow-Origin`，也就表示，如果服务端允许跨域请求，那么会在响应头中添加一个`Access-Control-Allow-Origin`字段，如果不允许跨域，就像现在这样。那么，什么才算是跨域呢：

1. 请求协议`如http、https`不同
2. 请求的地址/域名不同
3. 端口不同

因为我们现在相当于前端页面访问的是静态资源服务器，而后端数据是由我们的SpringBoot项目提供，它们是两个不同的服务器，所以在垮服务器请求资源时，会被判断为存在安全风险。

但是现在，由于我们前后端是分离状态，我们希望的是能够实现跨域请求，这时我们就需要添加一个过滤器来处理跨域问题：

```
@Bean
public CorsFilter corsFilter() {
    //创建CorsConfiguration对象后添加配置
    CorsConfiguration config = new CorsConfiguration();
    //设置放行哪些原始域，这里直接设置为所有
    config.addAllowedOriginPattern("*");
    //你可以单独设置放行哪些原始域 config.addAllowedOrigin("http://localhost:2222");
    //放行哪些原始请求头部信息
    config.addAllowedHeader("*");
    //放行哪些请求方式，*代表所有
    config.addAllowedMethod("*");
    //是否允许发送Cookie，必须要开启，因为我们的JSESSIONID需要在Cookie中携带
    config.setAllowCredentials(true);
    //映射路径
    UrlBasedCorsConfigurationSource corsConfigurationSource = new
UrlBasedCorsConfigurationSource();
    corsConfigurationSource.registerCorsConfiguration("/**", config);
    //返回CorsFilter
    return new CorsFilter(corsConfigurationSource);
}
```

这样，我们的SpringBoot项目就支持跨域访问了，接着我们再来尝试进行登陆，可以发现已经能够正常访问了，并且响应头中包含了以下信息：

```
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Access-Control-Allow-Origin: http://localhost:63342
Access-Control-Expose-Headers: *
Access-Control-Allow-Credentials: true
```

可以看到我们当前访问的原始域已经被放行了。

但是还有一个问题，我们的Ajax请求中没有携带Cookie信息（这个按理说属于前端知识了）这里我们稍微改一下，不然我们的请求无法确认身份：

```
function get(url, success){
    $.ajax({
        type: "get",
        url: url,
        async: true,
```

```
        dataType: 'json',
        xhrFields: {
            withCredentials: true
        },
        success: success
    });
}

function post(url, data, success){
    $.ajax({
        type: "post",
        url: url,
        async: true,
        data: data,
        dataType: 'json',
        xhrFields: {
            withCredentials: true
        },
        success: success
    });
}
```

添加两个封装好的方法，并且将`withCredentials`开启，这样在发送异步请求时，就会携带Cookie信息了。

在学习完成Linux之后，我们会讲解如何在Linux服务器上部署Nginx反向代理服务器。

接口管理：Swagger

在前后端分离项目中，前端人员需要知道我们后端会提供什么数据，根据后端提供的数据来进行前端页面渲染（在之前我们也演示过）这个时候，我们就需要编写一个API文档，以便前端人员随时查阅。

但是这样的一个文档，我们也不可能单独写一个项目去进行维护，并且随着我们的后端项目不断更新，文档也需要跟随更新，这显然是很麻烦的一件事情，那么有没有一种比较好的解决方案呢？

当然有，那就是丝袜哥：Swagger

走进Swagger

Swagger的主要功能如下：

- 支持 API 自动生成同步的在线文档：使用 Swagger 后可以直接通过代码生成文档，不再需要自己手动编写接口文档了，对程序员来说非常方便，可以节约写文档的时间去学习新技术。
- 提供 Web 页面在线测试 API：光有文档还不够，Swagger 生成的文档还支持在线测试。参数和格式都定好了，直接在界面上输入参数对应的值即可在线测试接口。

结合Spring框架（Spring-fox），Swagger可以很轻松地利用注解以及扫描机制，来快速生成在线文档，以实现当我们项目启动之后，前端开发人员就可以打开Swagger提供的前端页面，查看和测试接口。依赖如下：

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

SpringBoot 2.6以上版本修改了路径匹配规则，但是Swagger3还不支持，这里换回之前的，不然启动直接报错：

```
spring:
    mvc:
        pathmatch:
      matching-strategy: ant_path_matcher
```

项目启动后，我们可以直接打开：http://localhost:8080/swagger-ui/index.html，这个页面（要是觉得丑，UI是可以换的，支持第三方）会显示所有的API文档，包括接口的路径、支持的方法、接口的描述等，并且我们可以直接对API接口进行测试，非常方便。

我们可以创建一个配置类去配置页面的相关信息：

```
@Configuration
public class SwaggerConfiguration {

    @Bean
    public Docket docket() {
        return new Docket(DocumentationType.OAS_30).apiInfo(
                new ApiInfoBuilder()
                        .contact(new Contact("你的名字",
"https://www.bilibili.com", "javastudy111*@163.com"))
                        .title("图书管理系统 - 在线API接口文档")
                        .build()
        );
    }
}
```

接口信息配置

虽然Swagger的UI界面已经可以很好地展示后端提供的接口信息了，但是非常的混乱，我们来看看如何配置接口的一些描述信息。

首先我们的页面中完全不需要显示ErrorController相关的API，所以我们配置一下选择哪些Controller才会生成API信息：

```
@Bean
public Docket docket() {
    ApiInfo info = new ApiInfoBuilder()
            .contact(new Contact("你的名字", "https://www.bilibili.com",
"javastudy111@163.com"))
            .title("图书管理系统 - 在线API接口文档")
            .description("这是一个图书管理系统的后端API文档，欢迎前端人员查阅！")
            .build();
    return new Docket(DocumentationType.OAS_30)
            .apiInfo(info)
            .select()       //对项目中的所有API接口进行选择
            .apis(RequestHandlerSelectors.basePackage("com.example.controller"))
            .build();
}
```

接着我们来看看如何为一个Controller编写API描述信息：

```
@Api(tags = "账户验证接口", description = "包括用户登录、注册、验证码请求等操作。")
@RestController
@RequestMapping("/api/auth")
public class AuthApiController {
```

我们可以直接在类名称上面添加`@Api`注解，并填写相关信息，来为当前的Controller设置描述信息。

接着我们可以为所有的请求映射配置描述信息：

```
@ApiResponses({
        @ApiResponse(code = 200, message = "邮件发送成功"),
        @ApiResponse(code = 500, message = "邮件发送失败")   //不同返回状态码描述
})
@ApiOperation("请求邮件验证码")   //接口描述
@GetMapping("/verify-code")
public RestBean<Void> verifyCode(@ApiParam("邮箱地址")   //请求参数的描述
                                 @RequestParam("email") String email){
```

```
@ApiIgnore     //忽略此请求映射
@PostMapping("/login-success")
public RestBean<Void> loginSuccess(){
    return new RestBean<>(200, "登陆成功");
}
```

我们也可以为实体类配置相关的描述信息：

```
@Data
@ApiModel(description = "响应实体封装类")
@AllArgsConstructor
public class RestBean<T> {

    @ApiModelProperty("状态码")
    int code;
    @ApiModelProperty("状态码描述")
    String reason;
    @ApiModelProperty("数据实体")
```

```
    T data;

    public RestBean(int code, String reason) {
        this.code = code;
        this.reason = reason;
    }
}
```

这样，我们就可以在文档中查看实体类简介以及各个属性的介绍了。

最后我们再配置一下多环境：

```
<profiles>
    <profile>
        <id>dev</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <environment>dev</environment>
        </properties>
    </profile>
    <profile>
        <id>prod</id>
        <activation>
            <activeByDefault>false</activeByDefault>
        </activation>
        <properties>
            <environment>prod</environment>
        </properties>
    </profile>
</profiles>
```

```
<resources>
    <resource>
        <directory>src/main/resources</directory>
        <excludes>
            <exclude>application*.yaml</exclude>
        </excludes>
    </resource>
    <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
        <includes>
            <include>application.yaml</include>
            <include>application-${environment}.yaml</include>
        </includes>
    </resource>
</resources>
```

首先在Maven中添加两个环境，接着我们配置一下不同环境的配置文件：

```
  jpa:
    show-sql: false
    hibernate:
      ddl-auto: update
springfox:
  documentation:
    enabled: false
```

在生产环境下，我们选择不开启Swagger文档以及JPA的数据库操作日志，这样我们就可以根据情况选择两套环境了。