



下载APP



## 34 | 业务开发（下）：问答业务开发

2021-12-08 叶剑峰

《手把手带你写一个Web框架》

课程介绍 &gt;



讲述：叶剑峰

时长 18:38 大小 17.08M



你好，我是轩脉刃。

上节课我们已经完成了问答业务的一部分开发，主要是两部分，前后端接口设计，把接口的输出、输入以 swagger-UI 的形式表现；以及后端问答服务的接口设计，一共 14 个接口。这节课我们就继续完成问答的业务开发。

还是先划一下今天的重点，我们先使用前面定义的问答服务协议接口，来完成业务模块的接口开发，验证问答服务的协议接口是否满足需求，然后再实现我们的问答服务协议。不过因为这次问答服务实现的接口比较多，0 bug 有一定难度，所以会为问答服务写一元测试，希望你重点掌握。



最后，我们实现前端的 Vue 页面，同样，由于前端页面的编写不是课程重点，还是挑重点的实现难点解说一下。

下面开始我们今天的实战吧。

## 开发模块接口

上一节课定义好了问答服务的 14 个接口，可以使用这 14 个接口来实现业务模块了。我们的业务模块接口有七个接口需要开发：

问题创建接口 /question/create

问题列表接口 /question/list

问题详情接口 /question/detail

问题删除接口 /question/delete

更新问题接口 /question/edit

回答创建接口 /answer/create

回答删除接口 /answer/delete

这里就挑选关于问题的两个复杂一点的接口来具体说明一下，问题创建接口 /question/create、问题详情接口 /question/detail。

## 问题创建接口

问题创建接口，为 post 方法，上一节课我们已经定义了它的参数结构 questionCreateParam。那么要实现这个接口的剩余步骤就是：

解析参数

获取登录用户信息

使用登录用户创建一个问题

返回“操作成功”


解析参数我们在用户模块说过了，使用 Gorm 的 Bind 系列方法，能方便解析参数的时候并且验证参数：

 复制代码

```
1 param := &questionCreateParam{}
2 if err := c.ShouldBind(param); err != nil {
3     c.ISetStatus(400).IText(err.Error())
4     return
5 }
```

获取登录用户信息，也使用在用户模块实现的中间件。

这个中间件，我们在第 32 章的课后问题中布置给你作为课后作业，这里也说一下答案。**对于需要用户登录才能操作的接口，我们都让这些接口通过一个中间件 Auth。**在 Auth 中，验证前端传递的 cookie，从 cookie 中获取到 token，然后通过 token 去缓存中获取用户信息。

 复制代码

```
1 // AuthMiddleware 登录中间件
2 func AuthMiddleware() gin.HandlerFunc {
3     return func(c *gin.Context) {
4         envService := c.MustMake(contract.EnvKey).(contract.Env)
5         userService := c.MustMake(user.UserKey).(user.Service)
6         // 如果在调试模式下，根据参数的user_id 获取信息
7         if envService.AppEnv() == contract.EnvDevelopment {
8             userID, exist := c.DefaultQueryInt64("user_id", 0)
9             if exist {
10                 authUser, _ := userService.GetUser(c, userID)
11                 if authUser != nil {
12                     c.Set("auth_user", authUser)
13                     c.Next()
14                     return
15                 }
16             }
17         }
18
19         token, err := c.Cookie("hade_bbs")
20         if err != nil || token == "" {
21             c.ISetStatus(401).IText("请登录操作")
22             return
23         }
24
25         authUser, err := userService.VerifyLogin(c, token)
26         if err != nil || authUser == nil {
```

```
27         c.ISetStatus(401).IText("请登录后再操作")
28         return
29     }
30
31     c.Set("auth_user", authUser)
32
33     c.Next()
34 }
35 }
```

在这个中间件中，我们获取到用户信息，将用户信息存储到 Gin 的 context 中，key 名字为 auth\_user。同时为这个中间件创建一个获取这个用户信息的方法，GetAuthUser。这个方法比较简单，直接从 gin.Context 中获取认证用户信息。

为什么获取用户信息的方法，也定义在 Auth 中间件中呢？因为这样设计之后，**认证用户的所有逻辑，都放在这个 Auth 中间件中了，能保证“同类”逻辑封装在一个模块或者一个文件中**，不管是从代码优雅角度，还是查找追查问题角度都很方便，这个算是一个编程经验吧。

[复制代码](#)

```
1 // GetAuthUser 获取已经验证的用户
2 func GetAuthUser(c *gin.Context) *user.User {
3     t, exist := c.Get("auth_user")
4     if !exist {
5         return nil
6     }
7     return t.(*user.User)
8 }
```

回到我们的问题创建接口，在第二步，调用一下 Auth 中间件的 GetAuthUser 方法，就能获取到当前的登录用户了。

第三步，就是最核心的创建问题接口，我们调用用户服务的 PostQuestion 就能完成这个逻辑了。注意的是，要创建问题的 AuthorID 字段，我们使用的是上一步的登录用户 ID。

[复制代码](#)

```
1 question := &provider.Question{
2     Title:    param.Title,
3     Context:  param.Content,
4     AuthorID: user.ID, // 这里的user是我们的登录用户
```

```
5 }  
6 // 创建问题  
7 if err := qaService.PostQuestion(c, question); err != nil {  
8     c.ISetStatus(500).IText(err.Error())  
9     return  
10 }
```

最后，使用链式调用方法返回操作成功：

```
1 c.ISetOkStatus().IText("操作成功")
```

[复制代码](#)

## 问题详情接口

问题详情接口，可能是所有接口里面使用 qa 服务最多的业务接口了，我们单独拿出来梳理一下逻辑，分为六个步骤：

1. 解析参数
2. 获取问题详情
3. 加载问题作者
4. 加载问题答案
5. 问题答案加载答案作者
6. 返回数据

不过步骤多一点，实现倒不复杂，第二步到第五步，分别使用了 qaService 的 GetQuestion、QuestionLoadAuthor、QuestionLoadAnswers、AnswersLoadAuthor，代码如下：

```
1 func (api *QAApi) QuestionDetail(c *gin.Context) {  
2     qaService := c.MustMake(provider.QaKey).(provider.Service)  
3     id, exist := c.DefaultQueryInt64("id", 0)  
4     if !exist {  
5         c.ISetStatus(400).IText("参数错误")  
6         return  
7     }  
8     // 获取问题详情
```

[复制代码](#)

```
9    question, err := qaService.GetQuestion(c, id)
10    if err != nil {
11        c.ISetStatus(500).IText(err.Error())
12        return
13    }
14    // 加载问题作者
15    if err := qaService.QuestionLoadAuthor(c, question); err != nil {
16        c.ISetStatus(500).IText(err.Error())
17        return
18    }
19    // 加载所有答案
20    if err := qaService.QuestionLoadAnswers(c, question); err != nil {
21        c.ISetStatus(500).IText(err.Error())
22        return
23    }
24    // 加载答案的所有作者
25    if err := qaService.AnswersLoadAuthor(c, &question.Answers); err != nil {
26        c.ISetStatus(500).IText(err.Error())
27        return
28    }
29    // 输出转换
30    questionDTO := ConvertQuestionToDTO(question, nil)
31    c.ISetOkStatus().IJson(questionDTO)
32 }
33
```

这里就实现了两个接口，问题创建接口和问题详情接口，其他接口的具体实现可以参考 GitHub 上的 [@geekbang/34](#) 分支。

## 实现用户服务协议

简单回顾一下上节课定义的 qa 服务的 14 个后端服务协议：

 复制代码

```
1 // Service 代表qa的服务
2 type Service interface {
3
4     // GetQuestions 获取问题列表，question简化结构
5     GetQuestions(ctx context.Context, pager *Pager) ([]*Question, error)
6
7     // GetQuestion 获取某个问题详情，question简化结构
8     GetQuestion(ctx context.Context, questionID int64) (*Question, error)
9
10    // PostQuestion 上传某个问题
11    // ctx必须带操作人id
12    PostQuestion(ctx context.Context, question *Question) error
13
```

```
14 // DeleteQuestion 删除问题，同时删除对应的回答
15 // ctx必须带操作人信息
16 DeleteQuestion(ctx context.Context, questionID int64) error
17
18 // UpdateQuestion 代表更新问题，只会对比其中的context，title两个字段，其他字段不会对
19 // ctx必须带操作人
20 UpdateQuestion(ctx context.Context, question *Question) error
21
22
23 // QuestionLoadAuthor 问题加载Author字段
24 QuestionLoadAuthor(ctx context.Context, question *Question) error
25
26 // QuestionsLoadAuthor 批量加载Author字段
27 QuestionsLoadAuthor(ctx context.Context, questions []*Question) error
28
29 // QuestionLoadAnswers 单个问题加载Answers
30 QuestionLoadAnswers(ctx context.Context, question *Question) error
31
32 // QuestionsLoadAnswers 批量问题加载Answers
33 QuestionsLoadAnswers(ctx context.Context, questions []*Question) error
34
35
36 // PostAnswer 上传某个回答
37 // ctx必须带操作人信息
38 PostAnswer(ctx context.Context, answer *Answer) error
39
40 // GetAnswer 获取回答
41 GetAnswer(ctx context.Context, answerID int64) (*Answer, error)
42
43 // AnswerLoadAuthor 问题加载Author字段
44 AnswerLoadAuthor(ctx context.Context, question *Answer) error
45 // AnswersLoadAuthor 批量加载Author字段
46 AnswersLoadAuthor(ctx context.Context, questions []*Answer) error
47
48 // DeleteAnswer 删除某个回答
49 // ctx必须带操作人信息
50 DeleteAnswer(ctx context.Context, answerID int64) error
51 }
```


## 服务实现

下面我们就来实现这 14 个协议接口，有两个需要注意的函数重点说明一下，PostAnswer、AnswersLoadAuthor。

### PostAnswer

增加回答的服务 PostAnswer，这个接口的逻辑会比其他逻辑复杂一些。




 复制代码

```
1 func (q *QaService) PostAnswer(ctx context.Context, answer *Answer) error
```

可以看到，它的参数就是一个 answer 结构，但是我们仔细想想，**并不是简单将 answer 表创建一条新数据就行的，还需要做一个事情，将这个回答所归属问题的回答数 +1**。所以这里有一个查询问题。

我们可以把在 answer 表创建一条新数据的逻辑，和增加问题回答数的逻辑放在一起。这里会有多次去数据库的操作，需要将它们封成一个事务来做，否则的话，就会出现比如创建回答了，但是问题回答数没有 +1；或者两个事务都对一个问题回答数操作，出现脏数据。

所以需要使用 Gorm 的事务函数 Transaction，将这个函数内的所有数据库操作封装起来。

 复制代码

```
1 func (q *QaService) PostAnswer(ctx context.Context, answer *Answer) error {
2     if answer.QuestionID == 0 {
3         return errors.New("问题不存在")
4     }
5     // 必须使用事务
6     err := q.ormDB.WithContext(ctx).Transaction(func(tx *gorm.DB) error {
7         question := &Question{ID: answer.QuestionID}
8         // 获取问题
9         if err := tx.First(question).Error; err != nil {
10             return err
11         }
12         // 增加回答
13         if err := tx.Create(answer).Error; err != nil {
14             return err
15         }
16         // 问题回答数量+1
17         question.AnswerNum = question.AnswerNum + 1
18         if err := tx.Save(question).Error; err != nil {
19             return err
20         }
21         return nil
22     })
23     if err != nil {
24         return err
25     }
26
27     return nil
28 }
```



```
28 }
```

从上面代码可以看到，我们将查询问题、插入回答、增加问题回答数封装在一个事务中，这样一旦其中有数据库操作失败，那么整个操作都会失败，并且回滚，保证几个数据表的数据一致性。

## AnswersLoadAuthor

再看一下对多个回答加载回答作者的方法，AnswersLoadAuthor：

[复制代码](#)

```
1 func (q *QaService) AnswersLoadAuthor(ctx context.Context, answers []*Answer)
```

参数中传递了带有回答 ID 的 answers 结构，所以我们需要将 ID 全部查询出来。

也就是说需要在一个指针数组中，将某个字段查询出来，并且做成数组，怎么做？这里我使用的是第三方库 [collection](#)，我之前开源的一个作品，目前有 560+ 的 star，基于 Apache 协议开源。这个库最大的特点就是对“数组”进行一些特殊操作。

比如这里的，将一个数组指针的 ID 字段获取出来，组装成一个 int64 数组，就可以这么用：

[复制代码](#)


```
1 answerColl := collection.NewObjPointCollection(*answers)
2 ids, err := answerColl.Pluck("ID").ToInt64s()
```

先 New 一个指针数组，然后使用 Pluck，将某个字段获取出来，再使用 toInt64s 将获取出来的数组转位 []int64 数组。

回到 Answer 结构，ID 已经获取了，怎么查找对应作者呢？

还记得吗，上一节课我们已经**在 Answer 结构中，设置了 Answer 和 User 结构的 Belongs To 关系**，所以这里可以直接使用 Preload 方法，加载 Answers 中的 Author 字

段：

 复制代码

```
1 q.ormDB.WithContext(ctx).Preload("Author").Order("created_at desc").Find(answers)
```

直接使用 Preload，是不是比每个 answer foreach 来的更为方便了？

了解了这两个方法中难点的事务使用方式和 Preload 的使用方式，其他的 12 个协议接口的实现，基本上，逻辑就和这两个方法差不多了，你可以去 GitHub 上的 [geekbang/34](https://github.com/geekbang/34) 上的代码比对查看。

## 单元测试

问答服务有 14 个协议接口这么多，我们编写好这些协议接口之后，是很有可能错误的，这个时候，我们会非常希望写一下单元测试来验证一下这些服务。那么对于 hade 框架的服务，怎么编写单元测试呢？我们一起做一下。

首先，要知道 **hade 框架最核心的就是一个服务容器 container**。所以我们需要创建一个单元测试使用的 container。框架的 test/env.go 中，我写好了初始化服务容器的函数 InitBaseContainer：

 复制代码

```
1 package test
2
3 import (
4     "github.com/gohade/hade/framework"
5     "github.com/gohade/hade/framework/provider/app"
6     "github.com/gohade/hade/framework/provider/env"
7 )
8
9 const (
10     BasePath = "/Users/yejianfeng/Documents/workspace/gohade/bbs"
11 )
12
13 func InitBaseContainer() framework.Container {
14     // 初始化服务容器
15     container := framework.NewHadeContainer()
16     // 绑定App服务提供者
17     container.Bind(&app.HadeAppProvider{BaseFolder: BasePath})
18     // 后续初始化需要绑定的服务提供者...
```

```
19     container.Bind(&env.HadeTestingEnvProvider{})
20     return container
21 }
```

这个函数中，我们初始化了一个服务容器，当然这里的 BasePath 表示框架的初始化路径，在具体环境里，请替换你的项目所在的路径。

然后这个初始化服务容器先是绑定了 AppProvider，再绑定了 HadeTestingEnvProvider，其他的服务容器，就需要你在单元测试中自己绑定了。具体在单元测试文件 app/provider/qa/service\_test.go 中，我们将如下的服务绑定到这个容器中：

[复制代码](#)

```
1 func Test_QA(t *testing.T) {
2     container := test.InitBaseContainer()
3     container.Bind(&config.HadeConfigProvider{})
4     container.Bind(&log.HadeLogServiceProvider{})
5     container.Bind(&orm.GormProvider{})
6     container.Bind(&redis.RedisProvider{})
7     container.Bind(&cache.HadeCacheProvider{})
8     container.Bind(&user.UserProvider{})
}
```

包含 config、gorm、redis、cache、user 等服务。这里注意下，由于在 InitBaseContainer 中设置的 env 服务为 HadeTestingEnvProvider，这个服务会将 env 设置为 testing，所以这里 config 服务的所有配置，都会去 config/testing/ 目录下进行寻找。

我们是为 qa 服务写单元测试，qa 服务最本质使用的是数据库操作，所以如何模拟数据库操作来模拟单元测试就是绕不开的问题了。

## 模拟数据库操作

我们模拟数据库操作其实有多种方式，有的人直接创建一个测试的 MySQL 数据库，将所有操作在 MySQL 数据库中进行；也有的人使用一些第三方库，比如 [go-sqlmock](#)，将所有数据库操作都进行 mock。

但是这两种方式都有一些弊端，第一种需要单独搭建一个测试 MySQL，MySQL 搭建在哪里、使用什么镜像，又有很多需要讨论的点；第二种，则需要单独使用 sqlmock 来写一些 mock 的代码，增加了代码量。

hade 的模拟数据库操作，我们使用一个更为巧妙的方式：**使用 SQLite 驱动，并且将 SQLite 数据在内存中进行操作，来模拟 MySQL 的操作。**

因为 SQLite 数据库和 MySQL 数据库的操作基本上是一样的，我们对 SQLite 的操作能等同于对 MySQL 进行操作，而且 SQLite 还有一个非常好的功能，只要将 DSN 设置为 "file::memory:?cache=shared"，就能将 SQLite 的数据库保存在内存中。当然保存在内存中的代价就是，进程结束，这个内存也就消失了。但是这个对于跑一次的单元测试来说并没有什么影响。

那么具体怎么操作呢？

我们先把 config/testing/database.yaml 配置一下：

```
1 driver: sqlite # 连接驱动
2 dsn: "file::memory:?cache=shared"
```

[复制代码](#)

driver 代表使用 SQLite 驱动，dsn 代表在内存中创建一个数据库来提供给 ORM 进行操作。

然后在测试用例中，我们正常使用 hade 封装的 Gorm 就可以了。

```
1 ormService := container.MustMake(contract.ORMKey).(contract.ORMService)
2 db, err := ormService.GetDB(orm.WithGormConfig(func(options *contract.DBConfig
3     options.DisableForeignKeyConstraintWhenMigrating = true
4 })))
5
6 // 创建问题1
7 {
8     question1.AuthorID = user1.ID
9     err := qaService.PostQuestion(ctx, question1)
10     So(err, ShouldBeNil)
11 }
```

[复制代码](#)

```
12     question1, err = qaService.GetQuestion(ctx, question1.ID)
13     So(err, ShouldBeNil)
14     So(question1.CreatedAt, ShouldNotBeNil)
15 }
```

这样不仅省去了创建测试数据库的操作，也省去了写一些 mock 方法的代码量。

关于单元测试的断言，我们就使用一个第三方库 [goconvey](#)，这个第三方库是 SmartyStreets 公司开源的，目前有 6.8k 个 star，使用的是自有 [开源协议](#)，协议说明是允许用户使用下载的。

goconvey 是非常好用的一个单元测试的库，我现在的項目基本都是使用这个库来做单元测试的。它的好处有几个：

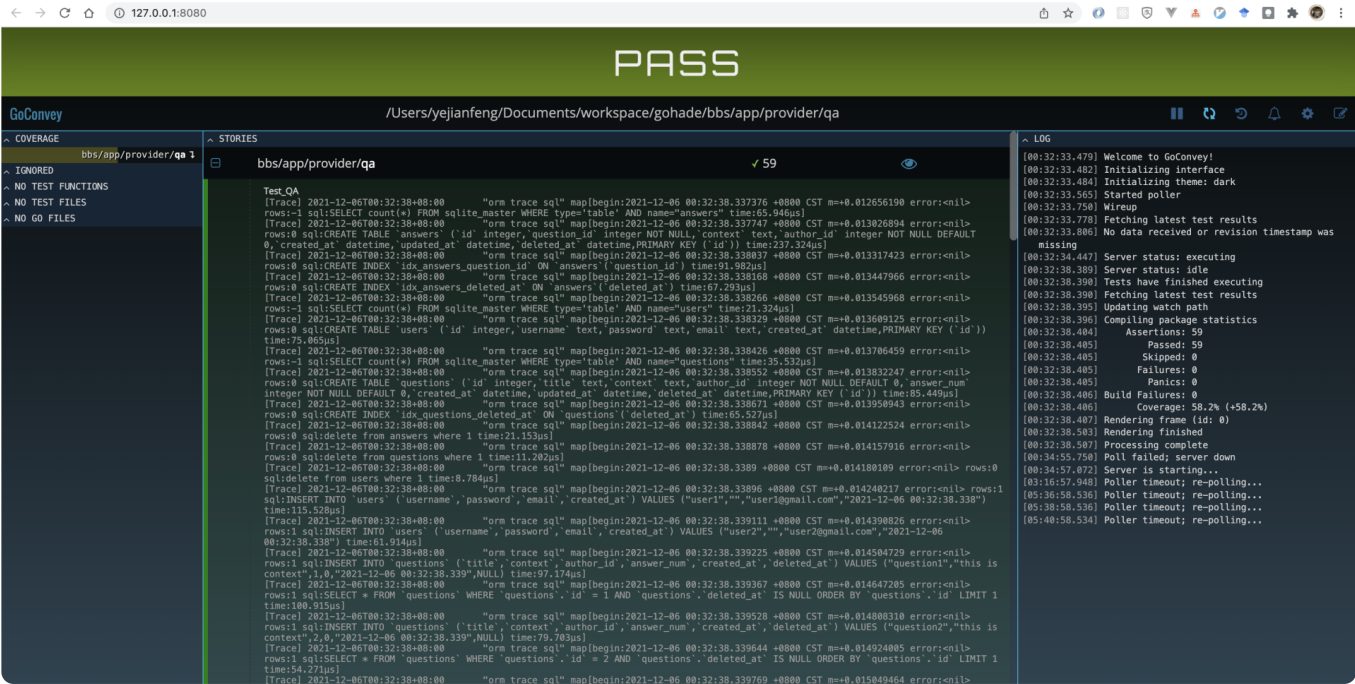
一是提供丰富的断言。比如：

```
1 So(err, ShouldBeNil)
2 So(question1.CreatedAt, ShouldNotBeNil)
3 So(q.Title, ShouldEqual, question1.Title)
```

[复制代码](#)

这种 So 系列，带上一个语义化的函数语法 ShouldNotBeNil / ShouldBeNil / ShouldEqual，能让整个断言的判断可读性更高。

其次这个库有丰富的 Web 界面。看它提供的一个可视化的工具界面，我们可以使用这个工具快速启动一个小的、简单的测试用例结果库：



在这个页面中，还有一个编写测试逻辑自动生成测试代码的功能：



上图，我们在左侧使用中文输入测试逻辑，在右侧就能生成我们的测试代码。

当然这个测试代码只有框架，没有具体的逻辑，后续只需要将这个测试代码直接拷贝进入我们的单元测试 `app/provider/qa/service_test.go` 中，然后再一个个填充其中的测试方法，就可以了：

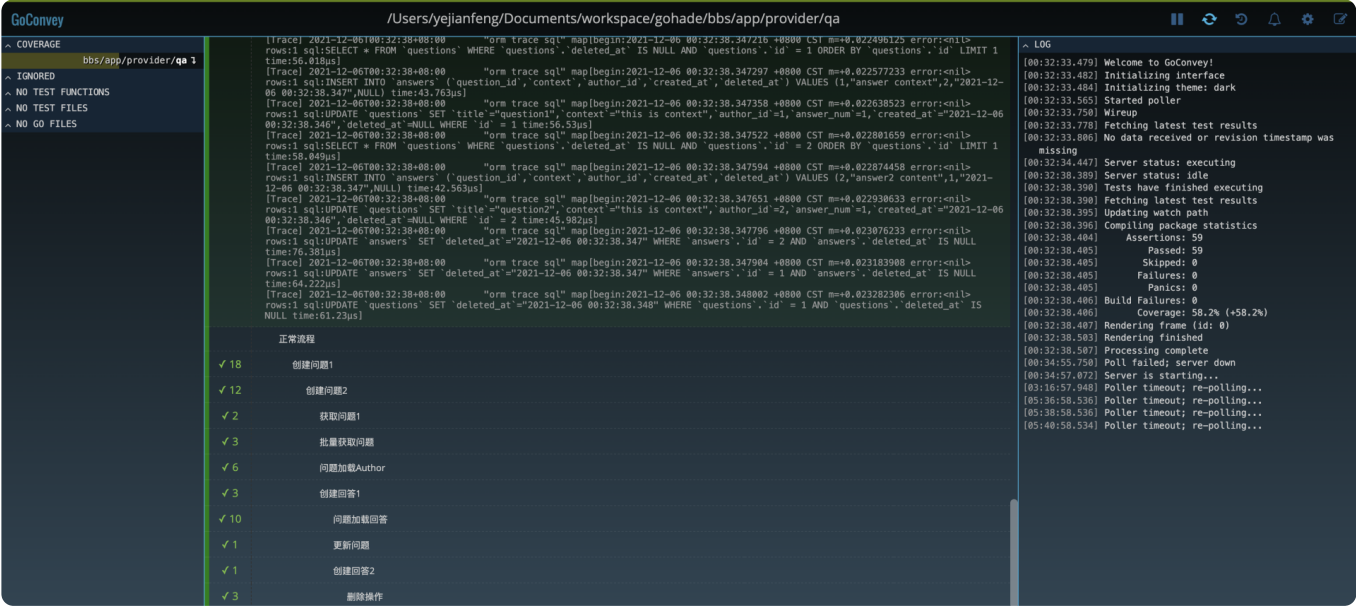
复制代码

```
1 Convey("创建问题1", func() {
2     question1 = &Question{
3         Title: "question1",
4         Context: "this is context",
```

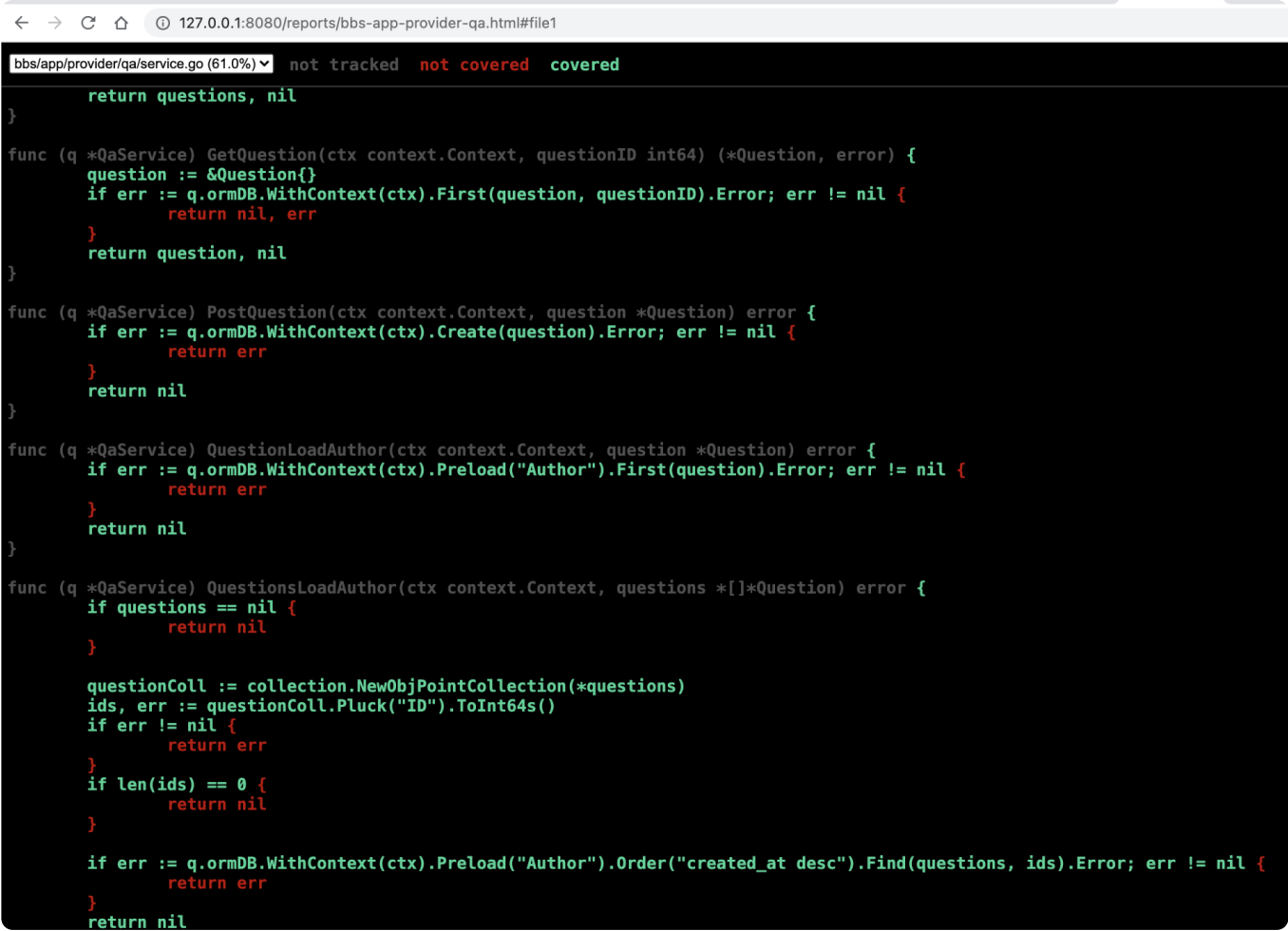
```
5     AnswerNum: 0,
6 }
7
8 question1.AuthorID = user1.ID
9 err := qaService.PostQuestion(ctx, question1)
10 So(err, ShouldBeNil)
11
12 question1, err = qaService.GetQuestion(ctx, question1.ID)
13 So(err, ShouldBeNil)
14 So(question1.CreatedAt, ShouldNotBeNil)
15
16 // 创建问题2
17 Convey("创建问题2", func() {
18     question2 = &Question{
19         Title:     "question2",
20         Context:    "this is context",
21         AnswerNum: 0,
22     }
23
24     question2.AuthorID = user2.ID
25     err := qaService.PostQuestion(ctx, question2)
26     So(err, ShouldBeNil)
27
28     question2, err = qaService.GetQuestion(ctx, question2.ID)
29     So(err, ShouldBeNil)
30
31     Convey("获取问题1", func() {
32         q, err := qaService.GetQuestion(ctx, question1.ID)
33         So(err, ShouldBeNil)
34         So(q.Title, ShouldEqual, question1.Title)
35     })
36 })
```

同时这个 Web 控制台是实时更新的，我们在编写测试用例的时候，每次保存结束之后，测试结果页面就会同步刷新：





还可以通过控制台直接查看我们的代码覆盖率：



总而言之 goconvey 是一个非常好用的第三方库，强烈推荐你在后续的项目中使用这个第三方库进行单元测试。

好我们理解了如何 mock 数据库，如何使用 goconvey，那么问答服务 14 个接口的单元测试编写逻辑就不是什么难事了，剩下的都是工作量，具体的代码就不在这里展开，可以参考 GitHub 上的这个 [🔗 service\\_test.go](#) 代码。

编写好单元测试之后，我们的后端问答服务具体实现就完成了。讲到这里，后端的四个开发步骤也就都完成了。下面我们来看下前端 Vue 开发。

## 前端 Vue 开发

前端的 Vue 开发我们要实现的业务在上一节课也梳理过了，一共 4 个页面：

问题创建页

问题列表页

问题详情页

问题更新页

我们拿比较复杂的“问题详情页”来描述一下。

1111111111

修改 | 删除

222222222222

所有回答

yejianfeng | 2021-12-06 06:22

删除

这是第二个回答

yejianfeng | 2021-12-06 06:22

删除

这是第一个回答

我来回答

H B I S | — 66 | ≡ 1≡ ☑ ...

# title1  
## title2  
### title3

title1  
title2  
title3

提交

https://time.geekbang.org/column/article/464834?utm\_source=u\_nav\_web&utm\_medium=u\_nav\_web&utm\_term=u\_nav\_web

18/24

这里其实用了富文本编辑器的两个形式，一个是富文本编辑器的编辑形式，就是最下方“我来回答”的这个输入框；另外一个富文本编辑器的查看形式，就是上方问题和所有回答的内容部分。

Vue 的组件开发中最复杂的就是这个富文本编辑器了。

富文本编辑器，我们使用第三方组件 [toast-ui-editor](#)，这个组件库提供的 viewer 模式和 editor 模式能满足我们编辑和展示的需求。

首先需要在 package.json 中引入这个组件库：

```
1 "@toast-ui/vue-editor": "^3.1.1",
```

[复制代码](#)

然后在需要的页面组件，就是这里详情页的组件中 import 引入组件库：

```
1 import { Viewer } from '@toast-ui/vue-editor';
2 import { Editor } from '@toast-ui/vue-editor';
3
4 export default {
5   components: {
6     viewer: Viewer,
7     editor: Editor,
8   },
```

[复制代码](#)


在使用 toast-ui-editor viewer 模式的地方，直接使用 viewer 标签来展示内容：

```
1 <viewer ref="answerViewer" :initialValue="answer.content" />
```

[复制代码](#)


这里的 answer.context，就是从后端接口中返回来的回答内容，这个内容是支持带有 HTML 标签的富文本的。

而在需要富文本编辑的回答框中，我们使用 editor 标签来放置一个编辑器：

 复制代码

```
1 <editor :options="editorOptions"
2     :initialValue = "answerContext"
3     initialEditType="markdown"
4     ref="toastuiEditor"
5     previewStyle="vertical" />
```


注意一下标签的几个属性。:options 属性代表这个编辑器的所有属性，它的属性值 editorOptions，我们在组件的 data 数据中进行设置，比如编辑器高度、编辑器头部的编辑功能有哪些等等：

 复制代码

```
1 editorOptions: {
2   minHeight: '200px',
3   language: 'en-US',
4   useCommandShortcut: true,
5   usageStatistics: true,
6   hideModeSwitch: true,
7   toolbarItems: [
8     ['heading', 'bold', 'italic', 'strike'],
9     ['hr', 'quote'],
10    ['ul', 'ol', 'task', 'indent', 'outdent'],
11    ['table', 'link'],
12    ['code', 'codeblock'],
13    ['scrollSync'],
14  ]
15 }
```

这些设置项都可以在 [官网](#) 查到说明。

掌握了如何使用 toast-ui-editor 作为富文本编辑器之后，前端页面的开发逻辑就并不复杂了。先使用 element-UI 搭建页面框架：

 复制代码

```
1 <template>
2   <el-row type="flex" justify="center" align="middle">
3     <el-col :span="8">
4       <el-card v-if="question" class="box-card" shadow="never">
5         ...
6         <div>
7           <viewer ref="questionViewer" :options="questionViewerOptions" :initi
8           </div>
```

```
9      </el-card>
10      ...
```

在页面加载的时候，调用 `/question/detail` 来获取后端数据：

[复制代码](#)

```
1  methods: {
2    getDetail: function (id) {
3      const that = this
4      this.id = id
5      // 调用后端接口
6      request({
7        url: "/question/detail",
8        method: 'GET',
9        params: {
10          "id": id
11        }
12      }).then(function (response) {
13        that.question = response.data;
14      })
15    },
```

在提交回答的时候，触发 `/answer/create` 来提交回答数据：

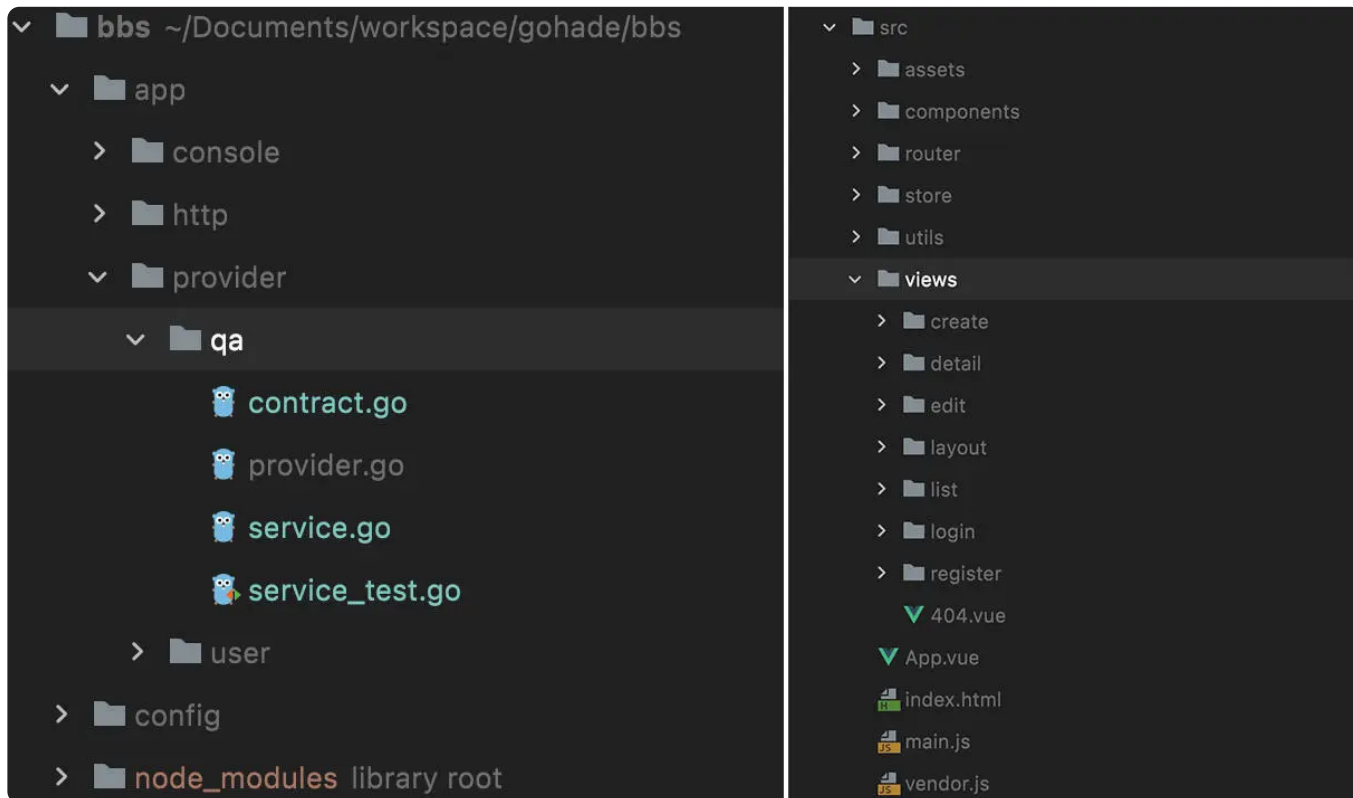
[复制代码](#)

```
1  postAnswer: function () {
2    // 获取富文本编辑器内容
3    let html = this.$refs.toastuiEditor.$data.editor.getHTML()
4    this.answerContext = html
5    const that = this
6    // 调用后端接口
7    request({
8      method: 'POST',
9      url: "/answer/create",
10     data: {
11       "question_id": that.id,
12       "context": that.answerContext,
13     },
14   }).then(function () {
15     that.$router.go(0)
16   })
17 },
```

更多代码可以参考 GitHub 上的 [@geekbang/34](#) 分支。

开发完前端和后端，别忘记使用 `./bbs dev all` 开启前后端联调模式，进行前后端联调。

这节课我们开发了问答模块的前端和后端，所有的代码都放在 [@geekbang/34](#) 分支，你可以对比查看。



## 小结

我们开发了问答模块的前端和后端，开发的流程，基本上和用户模块是一致的。只是其中有一些特殊的地方要注意掌握，比如，如何使用 Gorm 的预加载功能、如何在单元测试里面用 SQLite mock SQL 操作、如何使用容器做 hade 的单元测试、如何使用 goconvey 做测试框架、如何使用 toast-ui-editor 做富文本编辑器。这些都是我们问答模块实现的重点和难点。

到这里课程主体就要结束了。因为是 Web 框架的搭建课，所以这个课程除了专栏的文字之外，还有很大一部分在 GitHub 上，也就是代码。我们一共完成了两个项目，一个是 hade 框架项目，一个是类知乎问答网站 bbs，都是开源项目，每个项目也都保留着按章节的演进步骤和代码。



当然，专栏的文字不能穷尽所有知识点，有一些代码的实现细节，需要你自己在动手写的时候才能发现，如果有疑惑，不妨去 GitHub 上对比代码进行查看。非常欢迎你把这两个项目作为自己学习 Golang 的第一个开源项目，提交并合并你的修改。

## 思考题

看 GitHub 上的代码，bbs 项目中的 error，我使用的并不是官方的 error 库，而是 [github.com/pkg/errors](https://github.com/pkg/errors) 库。这个库，使用方式和标准的 error 库是一样的，但是它有很多额外的好处，你可以研究一下这个第三方 error 库，并且描述下它比官方 error 库好的地方有哪些么？

欢迎在留言区分享你的学习笔记。感谢你的收听，如果你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。

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 33 | 业务开发（上）：问答业务开发

下一篇 结束语 | 在语言的上升期投入最大热情，是最正确的投资

训练营推荐

# Java 学习包免费领<sup>NEW</sup>

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。