



下载APP



33 | 业务开发（上）：问答业务开发

2021-12-06 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 18:19 大小 16.78M



你好，我是轩脉刃。

上两节课我们开发了一个完整的用户模块的前后端，并且运用了 hade 框架的不少命令行工具和基础服务。这节课，我们继续开发这个类知乎问答网站的另外一个比较大的业务模块：问答业务模块。

关于问答业务模块的开发，整体的开发流程和基本的使用方式和用户模块其实差不多，说到底这两个模块都是操作数据库中对应的数据表，我们同样使用先分析需求，再实现后端接口，最后是实现前端接口的流程。



问答模块，包含问题表、回答表和之前的用户表，这三个表之间有一些关联关系，在 GORM 中，如何使用这些关联关系建模，并且封装问答服务，接着对这些问答服务的方法

提供足够的测试，是我们今天的解说重点。

页面和接口设计

还是先梳理一下问答模块页面，它包含四个页面：**问题创建页**、**问题列表页**、**问题详情页**、**问题更新页**。名称都很清晰，在问题更新页中，我们可以对某个问题进行更新修改。不过我们暂时不提供回答的修改功能，只提供回答的创建和删除功能。

问题创建页

在这个页面中，用户可以提出一个问题。提出问题的時候，让用户输入问题的标题和内容。通过点击提交，这个问题就提交进入数据库，并且在列表页面展示了。

hadecast

我要提问 | jianfengye | 登出

hade框架的初始化出现错误?

H B I S | — 66 | ☰ ☷ ☑ ☒ ☑ ☒ | ☐ ☑ | </> ...


我在按照官网的步骤执行的时候，出现下列的问题，请教，这个怎么处理：

golang.org无法访问

立即提问

取消

问题创建页明显就只会和后端有一个接口的交互，问题创建接口 `/question/create`。它是 POST 请求，请求参数包括问题标题 `title` 和问题内容 `context`。我们用一个结构来表示这个接口的请求内容：

 复制代码

```
1 type questionCreateParam struct {  
2     Title    string `json:"title" binding:"required"`  
3     Content  string `json:"content" binding:"required"`  
4 }
```

返回值为问题是否创建成功的字符串说明：“操作成功”。

问题列表页

在列表页面中，我们按照创建时间顺序展示问题列表。列表页中的每一项都代表一个问题，展示的时候列出问题的标题、问题的内容（只显示 200 个字）、问题的创建时间、问题的创建者，以及问题的回答数。



考虑到当问题数比较多的时候，一个页面展示不下，我们为列表页设计一个分页逻辑，当页面下拉到底部的时候，会有“加载中”的字样去后端获取更多的列表信息。



所以问题列表页的接口也比较简单。我们可以把这个页面开始的获取问题列表，和“加载中”功能的接口，设计为同一个：问题列表接口 `/question/list`。这个接口请求方法为 GET，参数需要设计两个，一个参数 `start` 表示要从第几个问题开始加载，而另外一个参数 `size` 表示请求的问题个数。

对于页面初始化的问题列表，`start` 为 0，`size` 为 10，表示页面初始化，我们向后端获取 10 个问题；而对于后面的“加载中”的功能，我们的 `start` 为当前页面已经展示的问题数量，`size` 同样为 10，表示再加载 10 个问题，增加到问题列表页中。

然后这个接口最终返回的是一个问题数组，包含问题的标题、问题的内容、问题创建时间、问题创建用户，以及问题的回答数。

问题详情页

到达列表页之后，用户会进入问题详情页查看某个具体的问题，但是这个页面承载的功能远不止查看问题详情这么简单。

首先因为列表页只显示 200 字，这个页面要能展示问题详情。用户要能回答这个问题，那么这个页面的最下方还要有用户回答框，如果查看人想对某个问题进行回答，可以输入回答内容进行提交。所以也需要展示这个问题的所有回答列表。

有了问题和回答的新增，我们当然要考虑删除。这个页面展示的问题如果是查看人创建的，查看人可以操作将这个问题进行删除。同时，如果回答列表中展示的某个回答是查看人创建的，查看人有权限将这个回答进行删除。

hadecast

我要提问 | jianfengye | 登出

hade框架的初始化出现错误?

修改 | 删除

我在按照官网的步骤执行的时候，出现下列的问题，请教，这个怎么处理：

golang.org无法访问

所有回答

yejianfeng | 2021-12-05 08:02

删除

可是使用

export GOPROXY=https://goproxy.io,direct

我来回答

H B I S | — 66 | :≡ 1≡ ☑ ▶◀ | ☐ 🖼 | </> ...

提交

所以问题详情页的接口就比较多，有 4 个接口。

问题详情接口 /question/detail

查看某个问题详情，并且在这个问题详情中，同时带有这个问题的所有回答，按照回答的创建时间倒序排列。

这个接口为 GET 请求，它的参数为一个 id，表示问题的 ID。返回值是问题详情，这个问题详情基本上和问题列表页中的问题是一个模型，但是还要带有一个回答列表信息，把这个问题的所有回答都返回。

回答创建接口 /answer/create

这个接口的功能是创建一个回答，它是 POST 请求，参数有两个：question_id，代表回答对应的问题 ID；content，代表回答的具体内容。我们用一个数据结构来代表这个接口的参数：

 复制代码

```
1 type answerCreateParam struct {  
2     QuestionID int64 `json:"question_id" binding:"required"`  
3     Content     string `json:"content" binding:"required"`  
4 }
```

接口的返回值是操作成功或者失败的信息。

回答删除接口 /answer/delete

这个接口功能是删除某个回答，它是 GET 请求，参数为 id，表示回答的具体 ID。当然在接口的后端逻辑中，我们必须判断这个回答是否是查看人所创建的，如果不是的话，这个接口是不允许进行操作的。接口的返回值就返回操作成功或者失败的信息即可。

问题删除接口 /question/delete

这个接口功能是删除某个问题，它是 GET 请求，参数为 id，表示问题的具体 ID，和回答的删除接口一个操作。

问题更新页

在这个页面中，用户可以对某个自己提出的问题的内容进行修改。这个页面和问题创建页有类似的页面布局，不同的是进入的时候，问题标题和内容都是有具体内容的。

hadecast

我要提问 | jianfengye | 登出

问题标题

hade框架的初始化出现错误?

问题描述

H B I | — 66 | | ...

我在按照官网的步骤执行的时候，出现下列的问题，请教，这个怎么处理：

golang.org无法访问

更新

问题更新页接口就一个，负责完成更新某个问题的功能。更新问题接口 `/question/edit`，我们允许更新问题的标题和内容，所以这个接口参数有三个：问题 ID，表示更新的哪个问题；标题 `title`，表示更新的问题标题；内容 `content`，表示要更新的问题内容。我们定义一个数据结构来表示这个接口的参数：

[复制代码](#)

```
1 type questionEditParam struct {  
2     ID      int64 `json:"id" binding:"required"`  
3     Title   string `json:"title" binding:"required"`  
4     Content string `json:"content" binding:"required"`  
5 }
```


返回值是操作成功或者失败的消息。

好最后我们梳理一下，关于问答模块，一共要开发七个接口。

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


后端开发

接口定义好，下面就是后端开发了。还记得开发用户模块的时候说过的后端开发四个步骤吗，接口 swagger 化、定义用户服务协议、开发模块接口、实现用户服务协议，这四个步骤具体负责的内容就不赘述了。今天 qa 模块的开发，我们仍然沿用这四个步骤。

接口 swagger 化

首先使用注释将前面定义的七个接口的说明、参数、返回值全部 swagger 化。

因为问题列表页面和问题详情页面，都会使用到输出“问题”和“回答”这两种结构，还记得第 31 章我们讨论的模型设计吗，DTO 层模型负责前端和后端接口的数据传输，定义了这个 DTO 层的模型，前端和后端的同学就能依照这个模型来并行开发了。所以我们设计 DTO 层的模型。

 复制代码

```
1 // QuestionDTO 问题列表返回结构
2 type QuestionDTO struct {
3     ID          int64      `json:"id,omitempty"`
4     Title       string     `json:"title,omitempty"`
5     Context     string     `json:"context,omitempty"` // 在列表页，只显示前200个字符
6     AnswerNum   int        `json:"answer_num"`
7     CreatedAt   time.Time  `json:"created_at"`
8     UpdatedAt   time.Time  `json:"updated_at"`
```

```
9     Author *user.UserDTO `json:"author,omitempty"` // 作者
10    Answers []*AnswerDTO `json:"answers,omitempty"` // 回答
11 }
12 // AnswerDTO 回答返回结构
13 type AnswerDTO struct {
14     ID          int64      `json:"id,omitempty"`
15     Content     string     `json:"content,omitempty"`
16     CreatedAt   time.Time `json:"created_at"`
17     UpdatedAt   time.Time `json:"updated_at"`
18     Author *user.UserDTO `json:"author,omitempty"` // 作者
19 }
```

我们可以看到，在 DTO 层，各个 DTO 是有关联的，QuestionDTO 关联了 UserDTO 和 AnswerDTO，而 AnswerDTO 关联了 UserDTO。**这样关联其实是非常合理的。后续我们输出给前端的数据模型就固定了**，比如要输出用户，前端就知道我们一定会输出一个 UserDTO 的数据模型，能减少前后端的沟通障碍。

然后编写接口方法并注册到路由中：

[复制代码](#)

```
1 // RegisterRoutes 注册路由
2 func RegisterRoutes(r *gin.Engine) error {
3     api := &QAApi{}
4     if !r.IsBind(qa.QaKey) {
5         r.Bind(&qa.QaProvider{})
6     }
7     questionApi := r.Group("/question", auth.AuthMiddleware())
8     {
9         // 问题列表
10        questionApi.GET("/list", api.QuestionList)
11        // 问题详情
12        questionApi.GET("/detail", api.QuestionDetail)
13        // 创建问题
14        questionApi.POST("/create", api.QuestionCreate)
15        // 删除问题
16        questionApi.POST("/delete", api.QuestionDelete)
17        // 更新问题
18        questionApi.POST("/edit", api.QuestionEdit)
19    }
20    answerApi := r.Group("/answer", auth.AuthMiddleware())
21    {
22        // 创建回答
23        answerApi.POST("/create", api.AnswerCreate)
24        // 删除回答
25        answerApi.POST("/delete", api.AnswerDelete)
26    }
27 }
```

```
28     return nil
29 }
```

最后按照 swaggo 的方式来编写 swagger 的注释，以获取问题详情的接口为例：

[复制代码](#)

```
1 // QuestionDetail 获取问题详情
2 // @Summary 获取问题详细
3 // @Description 获取问题详情，包括问题的所有回答
4 // @Accept json
5 // @Produce json
6 // @Tags qa
7 // @Param id query int true "问题id"
8 // @Success 200 QuestionDTO question "问题详情，带回答和作者"
9 // @Router /question/detail [get]
10 func (api *QAApi) QuestionDetail(c *gin.Context) {
11     ...
12 }
```

最后我们使用 `./bbs swagger gen` 生成 swagger 文件，并且编译 `./bbs build self`，编译进入 bbs 文件，最后再使用 `./bbs dev backend` 展示 swagger-UI 界面如图：

qa ▾		
POST	/answer/create	创建回答
GET	/answer/delete	删除回答
POST	/question/create	创建问题
GET	/question/delete	删除问题
GET	/question/detail	获取问题详情
POST	/question/edit	编辑问题
GET	/question/list	获取问题列表

qa 服务设计

接口 swagger 化之后，接下来就要设计 qa 服务了。关于 qa 服务，我们同样先处理模型，将 DO 层模型和 PO 层模型合并，统一使用一个数据模型来定义。

问题 / 回答模型

代表问题的模型 Question 和代表回答的模型 Answer。

[复制代码](#)

```

1 // Question 代表问题
2 type Question struct {
3     ID          int64          `gorm:"column:id;primaryKey"`
4     Title       string         `gorm:"column:title;comment:标题"`
5     Context     string         `gorm:"column:context;comment:内容"`
6     AuthorID    int64          `gorm:"column:author_id;comment:作者id;not null;de`
7     AnswerNum   int           `gorm:"column:answer_num;comment:回答数;not null;de`
8     CreatedAt   time.Time      `gorm:"column:created_at;autoCreateTime;comment:创`
9     UpdatedAt   time.Time      `gorm:"column:updated_at;autoUpdateTime;<=:false;c`
10    DeletedAt    gorm.DeletedAt `gorm:"index"`
11    Author       *user.User     `gorm:"foreignKey:AuthorID"`
12    Answers      []*Answer      `gorm:"foreignKey:QuestionID"`
13 }
14
15 // Answer 代表一个回答
16 type Answer struct {
17     ID          int64          `gorm:"column:id;primaryKey"`
18     QuestionID  int64          `gorm:"column:question_id;index;comment:问题id;not`
19     Content     string         `gorm:"column:context;comment:内容"`
20     AuthorID    int64          `gorm:"column:author_id;comment:作者id;not null;de`
21     CreatedAt   time.Time      `gorm:"column:created_at;autoCreateTime;comment:创`
22     UpdatedAt   time.Time      `gorm:"column:updated_at;autoUpdateTime;<=:false;`
23     DeletedAt   gorm.DeletedAt `gorm:"index"`
24     Author       *user.User     `gorm:"foreignKey:AuthorID"`
25     Question    *Question      `gorm:"foreignKey:QuestionID"`
26 }

```

你可以看到，我们使用了非常丰富的 Gorm 的 tag 标签。在 Gorm 的使用中，一个必须要掌握的就是 tag 标签的运用，**你的 tag 标签使用的好，就能节省很多代码量**。这是今天的重点，我们来详细说明一下。

index

在我们的数据表中，除了主键索引之外，很有可能需要建立其他某个字段的索引，比如回答模型一定少不了根据问题 ID 查询出所有的回答。那么我们需要针对问题 ID，在回答表中建立一个索引，就可以使用 index 的标签来表示这个索引。

[复制代码](#)

```

1     QuestionID int64          `gorm:"column:question_id;index;comment:问题id;not

```

not null 和 default

还有一个细节，数据库中每个字段默认都是允许为 null 的，但是我们在获取数据的时候，并不希望这个数据会为 null，比如问题表中的回答数字段，我们希望它不为空，默认为 0，就可以使用 not null 和 default 两个标签来设置。

```
1 AnswerNum int `gorm:"column:answer_num;comment:回答数;not null;de
```

复制代码

time

另外，问题表和回答表都有创建时间和更新时间，其中，创建时间我们在使用创建数据的方法 Create 时自动填充，而更新时间也希望能在更新时自动填充。一方面，这样服务调用者就能少顾虑到一些“时间”方面的逻辑，另一方面，这种“时间”的管理，我们封闭在服务内部，如果调用者逻辑错误，也不会导致这两个时间是有问题的。

所以我们使用 autoCreateTime、autoUpdateTime、< -:false 分别表示创建数据自动更新时间、更新数据自动更新时间、禁止写入。

```
1 CreatedAt time.Time `gorm:"column:created_at;autoCreateTime;comment:创建时
2 UpdatedAt time.Time `gorm:"column:updated_at;autoUpdateTime;< -:false;com
```

复制代码

DeletedAt

问题和回答的数据一定存在需要删除的行为，但是删除时，我们又不希望真正删除数据，而是希望采用软删除的方式，也就是为数据某个字段打一个标记来标记删除。

这种软删除的方式在实际业务中是有可能有需求的，比如有的问题和回答是先审批再展示出来的，我们可以先标记为软删除，审批完成之后再放出来；或者用户或者运营同学点击了删除某个问题，但是属于误操作，软删除就为恢复数据提供了可能性。

Gorm 提供了 gorm.DeletedAt 的字段类型来表示这个软删除的逻辑，所以在问题表和回答表中我们加上这个 DeletedAt 字段来标记；同时由于这个字段用来标记是否删除，所以

我们在查询的时候一定会经常使用到这个字段进行索引，对这个字段使用 index 的标签来创建一个索引也是非常必要的。

[复制代码](#)

```
1 DeletedAt gorm.DeletedAt `gorm:"index"`
```

foreignKey

最后，对于 ORM 来说，问题对象和回答对象其实是一对多的关系，它们之间其实是有外键关联的，回答对象中的 QuestionID 和问题对象的 ID 字段是关联的。

我们可以为回答表创建一个外键：

[复制代码](#)

```
1 type Answer struct {
2     ...
3     QuestionID int64           `gorm:"column:question_id;index;comment:问题id;not
4     Question    *Question      `gorm:"foreignKey:QuestionID"`
5 }
```

Answer 结构和 Question 结构是“属于关系”（[🔗 Belongs To](#)），一个回答属于一个问题，所以这里的 Question 结构，它使用了一个外键，告诉 DB，Answer 结构中的 QuestionID 字段，是我的属主的主键，根据 QuestionID 字段去查找 Question 结构。

同时相对应的，我们为问题表创建一个回答表的数组：

[复制代码](#)

```
1 type Question struct {
2     ...
3     Answers    []*Answer      `gorm:"foreignKey:QuestionID"`
4 }
```

相反的，Question 结构和 Answer 结构就属于“包含许多”（[🔗 Has Many](#)），一个问题包含许多个回答，它这里的外键 tag 标记为 QuestionID，表示，我这个问题的回答有很多，它们为 Answer 结构中 QuestionID 为主键的数据。

BelongsTo、HasMany，是 Gorm 中的关联逻辑，更多的解释和查看用法可以参考官网的 [关联](#) 部分的说明。

ORM 做这个外键约束有什么好处呢？它能让 Gorm 提供的“[预加载](#)”功能成为可能。

这个预加载的功能在实际开发过程中是非常好用的。比如现在有多个问题的数组对象 questions，想要获取每一个问题的所有回答，原本我们是需要自己再手写一个 ORM 的 SQL 查询来获取。

[复制代码](#)

```
1 questionIds := []int64{}
2 for _, question := range questions {
3     questionIds := append(questionIds, question.ID)
4 }
5
6 db.Where(map[string]interface{}{"question_id", questionIds}).Find(&answers)
```

但是一旦有了外键约束，我们就可以使用预加载的功能，一行代码直接将这些问题数组对应的回答获取回来了：

[复制代码](#)

```
1 db.Preload("Answers").Find(questions)
```

这样在获取的 questions 中，每个问题对象的 Answers 字段都带有一个回答数组了，非常方便。

分页模型

除了问题和回答两个模型，在问题列表页还会根据分页信息来获取每一页的问题列表。所以我們还需要一个分页模型 Pager，包含起始位置 Start、获取的数据个数 Size，还有一个 Total 代表一共有多少数据。

[复制代码](#)

```
1 // Pager 代表分页机制
2 type Pager struct {
3     Total int64 // 共有多少数据，只有返回值使用
4     Start int    // 起始位置
5     Size  int    // 获取的数据个数
```

```
6 }
```

协议

模型定义完成，下面我们就要来定义服务对外提供的协议接口了。qa 服务虽然接口比较多，但是它的接口逻辑却并不复杂，基本上都围绕问题、回答两个模型的增删改查进行，也就是说，我们 qa 服务对外提供的协议，基本上也就是围绕这两个对象的增删改查进行的。

首先围绕问题这个模型。

需要创建问题的接口 `PostQuestion`，直接把 `Question` 模型作为参数即可。创建完问题，我们需要获取问题，那么就要有 `GetQuestion` 接口，同时也需要有批量获取 `Question` 的接口 `GetQuestions`。创建问题结束，我们可能要修改问题，那么可以有一个修改问题的接口 `UpdateQuestion`。最后就是删除问题接口 `DeleteQuestion`。

[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，其他字段不会变
19    // ctx必须带操作人
20    UpdateQuestion(ctx context.Context, question *Question) error
21 }
```

这里我们关注一下获取问题的两个接口，`GetQuestion` 和 `GetQuestions`，它们返回的是 `Question` 模型和 `Question` 模型数组。

但是有一点要注意，在前面，我们定义的 Question 模型是带有“外键”属性的，比如问题的作者 Author、问题的回答 Answer。**这些属性，我们希望由上层业务“按需加载”。**

也就是说在服务层，获取问题和获取问题列表默认是没有作者和回答的，如果上层业务需要的话，请重新调用接口来获取。所以这里我们多出了四个接口：单个问题加载作者、多个问题加载作者、单个问题加载回答、多个问题加载回答。

[复制代码](#)

```
1 // Service 代表qa的服务
2 type Service interface {
3
4     // QuestionLoadAuthor 问题加载Author字段
5     QuestionLoadAuthor(ctx context.Context, question *Question) error
6
7     // QuestionsLoadAuthor 批量加载Author字段
8     QuestionsLoadAuthor(ctx context.Context, questions []*Question) error
9
10    // QuestionLoadAnswers 单个问题加载Answers
11    QuestionLoadAnswers(ctx context.Context, question *Question) error
12
13    // QuestionsLoadAnswers 批量问题加载Answers
14    QuestionsLoadAnswers(ctx context.Context, questions []*Question) error
15
16 }
```

在使用的时候注意一下，多个问题加载的方法中，第二个参数传递的是指向 slice 的指针 `*[]*Question`。因为我们在调用接口的时候，会重新修改这个指针指向的 slice。修改的时候是有可能变更原先 slice 地址的，所以这里使用了“指向 slice 的指针”。

再看围绕“回答”这个模型。

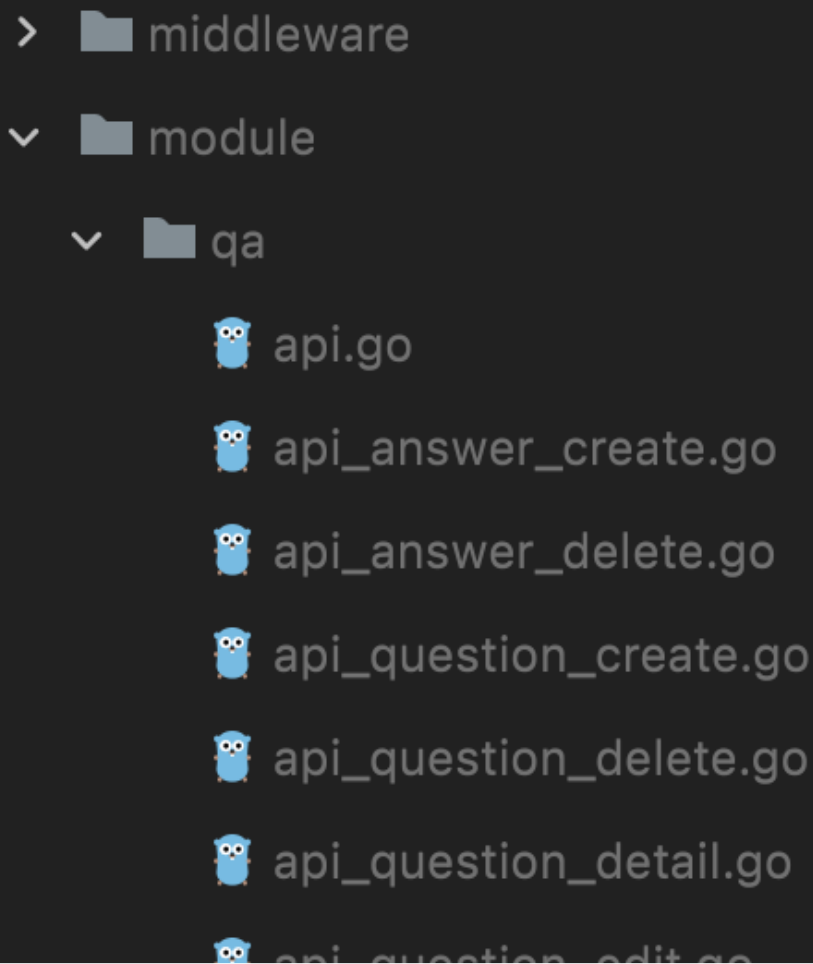
我们一样需要有创建回答接口 `PostAnswer`、删除回答接口 `DeleteAnswer`、获取回答接口 `GetAnswer`。由于产品设计上并不允许对回答进行修改，所以这里暂时不需要更新回答的接口。

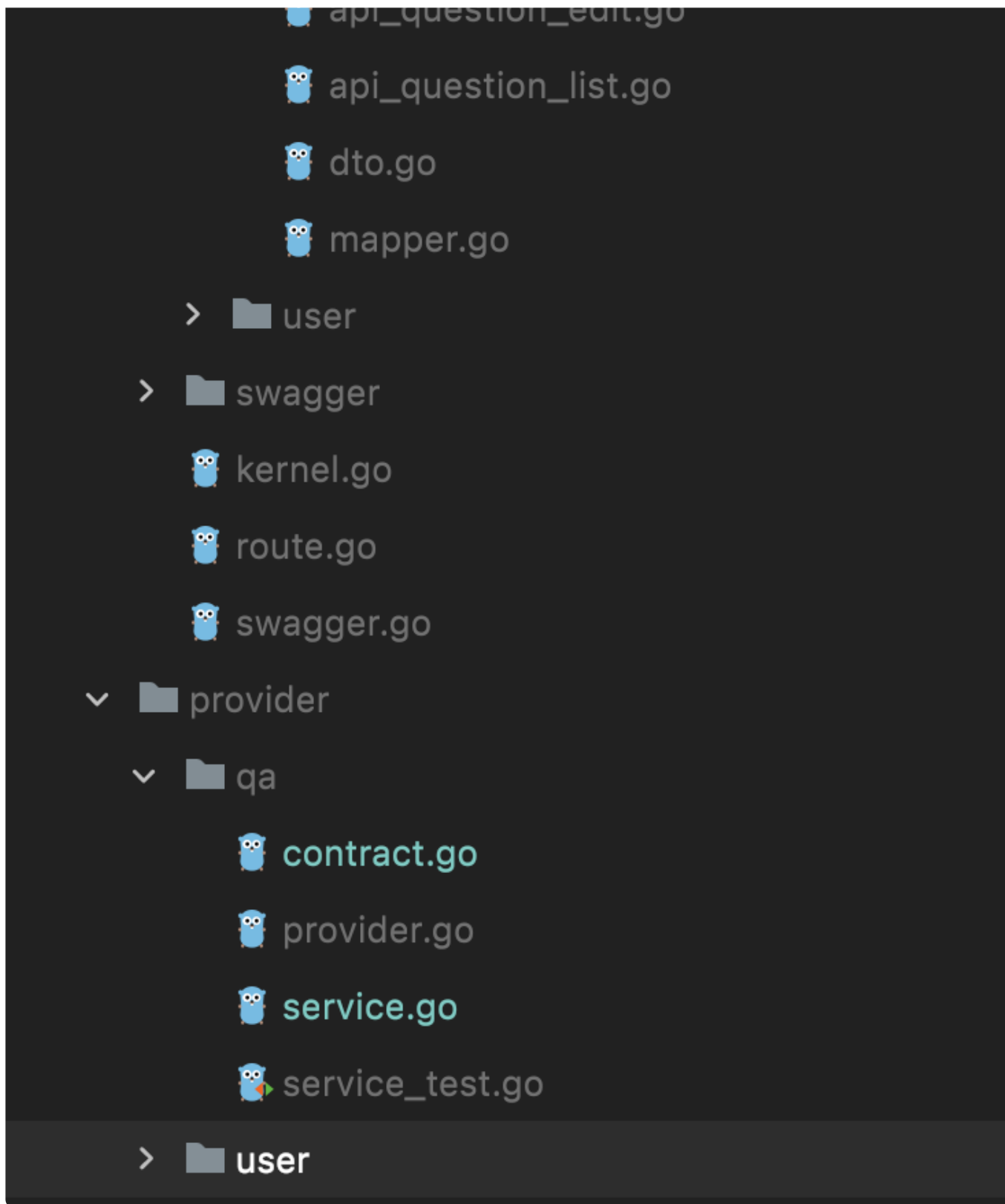
同样我们也提供“回答”作者信息的按需加载，也就是单个回答的按需加载 `AnswerLoadAuthor` 和多个回答的按需加载 `AnswersLoadAuthor` 两个方法：

[复制代码](#)

```
1 // Service 代表qa的服务
2 type Service interface {
3
4     // PostAnswer 上传某个回答
5     // ctx必须带操作人信息
6     PostAnswer(ctx context.Context, answer *Answer) error
7
8     // GetAnswer 获取回答
9     GetAnswer(ctx context.Context, answerID int64) (*Answer, error)
10
11     // AnswerLoadAuthor 问题加载Author字段
12     AnswerLoadAuthor(ctx context.Context, question *Answer) error
13     // AnswersLoadAuthor 批量加载Author字段
14     AnswersLoadAuthor(ctx context.Context, questions []*Answer) error
15
16     // DeleteAnswer 删除某个回答
17     // ctx必须带操作人信息
18     DeleteAnswer(ctx context.Context, answerID int64) error
19 }
```

好了，qa 服务的后端服务协议我们就定义完成了，一共有 14 个协议接口，代表 qa 服务对外提供的 14 种能力。所有代码都存放到 GitHub 上的 [geekbang/33](https://github.com/geekbang/33) 上了。对应的文档截图也放在这里，欢迎对比查看。





小结

今天我们主要定义了问答服务的两个协议，一个是前端和后端的协议接口，将接口的输出、输入以 swagger-UI 的形式表现，另外一个后端问答服务的协议，一共 14 个接口。

除了让你再熟悉一遍后端开发模块的四步骤之外，通过今天的实战，希望你能熟练掌握 Gorm 的模型定义，Gorm 的 tag 是个非常强大的存在，定义好了这个 tag，才能真正将之前我们引入 ORM 的利益最大化，这一点在下节课实现 qa 服务协议的时候也会领略到。

思考题

定义好 Gorm 模型的 tag，不仅仅能节省我们操作数据库的逻辑，还能根据 ORM 创建数据表，这里需要用到 Gorm 中提供的 [🔗 Auto Migrations](#) 功能。实际上，我在单元测试的时候，往测试数据库中创建表就是使用这个功能，你不妨尝试根据这节课定义的 Question 和 Answer，往自己的测试数据库中创建两张表 questions 和 answers。

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。我们下节课实战继续。

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 32 | 通用模块（下）：用户模块开发

下一篇 34 | 业务开发（下）：问答业务开发

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。