



下载APP



35 | 效率神器：如何设计和实现一个命令行客户端工具？

2021-08-14 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 16:03 大小 14.70M



你好，我是孔令飞。今天我们来聊聊，如何实现一个命令行客户端工具。

如果你用过 Kubernetes、Istio、etcd，那你一定用过这些开源项目所提供的命令行工具：kubectl、istioctl、etcdctl。一个 xxx 项目，伴随着一个 xxxctl 命令行工具，这似乎已经成为一种趋势，在一些大型系统中更是常见。提供 xxxctl 命令行工具有这两个好处：

实现自动化：可以通过在脚本中调用 xxxctl 工具，实现自动化。

提高效率：通过将应用的功能封装成命令和参数，方便运维、开发人员在 Linux 服务器上调用。



其中，kubectl 命令设计的功能最为复杂，也是非常优秀的命令行工具，IAM 项目的 iamctl 客户端工具就是仿照 kubectl 来实现的。这一讲，我就通过剖析 iamctl 命令行工具的实现，来介绍下如何实现一个优秀的客户端工具。

常见客户端介绍

在介绍 iamctl 命令行工具的实现之前，我们先来看下常见的客户端。

客户端又叫用户端，与后端服务相对应，安装在客户机上，用户可以使用这些客户端访问后端服务。不同的客户端面向的人群不同，所能提供的访问能力也有差异。常见的客户端有下面这几种：

前端，包括浏览器、手机应用；

SDK；

命令行工具；

其他终端。

接下来，我就来分别介绍下。

浏览器和手机应用提供一个交互界面供用户访问后端服务，使用体验最好，面向的人群是最终的用户。这两类客户端也称为前端。前端由前端开发人员进行开发，并通过 API 接口，调用后端的服务。后端开发人员不需要关注这两类客户端，只需要关注如何提供 API 接口即可。

SDK (Software Development Kit) 也是一个客户端，供开发者调用。开发者调用 API 时，如果是通过 HTTP 协议，需要编写 HTTP 的调用代码、HTTP 请求包的封装和返回包的解封，还要处理 HTTP 的状态码，使用起来不是很方便。SDK 其实是封装了 API 接口的一系列函数集合，开发者通过调用 SDK 中的函数调用 API 接口，提供 SDK 主要是方便开发者调用，减少工作量。

命令行工具是可以在操作系统上执行的一个二进制程序，提供了一种比 SDK 和 API 接口更方便快捷的访问后端服务的途径，供运维或者开发人员在服务器上直接执行使用，或者在自动化脚本中调用。

还有其他各类客户端，这里我列举一些常见的。

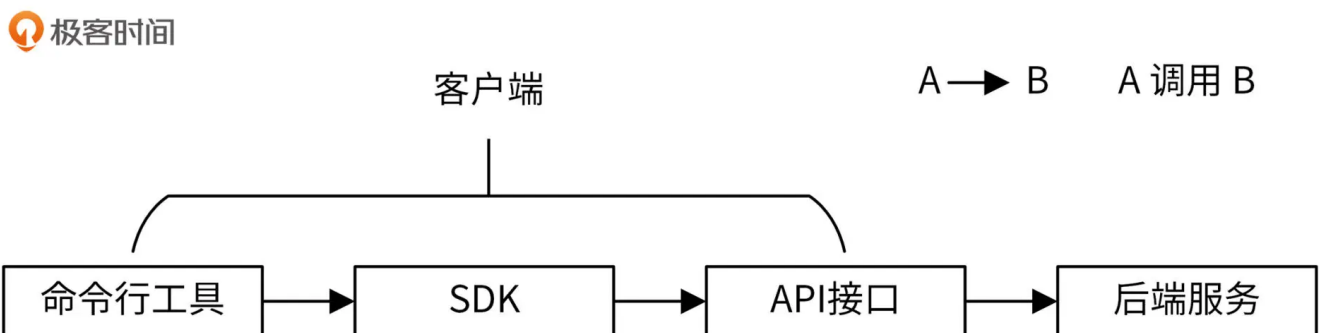
终端设备：POS 机、学习机、智能音箱等。

第三方应用程序：通过调用 API 接口或者 SDK，调用我们提供的后端服务，从而实现自身的功能。

脚本：脚本中通过 API 接口或者命令行工具，调用我们提供的后端服务，实现自动化。

这些其他的各类客户端，都是通过调用 API 接口使用后端服务的，它们跟前端一样，也不需要后台开发人员开发。

需要后台开发人员投入工作量进行研发的客户端是 SDK 和命令行工具。这两类客户端工具有个调用和被调用的顺序，如下图所示：



你可以看到，命令行工具和 SDK 最终都是通过 API 接口调用后端服务的，通过这种方式可以保证服务的一致性，并减少为适配多个客户端所带来的额外开发工作量。

大型系统客户端（xxxctl）的特点


通过学习 kubectl、istioctl、etcdctl 这些优秀的命令行工具，可以发现一个大型系统的命令行工具，通常具有下面这些特点：

支持命令和子命令，命令 / 子命名有自己独有的命令行参数。

支持一些特殊的命令。比如支持 completion 命令，completion 命令可以输出 bash/zsh 自动补全脚本，实现命令行及参数的自动补全。还支持 version 命令，version 命令不仅可以输出客户端的版本，还可以输出服务端的版本（如果有需要）。

支持全局 option，全局 option 可以作为所有命令及子命令的命令行参数。

支持 `-h/help` , `-h/help` 可以打印 `xxxctl` 的帮助信息，例如：


 复制代码

```

1 $ iamctl -h
2 iamctl controls the iam platform, is the client side tool for iam platform.
3
4 Find more information at:
5 https://github.com/marmotedu/iam/blob/master/docs/guide/en-US/cmd/iamctl/iamct
6
7 Basic Commands:
8   info          Print the host information
9   color         Print colors supported by the current terminal
10  new           Generate demo command code
11  jwt           JWT command-line tool
12
13 Identity and Access Management Commands:
14  user          Manage users on iam platform
15  secret        Manage secrets on iam platform
16  policy        Manage authorization policies on iam platform
17
18 Troubleshooting and Debugging Commands:
19  validate      Validate the basic environment for iamctl to run
20
21 Settings Commands:
22  set           Set specific features on objects
23  completion    Output shell completion code for the specified shell (bash or zs
24
25 Other Commands:
26  version       Print the client and server version information
27
28 Usage:
29  iamctl [flags] [options]
30
31 Use "iamctl <command> --help" for more information about a given command.
32 Use "iamctl options" for a list of global command-line options (applies to all

```

支持 `xxxctl help [command | command subcommand] [command | command subcommand] -h` , 打印命令 / 子命令的帮助信息，格式通常为 命令描述 + 使用方法 。例如：

 复制代码

```

1 $ istioctl help register
2 Registers a service instance (e.g. VM) joining the mesh
3
4 Usage:
5  istioctl register <svcname> <ip> [name1:]port1 [name2:]port2 ... [flags]

```

除此之外，一个大型系统的命令行工具还可以支持一些更高阶的功能，例如：支持命令分组，支持配置文件，支持命令的使用 example，等等。

在 Go 生态中，如果我们要找一个符合上面所有特点的命令行工具，那非 [kubectI](#) 莫属。因为我今天要重点讲的 iamctl 客户端工具，就是仿照它来实现的，所以这里就不展开介绍 kubectI 了，不过还是建议你认真研究下 kubectI 的实现。

iamctl 的核心实现

接下来，我就来介绍 IAM 系统自带的 iamctl 客户端工具，它是仿照 kubectI 来实现的，能够满足一个大型系统客户端工具的需求。我会从 iamctl 的功能、代码结构、命令行选项和配置文件解析 4 个方面来介绍。

iamctl 的功能

iamctl 将命令进行了分类。这里，我也建议你对命令进行分类，因为通过分类，不仅可以协助你理解命令的用途，还能帮你快速定位某类命令。另外，当命令很多时，分类也可以使命令看起来更规整。

iamctl 实现的命令如下：



命令		功能
Basic Commands	info	打印机器的信息
	color	打印终端支持的颜色
	new	生成命令和子命令
	jwt	签发、解析和验证JWT Token
Identity and Access Management Commands	user	IAM系统用户的增删改查
	secret	IAM系统密钥的增删改查
	policy	IAM系统策略的增删改查
Troubleshooting and Debugging Commands	validate	验证iamctl是否处在可用状态，主要是检查是否能够连通iam-apiserver
Settings Commands	set	做一些系统设置相关的工作
	completion	生成bash/zash自动补全脚本
Other Commands	version	打印IAM客户端和服务端版本

更详细的功能，你可以参考 `iamctl -h`。我建议你在实现 `xxxctl` 工具时，考虑实现下面这几个功能。

API 功能：平台具有的 API 功能，都能通过 `xxxctl` 方便地进行调用。

工具：一些使用 IAM 系统时有用的功能，比如签发 JWT Token。

`version`、`completion`、`validate` 命令。

代码结构


`iamctl` 工具的 `main` 函数位于 `iamctl.go` 文件中。命令的实现存放在 `internal/iamctl/cmd/cmd.go` 文件中。`iamctl` 的命令统一存放在 `internal/iamctl/cmd` 目录下，每个命令都是一个 Go 包，包名即为命令名，具体实现存

放在 `internal/iamctl/cmd/<命令>/<命令>.go` 文件中。如果命令有子命令，则子命令的实现存放在 `internal/iamctl/cmd/<命令>/<命令>_<子命令>.go` 文件中。

使用这种代码组织方式，即使是在命令很多的情况下，也能让代码井然有序，方便定位和维护代码。

命令行选项

添加命令行选项的代码在 [🔗 NewIAMCtlCommand](#) 函数中，核心代码为：

 复制代码

```
1 flags := cmds.PersistentFlags()
2 ...
3 iamConfigFlags := genericclioptions.NewConfigFlags(true).WithDeprecatedPasswor
4 iamConfigFlags.AddFlags(flags)
5 matchVersionIAMConfigFlags := cmdutil.NewMatchVersionFlags(iamConfigFlags)
6 matchVersionIAMConfigFlags.AddFlags(cmds.PersistentFlags())
```

`NewConfigFlags(true)` 返回带有默认值的参数，并通过 `iamConfigFlags.AddFlags(flags)` 添加到 cobra 的命令行 flag 中。

`NewConfigFlags(true)` 返回结构体类型的值都是指针类型，这样做的好处是：程序可以判断出是否指定了某个参数，从而可以根据需要添加参数。例如：可以通过 `WithDeprecatedPasswordFlag()` 和 `WithDeprecatedSecretFlag()` 添加密码和密钥认证参数。

`NewMatchVersionFlags` 指定是否需要服务端版本和客户端版本一致。如果不一致，在调用服务接口时会报错。

配置文件解析

iamctl 需要连接 iam-apiserver，来完成用户、策略和密钥的增删改查，并且需要进行认证。要完成这些功能，需要有比较多的配置项。这些配置项如果每次都在命令行选项指定，会很麻烦，也容易出错。

最好的方式是保存到配置文件中，并加载配置文件。加载配置文件的代码位于 `NewIAMCtlCommand` 函数中，代码如下：

[复制代码](#)

```
1 _ = viper.BindPFlags(cmds.PersistentFlags())
2 cobra.OnInitialize(func() {
3     genericapiserver.LoadConfig(viper.GetString(genericclioptions.FlagIAMConfi
4 })
```

iamctl 会按以下优先级加载配置文件：

1. 命令行参 `--iamconfig` 指定的配置文件。
2. 当前目录下的 `config.yaml` 文件。
3. `$HOME/.iam/config.yaml` 文件。

这种加载方式具有两个好处。首先是可以手动指定不同的配置文件，这在多环境、多配置下尤为重要。其次是方便使用，可以把配置存放在默认的加载路径中，在执行命令时，就不用再指定 `--iamconfig` 参数。

加载完配置文件之后，就可以通过 `viper.Get<Type>()` 函数来获取配置。例如，iamctl 使用了以下 `viper.Get<Type>` 方法：

```
[colin@dev iamctl]$ grep -R viper.Get *
cmd/new/new.go: // persistent flag, we can get the value in subcommand via {{.Dot}}viper.Get{{.Dot}}
cmd/new/new.go: o.PersistentOption = viper.GetString("persistent")
cmd/set/set_db.go: // o.host = viper.GetString("host")
cmd/set/set_db.go: // o.username = viper.GetString("username")
cmd/set/set_db.go: // o.password = viper.GetString("password")
cmd/set/set_db.go: // o.Database = viper.GetString("database")
cmd/set/set_db.go: // o.drop = viper.GetBool("drop")
cmd/set/set_db.go: // o.admin = viper.GetBool("admin")
cmd/secret/user.go: // persistent flag, we can get the value in subcommand via `viper.Get`
cmd/util/factory.go: secretID := viper.GetString("secret-id")
cmd/util/factory.go: secretKey := viper.GetString("secret-key")
cmd/util/factory.go: timeout := viper.GetDuration("token-timeout")
cmd/util/helpers.go: Host: viper.GetString("server.address"),
cmd/util/helpers.go: BearerToken: viper.GetString("user.token"),
cmd/util/helpers.go: Username: viper.GetString("user.username"),
cmd/util/helpers.go: Password: viper.GetString("user.password"),
cmd/util/helpers.go: SecretID: viper.GetString("user.secret-id"),
cmd/util/helpers.go: SecretKey: viper.GetString("user.secret-key"),
cmd/util/helpers.go: Timeout: viper.GetDuration("server.timeout"),
cmd/util/helpers.go: MaxRetries: viper.GetInt("server.max-retries"),
cmd/util/helpers.go: RetryInterval: viper.GetDuration("server.retry-interval"),
cmd/validate/validate.go: target, err := url.Parse(viper.GetString("server.address"))
cmd/cmd.go: genericapiserver.LoadConfig(viper.GetString(genericclioptions.FlagIAMConfig), "config")
```

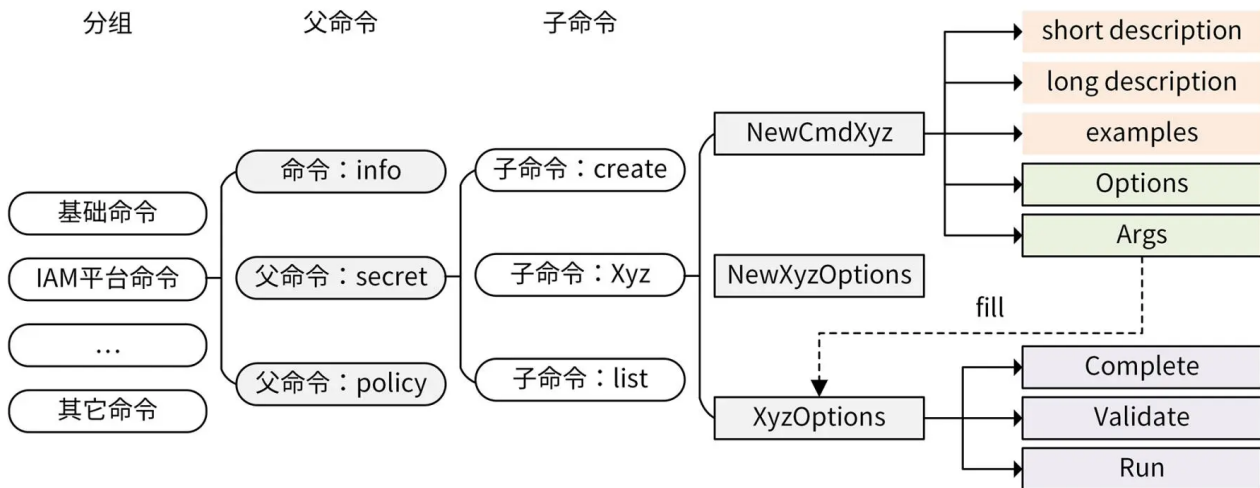
iamctl 中子命令是如何构建的？

讲完了 iamctl 命令行工具的核心实现，我们再来看看 iamctl 命令行工具中，子命令是如何构建的。

命令行工具的核心是命令，有很多种方法可以构建一个命令，但还是有一些比较好的构建方法，值得我们去参考。接下来，我来介绍下如何用比较好的方式去构建命令。

命令构建

命令行工具的核心能力是提供各类命令，来完成不同功能，每个命令构建的方式可以完全不同，但最好能按相同的方式去构建，并抽象成一个模型。如下图所示：



你可以将一个命令行工具提供的命令进行分组。每个分组包含多个命令，每个命令又可以具有多个子命令，子命令和父命令在构建方式上完全一致。

每个命令可以按下面的四种方式构建。具体代码你可以参考

[internal/iamctl/cmd/user/user_update.go](https://internal.iamctl/cmd/user/user_update.go)。

通过 NewCmdXyz 函数创建命令框架。NewCmdXyz 函数通过创建一个 cobra.Command 类型的变量来创建命令；通过指定 cobra.Command 结构体类型的 Short、Long、Example 字段，来指定该命令的使用文档 iamctl -h、详细使用文档 iamctl xyz -h 和使用示例。

通过 cmd.Flags().XxxxVar 来给该命令添加命令行选项。

为了在不指定命令行参数时，能够按照默认的方式执行命令，可以通过 `NewXYZOptions` 函数返回一个设置了默认选项的 `XYZOptions` 类型的变量。

`XYZOptions` 选项具有 `Complete`、`Validate` 和 `Run` 三个方法，分别完成选项补全、选项验证和命令执行。命令的执行逻辑可以在 `func (o *XYZOptions) Run(args []string) error` 函数中编写。

按相同的方式去构建命令，抽象成一个通用模型，这种方式有下面四个好处。

减少理解成本：理解一个命令的构建方式，就可以理解其他命令的构建方式。

提高新命令的开发效率：可以复用其他命令的开发框架，新命令只需填写业务逻辑即可。

自动生成命令：可以按照规定的命令模型，自动生成新的命令。

易维护：因为所有的命令都来自于同一个命令模型，所以可以保持一致的代码风格，方便后期维护。


自动生成命令

上面讲到，自动生成命令模型的好处之一是可以自动生成命令，下面让我们来具体看下。

`iamctl` 自带了命令生成工具，下面我们看看生成方法，一共可以分成 5 步。这里假设生成 `xyz` 命令。


第一步，新建一个 `xyz` 目录，用来存放 `xyz` 命令源码：

```
1 $ mkdir internal/iamctl/cmd/xyz
```

 复制代码

第二步，在 `xyz` 目录下，使用 `iamctl new` 命令生成 `xyz` 命令源码：

```
1 $ cd internal/iamctl/cmd/xyz/  
2 $ iamctl new xyz  
3 Command file generated: xyz.go
```

 复制代码

第三步，将 xyz 命令添加到 root 命令中，假设 xyz 属于 Settings Commands 命令分组。

在 NewIAMCtlCommand 函数中，找到 Settings Commands 分组，将 NewCmdXyz 追加到 Commands 数组后面：

[复制代码](#)

```
1      {
2          Message: "Settings Commands:",
3          Commands: []*cobra.Command{
4              set.NewCmdSet(f, ioStreams),
5              completion.NewCmdCompletion(ioStreams.Out, ""),
6              xyz.NewCmdXyz(f, ioStreams),
7          },
8      },
```

第四步，编译 iamctl：

[复制代码](#)

```
1 $ make build BINS=iamctl
```

第五步，测试：

[复制代码](#)

```
1 $ iamctl xyz -h
2 A longer description that spans multiple lines and likely contains examples an
3 example:
4
5 Cobra is a CLI library for Go that empowers applications. This application is
6 quickly create a Cobra application.
7
8 Examples:
9 # Print all option values for xyz
10 iamctl xyz marmotedu marmotedupass
11
12 Options:
13 -b, --bool=false: Bool option.
14 -i, --int=0: Int option.
15     --slice=[]: String slice option.
16     --string='default': String option.
17
```

```
18 Usage:
19   iamctl xyz USERNAME PASSWORD [options]
20
21 Use "iamctl options" for a list of global command-line options (applies to all
22 $ iamctl xyz marmotedu marmotedupass
23 The following is option values:
24 ==> --string: default(complete)
25 ==> --slice: []
26 ==> --int: 0
27 ==> --bool: false
28
29 The following is args values:
30 ==> username: marmotedu
31 --\ password: marmotedupass
```

你可以看到，经过短短的几步，就添加了一个新的命令 xyz。iamctl new 命令不仅可以生成不带子命令的命令，还可以生成带有子命令的命令，生成方式如下：

```
1 $ iamctl new -g xyz
2 Command file generated: xyz.go
3 Command file generated: xyz_subcmd1.go
4 Command file generated: xyz_subcmd2.go
```

[复制代码](#)

命令自动补全

cobra 会根据注册的命令自动生成补全脚本，可以补全父命令、子命令和选项参数。在 bash 下，可以按下面的方式配置自动补全功能。

第一步，生成自动补全脚本：

```
1 $ iamctl completion bash > ~/.iam/completion.bash.inc
```

[复制代码](#)

第二步，登陆时加载 bash，自动补全脚本：

```
1 $ echo "source '$HOME/.iam/completion.bash.inc'" >> $HOME/.bash_profile
2 $ source $HOME/.bash_profile
```

[复制代码](#)

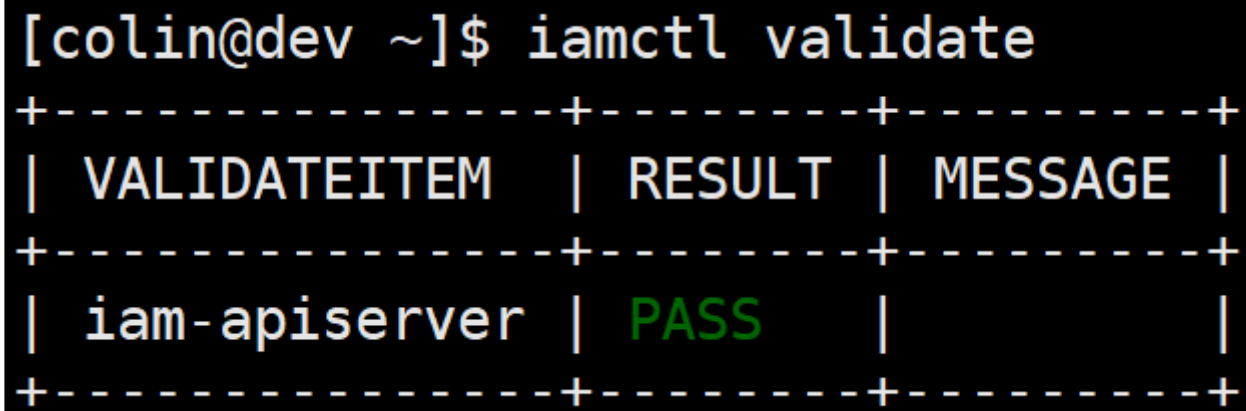
第三步，测试自动补全功能：

[复制代码](#)

```
1 $ iamctl xy<TAB> # 按TAB键，自动补全为：iamctl xyz
2 $ iamctl xyz --b<TAB> # 按TAB键，自动补全为：iamctl xyz --bool
```

更友好的输出

在开发命令时，可以通过一些技巧来提高使用体验。我经常会在输出中打印一些彩色输出，或者将一些输出以表格的形式输出，如下图所示：



```
[colin@dev ~]$ iamctl validate
+-----+-----+-----+
| VALIDATEITEM | RESULT | MESSAGE |
+-----+-----+-----+
| iam-apiserver | PASS   |         |
+-----+-----+-----+
```

这里，使用 github.com/olekukonko/tablewriter 包来实现表格功能，使用 github.com/fatih/color 包来打印带色彩的字符串。具体使用方法，你可以参考 [@internal/iamctl/cmd/validate/validate.go](#) 文件。

github.com/fatih/color 包可以给字符串标示颜色，字符串和颜色的对应关系可通过 `iamctl color` 来查看，如下图所示：

```
[colin@dev ~]$ iamctl color
```

CATEGORY	COLOR NAME	EFFECT
fg	black	
	red	color.RedString
	green	color.GreenString
	yellow	color.YellowString
	blue	color.BlueString
	magenta	color.MagentaString
	Cyan	color.CyanString
bg	white	color.WhiteString
	black	color.BgBlack
	red	color.BgRed
	greep	color.BgGreen
	yellow	color.BgYellow
	blue	color.BgBlue
	magenta	color.BgMagenta
	cyan	color.BgCyan
	white	color.BgWhite

iamctl 是如何进行 API 调用的？

上面我介绍了 iamctl 命令的构建方式，那么这里我们再来看下 iamctl 是如何请求服务端 API 接口的。

Go 后端服务的功能通常通过 API 接口来对外暴露，一个后端服务可能供很多个终端使用，比如浏览器、命令行工具、手机等。为了保持功能的一致性，这些终端都会调用同一套 API 来完成相同的功能，如下图所示：



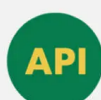
客户端

命令行工具

API调用

浏览器

移动端



服务端

如果命令行工具需要用到后端服务的功能，也需要通过 API 调用的方式。理想情况下，Go 后端服务对外暴露的所有 API 功能，都可以通过命令行工具来完成。一个 API 接口对应一个命令，API 接口的参数映射到命令的参数。

要调用服务端的 API 接口，最便捷的方法是通过 SDK 来调用，对于一些没有实现 SDK 的接口，也可以直接调用。所以，在命令行工具中，需要支持以下两种调用方式：

通过 SDK 调用服务端 API 接口。

直接调用服务端的 API 接口（本专栏是 REST API 接口）。

iamctl 通过 [cmdutil.NewFactory](#) 创建一个 Factory 类型的变量 f，Factory 定义为：

[复制代码](#)

```
1 type Factory interface {  
2     genericcliptions.RESTClientGetter  
3     IAMClientSet() (*marmotedu.Clientset, error)  
4     RESTClient() (*restclient.RESTClient, error)  
5 }
```

将变量 f 传入到命令中，在命令中使用 Factory 接口提供的 RESTClient() 和 IAMClientSet() 方法，分别返回 RESTful API 客户端和 SDK 客户端，从而使用客户端提供的接口函数。代码可参考 [internal/iamctl/cmd/version/version.go](#)。

客户端配置文件

如果要创建 RESTful API 客户端和 SDK 的客户端，需要调用 f.ToRESTConfig() 函数返回 *github.com/marmotedu/marmotedu-sdk-go/rest.Config 类型的配置变量，然后再基于 rest.Config 类型的配置变量创建客户端。

f.ToRESTConfig 函数最终是调用 [toRawIAMConfigLoader](#) 函数来生成配置的，代码如下：

[复制代码](#)

```
1 func (f *ConfigFlags) toRawIAMConfigLoader() clientcmd.ClientConfig {  
2     config := clientcmd.NewConfig()
```



```
3     if err := viper.Unmarshal(&config); err != nil {
4         panic(err)
5     }
6
7     return clientcmd.NewClientConfigFromConfig(config)
8 }
```

toRawIAMConfigLoader 返回 clientcmd.ClientConfig 类型的变量，clientcmd.ClientConfig 类型提供了 ClientConfig 方法，用来返回 *rest.Config 类型的变量。

在 toRawIAMConfigLoader 函数内部，通过 viper.Unmarshal 将 viper 中存储的配置解析到 clientcmd.Config 类型的结构体变量中。viper 中存储的配置，是在 cobra 命令启动时通过 LoadConfig 函数加载的，代码如下（位于 NewIAMCtlCommand 函数中）：

[复制代码](#)

```
1 cobra.OnInitialize(func() {
2     genericapiserver.LoadConfig(viper.GetString(genericclioptions.FlagIAMConf
3 })
```

你可以通过 --config 选项，指定配置文件的路径。

SDK 调用

通过 [IAMClient](#) 返回 SDK 客户端，代码如下：

[复制代码](#)

```
1 func (f *factoryImpl) IAMClient() (*iam.IamClient, error) {
2     clientConfig, err := f.ToRESTConfig()
3     if err != nil {
4         return nil, err
5     }
6     return iam.NewForConfig(clientConfig)
7 }
```

marmotedu.Clientset 提供了 iam-apiserver 的所有接口。

REST API 调用

通过 `RESTClient()` 返回 RESTful API 客户端，代码如下：

[复制代码](#)

```
1 func (f *factoryImpl) RESTClient() (*restclient.RESTClient, error) {
2     clientConfig, err := f.ToRESTConfig()
3     if err != nil {
4         return nil, err
5     }
6     setIAMDefaults(clientConfig)
7     return restclient.RESTClientFor(clientConfig)
8 }
```

可以通过下面的方式访问 RESTful API 接口：

[复制代码](#)

```
1 serverVersion *version.Info
2
3 client, _ := f.RESTClient()
4 if err := client.Get().AbsPath("/version").Do(context.TODO()).Into(&serverVers
5     return err
6 }
```

上面的代码请求了 iam-apiserver 的 /version 接口，并将返回结果保存在 serverVersion 变量中。

总结

这一讲，我主要剖析了 iamctl 命令行工具的实现，进而向你介绍了如何实现一个优秀的客户端工具。

对于一个大型系统 xxx 来说，通常需要有一个 xxxctl 命令行工具，xxxctl 命令行工具可以方便开发、运维使用系统功能，并能实现功能自动化。

IAM 项目参考 kubectl，实现了命令行工具 iamctl。iamctl 集成了很多功能，我们可以通过 iamctl 子命令来使用这些功能。例如，我们可以通过 iamctl 对用户、密钥和策略进行

CURD 操作；可以设置 iamctl 自动补全脚本；可以查看 IAM 系统的版本信息。甚至，你还可以使用 iamctl new 命令，快速创建一个 iamctl 子命令模板。

iamctl 使用了 cobra、pflag、viper 包来构建，每个子命令又包含了一些基本的功能，例如短描述、长描述、使用示例、命令行选项、选项校验等。iamctl 命令可以加载不同的配置文件，来连接不同的客户端。iamctl 通过 SDK 调用、REST API 调用两种方式来调用服务端 API 接口。

课后练习

1. 尝试在 iamctl 中添加一个 cliprint 子命令，该子命令会读取并打印命令行选项。
2. 思考下，还有哪些好的命令行工具构建方式，欢迎在留言区分享。

欢迎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 34 | SDK 设计（下）：IAM 项目 Go SDK 设计和实现

下一篇 36 | 代码测试（上）：如何编写 Go 语言单元测试和性能测试用例？

专栏上新

陈天 · Rust 编程第一课

实战驱动，快速上手 Rust

早鸟优惠 **¥99** 原价¥129



精选留言 (4)

写留言



Sch0ng

2021-08-17

命令行客户端工具是大型项目交互和管理的一部分。
又学到了一个最佳实践，赞👍



我来也

2021-08-14

有新的发现：

ko 自动生成的代码中，嵌套了中括号，就会导致补全失败。

ko completion zsh 生成的代码是这样：

```
'--platform[Which platform to use when pulling a multi-platform base. Format: a...
```

展开▼

作者回复: 牛批！



1



我来也

2021-08-14

老师的项目真全，连 ctl 都有了。😄

本地一个 make 命令，就构建出来了对应架构的可执行文件。

有个地方比较好奇，咨询一下老师：

老师的 zsh 补全脚本是如何生成的，为什么是在代码中定义的常量？...

展开▼

作者回复: zsh脚本是cobra框架自动生成的。如果生成的zsh自动补全脚本有问题，可能说明没有正确使用cobra或者cobra有问题，提议google下



1



pedro

2021-08-14

cool!

展开

