链滴

go-micro 动态加载插件源码分析

作者：Allenxuxu

原文链接：https://ld246.com/article/1585471187985

来源网站：链滴

go-micro 框架支持动态加载插件，无需修改代码。

源码分析

启动服务前，设定 MICRO_PLUGIN 环境变量指定动态库 .so 文件路径，支持多个插件，逗号分割。序启动前会读取 MICRO_PLUGIN 环境变量，并完成插件设定。

下面是其内部实现：

go-micro/service.go

```
func (s *service) Init(opts ...Option) {
...

    // setup the plugins
    for _, p := range strings.Split(os.Getenv("MICRO_PLUGIN"), ",") {
        if len(p) == 0 {
            continue
        }

        // 加载 .so 文件
        c, err := plugin.Load(p)
        if err != nil {
            logger.Fatal(err)
        }

        // go-micro 初始化插件
        if err := plugin.Init(c); err != nil {
            logger.Fatal(err)
        }
    }
```

从上面的代码可以看出，service 初始化化的时候，读取 MICRO_PLUGIN 环境变量中指定的 .so 文路径。并且调用 plugin 包，逐个 Init 。

下面我们看下 plugin 包的实现：

```
plugin
├── default.go
├── plugin.go
└── template.go

0 directories, 3 files
```

plugin 包的实现非常简单，只有三个文件。

go-micro/plugin/plugin.go

```
// Plugin is a plugin loaded from a file
type Plugin interface {
    // Initialise a plugin with the config
    Init(c *Config) error
    // Load loads a .so plugin at the given path
    Load(path string) (*Config, error)
```

```
    // Build a .so plugin with config at the path specified
    Build(path string, c *Config) error
}
```

plugin 包定义了这样一个接口

- Init(c *Config) error 方法用来注册插件；
- Load(path string) (*Config, error) 用来加载一个 .so 文件，并返回一个 Config ；
- Build(path string, c *Config) error 用来根据指定的 Config 变量生成一个 .so 文件。

go-micro 提供了一个默认的实现，在 go-micro/plugin/default.go 。

先来看一下默认实现的 Load 方法：

```
import (
//...
    pg "plugin"
//...
)

// Load loads a plugin created with `go build -buildmode=plugin`
func (p *plugin) Load(path string) (*Config, error) {
    // 调用标准库打开 .so 文件
    plugin, err := pg.Open(path)
    if err != nil {
        return nil, err
    }
    // 在加载成功的动态库文件中寻找 Plugin 变量/函数
    s, err := plugin.Lookup("Plugin")
    if err != nil {
        return nil, err
    }
    // 类型转换成 go-micro 定义的 Config 类型指针
    pl, ok := s.(*Config)
    if !ok {

        return nil, errors.New("could not cast Plugin object")
    }
    return pl, nil
}
```

Load 方法主要就是调用标准库 plugin open 一个 .so 文件，然后寻找 Plugin 这个变量，并通过类断言它转换成 *Config 。Config 是 go-micro 定义的一个类型：

```
// Config is the plugin config
type Config struct {
    // Name of the plugin e.g rabbitmq
    Name string
    // Type of the plugin e.g broker
    Type string
    // Path specifies the import path
    Path string
    // NewFunc creates an instance of the plugin
    NewFunc interface{}
```

```
}
```

关于标准库 plugin 的用法，这里不再描述可以查看源码文件，里面有用法说明。需要特别说明的一是，标准库 plugin 的 Lookup 方法返回 Symbol 类型，它可以类型转换成一个函数或者指向变量的针。

我们继续看 go-micro 的 Init 方法实现：

```
// Init sets up the plugin
func (p *plugin) Init(c *Config) error {
    switch c.Type {
    case "broker":
        pg, ok := c.NewFunc.(func(...broker.Option) broker.Broker)
        if !ok {
            return fmt.Errorf("Invalid plugin %s", c.Name)
        }
        cmd.DefaultBrokers[c.Name] = pg
    case "client":
        pg, ok := c.NewFunc.(func(...client.Option) client.Client)
        if !ok {
            return fmt.Errorf("Invalid plugin %s", c.Name)
        }
        cmd.DefaultClients[c.Name] = pg
    case "registry":
        pg, ok := c.NewFunc.(func(...registry.Option) registry.Registry)
        if !ok {
            return fmt.Errorf("Invalid plugin %s", c.Name)
        }
        cmd.DefaultRegistries[c.Name] = pg
    // .... 省略一些 case
    default:
        return fmt.Errorf("Unknown plugin type: %s for %s", c.Type, c.Name)
    }

    return nil
}
```

这个函数是 micro 实现动态加载的重点，Init 函数通过 Load 方法返回的 Config 变量进行选择，然通过类型转换得到 对应的构建函数赋值给 go-micro 的 cmd 包里的全局变量 DefaultXXXs 。

go-micro/config/cmd/cmd.go

```
    DefaultBrokers = map[string]func(...broker.Option) broker.Broker{
        "service": brokerSrv.NewBroker,
        "memory":  memory.NewBroker,
        "nats":    nats.NewBroker,
    }

    DefaultClients = map[string]func(...client.Option) client.Client{
        "mucp": cmucp.NewClient,
        "grpc": cgrpc.NewClient,
    }
```

以 DefaultClients 为例，假设我们实现了一个 client 插件(需要实现 go-micro 的 client.Client 接口)并实现了自己的创建函数 xrpc.NewClient 。那加载插件成功后， DefaultClients 变量就是

```
map[string]func(...client.Option) client.Client{
        "mucp": cmucp.NewClient,
        "grpc": cgrpc.NewClient,
        "xrpc": xrpc.NewClient,
    }
```

在 cmd 对象的 Before 方法中会根据程序启动时传入的参数来选择对应的插件。

go-micro/config/cmd/cmd.go

```
// Set the client
if name := ctx.String("client"); len(name) > 0 {
        // only change if we have the client and type differs
    if cl, ok := c.opts.Clients[name]; ok && (*c.opts.Client).String() != name {
        *c.opts.Client = cl()
    }
}
```

```
GLOBAL OPTIONS:
--client value              Client for go-micro; rpc [$MICRO_CLIENT]
```

如果启动程序时设定了 client=xrpc ，必须记得设定 MICRO_PLUGIN 环境变量指定动态库。这里定的 c.opts.Client 会被 micro 服务所使用，可以一步一步向上追溯，这里就不追踪了。

上面的源码分析中，我们没有看 go-micro plugin 包的 Build 方法默认实现，现在我们来看一下：

```
// Build generates a dso plugin using the go command `go build -buildmode=plugin`
func (p *plugin) Build(path string, c *Config) error {
    path = strings.TrimSuffix(path, ".so")

    // 在 tmp 目录创建一个临时go源码文件
    temp := os.TempDir()
    base := filepath.Base(path)
    goFile := filepath.Join(temp, base+".go")

    // 根据模版生成 go 代码到文件中
    if err := p.Generate(goFile, c); err != nil {
        return err
    }
    // defer 函数执行完成时候删除这个临时go源码文件
    defer os.Remove(goFile)

    if err := os.MkdirAll(filepath.Dir(path), 0755); err != nil && !os.IsExist(err) {
        return fmt.Errorf("Failed to create dir %s: %v", filepath.Dir(path), err)
    }
    // 将这个文件编译成动态库
    cmd := exec.Command("go", "build", "-buildmode=plugin", "-o", path+".so", goFile)
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    return cmd.Run()
}
```

可以看出，主要就是更具传入的文件路径，创建目录，创建一个临时的go文件，然后调用 go build -uildmode=plugin 生成动态库。

这里调用的一个 Generate 方法，这个方法通过 go 模版生成 go 文件。

```
// Generate creates a go file at the specified path.
// You must use `go build -buildmode=plugin`to build it.
func (p *plugin) Generate(path string, c *Config) error {
    f, err := os.Create(path)
    if err != nil {
        return err
    }
    defer f.Close()
    t, err := template.New(c.Name).Parse(tmpl)
    if err != nil {
        return err
    }
    return t.Execute(f, c)
}

// ...

var (
    tmpl = `
package main

import (
    "github.com/micro/go-micro/v2/plugin"

    "{{.Path}}"
)

var Plugin = plugin.Config{
    Name: "{{.Name}}",
    Type: "{{.Type}}",
    Path: "{{.Path}}",
    NewFunc: {{.Name}}.{{.NewFunc}},
}
`

)
```

根据模版生成 go 文件中会有一个全局变量 Plugin，这也印证了 Load 方法中的 plugin.Lookup("Plugin") 。

简单使用

```
package main

import (
    "fmt"
    "github.com/micro/go-micro/v2/plugin"
)

func main() {
    p := plugin.NewPlugin()
    if err := p.Build("/tmp/test.so", &plugin.Config{
        Name:    "client",
```

```
        Type:    "client",
        Path:    "github.com/micro/go-micro/v2/client",
        NewFunc: "NewClient",
    }); err != nil {
        panic(err)
    }

    c, err := plugin.Load("/tmp/test.so")
    if err != nil {
        panic(err)
    }

    fmt.Println(c.Name, c.Type, c.Path, c.NewFunc)
}
```

上面的例子，主要是使用了 go-micro 的 plugin 包。先生成了 /tmp/test.so ，然后在 Load 这个动库，打印 config 的内容。

当然也可以，不通过 plugin.Build 生成动态库，直接手写一个 go 文件，手动编译成动态库。

test.go

```
package main

import (
    "github.com/micro/go-micro/v2/client"

    "github.com/micro/go-micro/v2/plugin"
)

var Plugin = plugin.Config{
    Name:    "test",
    Type:    "client",
    Path:    "github.com/micro/go-micro/v2/client",
    NewFunc: client.NewClient,
}


go build -buildmode=plugin -o ./test.so test.go
```

go-micro 动态加载的主要场景

假设我们的 micro 服务 client 使用的是 grpc 的形式，现在希望改成 brpc 的形式。go-micro 支持的 client 插件中并不包含 brpc，我们自己使用 brpc 实现一个 client 插件，然后将其编译成动态库。

在运行环境 MICRO_PLUGIN 变量指定动态库路径，并且修改程序的启动命令，指定 client=brpc 。样就可以做到无需重新编译二进制，替换自己想要的插件。

当然，自己在代码中重新 import 自己实现的插件库，显示指定 client 也是可以的。这样还可以将变纳入版本管理，也是极好的。