



下载APP



32 | 通用模块（下）：用户模块开发

2021-12-03 叶剑峰

《手把手带你写一个Web框架》

课程介绍 >

**讲述：叶剑峰**

时长 15:19 大小 14.04M



你好，我是轩脉刃。

上一节课我们设计好用户模块的需求后，开始了后端开发。在后端开发中我们明确了开发流程的四个步骤，先将接口 swagger 化，再定义用户服务协议，接着开发模块接口，最后实现用户服务协议。而且上节课已经完成了接口 swagger 化，以及用户服务协议设计的模型部分。

这节课，我们就继续完成用户服务协议的定义，再开发模块接口和实现用户服务协议。



用户服务协议

前面我们设计好了一个模型 User 了，“接口优于实现”，来设计这个服务的接口，看看要提供哪些能力。

首先用户服务一定要提供的是预注册能力，所以提供了一个 Register 方法。预注册之后，我们还要提供发送邮件的能力，再提供一个发送邮件的接口 SendRegisterMail。当然最后要提供一个确认注册用户的接口 VerfityRegister。

在登录这块，用户服务一定要提供登录、登出的接口 Login 和 Logout。同时由于所有业务请求，比如创建问题等逻辑，我们需要使用 token 来获取用户信息，所以我们也要提供验证登录的接口 VerifyLogin。

于是整体的接口设计如下，详细信息都写在注释中了：

[复制代码](#)

```
1 // Service 用户相关的服务
2 type Service interface {
3
4     // Register 注册用户,注意这里只是将用户注册,并没有激活,需要调用
5     // 参数:user必填,username,password,email
6     // 返回值: user 带上token
7     Register(ctx context.Context, user *User) (*User, error)
8     // SendRegisterMail 发送注册的邮件
9     // 参数:user必填: username,password,email,token
10    SendRegisterMail(ctx context.Context, user *User) error
11    // VerifyRegister 注册用户,验证注册信息,返回验证是否成功
12    VerifyRegister(ctx context.Context, token string) (bool, error)
13
14    // Login 登录相关,使用用户名密码登录,获取完成User信息
15    Login(ctx context.Context, user *User) (*User, error)
16    // Logout 登出
17    Logout(ctx context.Context, user *User) error
18    // VerifyLogin 登录验证
19    VerifyLogin(ctx context.Context, token string) (*User, error)
20 }
```

这里也说明一下，要抽象设计出一个服务模块的协议确实不是一件很简单的事情，也不一定能够一次性设计好。

我们说过，从“服务需要提供哪些对外能力”的角度来思考，会比较完善。比如这里为什么要设计一个 VerifyLogin 能力呢？系统对外提供的接口并没有这个服务，但是在内部，

我们每次验证 token 的时候，会需要用 token 来验证和换取 user 的。所以这个接口的设计是有“需要”的。

服务协议的设计从需求出发，当遇到新的需求，不断迭代就可以了。

用户模块接口实现

设计了用户服务的协议，下一步我们也不是急于实现它，需要先验证下这些服务协议是否能满足我们的“需求”。如何验证呢？可以直接开发用户接口，确认是否有未满足的需求。

上一节课梳理了，要实现四个接口：

```
app/http/module/user/api_register.go
```

```
app/http/module/user/api_verify.go
```


```
app/http/module/user/api_login.go
```

```
app/http/module/user/api_logout.go
```

我们还是拿其中比较复杂的注册接口 api_register.go 做一下说明，其他接口的实现没什么难点，你可以参考 GitHub 上的代码。

注册接口我们要做几个事情？**首先验证接口参数，其次要进行预注册，然后发送预注册的验证邮件，最后返回成功状态。**


验证接口参数之前讲过，使用定义好的 registerParam 结构和 Gin 带有的 binding 逻辑就可以做参数的获取和验证了。预注册的逻辑，既然已经定义好了用户服务的预注册接口，这里可以直接调用这个接口 Register。同样，发送验证邮件的接口我们也已经在用户服务中定义好了，直接调用 SendRegisterMail 即可。最终，返回成功状态，我们使用 hade 框架对 Gin 扩展的 IStatusOk。

 复制代码

```
1 func (api *UserApi) Register(c *gin.Context) {  
2     // 验证参数  
3     userService := c.MustMake(provider.UserKey).(provider.Service)  
4     logger := c.MustMake(contract.LogKey).(contract.Log)
```

```
5   param := &registerParam{}
6   if err := c.ShouldBind(param); err != nil {
7       c.ISetStatus(400).IText("参数错误"); return
8   }
9
10  // 注册对象
11  model := &provider.User{
12      UserName: param.UserName,
13      Password: param.Password,
14      Email: param.Email,
15      CreatedAt: time.Now(),
16  }
17  // 注册
18  userWithToken, err := userService.Register(c, model)
19  if err != nil {
20      logger.Error(c, err.Error(), map[string]interface{}{
21          "stack": fmt.Sprintf("%+v", err),
22      })
23      c.ISetStatus(500).IText(err.Error()); return
24  }
25  if userWithToken == nil {
26      c.ISetStatus(500).IText("注册失败"); return
27  }
28
29  if err := userService.SendRegisterMail(c, userWithToken); err != nil {
30
31      c.ISetStatus(500).IText("发送电子邮件失败"); return
32  }
33
34  c.ISetOkStatus().IText("注册成功，请前往邮箱查看邮件"); return
35 }
36
37
```

这里使用了之前我们对 Gin 框架扩展定义的 Response 结构中的链式方法：

 复制代码

```
1 c.ISetOkStatus().IText("注册成功，请前往邮箱查看邮件");
```

很明显，这种方法确实比 Gin 框架自带的 Response 方法更为优雅轻便了。

实现好 Register 接口，我们基本确认了之前设计的用户服务中注册部分是满足需求的。再一个个接口 Verify/Login/Logout 都实现一下，基本能确定之前用户服务的设计是可以的。

开发模块接口

既然我们已经确定了用户服务设计可行，进入最后一步，实现这些用户服务定义的协议方法。在实现中，你能看到很多之前定义的各种服务的具体使用。用户注册的三个相关协议接口的实现，我们详细说一下，其他登录相关的接口协议，你可以参考 GitHub 上的代码。

预注册协议接口

[复制代码](#)

```
1 // Register 注册用户,注意这里只是将用户注册,并没有激活,需要调用
2 // 参数: user必填, username, password, email
3 // 返回值: user 带上token
4 Register(ctx context.Context, user *User) (*User, error)
```

预注册协议接口的具体实现要做几个事情：

1. 去数据库判断邮箱是否已经注册用户了，如果邮箱已经注册，那么这个预注册操作是不能执行的；
2. 去数据库判断用户名是否已经被注册了，如果用户名已经被注册了，那么预注册操作也是不能执行的；
3. 生成预注册的验证 token；
4. 将要注册的用户存储在缓存中，存储 1 天，待用户注册验证。

我们注意到四步操作里**前面两步是去数据库的查询操作**，所以可以使用 **hade 框架的 ORM 服务**，先从容器中获取 ORM 服务，使用 GetDB 获取 gorm.DB，接着就可以使用 gorm 的 Where、First 等方法了。由于之前已经定义好了 User 结构作为数据库模型，所以我们直接使用这个模型：

[复制代码](#)

```
1 // 判断邮箱是否已经注册了
2 ormService := u.container.MustMake(contract.ORMKey).(contract.ORMService)
3 db, err := ormService.GetDB()
4 if err != nil {
5     return nil, err
6 }
7 userDB := &User{}
```



```
8 if db.Where(&User{Email: user.Email}).First(userDB).Error != gorm.ErrRecordNot
9     return nil, errors.New("邮箱已注册用户，不能重复注册")
10 }
11 if db.Where(&User{UserName: user.UserName}).First(userDB).Error != gorm.ErrRec
12     return nil, errors.New("用户名已经被注册，请换一个用户名")
13 }
```

而第三步生成 token，就使用一个简单的随机生成 token 的算法，直接去一排字符串中随机获取下标来生成 token。

[复制代码](#)

```
1 const letterBytes = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
2
3 func genToken(n int) string {
4     b := make([]byte, n)
5     for i := range b {
6         // 这里是随机获取的
7         b[i] = letterBytes[rand.Intn(len(letterBytes))]
8     }
9     return string(b)
10 }
```

最后一步，需要将 User 对象存储到缓存中。我们使用 key 为"user:register:[token]"来存储 User 对象。

[复制代码](#)

```
1 // 将请求注册进入redis，保存一天
2 cacheService := u.container.MustMake(contract.CacheKey).(contract.CacheService
3
4 key := fmt.Sprintf("user:register:%v", user.Token)
5 if err := cacheService.SetObj(ctx, key, user, 24*time.Hour); err != nil {
6     return nil, err
7 }
8 return user, nil
```

但是你还记得吗，在 hade 的 Cache 服务中，如果直接使用 SetObj 和 GetObj 操作对象，那么这个对象必须实现 BinaryMarshaler 和 BinaryUnMarshaler。所以我们再给 User 对象实现这两个接口。在 app/provider/user/service.go 中：

[复制代码](#)

```
1 // MarshalBinary 实现BinaryMarshaler 接口
```

```
2 func (b *User) MarshalBinary() ([]byte, error) {
3     return json.Marshal(b)
4 }
5
6 // UnmarshalBinary 实现 BinaryUnMarshaler 接口
7 func (b *User) UnmarshalBinary(bt []byte) error {
8     return json.Unmarshal(bt, b)
9 }
```

于是，用户服务的 Register 方法实现就写好了。在这个小小的方法中，我们已经演示了之前定义的 ORM 服务、Cache 服务，所有的这些服务在服务容器中都可以得到。你可以具体感受一下，容器加服务协议在具体的业务代码中带来的便利。

发送邮件协议接口

[复制代码](#)

```
1 // SendRegisterMail 发送注册的邮件
2 // 参数: user必填: username, password, email, token
3 SendRegisterMail(ctx context.Context, user *User) error
```

发送邮件协议接口要实现的就是一个邮件发送的功能。在 Golang 中邮件发送功能也是有现成库的，[gomail](#)。这个库目前已经有 3.4k star 了，基于限制比较少的 MIT 协议。

发送电子邮件的方式其实有很多种，但是我们最好使用 SMTP 的方式来发送邮件，因为 SMTP 的服务提供方，基本上都是在互联网上已经认证的服务提供商，比如 Gmail、126 等。通过这些邮件服务提供商注册的 SMTP 账号发送邮件，基本上不会进入对方邮箱的“垃圾箱”中。

不过所有邮件服务提供商的 SMTP 账号都需要单独申请，但是基本都是免费的。这里是我使用 126 注册的邮箱申请了一个 126 的 SMTP 发送账号。

我们把 SMTP 的账号信息存储在配置文件 config/development/app.yaml 中：

[复制代码](#)

```
1 domain: "http://hadecast.funaiio.cn"
2
3 smtp:
4     host: "smtp.126.com"
```

```
5     port: 25
6     from: "jianfengye110@126.com"
7     username: "jianfengye110"
8     password: "123456"
```

下面来演示如何使用 gmail 来通过 SMTP 账号发送邮件，我们直接看 app/provider/user/service.go 的具体实现：

[复制代码](#)

```
1 func (u *UserService) SendRegisterMail(ctx context.Context, user *User) error {
2     logger := u.container.MustMake(contract.LogKey).(contract.Log)
3     configer := u.container.MustMake(contract.ConfigKey).(contract.Config)
4
5     // 配置服务中获取发送邮件需要的参数
6     host := configer.GetString("app.smtp.host")
7     port := configer.GetInt("app.smtp.port")
8     username := configer.GetString("app.smtp.username")
9     password := configer.GetString("app.smtp.password")
10    from := configer.GetString("app.smtp.from")
11    domain := configer.GetString("app.domain")
12
13    // 实例化gmail
14    d := gmail.NewDialer(host, port, username, password)
15
16    // 组装message
17    m := gmail.NewMessage()
18    m.SetHeader("From", from)
19    m.SetAddressHeader("To", user.Email, user.UserName)
20    m.SetHeader("Subject", "感谢您注册我们的hadecast")
21    link := fmt.Sprintf("%v/user/register/verify?token=%v", domain, user.Token)
22    m.SetBody("text/html", fmt.Sprintf("请点击下面的链接完成注册：%s", link))
23
24    // 发送电子邮件
25    if err := d.DialAndSend(m); err != nil {
26        logger.Error(ctx, "send email error", map[string]interface{}{
27            "err": err,
28            "message": m,
29        })
30        return err
31    }
32    return nil
33 }
```

首先通过 hade 的配置服务来获取 SMTP 的所有配置，使用这些配置实例化一个 gmail.Dialer 对象；然后创建邮件的内容，内容的 From 和 To 分别代表发送方和接收


方，接收方自然就是我们的预注册用户填写的邮箱。将链接放在邮件的 Body 里面。组装好邮件内容之后，我们使用 `DialAndSend` 就可以直接发送一个邮件到预注册的用户邮箱了。

在这个过程中，**我们会希望如果发送邮箱失败的话，使用日志记录一下发送失败的原因和内容**。这个是很有必要的，因为后续如果希望有一些脚本能补发邮件，这个日志就很有帮助了。所以使用 `hade` 定义的日志服务，这里使用日志服务的 `Error` 方法来记录发送邮件错误信息。

不管配置服务还是日志服务，都是从服务容器中可以获取到。按照上述逻辑，发送邮件的接口就完成了。

注册验证协议接口


注册相关的最后一个协议接口是验证注册。

 复制代码

```
1 // VerifyRegister 注册用户，验证注册信息，返回验证是否成功
2 VerifyRegister(ctx context.Context, token string) (bool, error)
```


这个协议接口逻辑会复杂一些了。

它的参数为一个 token，我们首先要拿着这个 token 去缓存中，获取到这个 token 对应的预注册用户。由于之前已经将 `User` 实现了 `BinaryUnmarshaler` 接口，这里就使用缓存服务的 `GetObj` 方法：

 复制代码

```
1 //验证token
2 cacheService := u.container.MustMake(contract.CacheKey).(contract.CacheService)
3 key := fmt.Sprintf("user:register:%v", token)
4 user := &User{}
5 if err := cacheService.GetObj(ctx, key, user); err != nil {
6     return false, err
7 }
8 if user.Token != token {
9     return false, nil
10 }
```

然后下一步，**由于预注册和注册验证过程是异步的，中间数据库是有可能发生变化的，**所以我们需要再次验证一下这个用户在数据库中是否已经存在了，他的用户名和邮箱是否是唯一的。


 复制代码

```
1 //验证邮箱，用户名的唯一
2 ormService := u.container.MustMake(contract.ORMKey).(contract.ORMService)
3 db, err := ormService.GetDB()
4 if err != nil {
5     return false, err
6 }
7 userDB := &User{}
8 if db.Where(&User{Email: user.Email}).First(userDB).Error != gorm.ErrRecordNot
9     return false, errors.New("邮箱已注册用户，不能重复注册")
10 }
11 if db.Where(&User{UserName: user.UserName}).First(userDB).Error != gorm.ErrRec
12     return false, errors.New("用户名已经被注册，请换一个用户名")
13 }
```

最后准备将这个缓存中的用户存储进入数据库 users 表。这里我们知道，缓存中预注册用户的密码是用户填写的真实密码。但是将真实密码直接存储进入数据库，是一个非常不安全的做法。如果我们的数据库被黑客攻击拖库了，这对我们的网站用户是个非常大的影响。所以这里我们**有必要对用户的密码做一次加密操作。**

Golang 的 golang.org/x/crypto/bcrypt 库提供了对密码进行加密的标准方法。还记得这种 golang.org/x/ 开头的库么，可以说是 Golang 标准库的预备库，我们可以直接放心使用。在这个库中，提供了加密密码的方法 `GenerateFromPassword` 和验证密码的方法 `CompareHashAndPassword`。


在这个函数中，我们就使用到加密密码的方法：

 复制代码

```
1 // 验证成功将密码存储数据库之前需要加密，不能原文存储进入数据库
2 hash, err := bcrypt.GenerateFromPassword([]byte(user.Password), bcrypt.MinCost)
3 if err != nil {
4     return false, err
5 }
```

GenerateFromPassword 的第二个参数 cost，是表示加密密码的复杂度，最小必须为 MinCost。

最后一步，就是将用户存储到数据库中了。同样使用的是 Gorm，其中有个 Create 方法，能将对象保存进入数据库：

 复制代码

```
1 user.Password = string(hash)
2
3 // 具体在数据库创建用户
4 if err := db.Create(user).Error; err != nil {
5     return false, err
6 }
7 return true, nil
```

以上，就完成了注册实现的具体方法了。

后端调试

现在，汇总两节课的成果，我们完成了用户服务、用户模块接口以及 swagger 的搭建。之后就可以很方便地使用 swagger 来调试用户模块和用户服务了。

记得开启 hade 特有的调试模式：`./bbs dev backend`

```
~/Documents/workspace/gohade/bbs  ↗ geekbang/31 ●  ./bbs dev backend
启动后端服务： http://127.0.0.1:8072
监控文件夹： /Users/yejianfeng/Documents/workspace/gohade/bbs/app
后端服务pid: 18690
代理服务启动： http://127.0.0.1:8070
[PID] 18690
app serve url: :8072
```

打开浏览器 <http://localhost:8070/swagger/index.html> 看到 swagger-UI 界面。点击要调试接口的 “try it out” 按钮进入接口调用，填写要调用的接口参数，点击 “Execute” 调用接口，并且获取接口返回值。

POST

/user/register 用户注册

用户注册接口

Parameters

Cancel

Name

registerParam

required

(body)

Description

注册参数

Example Value

Model

```
{
  "email": "jianfengye110@gmail.com",
  "password": "123456789",
  "username": "yejianfeng"
}
```

Cancel

Parameter content type

application/json

Execute

Clear

Responses

Response content type application/json

Curl

```
curl -X POST "http://localhost:8070/user/register" -H "accept: application/json" -H "Content-Type: application/json" -d '{"email": "jianfengye110@gmail.com", "password": "123456789", "username": "yejianfeng"}'
```

Request URL

http://localhost:8070/user/register

Server response

Code

500

Undocumented

Details

Error: Internal Server Error

Response body

邮箱已注册用户，不能重复注册

Response headers

content-length: 42
content-type: application/text
date: Thu, 25 Nov 2021 01:46:01 GMT

Responses

Code

200

Description

注册成功

Example Value

Model

"string"

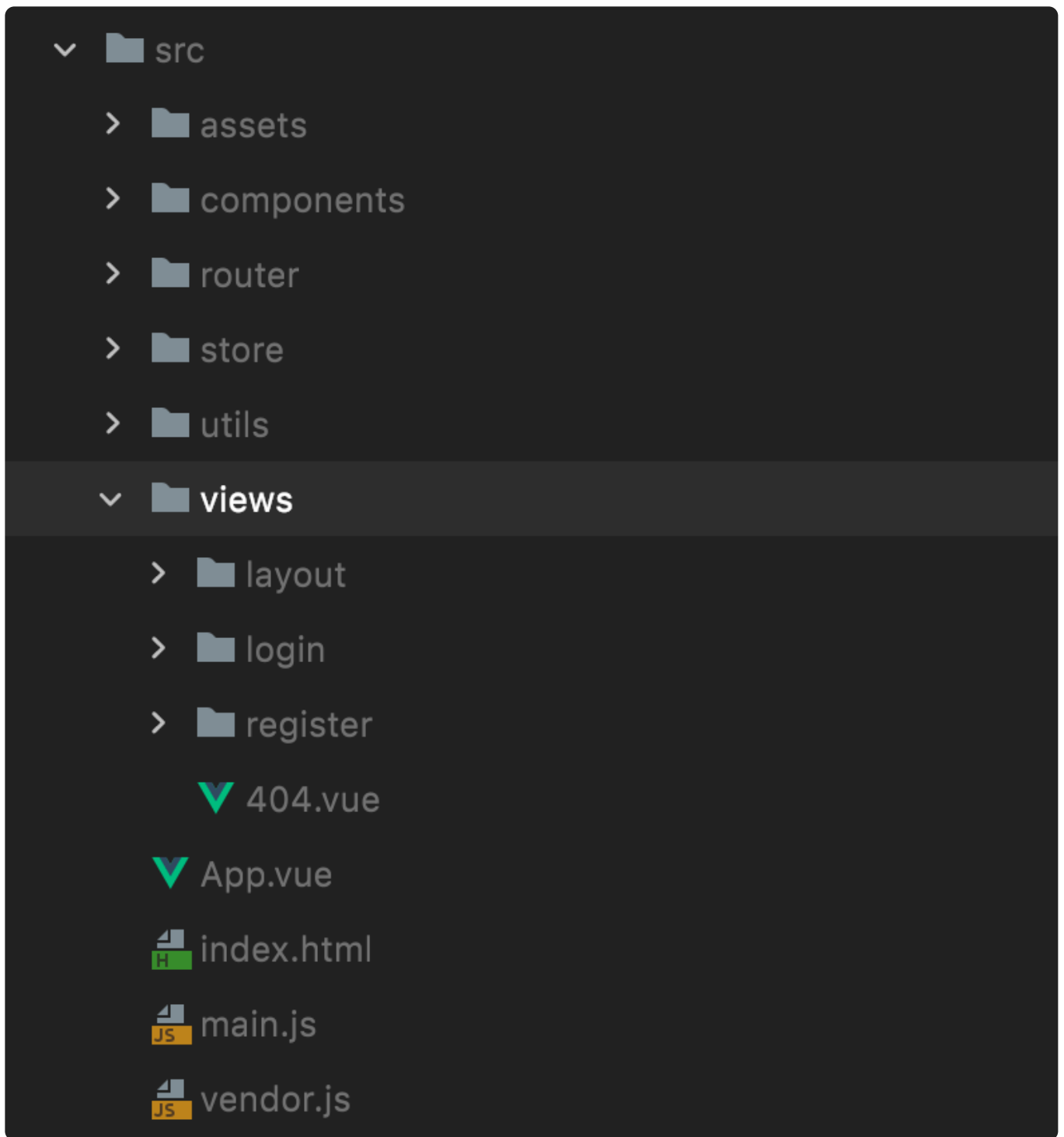
如果接口调用错误，我们要修改接口，只需要直接在 IDE 上修改代码，并且直接保存，hade 就会检测到文件更新，并且重新编译重启服务，立刻生效。

```
~/Documents/workspace/gohade/bbs  geekbang/31  ●  ./bbs dev backend
启动后端服务： http://127.0.0.1:8072
监控文件夹： /Users/yejianfeng/Documents/workspace/gohade/bbs/app
后端服务pid: 19346
代理服务启动： http://127.0.0.1:8070
[PID] 19346
app serve url: :8072
...检测到文件更新，重启服务开始...
编译hade成功
启动后端服务： http://127.0.0.1:8072
后端服务pid: 19390
...检测到文件更新，重启服务结束...
[PID] 19390
app serve url: :8072
```

前端开发

关于用户接口的前端开发部分，由于并不是我们课程的重点，就简要描述一下关键实现点。

前端一共就两个页面，注册页面和登录页面，所以我们在 `src/views/` 中创建两个文件夹，`register` 和 `login`，分别存储这两个页面。这里我主要也描述一下注册页面的具体实现。



在注册页面上，实际上是搭建了一个表单，在 element-UI 中我们可以使用 el-form 来方便搭建一个漂亮的表单，这个表单包含用户名、邮箱、密码等信息，并且这些信息都对应 script 中的 form 数据。在表单的按钮，按钮点击行为我们设置成触发 submitFrom 方法。在 src/views/register/index.vue 中：

[复制代码](#)

```
1 <el-form v-model="form" class="register-form">
2   <el-form-item >
3     <el-input v-model="form.username" placeholder="用户名" ></el-input>
4   </el-form-item>
5   <el-form-item >
```




```
6     <el-input v-model="form.email" placeholder="邮箱"></el-input>
7   </el-form-item>
8   <el-form-item >
9     <el-input
10       placeholder="密码"
11       type="password"
12       v-model="form.password"
13     ></el-input>
14   </el-form-item>
15   <el-form-item >
16     <el-input
17       placeholder="确认密码"
18       type="password"
19       v-model="form.repassword"
20     ></el-input>
21   </el-form-item>
22   <el-form-item>
23     <el-button
24       :loading="loading"
25       class="login-button"
26       type="primary"
27       native-type="submit"
28       @click="submitForm"
29       block
30     >注册</el-button>
31   </el-form-item>
32 </el-form>
```

 复制代码

```
1 <script>
2 export default {
3   name: "register",
4   data() {
5     return {
6       form: {
7         username: '', // 用户名
8         password: '', // 密码
9         email: '', // 邮箱
10        repassword: '' // 重复输入密码
11      },
12      loading: false,
13    };
14  },
```

对应的 submitForm 方法，就调用第 30 节课介绍的封装了 axios 库的 request.js。我们使用 request，并且传递上面输入的 form 对象数据给后端，如果请求返回成功，就返回返

回体中的成功信息。

 复制代码

```
1  methods: {
2    submitForm: function(e) {
3      if (this.form.repassword !== this.form.password) {
4        this.$message.error("两次输入密码不一致");
5        return;
6      }
7      const that = this;
8      request({
9        url: '/user/register',
10       method: 'post',
11       data: this.form
12     }).then(function (response) {
13       const msg = response.data
14       that.$message.success(msg);
15     })
16   }
17 }
```

注册界面的开发就完成了，虽然逻辑比较简单，但也是使用了前面介绍的几个前端组件 vue、element-Ui、axios，所以如果你看源码，对这几个组件的使用有一些疑惑的话，还是要研究一下每一个前端组件。

写完前端之后，别忘记我们的 hade 模块的强大调试功能之一：可以前后端同时调试。

使用命令 `./hade dev all` 开启前后端同时调试模式：

```
~/Documents/workspace/gohade/bbs  geekbang/33  ./bbs dev all
启动后端服务: http://127.0.0.1:8072
监控文件夹: /Users/yejianfeng/Documents/workspace/gohade/bbs/app
后端服务pid: 88976
启动前端服务: http://127.0.0.1:8010
前端服务pid: 88977
代理服务启动: http://127.0.0.1:8070
[PID] 88976
app serve url: :8072

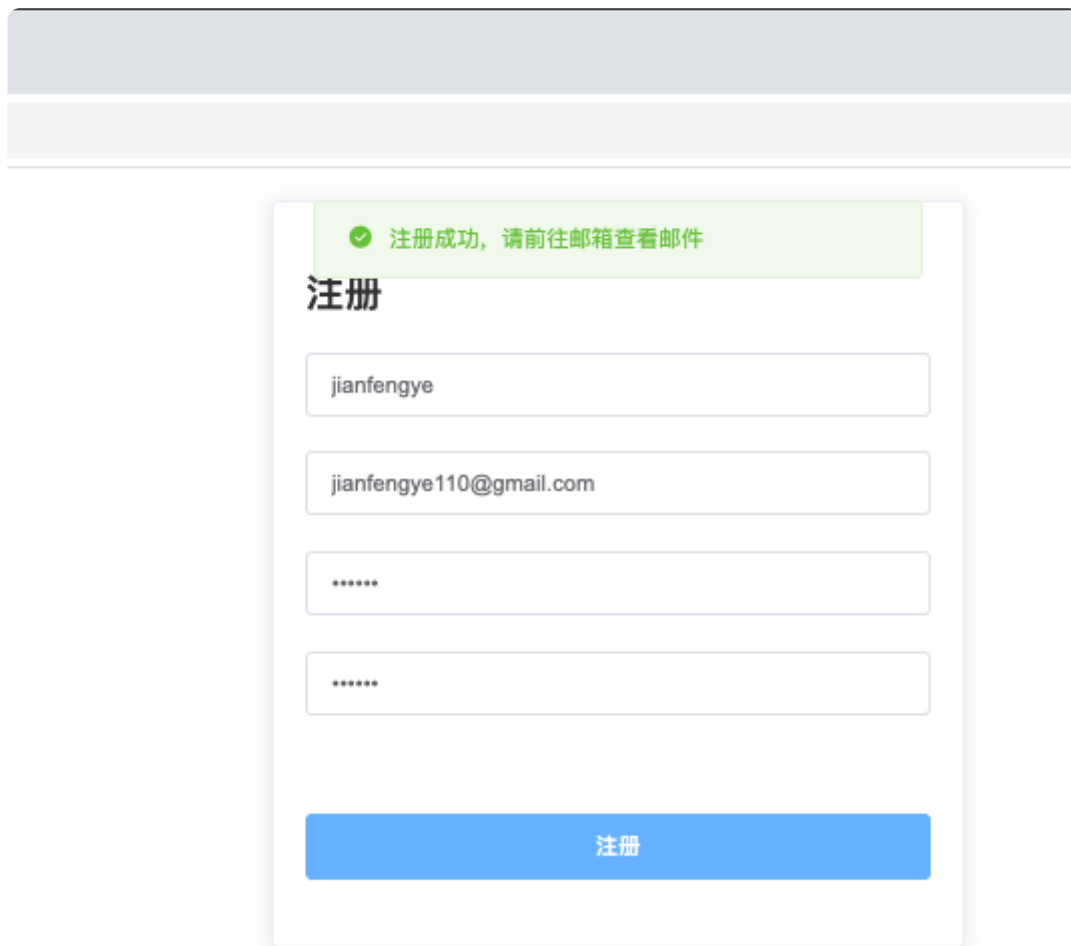
> element-starter@ dev /Users/yejianfeng/Documents/workspace/gohade/bbs
> webpack-dev-server --inline --hot --env.dev

(node:88979) [DEP0131] DeprecationWarning: The legacy HTTP parser is deprecated.
Project is running at http://127.0.0.1:8010/
webpack output is served from /assets/
404s will fallback to /assets/
{ parser: "babylon" } is deprecated; we now treat it as { parser: "babel" }.
Hash: 7327e265de0b67b6346e
Version: webpack 2.7.0
Time: 6344ms

      Asset      Size  Chunks             Chunk Names
732389ded34cb9c52dd88271f1345af9.ttf    56 kB          [emitted]
535877f50039c0cb49a6196a5b7517cd.woff  28.2 kB          [emitted]
      vendor.js   13.5 MB          0  [emitted] [big]  vendor
      index.js    3.22 kB          1  [emitted]          index
      manifest.js  30.5 kB          2  [emitted]          manifest
      index.html   362 bytes          [emitted]
chunk    {0} vendor.js (vendor) 4.99 MB {2} [initial] [rendered]
```

控制台可以看到前端和后端都已经编译运行了。然后我们通过

🔗 <http://localhost:8070/#/register> 直接看到前端页面：



注册成功, 请前往邮箱查看邮件

注册

jianfengye

jianfengye110@gmail.com

注册

如果我们发现接口或者页面有需要修改的地方，直接修改前后端的代码即可重新编译，直接调试：

```
...检测到文件更新, 重启服务开始...  
...期间请不要发送任何请求...  
编译hade成功  
启动后端服务: http://127.0.0.1:8072  
后端服务pid: 89412  
...检测到文件更新, 重启服务结束...  
[PID] 89412  
app serve url: :8072
```

这节课我们实现了用户模块的前后端的开发，代码改动量较大，已经提交到 [geekbang/32](https://github.com/geekbang/geekbang/pull/32) 分支了。欢迎比对查看。

小结

这节课我们就完完整整做好了用户模块的开发。还是再啰嗦强调一下，后端开发的四个步骤：先将接口 swagger 化、再定义用户服务协议、接着开发模块接口、最后实现用户服务协议。**服务模块的协议设计不一定能一次性抽象好，可以从服务需要提供哪些对外能力”的角度来思考，从需求出发，遇到新的需求，不断迭代你的设计就可以。**

同时关于前端开发，我们重点讲了一下如何使用 element-UI 来构建页面，以及如何使用 axios 来向后端发送请求。要掌握前后端都开发完成之后的调试方式，使用 dev all 的调试模式来同时调试前后端，这个能让你的开发速度提高不少。

思考题

对于用户服务来说，我们定义了一个 VerifyLogin 的接口，根据 token 来获取对应的 user 信息。这个你觉得应该在哪里使用？怎么使用呢？

欢迎在留言区分享你的学习笔记。感谢你的收听，如果觉得今天的内容对你有所帮助，也欢迎分享给你身边的朋友，邀请他一起学习。下节课我们实战继续。

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**



生成海报并分享

👍 赞 0

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 31 | 通用模块（上）：用户模块开发

训练营推荐

Java 学习包免费领^{NEW}

面试题答案均由大厂工程师整理

阿里、美团等
大厂真题

18 大知识点
专项练习

大厂面试
流程解析

可复用的
面试方法

面试前
要做的准备

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。