五、Micro

Micro的介绍

Micro解决了构建云本地系统的关键需求。它采用了微服务体系结构模式，并将其转换为一组工具，作为可伸缩平台的构建块。Micro隐藏了分布式系统的复杂性，并为开发人员提供了很好的理解概念。

Micro是一个专注于简化分布式系统开发的微服务生态系统。是一个工具集合, 通过将微服务架构抽象成一组工具。隐藏了分布式系统的复杂性，为开发人员提供了更简洁的概念。

micro的安装

下载micro

```
$ go get -u -v github.com/go-log/log
$ go get -u -v github.com/gorilla/handlers
$ go get -u -v github.com/gorilla/mux
$ go get -u -v github.com/gorilla/websocket
$ go get -u -v github.com/mitchellh/hashstructure
$ go get -u -v github.com/nlopes/slack
$ go get -u -v github.com/pborman/uuid
$ go get -u -v github.com/pkg/errors
$ go get -u -v github.com/serenize/snaker
# hashicorp_consul.zip包解压在github.com/hashicorp/consul
$ unzip hashicorp_consul.zip -d github.com/hashicorp/consul
# miekg_dns.zip 包解压在github.com/miekg/dns
$ unzip miekg_dns.zip -d github.com/miekg/dns
$ go get github.com/micro/micro
```

编译安装micro

```
$ cd $GOPATH/src/github.com/micro/micro
$ go build   -o micro  main.go
$ sudo cp micro /bin/
```

插件安装

```
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
go get -u github.com/micro/protoc-gen-micro
```

micro基本演示

创建微服务命令说明

```
new     Create a new Micro service by specifying a directory path relative to your $GOPATH
#创建 通过指定相对于$GOPATH的目录路径，创建一个新的微服务。

USAGE:
#用法
micro new [command options][arguments...]

--namespace "go.micro"  Namespace for the service e.g com.example
                        #服务的命名空间
--type "srv"            Type of service e.g api, fnc, srv, web
                        #服务类型
--fqdn                  FQDN of service e.g com.example.srv.service (defaults to namespace.type.alias)
                        #服务的正式定义全面
--alias                 Alias is the short name used as part of combined name if specified

                        #别名是在指定时作为组合名的一部分使用的短名称
run     Run the micro runtime
#运行 运行这个微服务时间
```

创建2个服务

```
$micro new --type "srv" micro/rpc/srv
#"srv" 是表示当前创建的微服务类型
#sss是相对于go/src下的文件夹名称 可以根据项目进行设置
#srv是当前创建的微服务的文件名
Creating service go.micro.srv.srv in /home/itcast/go/src/micro/rpc/srv

.
#主函数
├── main.go
#插件
├── plugin.go
#被调用函数
├── handler
|   └── example.go
#订阅服务
├── subscriber
|   └── example.go
```

```
#proto协议
├── proto/example
|   └── example.proto
#docker生成文件
├── Dockerfile
├── Makefile
└── README.md


download protobuf for micro:

brew install protobuf
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
go get -u github.com/micro/protoc-gen-micro

compile the proto file example.proto:

cd /home/itcast/go/src/micro/rpc/srv
protoc --proto_path=. --go_out=. --micro_out=. proto/example/example.proto

#使用创建srv时给的protobuf命令保留用来将proto文件进行编译

micro new --type "web" micro/rpc/web
Creating service go.micro.web.web in /home/itcast/go/src/micro/rpc/web
.
#主函数
├── main.go
#插件文件
├── plugin.go
#被调用处理函数
├── handler
|   └── handler.go
#前端页面
├── html
|   └── index.html
#docker生成文件
├── Dockerfile
├── Makefile
└── README.md

#编译后将web端呼叫srv端的客户端连接内容修改为srv的内容
#需要进行调通
```

启动consul进行监管

```
consul agent -dev
```

对srv服务进行的操作

```
#根据提示将proto文件生成为.go文件
cd /home/itcast/go/src/micro/rpc/srv
protoc --proto_path=. --go_out=. --micro_out=. proto/example/example.proto
#如果报错就按照提示将包进行下载
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
go get -u github.com/micro/protoc-gen-micro
#如果还不行就把以前的包删掉从新下载
```

对web服务进行的操作

main文件

```
package main

import (
    "github.com/micro/go-log"
    "net/http"
    "github.com/micro/go-web"
    "micro/rpc/web/handler"
)

func main() {
    // 创建1个web服务
        service := web.NewService(
           //注册服务名
                web.Name("go.micro.web.web"),
            //服务的版本号
                web.Version("latest"),
                //! 添加端口
                web.Address(":8080"),
        )

    //服务进行初始化
        if err := service.Init(); err != nil {
                log.Fatal(err)
        }

    //处理请求  / 的路由   //当前这个web微服务的 html文件进行映射
    service.Handle("/", http.FileServer(http.Dir("html")))

    //处理请求 /example/call  的路由   这个相应函数 在当前项目下的handler
    service.HandleFunc("/example/call", handler.ExampleCall)

    //运行服务
        if err := service.Run(); err != nil {
                log.Fatal(err)
        }
}

```

将准备好的html文件替换掉原有的文件

handler文件

```
package handler

import (
    "context"
    "encoding/json"
    "net/http"
    "time"

    "github.com/micro/go-micro/client"
    //将srv中的proto的文件导入进来进行通信的使用
    example "micro/rpc/srv/proto/example"
)
//相应请求的业务函数
func ExampleCall(w http.ResponseWriter, r *http.Request) {
    // 将传入的请求解码为json
    var request map[string]interface{}
    if err := json.NewDecoder(r.Body).Decode(&request); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }

    // 调用服务
    //替换掉原有的服务名
    //通过服务名和
    exampleClient := example.NewExampleService("go.micro.srv.srv",
client.DefaultClient)
    rsp, err := exampleClient.Call(context.TODO(), &example.Request{
        Name: request["name"].(string),
    })
    if err != nil {
        http.Error(w, err.Error(), 500)
        return
    }

    // we want to augment the response
    response := map[string]interface{}{
        "msg": rsp.Msg,
        "ref": time.Now().UnixNano(),
    }

    // encode and write the response as json
    if err := json.NewEncoder(w).Encode(response); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }
}

```

升级成为grpc的版本

重新生成proto文件

srv的main.go

```go
package main

import (
    "github.com/micro/go-log"
    "github.com/micro/go-micro"
    "micro/grpc/srv/handler"
    "micro/grpc/srv/subscriber"
    example "micro/grpc/srv/proto/example"
    "github.com/micro/go-grpc"
)

func main() {
    // 创建新服务

    service := grpc.NewService(
        //当前微服务的注册名
        micro.Name("go.micro.srv.srv"),
        //当前微服务的版本号
        micro.Version("latest"),
    )

    // 初始化服务
    service.Init()

    // Register Handler
    //通过protobuf的协议注册我们的handler
    //参数1是我们创建好的服务返回的句柄
    //参数2 使我们new的handler包中的类
    example.RegisterExampleHandler(service.Server(), new(handler.Example))

    // Register Struct as Subscriber
    //注册结构体来自于Subscriber
    micro.RegisterSubscriber("go.micro.srv.srv", service.Server(),
new(subscriber.Example))

    // Register Function as Subscriber
    // 注册函数自于Subscriber
    micro.RegisterSubscriber("go.micro.srv.srv", service.Server(),
subscriber.Handler)

    // Run service
    if err := service.Run(); err != nil {
        log.Fatal(err)
    }
```

```
}

```

srv的example.go

```
package handler

import (
    "context"

    "github.com/micro/go-log"
    //更换了相关proto文件
    example "micro/grpc/srv/proto/example"
)

type Example struct{}

// Call is a single request handler called via client.Call or the generated client code
func (e *Example) Call(ctx context.Context, req *example.Request, rsp *example.Response) error {
    log.Log("Received Example.Call request")
    rsp.Msg = "Hello " + req.Name
    return nil
}

// Stream is a server side stream handler called via client.Stream or the generated client code
//流数据的检测操作
func (e *Example) Stream(ctx context.Context, req *example.StreamingRequest, stream example.Example_StreamStream) error {
    log.Logf("Received Example.Stream request with count: %d", req.Count)

    for i := 0; i < int(req.Count); i++ {
        log.Logf("Responding: %d", i)
        if err := stream.Send(&example.StreamingResponse{
            Count: int64(i),
        }); err != nil {
            return err
        }
    }

    return nil
}

// PingPong is a bidirectional stream handler called via client.Stream or the generated client code
//心跳检测机制
func (e *Example) PingPong(ctx context.Context, stream example.Example_PingPongStream) error {
    for {
        req, err := stream.Recv()
        if err != nil {
```

```
            return err
        }
        log.Logf("Got ping %v", req.Stroke)
        if err := stream.Send(&example.Pong{Stroke: req.Stroke}); err != nil {
            return err
        }
    }
}

```

修改web的main.go

```
package main
import (
        "github.com/micro/go-log"
    "net/http"

        "github.com/micro/go-web"
        "micro/grpc/web/handler"
)
func main() {
    // create new web service
        service := web.NewService(
                web.Name("go.micro.web.web"),
                web.Version("latest"),
                web.Address(":8080"),
        )
    // initialise service
        if err := service.Init(); err != nil {
                log.Fatal(err)
        }

    // register html handler
    service.Handle("/", http.FileServer(http.Dir("html")))
    // register call handler
    service.HandleFunc("/example/call", handler.ExampleCall)
    // run service
        if err := service.Run(); err != nil {
                log.Fatal(err)
        }
}

```

修改web的handler.go

```
package handler

import (
    "context"
    "encoding/json"
    "net/http"
    "time"

```

```
    example "micro/grpc/srv/proto/example"
    "github.com/micro/go-grpc"
)

func ExampleCall(w http.ResponseWriter, r *http.Request) {

    server :=grpc.NewService()
    server.Init()

    // decode the incoming request as json
    var request map[string]interface{}
    if err := json.NewDecoder(r.Body).Decode(&request); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }
    // call the backend service
    //exampleClient := example.NewExampleService("go.micro.srv.srv",
client.DefaultClient)
    //通过grpc的方法创建服务连接返回1个句柄
    exampleClient := example.NewExampleService("go.micro.srv.srv", server.Client())
    rsp, err := exampleClient.Call(context.TODO(), &example.Request{
        Name: request["name"].(string),
    })
    if err != nil {
        http.Error(w, err.Error(), 500)
        return
    }
    // we want to augment the response
    response := map[string]interface{}{
        "msg": rsp.Msg,
        "ref": time.Now().UnixNano(),
    }
    // encode and write the response as json
    if err := json.NewEncoder(w).Encode(response); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }
}
```

关于插件化

Go Micro跟其他工具最大的不同是它是插件化的架构，这让上面每个包的具体实现都可以切换出去。举个例子，默认的服务发现的机制是通过Consul，但是如果想切换成etcd或者zookeeper 或者任何你实现的方案，都是非常便利的。