

28-控制流（上）：通过iam-apiserver设计，看Web服务的构建

你好，我是孔令飞。

前面我们讲了很多关于应用构建的内容，你一定迫不及待地想看下IAM项目的应用是如何构建的。那么接下来，我就讲解下IAM应用的源码。

在讲解过程中，我不会去讲解具体如何Code，但会讲解一些构建过程中的重点、难点，以及Code背后的设计思路、想法。我相信这是对你更有帮助的。

IAM项目有很多组件，这一讲，我先来介绍下IAM项目的门面服务：iam-apiserver（管理流服务）。我会先给你介绍下iam-apiserver的功能和使用方法，再介绍下iam-apiserver的代码实现。

iam-apiserver服务介绍

iam-apiserver是一个Web服务，通过一个名为iam-apiserver的进程，对外提供RESTful API接口，完成用户、密钥、策略三种REST资源的增删改查。接下来，我从功能和使用方法两个方面来具体介绍下。

iam-apiserver功能介绍

这里，我们可以通过iam-apiserver提供的RESTful API接口，来看下iam-apiserver具体提供的功能。iam-apiserver提供的RESTful API接口可以分为四类，具体如下：

认证相关接口



接口名称	接口功能
POST /login	用户登录
POST /logout	用户登出
POST /refresh	刷新Token

用户相关接口

接口名称	接口功能
POST /v1/users	创建用户
DELETE /v1/users	批量删除用户
DELETE /v1/users/:name	删除用户
PUT /v1/users/:name/change_password	修改用户密码
PUT /v1/users/:name	修改用户属性
GET /v1/users/:name	查询用户信息
GET /v1/users	查询用户列表

密钥相关接口

接口名称	接口功能
POST /v1/secrets	创建密钥
DELETE /v1/secrets/:name	删除密钥
PUT /v1/secrets/:name	修改密钥属性
GET /v1/secrets/:name	查询密钥信息
GET /v1/secrets	查询密钥列表

策略相关接口

接口名称	接口功能
POST /v1/policies	创建授权策略
DELETE /v1/policies	批量删除授权策略
DELETE /v1/policies/:name	删除授权策略
PUT /v1/policies/:name	修改授权策略属性
GET /v1/policies/:name	查询授权策略信息
GET /v1/policies	查询授权策略列表

iam-apiserver使用方法介绍

上面我介绍了iam-apiserver的功能，接下来就介绍下如何使用这些功能。

我们可以通过不同的客户端来访问iam-apiserver，例如前端、API调用、SDK、iamctl等。这些客户端最终都会执行HTTP请求，调用iam-apiserver提供的RESTful API接口。所以，我们首先需要有一个顺手的REST API客户端工具来执行HTTP请求，完成开发测试。

因为不同的开发者执行HTTP请求的方式、习惯不同，为了方便讲解，这里我统一通过cURL工具来执行HTTP请求。接下来先介绍下cURL工具。

标准的Linux发行版都安装了cURL工具。cURL可以很方便地完成RESTful API的调用场景，比如设置Header、指定HTTP请求方法、指定HTTP消息体、指定权限认证信息等。通过-v选项，也能输出REST请求的所有返回信息。cURL功能很强大，有很多参数，这里列出cURL工具常用的参数：

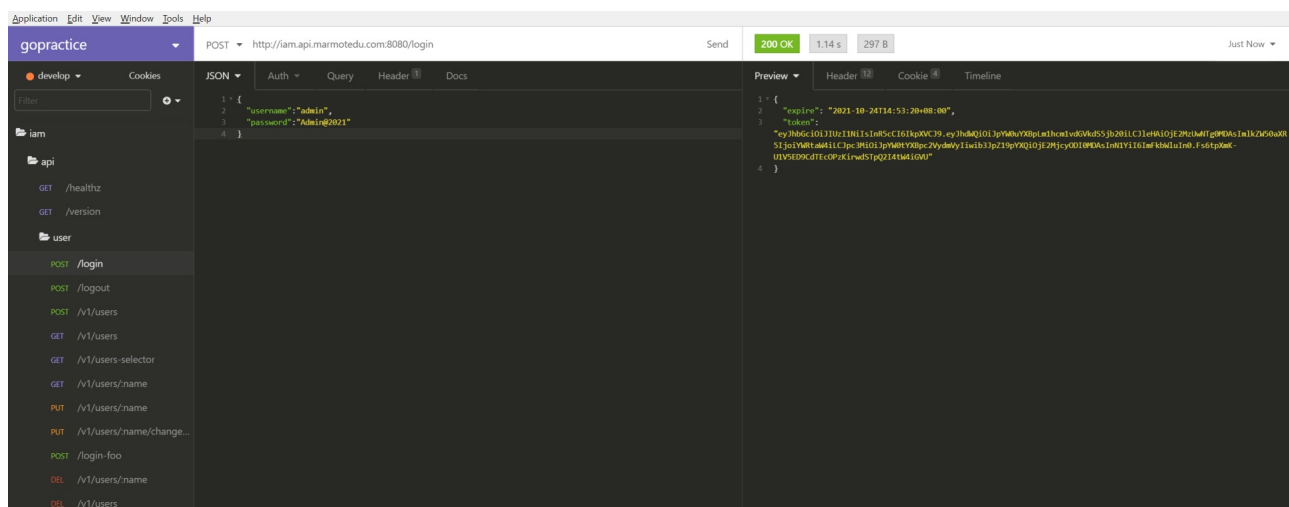
```
-X/--request [GET|POST|PUT|DELETE|...] 指定请求的 HTTP 方法
-H/--header                               指定请求的 HTTP Header
-d/--data                                  指定请求的 HTTP 消息体 (Body)
-v/--verbose                              输出详细的返回信息
-u/--user                                  指定账号、密码
-b/--cookie                               读取 cookie
```

此外，如果你想使用带UI界面的工具，这里我推荐你使用 Insomnia 。

Insomnia是一个跨平台的REST API客户端，与Postman、Apifox是一类工具，用于接口管理、测试。Insomnia功能强大，支持以下功能：

- 发送HTTP请求；
- 创建工作区或文件夹；
- 导入和导出数据；
- 导出cURL格式的HTTP请求命令；
- 支持编写swagger文档；
- 快速切换请求；
- URL编码和解码。
- ...

Insomnia界面如下图所示：



当然了，也有很多其他优秀的带UI界面的REST API客户端，例如 Postman、Apifox等，你可以根据需要自行选择。

接下来，我用对secret资源的CURD操作，来给你演示下**如何使用iam-apiserver的功能**。你需要执行6步操作。

1. 登录iam-apiserver，获取token。
2. 创建一个名为secret0的secret。
3. 获取secret0的详细信息。
4. 更新secret0的描述。
5. 获取secret列表。
6. 删除secret0。

具体操作如下：

1. 登录iam-apiserver，获取token：

```
$ curl -s -XPOST -H"Authorization: Basic `echo -n 'admin:Admin@2021'|base64`" http://127.0.0.1:8080/login |
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOiJpYW0uYXBpLm1hcm1vdGVkdS5jb20iLCJleHAiOiJlMzUwNTk4NDIsImklkZW
```

这里，为了便于使用，我们将token设置为环境变量：

```
TOKEN=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOiJpYW0uYXBpLm1hcm1vdGVkdS5jb20iLCJleHAiOiJlMzUwNTk4NDIs
```

2. 创建一个名为secret0的secret:

```
$ curl -v -XPOST -H "Content-Type: application/json" -H"Authorization: Bearer ${TOKEN}" -d'{"metadata":{"na
* About to connect() to iam.api.marmotedu.com port 8080 (#0)
*   Trying 127.0.0.1...
* Connected to iam.api.marmotedu.com (127.0.0.1) port 8080 (#0)
> POST /v1/secrets HTTP/1.1
> User-Agent: curl/7.29.0
> Host: iam.api.marmotedu.com:8080
> Accept: */*
> Content-Type: application/json
> Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOiJpYW0uYXBpLm1hcm1vdGVkdS5jb20iLCJleH
> Content-Length: 72
>
* upload completely sent off: 72 out of 72 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< X-Request-Id: ff825bea-53de-4020-8e68-4e87574bd1ba
< Date: Mon, 26 Jul 2021 07:20:26 GMT
< Content-Length: 313
<
* Connection #0 to host iam.api.marmotedu.com left intact
{"metadata":{"id":60,"instanceID":"secret-jedr3e","name":"secret0","createdAt":"2021-07-26T15:20:26.885+08:
```

可以看到，请求返回头中返回了X-Request-IdHeader，X-Request-Id唯一标识这次请求。如果这次请求失败，就可以将X-Request-Id提供给运维或者开发，通过X-Request-Id定位出失败的请求，进行排障。另外X-Request-Id在微服务场景中，也可以透传给其他服务，从而实现请求调用链。

3. 获取secret0的详细信息:

```
$ curl -XGET -H"Authorization: Bearer ${TOKEN}" http://iam.api.marmotedu.com:8080/v1/secrets/secret0
{"metadata":{"id":60,"instanceID":"secret-jedr3e","name":"secret0","createdAt":"2021-07-26T15:20:26+08:00",
```

4. 更新secret0的描述:

```
$ curl -XPUT -H"Authorization: Bearer ${TOKEN}" -d'{"metadata":{"name":"secret"},"expires":0,"description":
{"metadata":{"id":60,"instanceID":"secret-jedr3e","name":"secret0","createdAt":"2021-07-26T15:20:26+08:00",
```

5. 获取secret列表:

```
$ curl -XGET -H"Authorization: Bearer ${TOKEN}" http://iam.api.marmotedu.com:8080/v1/secrets  
{ "totalCount":1, "items":[{"metadata":{"id":60, "instanceID":"secret-jedr3e", "name":"secret0", "createdAt":"20
```

6. 删除secret0:

```
$ curl -XDELETE -H"Authorization: Bearer ${TOKEN}" http://iam.api.marmotedu.com:8080/v1/secrets/secret0  
null
```

上面，我给你演示了密钥的使用方法。用户和策略资源类型的使用方法跟密钥类似。详细的使用方法你可以参考[install.sh](#)脚本，该脚本是用来测试IAM应用的，里面包含了各个接口的请求方法。

这里，我还想顺便介绍下**如何测试IAM应用中的各个部分**。确保iam-apiserver、iam-authz-server、iam-pump等服务正常运行后，进入到IAM项目的根目录，执行以下命令：

```
$ ./scripts/install/test.sh iam::test::test # 测试整个IAM应用是否正常运行  
$ ./scripts/install/test.sh iam::test::login # 测试登陆接口是否可以正常访问  
$ ./scripts/install/test.sh iam::test::user # 测试用户接口是否可以正常访问  
$ ./scripts/install/test.sh iam::test::secret # 测试密钥接口是否可以正常访问  
$ ./scripts/install/test.sh iam::test::policy # 测试策略接口是否可以正常访问  
$ ./scripts/install/test.sh iam::test::apiserver # 测试iam-apiserver服务是否正常运行  
$ ./scripts/install/test.sh iam::test::authz # 测试authz接口是否可以正常访问  
$ ./scripts/install/test.sh iam::test::authzserver # 测试iam-authz-server服务是否正常运行  
$ ./scripts/install/test.sh iam::test::pump # 测试iam-pump是否正常运行  
$ ./scripts/install/test.sh iam::test::iamctl # 测试iamctl工具是否可以正常使用  
$ ./scripts/install/test.sh iam::test::man # 测试man文件是否正确安装
```

所以，每次发布完iam-apiserver后，你可以执行以下命令来完成iam-apiserver的冒烟测试：

```
$ export IAM_APISERVER_HOST=127.0.0.1 # iam-apiserver部署服务器的IP地址  
$ export IAM_APISERVER_INSECURE_BIND_PORT=8080 # iam-apiserver HTTP服务的监听端口  
$ ./scripts/install/test.sh iam::test::apiserver
```

iam-apiserver代码实现

上面，我介绍了iam-apiserver的功能和使用方法，这里我们再来看下iam-apiserver具体的代码实现。我会从配置处理、启动流程、请求处理流程、代码架构4个方面来讲解。

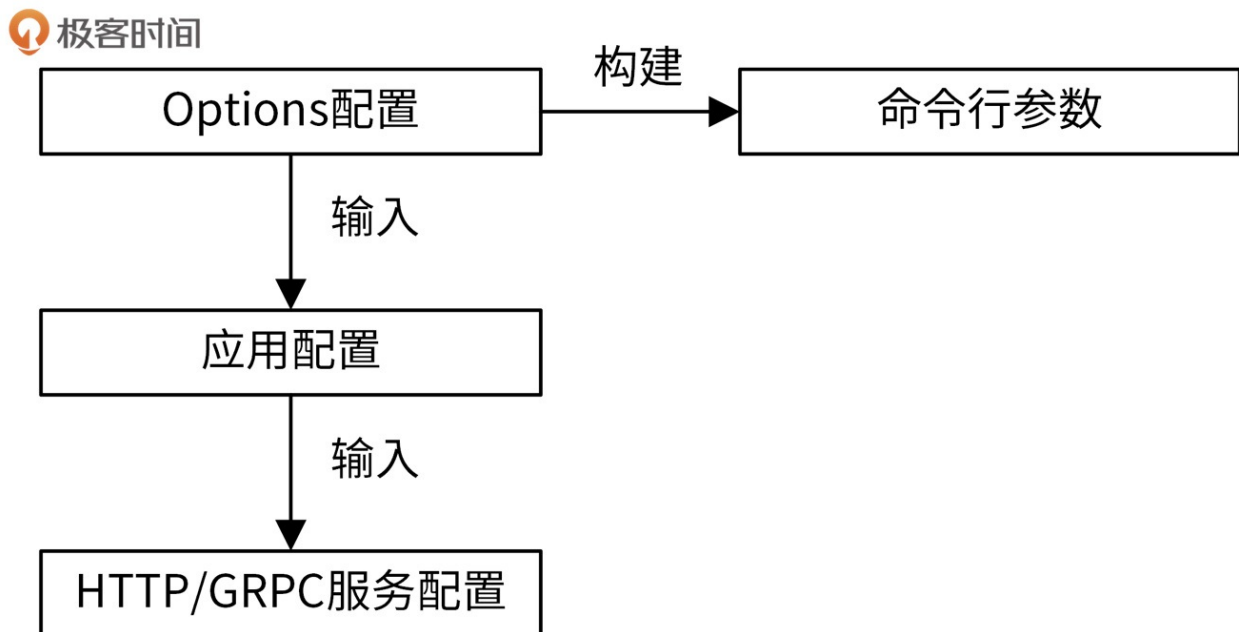
iam-apiserver配置处理

iam-apiserver服务的main函数位于[apiserver.go](#)文件中，你可以跟读代码，了解iam-apiserver的代码实现。这里，我来介绍下iam-apiserver服务的一些设计思想。

首先，来看下iam-apiserver中的3种配置：Options配置、应用配置和 HTTP/GRPC服务配置。

- **Options配置**：用来构建命令行参数，它的值来自于命令行选项或者配置文件（也可能是二者Merge后的配置）。Options可以用来构建应用框架，Options配置也是应用配置的输入。
- **应用配置**：iam-apiserver组件中需要的一切配置。有很多地方需要配置，例如，启动HTTP/GRPC需要配置监听地址和端口，初始化数据库需要配置数据库地址、用户名、密码等。
- **HTTP/GRPC服务配置**：启动HTTP服务或者GRPC服务需要的配置。

这三种配置的关系如下图：



Options配置接管命令行选项，应用配置接管整个应用的配置，HTTP/GRPC服务配置接管跟HTTP/GRPC服务相关的配置。这3种配置独立开来，可以解耦命令行选项、应用和应用内的服务，使得这3个部分可以独立扩展，又不相互影响。

iam-apiserver根据Options配置来构建命令行参数和应用配置。

我们通过github.com/marmotedu/iam/pkg/app包的[buildCommand](#)方法来构建命令行参数。这里的核心是，通过[NewApp](#)函数构建Application实例时，传入的[Options](#)实现了Flags() (fss cliflag.NamedFlagSets)方法，通过buildCommand方法中的以下代码，将option的Flag添加到cobra实例的FlagSet中：

```
if a.options != nil {
    namedFlagSets = a.options.Flags()
    fs := cmd.Flags()
    for _, f := range namedFlagSets.FlagSets {
        fs.AddFlagSet(f)
    }

    ...
}
```

通过[CreateConfigFromOptions](#)函数来构建应用配置：

```
cfg, err := config.CreateConfigFromOptions(opts)
if err != nil {
    return err
}
```

根据应用配置来构建HTTP/GRPC服务配置。例如，以下代码根据应用配置，构建了HTTP服务器的Address参数：

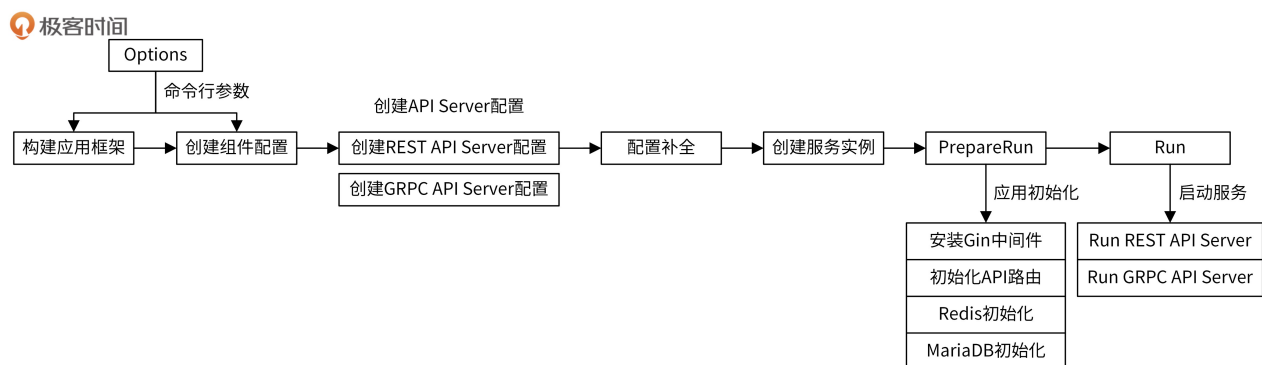
```
func (s *InsecureServingOptions) ApplyTo(c *server.Config) error {
    c.InsecureServing = &server.InsecureServingInfo{
        Address: net.JoinHostPort(s.BindAddress, strconv.Itoa(s.BindPort)),
    }

    return nil
}
```

其中，c *server.Config是HTTP服务器的配置，s *InsecureServingOptions是应用配置。

iam-apiserver启动流程设计

接下来，我们来详细看下iam-apiserver的启动流程设计。启动流程如下图所示：



首先，通过opts := options.NewOptions()创建带有默认值的Options类型变量opts。opts变量作为github.com/marmotedu/iam/pkg/app包的NewApp函数的输入参数，最终在App框架中，被来自于命令行参数或配置文件的配置（也可能是二者Merge后的配置）所填充，opts变量中各个字段的值会用来创建应用配置。

接着，会注册run函数到App框架中。run函数是iam-apiserver的启动函数，里面封装了我们自定义的启动逻辑。run函数中，首先会初始化日志包，这样我们就可以根据需要，在后面的代码中随时记录日志了。

然后，会创建应用配置。应用配置和Options配置其实是完全独立的，二者可能完全不同，但在iam-apiserver中，二者配置项是相同的。

之后，根据应用配置，创建HTTP/GRPC服务器所使用的配置。在创建配置后，会先分别进行配置补全，再使用补全后的配置创建Web服务实例，例如：

```
genericServer, err := genericConfig.Complete().New()
if err != nil {
    return nil, err
}
extraServer, err := extraConfig.complete().New()
if err != nil {
    return nil, err
}
...
func (c *ExtraConfig) complete() *completedExtraConfig {
    if c.Addr == "" {
        c.Addr = "127.0.0.1:8081"
    }

    return &completedExtraConfig{c}
}
```

上面的代码中，首先调用Complete/complete函数补全配置，再基于补全后的配置，New一个HTTP/GRPC服务实例。

这里有个设计技巧：complete函数返回的是一个*completedExtraConfig类型的实例，在创建GRPC实例时，是调用completedExtraConfig结构体提供的New方法，这种设计方法可以确保我们创建的GRPC实例一定是基于complete之后的配置（completed）。

在实际的Go项目开发中，我们需要提供一种机制来处理或补全配置，这在Go项目开发中是一个非常有用的步骤。

最后，调用PrepareRun方法，进行HTTP/GRPC服务器启动前的准备。在准备函数中，我们可以做各种初始化操作，例如初始化数据库，安装业务相关的Gin中间件、RESTful API路由等。

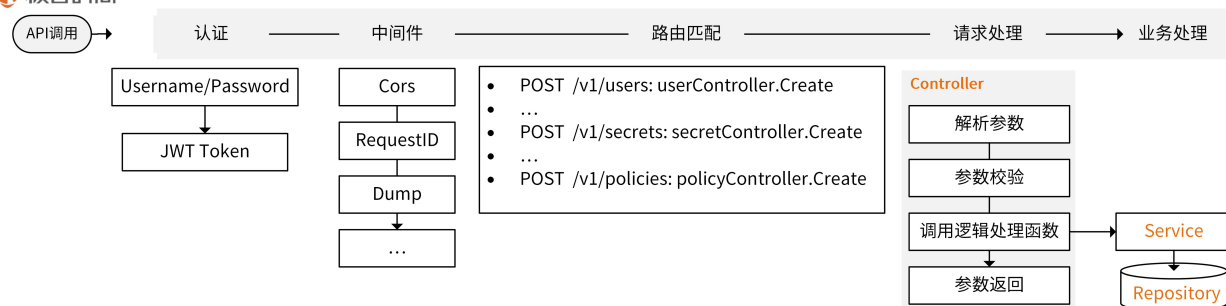
完成HTTP/GRPC服务器启动前的准备之后，调用Run方法启动HTTP/GRPC服务。在Run方法中，分别启动了GRPC和HTTP服务。

可以看到，整个iam-apiserver的软件框架是比较清晰的。

服务启动后，就可以处理请求了。所以接下来，我们再来看下iam-apiserver的RESTAPI请求处理流程。

iam-apiserver 的REST API请求处理流程

iam-apiserver的请求处理流程也是清晰、规范的，具体流程如下图所示：



结合上面这张图，我们来看下iam-apiserver的REST API请求处理流程，来帮你更好地理解iam-apiserver是如何处理HTTP请求的。

首先，我们通过API调用（<HTTP Method> + <HTTP Request Path>）请求iam-apiserver提供的RESTful API接口。

接着，Gin Web框架接收到HTTP请求之后，会通过认证中间件完成请求的认证，iam-apiserver提供了Basic认证和Bearer认证两种认证方式。

认证通过后，请求会被我们加载的一系列中间件所处理，例如跨域、RequestID、Dump等中间件。

最后，根据<HTTP Method> + <HTTP Request Path>进行路由匹配。

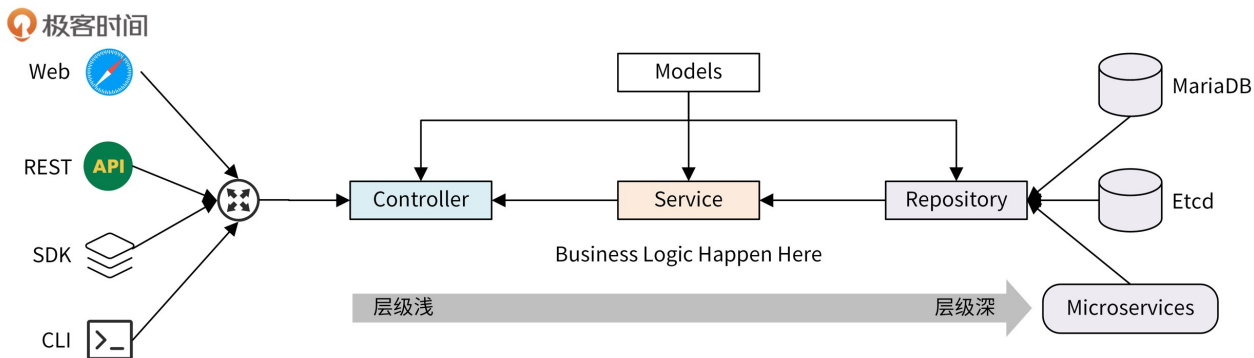
举个例子，假设我们请求的RESTful API是POST + /v1/secrets，Gin Web框架会根据HTTP Method和HTTP Request Path，查找注册的Controllers，最终匹配到secretController.CreateController。在CreateController中，我们会依次执行请求参数解析、请求参数校验、调用业务层的方法创建Secret、处理业务层的返回结果，最后返回最终的HTTP请求结果。

iam-apiserver代码架构

iam-apiserver代码设计遵循简洁架构设计，一个简洁架构具有以下5个特性：

- **独立于框架**：该架构不会依赖于某些功能强大的软件库存在。这可以让你使用这样的框架作为工具，而不是让你的系统陷入到框架的约束中。
- **可测试性**：业务规则可以在没有UI、数据库、Web服务或其他外部元素的情况下进行测试，在实际的开发中，我们通过Mock来解耦这些依赖。
- **独立于UI**：在无需改变系统其他部分的情况下，UI可以轻松地改变。例如，在没有改变业务规则的情况下，Web UI可以替换为控制台UI。
- **独立于数据库**：你可以用Mongo、Oracle、Etcd或者其他数据库来替换MariaDB，你的业务规则不要绑定到数据库。
- **独立于外部媒介**：实际上，你的业务规则可以简单到根本不去了解外部世界。

所以，基于这些约束，每一层都必须是独立的和可测试的。iam-apiserver代码架构分为4层：模型层（Models）、控制层（Controller）、业务层（Service）、仓库层（Repository）。从控制层、业务层到仓库层，从左到右层级依次加深。模型层独立于其他层，可供其他层引用。如下图所示：



层与层之间导入包时，都有严格的导入关系，这可以防止包的循环导入问题。导入关系如下：

- 模型层的包可以被仓库层、业务层和控制层导入；
- 控制层能够导入业务层和仓库层的包。这里需要注意，如果没有特殊需求，控制层要避免导入仓库层的包，控制层需要完成的业务功能都通过业务层来完成。这样可以使代码逻辑更加清晰、规范。
- 业务层能够导入仓库层的包。

接下来，我们就来详细看下每一层所完成的功能，以及其中的一些注意点。

1. 模型层 (Models)

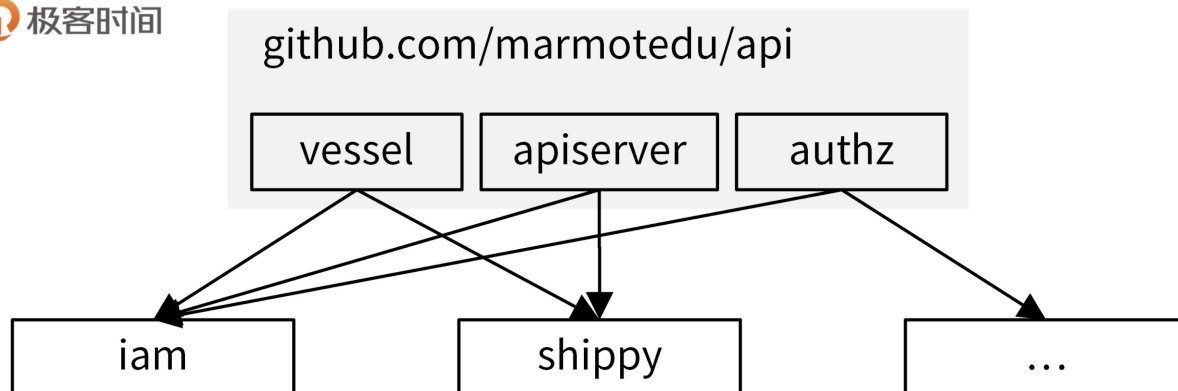
模型层在有些软件架构中也叫做实体层 (Entities)，模型会在每一层中使用，在这一层中存储对象的结构和它的方法。IAM项目模型层中的模型存放在github.com/marmotedu/api/apiserver/v1目录下，定义了 User、UserList、Secret、SecretList、Policy、PolicyList、AuthzPolicy 模型及其方法。例如：

```
type Secret struct {
    // May add TypeMeta in the future.
    // metav1.TypeMeta `json:",inline"`

    // Standard object's metadata.
    metav1.ObjectMeta `json:"metadata,omitempty"`
    Username          string `json:"username"          gorm:"column:username" validate:"omitempty"`
    SecretID           string `json:"secretID"           gorm:"column:secretID" validate:"omitempty"`
    SecretKey          string `json:"secretKey"          gorm:"column:secretKey" validate:"omitempty"`

    // Required: true
    Expires    int64 `json:"expires"    gorm:"column:expires"    validate:"omitempty"`
    Description string `json:"description" gorm:"column:description" validate:"description"`
}
```

之所以将模型层的模型存放在 github.com/marmotedu/api 项目中，而不是 github.com/marmotedu/iam 项目中，是为了让这些模型能够被其他项目使用。例如，iam 的模型可以被 github.com/marmotedu/shippy 应用导入。同样，shippy 应用的模型也可以被 iam 项目导入，导入关系如下图所示：



上面的依赖关系都是单向的，依赖关系清晰，不存在循环依赖的情况。

要增加shippy的模型定义，只需要在api目录下创建新的目录即可。例如，shippy应用中有一个vessel服务，其模型所在的包可以为github.com/marmotedu/api/vessel。

另外，这里的模型既可以作为数据库模型，又可以作为API接口的请求模型（入参、出参）。如果我们能够确保**创建资源时的属性**、**资源保存在数据库中的属性**、**返回资源的属性**三者一致，就可以使用同一个模型。通过使用同一个模型，可以使我们的代码更加简洁、易维护，并能提高开发效率。如果这三个属性有差异，你可以另外新建模型来适配。

2. 仓库层 (Repository)

仓库层用来跟数据库/第三方服务进行CURD交互，作为应用程序的数据引擎进行应用数据的输入和输出。这里需要注意，仓库层仅对数据库/第三方服务执行CRUD操作，不封装任何业务逻辑。

仓库层也负责选择应用中将要使用什么样的数据库，可以是MySQL、MongoDB、MariaDB、Etcd等。无论使用哪种数据库，都要在这层决定。仓库层依赖于连接数据库或其他第三方服务（如果存在的话）。

这一层也会起到数据转换的作用：将从数据库/微服务中获取的数据转换为控制层、业务层能识别的数据结构，将控制层、业务层的数据格式转换为数据库或微服务能识别的数据格式。

iam-apiserver的仓库层位于[internal/apiserver/store/mysql](#)目录下，里面的方法用来跟MariaDB进行交互，完成CURD操作，例如，从数据库中获取密钥：

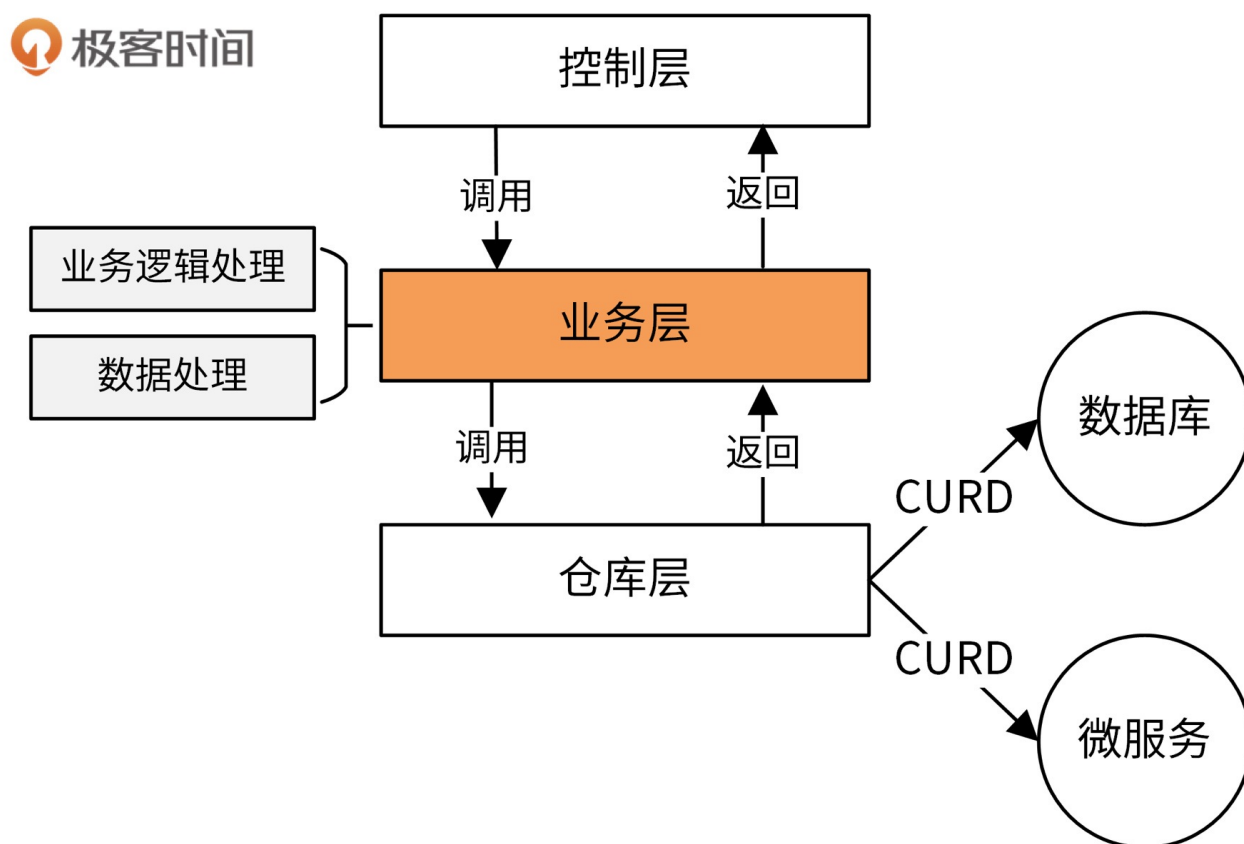
```

func (s *secrets) Get(ctx context.Context, username, name string, opts metav1.GetOptions) (*v1.Secret, error) {
    secret := &v1.Secret{}
    err := s.db.Where("username = ? and name= ?", username, name).First(&secret).Error
    if err != nil {
        if errors.Is(err, gorm.ErrRecordNotFound) {
            return nil, errors.WithCode(code.ErrSecretNotFound, err.Error())
        }
        return nil, errors.WithCode(code.ErrDatabase, err.Error())
    }
    return secret, nil
}

```

3. 业务层 (Service)

业务层主要用来完成业务逻辑处理，我们可以把所有的业务逻辑处理代码放在业务层。业务层会处理来自控制层的请求，并根据需要请求仓库层完成数据的CURD操作。业务层功能如下图所示：



iam-apiserver的业务层位于[internal/apiserver/service](#)目录下。下面是iam-apiserver业务层中，用来创建密钥的函数：

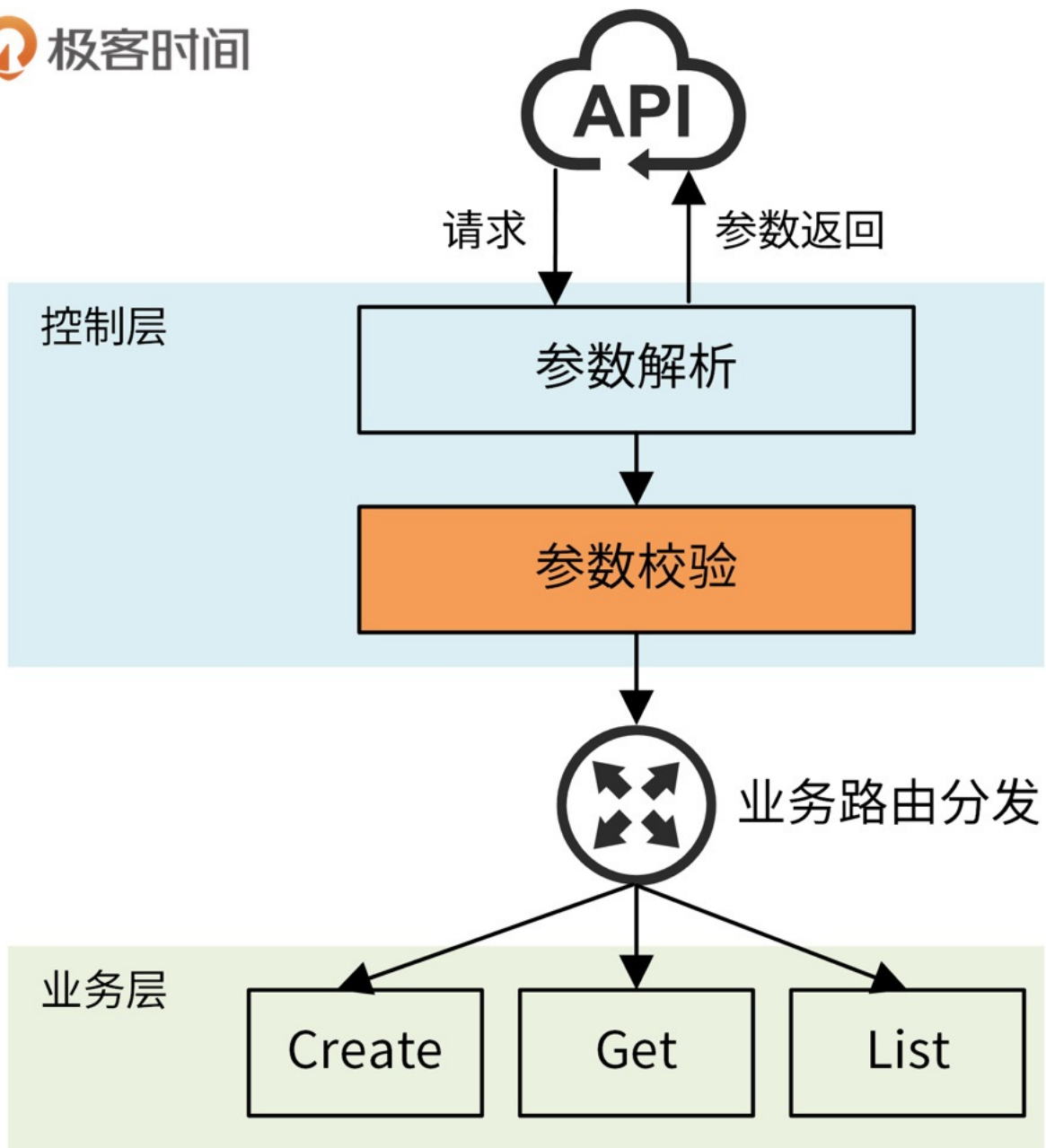
```
func (s *secretService) Create(ctx context.Context, secret *v1.Secret, opts metav1.CreateOptions) error {
    if err := s.store.Secrets().Create(ctx, secret, opts); err != nil {
        return errors.WithCode(code.ErrDatabase, err.Error())
    }

    return nil
}
```

可以看到，业务层最终请求仓库层的s.store的Create方法，将密钥信息保存在MariaDB数据库中。

4. 控制层 (Controller)

控制层接收HTTP请求，并进行参数解析、参数校验、逻辑分发处理、请求返回这些操作。控制层会将逻辑分发给业务层，业务层处理后返回，返回数据在控制层中被整合再加工，最终返回给请求方。控制层相当于实现了业务路由的功能。具体流程如下图所示：



这里我有个建议，不要在控制层写复杂的代码，如果需要，请将这些代码分发到业务层或其他包中。

iam-apiserver的控制层位于[internal/apiserver/controller](#)目录下。下面是iam-apiserver控制层中创建密钥的代码：

```
func (s *SecretHandler) Create(c *gin.Context) {
    log.L(c).Info("create secret function called.")

    var r v1.Secret

    if err := c.ShouldBindJSON(&r); err != nil {
        core.WriteResponse(c, errors.WithCode(code.ErrBind, err.Error()), nil)

        return
    }

    if errs := r.Validate(); len(errs) != 0 {
        core.WriteResponse(c, errors.WithCode(code.ErrValidation, errs.ToAggregate().Error()), nil)
    }
}
```

```

    return
}

username := c.GetString(middleware.UsernameKey)

secrets, err := s.srv.Secrets().List(c, username, metav1.ListOptions{
    Offset: pointer.ToInt64(0),
    Limit:  pointer.ToInt64(-1),
})
if err != nil {
    core.WriteResponse(c, errors.WithCode(code.ErrDatabase, err.Error()), nil)

    return
}

if secrets.TotalCount >= maxSecretCount {
    core.WriteResponse(c, errors.WithCode(code.ErrReachMaxCount, "secret count: %d", secrets.TotalCount), nil)

    return
}

// must reassign username
r.Username = username

if err := s.srv.Secrets().Create(c, &r, metav1.CreateOptions{}); err != nil {
    core.WriteResponse(c, err, nil)

    return
}

core.WriteResponse(c, nil, r)
}

```

上面的代码完成了以下操作：

1. 解析HTTP请求参数。
2. 进行参数验证，这里可以添加一些业务性质的参数校验，例如：`secrets.TotalCount >= maxSecretCount`。
3. 调用业务层`s.srv`的`Create`方法，完成密钥的创建。
4. 返回HTTP请求参数。

上面，我们介绍了iam-apiserver采用的4层结构，接下来我们再看看**每一层之间是如何通信的**。

除了模型层，控制层、业务层、仓库层之间都是通过接口进行通信的。通过接口通信，一方面可以使相同的功能支持不同的实现（也就是说具有插件化能力），另一方面也使得每一层的代码变得可测试。

这里，我用创建密钥API请求的例子，来给你讲解下层与层之间是如何进行通信的。

首先，来看下控制层如何跟业务层进行通信。

对密钥的请求处理都是通过`SecretController`提供的方法来处理的，创建密钥调用的是它的`Create`方法：

```

func (s *SecretController) Create(c *gin.Context) {

```

```

    ...
    if err := s.srv.Secrets().Create(c, &r, metav1.CreateOptions{}); err != nil {
        core.WriteResponse(c, err, nil)

        return
    }
    ...
}

```

在Create方法中，调用了s.srv.Secrets().Create()来创建密钥，s.srv是一个接口类型，定义如下：

```

type Service interface {
    Users() UserSrv
    Secrets() SecretSrv
    Policies() PolicySrv
}

type UserSrv interface {
    Create(ctx context.Context, user *v1.User, opts metav1.CreateOptions) error
    Update(ctx context.Context, user *v1.User, opts metav1.UpdateOptions) error
    Delete(ctx context.Context, username string, opts metav1.DeleteOptions) error
    DeleteCollection(ctx context.Context, usernames []string, opts metav1.DeleteOptions) error
    Get(ctx context.Context, username string, opts metav1.GetOptions) (*v1.User, error)
    List(ctx context.Context, opts metav1.ListOptions) (*v1.UserList, error)
    ListWithBadPerformance(ctx context.Context, opts metav1.ListOptions) (*v1.UserList, error)
    ChangePassword(ctx context.Context, user *v1.User) error
}

```

可以看到，控制层通过业务层提供的Service接口类型，剥离了业务层的具体实现。业务层的Service接口类型提供了Secrets()方法，该方法返回了一个实现了SecretSrv接口的实例。在控制层中，通过调用该实例的Create(ctx context.Context, user *v1.User, opts metav1.CreateOptions) error方法来完成密钥的创建。至于业务层是如何创建密钥的，控制层不需要知道，也就是说创建密钥可以有多种实现。

这里使用到了设计模式中的**工厂方法模式**。Service是工厂接口，里面包含了一系列创建具体业务层对象的工厂函数：Users()、Secrets()、Policies()。通过工厂方法模式，不仅隐藏了业务层对象的创建细节，而且还可以很方便地在Service工厂接口实现方法中添加新的业务层对象。

例如，我们想新增一个Template业务层对象，用来在iam-apiserver中预置一些策略模板，可以这么来加：

```

type Service interface {
    Users() UserSrv
    Secrets() SecretSrv
    Policies() PolicySrv
    Templates() TemplateSrv
}

func (s *service) Templates() TemplateSrv {

```



```
    return newTemplates(s)
}
```

接下来，新建一个template.go文件：

```
type TemplateSrv interface {
    Create(ctx context.Context, template *v1.Template, opts metav1.CreateOptions) error
    // Other methods
}

type templateService struct {
    store store.Factory
}

var _ TemplateSrv = (*userService)(nil)

func newTemplates(srv *service) *TemplateService {
    // more create logic
    return &TemplateService{store: srv.store}
}

func (u *templateService) Create(ctx context.Context, template *v1.Template, opts metav1.CreateOptions) error {
    // normal code

    return nil
}
```

可以看到，我们通过以下三步新增了一个业务层对象：

1. 在Service接口定义中，新增了一个入口：Templates() TemplateSrv。
2. 在service.go文件中，新增了一个函数：Templates()。
3. 新建了template.go文件，在template.go中定义了templateService结构体，并为它实现了TemplateSrv接口。

可以看到，我们新增的Template业务对象的代码几乎都闭环在template.go文件中。对已有的Service工厂接口的创建方法，除了新增一个工厂方法Templates() TemplateSrv外，没有其他任何入侵。这样做可以避免影响已有业务。

在实际项目开发中，你也有可能会想到下面这种错误的创建方式：

```
// 错误方法一
type Service interface {
    UserSrv
    SecretSrv
    PolicySrv
    TemplateSrv
}
```

上面的创建方式中，我们如果想创建User和Secret，那只能定义两个不同的方法：CreateUser和CreateSecret，远没有在有User和Secret各自的域中提供同名的Create方法来得优雅。

IAM项目中还有其他地方也使用了工厂方法模式，例如[Factory](#)工厂接口。

再来看下业务层和仓库层是如何通信的。

业务层和仓库层也是通过接口来通信的。例如，在业务层中创建密钥的代码如下：

```
func (s *secretService) Create(ctx context.Context, secret *v1.Secret, opts metav1.CreateOptions) error {
    if err := s.store.Secrets().Create(ctx, secret, opts); err != nil {
        return errors.WithCode(code.ErrDatabase, err.Error())
    }

    return nil
}
```

Create方法中调用了s.store.Secrets().Create()方法来将密钥保存到数据库中。s.store是一个接口类型，定义如下：

```
type Factory interface {
    Users() UserStore
    Secrets() SecretStore
    Policies() PolicyStore
    Close() error}
```

业务层与仓库层的通信实现，和控制层与业务层的通信实现类似，所以这里不再详细介绍。

到这里我们知道了，控制层、业务层和仓库层之间是通过接口来通信的。通过接口通信有一个好处，就是可以让各层变得可测。那接下来，我们就来看下**如何测试各层的代码**。因为**第38讲**和**第39讲**会详细介绍如何测试Go代码，所以这里只介绍下测试思路。

1. 模型层

因为模型层不依赖其他任何层，我们只需要测试其中定义的结构及其函数和方法即可。

2. 控制层

控制层依赖于业务层，意味着该层需要业务层来支持测试。你可以通过[golang/mock](#)来mock业务层，测试用例可参考[TestUserController_Create](#)。

3. 业务层

因为该层依赖于仓库层，意味着该层需要仓库层来支持测试。我们有两种方法来模拟仓库层：

- 通过golang/mock来mock仓库层。
- 自己开发一个fake仓库层。

使用golang/mock的测试用例，你可以参考[Test_secretService_Create](#)。

fake的仓库层可以参考[fake](#)，使用该fake仓库层进行测试的测试用例为 [Test_userService_List](#)。

4. 仓库层

仓库层依赖于数据库，如果调用了其他微服务，那还会依赖第三方服务。我们可以通过[sqlmock](#)来模拟数据库连接，通过[httpmock](#)来模拟HTTP请求。

总结

这一讲，我主要介绍了iam-apiserver的功能和使用方法，以及它的代码实现。iam-apiserver是一个Web服务，提供了REST API来完成用户、密钥、策略三种REST资源的增删改查。我们可以通过cURL、Insomnia等工具，来完成REST API请求。

iam-apiserver包含了3种配置：Options配置、应用配置、HTTP/GRPC服务配置。这三种配置分别用来构建命令行参数、应用和HTTP/GRPC服务。

iam-apiserver在启动时，会先构建应用框架，接着会设置应用选项，然后对应用进行初始化，最后创建HTTP/GRPC服务的配置和实例，最终启动HTTP/GRPC服务。

服务启动之后，就可以接收HTTP请求了。一个HTTP请求会先进行认证，接着会被注册的中间件处理，然后，会根据(HTTP Method, HTTP Request Path)匹配到处理函数。在处理函数中，会解析请求参数、校验参数、调用业务逻辑处理函数，最终返回请求结果。

iam-apiserver采用了简洁架构，整个应用分为4层：模型层、控制层、业务层和仓库层。模型层存储对象的结构和它的方法；仓库层用来跟数据库/第三方服务进行CURD交互；业务层主要用来完成业务逻辑处理；控制层接收HTTP请求，并进行参数解析、参数校验、逻辑分发处理、请求返回操作。控制层、业务层、仓库层之间通过接口通信，通过接口通信可以使相同的功能支持不同的实现，并使每一层的代码变得可测试。

课后练习

1. iam-apiserver和iam-authz-server都提供了REST API服务，阅读它们的源码，看看iam-apiserver和iam-authz-server是如何共享REST API相关代码的。
2. 思考一下，iam-apiserver的服务构建方式，能够再次抽象成一个模板（Go包）吗？如果能，该如何抽象？

欢迎你在留言区与我交流讨论，我们下一讲见。