



下载APP



## 33 | SDK 设计（上）：如何设计出一个优秀的 Go SDK？

2021-08-10 孔令飞

《Go 语言项目开发实战》

课程介绍 &gt;

**讲述：孔令飞**

时长 14:36 大小 13.38M



你好，我是孔令飞。接下来的两讲，我们来看下如何设计和实现一个优秀的 Go SDK。

后端服务通过 API 接口对外提供应用的功能，但是用户直接调用 API 接口，需要编写 API 接口调用的逻辑，并且需要构造入参和解析返回的数据包，使用起来效率低，而且有一定的开发工作量。

在实际的项目开发中，通常会提供对开发者更友好的 SDK 包，供客户端调用。很多大型服务在发布时都会伴随着 SDK 的发布，例如腾讯云很多产品都提供了 SDK：



对象存储	归档存储	数据传输服务 DTS	即时通信 IM
Android SDK 	Python SDK 	数据订阅的 SDK2.0 	Android SDK
C SDK 			iOS SDK
C++ SDK 			Mac SDK
更多 			更多 
腾讯移动推送	语言消息	实时音视频	移动直播 SDK
Android SDK 	Java SDK 	iOS SDK 	iOS 端集成
iOS SDK 	PHP SDK 	Android SDK 	Android 端集成
Mac SDK 	Python SDK 	Windows SDK 	Web 端 ( TcPlayerLite )
更多 	更多 	更多 	更多 

既然 SDK 如此重要，那么如何设计一个优秀的 Go SDK 呢？这一讲我就来详细介绍一下。

## 什么是 SDK？

首先，我们来看下什么是 SDK。

对于 SDK（Software Development Kit，软件开发工具包），不同场景下有不同的解释。但是对于一个 Go 后端服务来说，SDK 通常是指**封装了 Go 后端服务 API 接口的软件包**，里面通常包含了跟软件相关的库、文档、使用示例、封装好的 API 接口和工具。

调用 SDK 跟调用本地函数没有太大的区别，所以可以极大地提升开发者的开发效率和体验。SDK 可以由服务提供者提供，也可以由其他组织或个人提供。为了鼓励开发者使用其系统或语言，SDK 通常都是免费提供的。

通常，服务提供者会提供不同语言的 SDK，比如针对 Python 开发者会提供 Python 版的 SDK，针对 Go 开发者会提供 Go 版的 SDK。一些比较专业的团队还会有 SDK 自动生成工具，可以根据 API 接口定义，自动生成不同语言的 SDK。例如，Protocol Buffers 的编译工具 protoc，就可以基于 Protobuf 文件生成 C++、Python、Java、JavaScript、PHP 等语言版本的 SDK。阿里云、腾讯云这些一线大厂，也可以基于 API 定义，生成不同编程语言的 SDK。

## SDK 设计方法

那么，我们如何才能设计一个好的 SDK 呢？对于 SDK，不同团队会有不同的设计方式，我调研了一些优秀 SDK 的实现，发现这些 SDK 有一些共同点。根据我的调研结果，结合

我在实际开发中的经验，我总结出了一套 SDK 设计方法，接下来就分享给你。

## 如何给 SDK 命名？

在讲设计方法之前，我先来介绍两个重要的知识点：SDK 的命名方式和 SDK 的目录结构。

SDK 的名字目前没有统一的规范，但比较常见的命名方式是 xxx-sdk-go / xxx-sdk-python / xxx-sdk-java。其中，xxx 可以是项目名或者组织名，例如腾讯云在 GitHub 上的组织名为 tencentcloud，那它的 SDK 命名如下图所示：

### [tencentcloud-sdk-java](#)

Tencent Cloud API 3.0 SDK for Java

Java Apache-2.0 163 279 2 1 Updated 32 minutes ago

### [tencentcloud-sdk-python](#)

Tencent Cloud API 3.0 SDK for Python

Python Apache-2.0 150 319 3 0 Updated 1 hour ago

### [tencentcloud-sdk-php](#)

Tencent Cloud API 3.0 SDK for PHP

PHP Apache-2.0 133 223 1 3 Updated 22 hours ago

### [tencentcloud-sdk-go](#)

Tencent Cloud API 3.0 SDK for Golang

Go Apache-2.0 111 293 6 1 Updated 1 hour ago

## SDK 的目录结构

不同项目 SDK 的目录结构也不相同，但一般需要包含下面这些文件或目录。目录名可能会有所不同，但目录功能是类似的。

**README.md**：SDK 的帮助文档，里面包含了安装、配置和使用 SDK 的方法。


**examples/sample/**：SDK 的使用示例。

**sdk/**：SDK 共享的包，里面封装了最基础的通信功能。如果是 HTTP 服务，基本都是基于 net/http 包进行封装。

**api**：如果 xxx-sdk-go 只是为某一个服务提供 SDK，就可以把该服务的所有 API 接口封装代码存放在 api 目录下。

**services/{iam, tms}**：如果 xxx-sdk-go 中，xxx 是一个组织，那么这个 SDK 很可能会集成该组织中很多服务的 API，就可以把某类服务的 API 接口封装代码存放在 services/<服务名>下，例如 AWS 的 [Go SDK](#)。

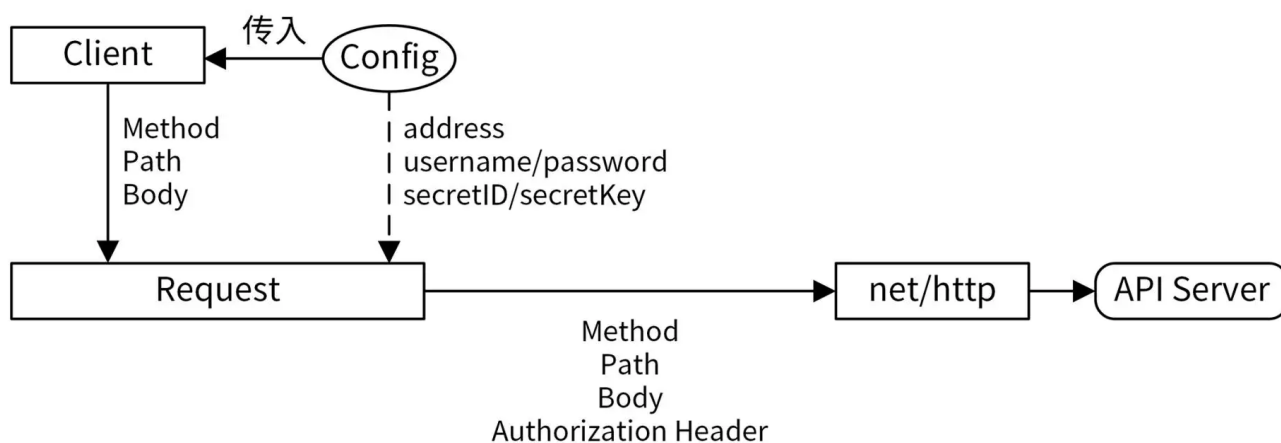
一个典型的目录结构如下：

 复制代码

```
1 |— examples          # 示例代码存放目录
2 |   |— authz.go
3 |— README.md        # SDK使用文档
4 |— sdk              # 公共包，封装了SDK配置、API请求、认证等代码
5 |   |— client.go
6 |   |— config.go
7 |   |— credential.go
8 |   |— ...
9 |— services          # API封装
10 |   |— common
11 |   |   |— model
12 |   |— iam          # iam服务的API接口
13 |   |   |— authz.go
14 |   |   |— client.go
15 |   |   |— ...
16 |— tms              # tms服务的API接口
```

## SDK 设计方法

SDK 的设计方法如下图所示：



我们可以通过 Config 配置创建客户端 Client，例如 `func NewClient(config sdk.Config) (Client, error)`，配置中可以指定下面的信息。

服务的后端地址：服务的后端地址可以通过配置文件来配置，也可以直接固化在 SDK 中，推荐后端服务地址可通过配置文件配置。

认证信息：最常用的认证方式是通过密钥认证，也有一些是通过用户名和密码认证。

其他配置：例如超时时间、重试次数、缓存时间等。

创建的 Client 是一个结构体或者 Go interface。这里我建议你使用 interface 类型，这样可以将定义和具体实现解耦。Client 具有一些方法，例如 `CreateUser`、`DeleteUser` 等，每一个方法对应一个 API 接口，下面是一个 Client 定义：

复制代码

```

1 type Client struct {
2     client *sdk.Request
3 }
4
5 func (c *Client) CreateUser(req *CreateUserRequest) (*CreateUserResponse, error) {
6     // normal code
7     resp := &CreateUserResponse{}
8     err := c.client.Send(req, resp)
9     return resp, err
10 }
  
```

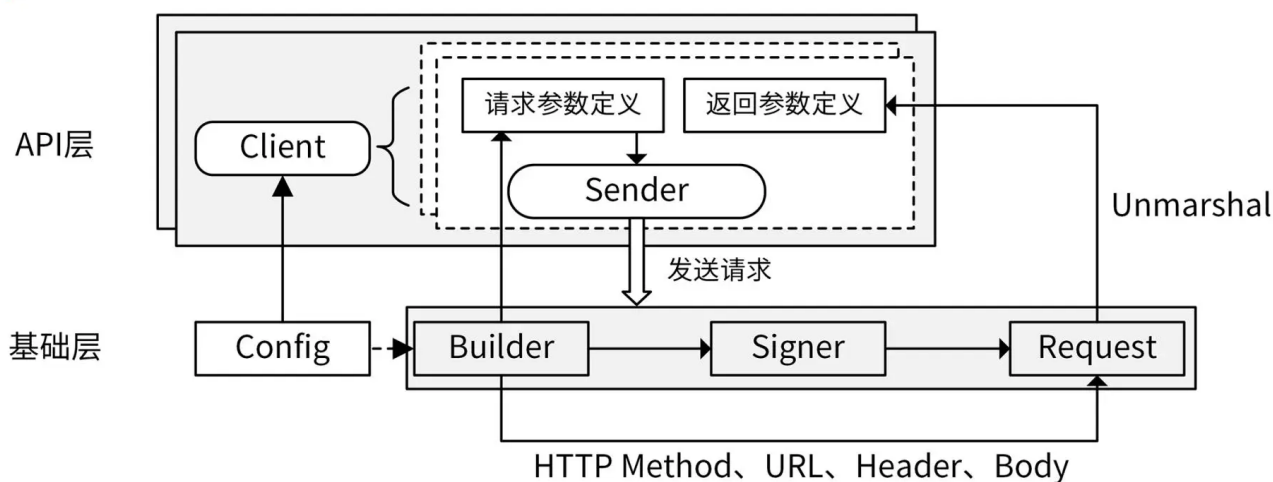
调用 `client.CreateUser(req)` 会执行 HTTP 请求，在 `req` 中可以指定 HTTP 请求的方法 `Method`、路径 `Path` 和请求 `Body`。`CreateUser` 函数中，会调用 `c.client.Send(req)` 执行具体的 HTTP 请求。

`c.client` 是 `*Request` 类型的变量，`*Request` 类型的变量具有一些方法，可以根据传入的请求参数 `req` 和 `config` 配置构造出请求路径、认证头和请求 `Body`，并调用 `net/http` 包完成最终的 HTTP 请求，最后将返回结果 `Unmarshal` 到传入的 `resp` 结构体中。

根据我的调研，目前有两种 SDK 设计方式可供参考，一种是各大公有云厂商采用的 SDK 设计方式，一种是 Kubernetes `client-go` 的设计方式。IAM 项目分别实现了这两种 SDK 设计方式，但我还是更倾向于对外提供 `client-go` 方式的 SDK，我会在下一讲详细介绍它。这两种设计方式的设计思路跟上面介绍的是一致的。

## 公有云厂商采用的 SDK 设计方式

这里，我先来简单介绍下公有云厂商采用的 SDK 设计模式。SDK 架构如下图所示：



SDK 框架分为两层，分别是 API 层和基础层。API 层主要用来构建客户端实例，并调用客户端实例提供的方法来完成 API 请求，每一个方法对应一个 API 接口。API 层最终会调用基础层提供的能力，来完成 REST API 请求。基础层通过依次执行构建请求参数（`Builder`）、签发并添加认证头（`Signer`）、执行 HTTP 请求（`Request`）三大步骤，来完成具体的 REST API 请求。



为了让你更好地理解公有云 SDK 的设计方式，接下来我会结合一些真实的代码，给你讲解 API 层和基础层的具体设计，SDK 代码见 [@medu-sdk-go](#)。

## API 层：创建客户端实例

客户端在使用服务 A 的 SDK 时，首先需要根据 Config 配置创建一个服务 A 的客户端 Client，Client 实际上是一个 struct，定义如下：

[复制代码](#)

```
1 type Client struct {  
2     sdk.Client  
3 }
```

在创建客户端时，需要传入认证（例如密钥、用户名 / 密码）、后端服务地址等配置信息。例如，可以通过 [@NewClientWithSecret](#) 方法来构建一个带密钥对的客户端：

[复制代码](#)

```
1 func NewClientWithSecret(secretID, secretKey string) (client *Client, err error)  
2     client = &Client{}  
3     config := sdk.NewConfig().WithEndpoint(defaultEndpoint)  
4     client.Init(serviceName).WithSecret(secretID, secretKey).WithConfig(config)  
5     return  
6 }
```

这里要注意，上面创建客户端时，传入的密钥对最终会在基础层中被使用，用来签发 JWT Token。

Client 有多个方法（Sender），例如 Authz 等，每个方法代表一个 API 接口。Sender 方法会接收 AuthzRequest 等结构体类型的指针作为输入参数。我们可以调用 `client.Authz(req)` 来执行 REST API 调用。可以在 `client.Authz` 方法中添加一些业务逻辑处理。`client.Authz` 代码如下：

[复制代码](#)

```
1 type AuthzRequest struct {  
2     *request.BaseRequest  
3     Resource *string `json:"resource"`  
4     Action *string `json:"action"`  
5 }
```

```
6     Subject *string `json:"subject"`
7     Context *ladon.Context
8 }
9
10 func (c *Client) Authz(req *AuthzRequest) (resp *AuthzResponse, err error) {
11     if req == nil {
12         req = NewAuthzRequest()
13     }
14
15     resp = NewAuthzResponse()
16     err = c.Send(req, resp)
17     return
18 }
```

请求结构体中的字段都是指针类型的，使用指针的好处是可以判断入参是否有被指定，如果 `req.Subject == nil` 就说明传参中没有 Subject 参数，如果 `req.Subject != nil` 就说明参数中有传 Subject 参数。根据某个参数是否被传入，执行不同的业务逻辑，这在 Go API 接口开发中非常常见。

另外，因为 Client 通过匿名的方式继承了基础层中的 `sdk.Client`：

```
1 type Client struct {
2     sdk.Client
3 }
```

[复制代码](#)

所以，API 层创建的 Client 最终可以直接调用基础层中的 Client 提供的 `Send(req, resp)` 方法，来执行 RESTful API 调用，并将结果保存在 `resp` 中。

为了方便和 API 层的 Client 进行区分，我下面统一将基础层中的 Client 称为 **sdk.Client**。

最后，一个完整的客户端调用示例代码如下：

```
1 package main
2
3 import (
4     "fmt"
5 )
```

[复制代码](#)



```
6     "github.com/ory/ladon"
7
8     "github.com/marmotedu/medu-sdk-go/sdk"
9     iam "github.com/marmotedu/medu-sdk-go/services/iam/authz"
10 )
11
12 func main() {
13     client, _ := iam.NewClientWithSecret("XhbY3aCrfdYcP10FJRu9xcno8JzSbUIvGE2",
14
15     req := iam.NewAuthzRequest()
16     req.Resource = sdk.String("resources:articles:ladon-introduction")
17     req.Action = sdk.String("delete")
18     req.Subject = sdk.String("users:peter")
19     ctx := ladon.Context(map[string]interface{}{"remoteIP": "192.168.0.5"})
20     req.Context = &ctx
21
22     resp, err := client.Authz(req)
23     if err != nil {
24         fmt.Println("err1", err)
25         return
26     }
27     fmt.Printf("get response body: `%s`\n", resp.String())
28     fmt.Printf("allowed: %v\n", resp.Allowed)
29 }
```

## 基础层：构建并执行 HTTP 请求

上面我们创建了客户端实例，并调用了它的 [Send](#) 方法来完成最终的 HTTP 请求。这里，我们来看下 Send 方法具体是如何构建 HTTP 请求的。

sdk.Client 通过 Send 方法，完成最终的 API 调用，代码如下：

```
1 func (c *Client) Send(req request.Request, resp response.Response) error {
2     method := req.GetMethod()
3     builder := GetParameterBuilder(method, c.Logger)
4     jsonReq, _ := json.Marshal(req)
5     encodedUrl, err := builder.BuildURL(req.GetURL(), jsonReq)
6     if err != nil {
7         return err
8     }
9
10    endPoint := c.Config.Endpoint
11    if endPoint == "" {
12        endPoint = fmt.Sprintf("%s/%s", defaultEndpoint, c.ServiceName)
13    }
14    reqUrl := fmt.Sprintf("%s://%s/%s%s", c.Config.Scheme, endPoint, req.GetVers
```

[复制代码](#)

```
15  body, err := builder.BuildBody(jsonReq)
16  if err != nil {
17      return err
18  }
19
20  sign := func(r *http.Request) error {
21      signer := NewSigner(c.signMethod, c.Credential, c.Logger)
22      _ = signer.Sign(c.ServiceName, r, strings.NewReader(body))
23      return err
24  }
25
26  rawResponse, err := c.doSend(method, reqUrl, body, req.GetHeaders(), sign)
27  if err != nil {
28      return err
29  }
30
31  return response.ParseFromHttpResponse(rawResponse, resp)
32 }
33
```

上面的代码大体上可以分为四个步骤。


## 第一步，Builder：构建请求参数。

根据传入的 AuthzRequest 和客户端配置 Config，构造 HTTP 请求参数，包括请求路径和请求 Body。

接下来，我们来看下如何构造 HTTP 请求参数。

### 1. HTTP 请求路径构建

在创建客户端时，我们通过 [NewAuthzRequest](#) 函数创建了 /v1/authz REST API 接口请求结构体 AuthzRequest，代码如下：

 复制代码

```
1  func NewAuthzRequest() (req *AuthzRequest) {
2      req = &AuthzRequest{
3          BaseRequest: &request.BaseRequest{
4              URL:      "/authz",
5              Method: "POST",
6              Header: nil,
7              Version: "v1",
8          },
9      }
```

```
9     }  
10    return  
11 }
```

可以看到，我们创建的 req 中包含了 API 版本（Version）、API 路径（URL）和请求方法（Method）。这样，我们就可以在 Send 方法中，构建出请求路径：

[复制代码](#)

```
1 endPoint := c.Config.Endpoint  
2 if endPoint == "" {  
3     endPoint = fmt.Sprintf("%s/%s", defaultEndpoint, c.ServiceName)  
4 }  
5 reqUrl := fmt.Sprintf("%s://%s/%s%s", c.Config.Scheme, endPoint, req.GetVersio
```

上述代码中，c.Config.Scheme=http/https、endPoint=iam.api.marmotedu.com:8080、req.GetVersion()=v1 和 encodedUrl，我们可以认为它们等于 /authz。所以，最终构建出的请求路径为 http://iam.api.marmotedu.com:8080/v1/authz。

## 2. HTTP 请求 Body 构建

在 [BuildBody](#) 方法中构建请求 Body。BuildBody 会将 req Marshal 成 JSON 格式的 string。HTTP 请求会以该字符串作为 Body 参数。

### 第二步，Signer：签发并添加认证头。

访问 IAM 的 API 接口需要进行认证，所以在发送 HTTP 请求之前，还需要给 HTTP 请求添加认证 Header。

medu-sdk-go 代码提供了 JWT 和 HMAC 两种认证方式，最终采用了 JWT 认证方式。JWT 认证签发方法为 [Sign](#)，代码如下：

[复制代码](#)

```
1 func (v1 SignatureV1) Sign(serviceName string, r *http.Request, body io.ReadSe  
2     tokenString := auth.Sign(v1.Credentials.SecretID, v1.Credentials.SecretKey,  
3     r.Header.Set("Authorization", fmt.Sprintf("Bearer %s", tokenString))  
4     return r.Header
```

```
5 }  
6 }
```

auth.Sign 方法根据 SecretID 和 SecretKey 签发 JWT Token。

接下来，我们就可以调用 [doSend](#) 方法来执行 HTTP 请求了。调用代码如下：

```
1 rawResponse, err := c.doSend(method, reqUrl, body, req.GetHeaders(), sign)  
2 if err != nil {  
3     return err  
4 }
```

[复制代码](#)

可以看到，我们传入了 HTTP 请求方法 method、HTTP 请求 URL reqUrl、HTTP 请求 Body body，以及用来签发 JWT Token 的 sign 方法。我们在调用 NewAuthzRequest 创建 req 时，指定了 HTTP Method，所以这里的 method := req.GetMethod()、reqUrl 和请求 Body 都是通过 Builder 来构建的。

### 第三步，Request：执行 HTTP 请求。

调用 [doSend](#) 方法执行 HTTP 请求，doSend 通过调用 net/http 包提供的 http.NewRequest 方法来发送 HTTP 请求，执行完 HTTP 请求后，会返回 \*http.Response 类型的 Response。代码如下：

```
1 func (c *Client) doSend(method, url, data string, header map[string]string, si  
2     client := &http.Client{Timeout: c.Config.Timeout}  
3  
4     req, err := http.NewRequest(method, url, strings.NewReader(data))  
5     if err != nil {  
6         c.Logger.Errorf("%s", err.Error())  
7         return nil, err  
8     }  
9  
10    c.setHeader(req, header)  
11  
12    err = sign(req)  
13    if err != nil {  
14        return nil, err  
15    }
```

[复制代码](#)

```
16
17     return client.Do(req)
18 }
```

#### 第四步，处理 HTTP 请求返回结果。

调用 `doSend` 方法返回 `*http.Response` 类型的 `Response` 后，`Send` 方法会调用 [ParseFromHttpResponse](#) 函数来处理 HTTP Response，`ParseFromHttpResponse` 函数代码如下：

[复制代码](#)

```
1 func ParseFromHttpResponse(rawResponse *http.Response, response Response) error {
2     defer rawResponse.Body.Close()
3     body, err := ioutil.ReadAll(rawResponse.Body)
4     if err != nil {
5         return err
6     }
7     if rawResponse.StatusCode != 200 {
8         return fmt.Errorf("request fail with status: %s, with body: %s", rawResponse.StatusCode, body)
9     }
10
11     if err := response.ParseErrorFromHTTPResponse(body); err != nil {
12         return err
13     }
14
15     return json.Unmarshal(body, &response)
16 }
```

可以看到，在 `ParseFromHttpResponse` 函数中，会先判断 HTTP Response 中的 `StatusCode` 是否为 200，如果不是 200，则会报错。如果是 200，会调用传入的 `resp` 变量提供的 [ParseErrorFromHTTPResponse](#) 方法，来将 HTTP Response 的 Body Unmarshal 到 `resp` 变量中。

通过以上四步，SDK 调用方调用了 API，并获得了 API 的返回结果 `resp`。

下面这些公有云厂商的 SDK 采用了此设计模式：

腾讯云 SDK：[tencentcloud-sdk-go](#)。

AWS SDK : [aws-sdk-go](#)。

阿里云 SDK : [alibaba-cloud-sdk-go](#)。

京东云 SDK : [jdcloud-sdk-go](#)。

Ucloud SDK : [ucloud-sdk-go](#)。

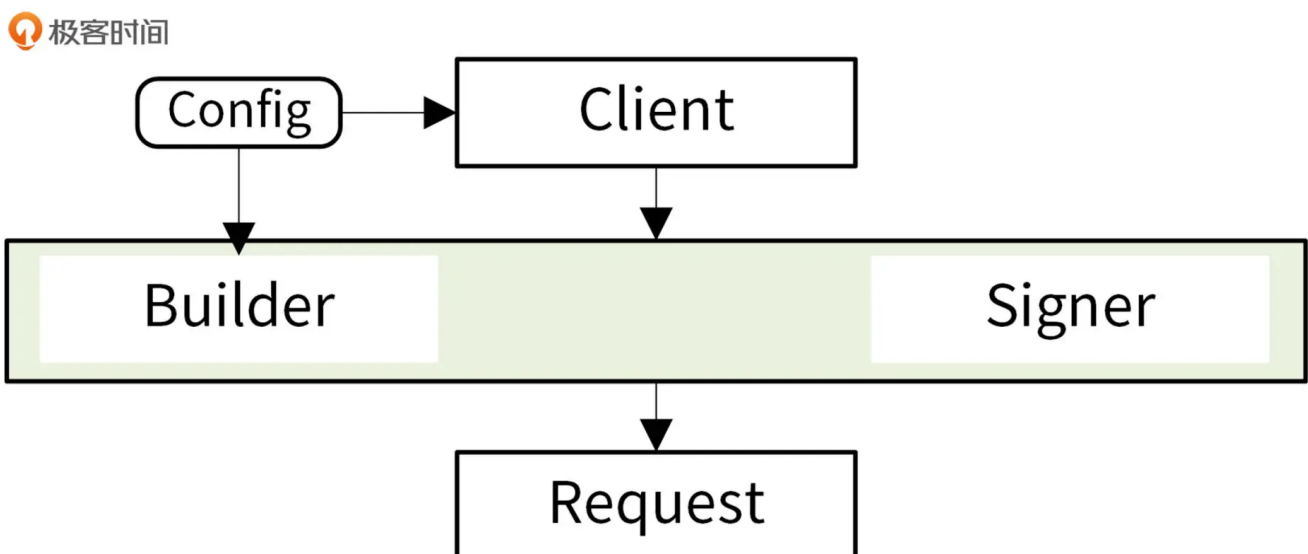
IAM 公有云方式的 SDK 实现为 [medu-sdk-go](#)。

此外，IAM 还设计并实现了 Kubernetes client-go 方式的 Go SDK : [marmotedu-sdk-go](#)，marmotedu-sdk-go 也是 IAM Go SDK 所采用的 SDK。下一讲中，我会具体介绍 marmotedu-sdk-go 的设计和实现。

## 总结

这一讲，我主要介绍了如何设计一个优秀的 Go SDK。通过提供 SDK，可以提高 API 调用效率，减少 API 调用难度，所以大型应用通常都会提供 SDK。不同团队有不同的 SDK 设计方法，但目前比较好的实现是公有云厂商采用的 SDK 设计方式。

公有云厂商的 SDK 设计方式中，SDK 按调用顺序从上到下可以分为 3 个模块，如下图所示：



Client 构造 SDK 客户端，在构造客户端时，会创建请求参数 req，req 中会指定 API 版本、HTTP 请求方法、API 请求路径等信息。

Client 会请求 Builder 和 Signer 来构建 HTTP 请求的各项参数：HTTP 请求方法、HTTP 请求路径、HTTP 认证头、HTTP 请求 Body。Builder 和 Signer 是根据 req 配置来构造这些 HTTP 请求参数的。

构造完成之后，会请求 Request 模块，Request 模块通过调用 net/http 包，来执行 HTTP 请求，并返回请求结果。

## 课后练习

1. 思考下，如何实现可以支持多个 API 版本的 SDK 包，代码如何实现？
2. 这一讲介绍了一种 SDK 实现方式，在你的 Go 开发生涯中，还有没有一些更好的 SDK 实现方法？欢迎在留言区分享。

期待你在留言区与我交流讨论，我们下一讲见。

分享给需要的人，Ta 订阅后你可得 **24 元现金奖励**

 赞 1     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 | 数据处理：如何高效处理应用程序产生的数据？

下一篇    特别放送 | 给你一份清晰、可直接套用的 Go 编码规范



更多课程推荐

# 说透区块链

拨开迷雾，还原区块链真相

赵铭

区块链服务平台资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

## 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。