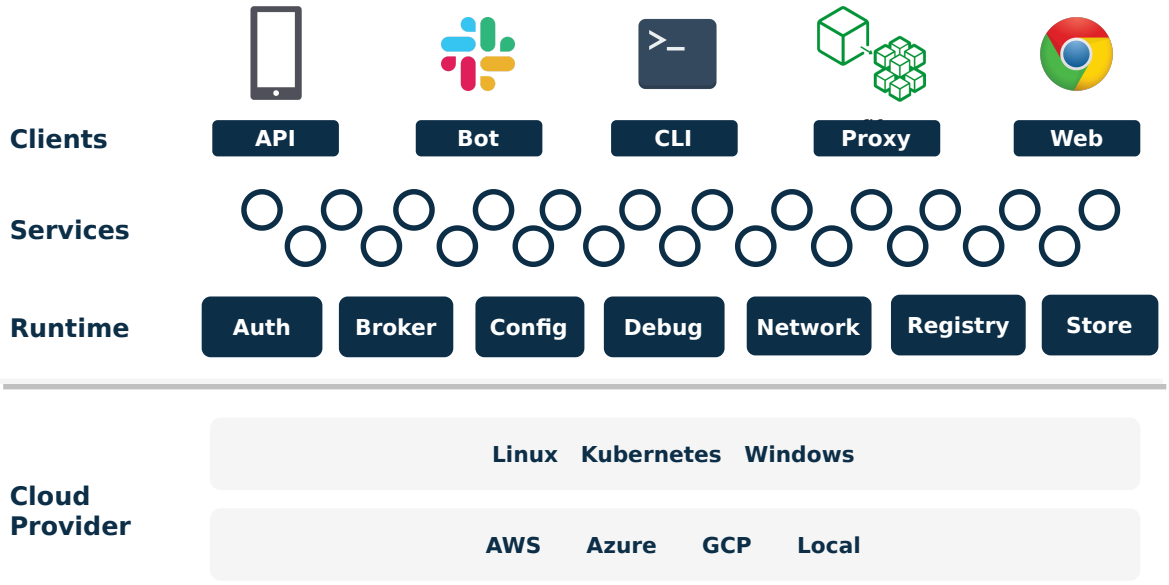
目　录

介绍

介绍

此专栏文档转自taadis编写的https://learnku.com/docs/go-micro/2.x

Micro 解决了在云内外构建分布式系统的关键要求. 它利用微服务体系结构模式, 并提供一组作为平台构建基块的服务. Micro 处理分布式系统的复杂性, 并提供更简单的可编程抽象.



技术在不断发展. 基础结构堆栈总是在变化. Micro 是一个平台, 它使用可插拔的基础和强定义的 api 来解决这些问题. 可插入任何堆栈或云.

在推特上关注我们, 或加入 Slack 社区.

特性

运行时由以下功能组成:

- **api:** api 网关. 使用服务发现具有动态请求路由的单个入口点. API 网关允许您在后端构建可扩展的微服务体系结构, 并在前端合并公共 api. micro api 通过发现和可插拔处理程序提供强大的路由, 为 http, grpc, Websocket, 发布事件等提供服务.
- **broker:** 允许异步消息的消息代理. 微服务是事件驱动的体系结构, 应该作为一等公民提供消息传递. 通知其他服务的事件, 而无需担心响应.

- **network:** 通过微网络服务构建多云网络. 只需跨任何环境连接网络服务, 创建单个平面网络即可全局路由. Micro 的网络根据每个数据中心中的本地注册表动态构建路由, 确保根据本地设置路由查询.
- **new:** 服务模板生成器. 创建新的服务模板以快速入门. Micro 提供用于编写微服务的预定义模板. 始终以相同的方式启动, 构建相同的服务以提高工作效率.
- **proxy:** 建立在 Go Micro 上的透明服务代理. 将服务发现, 负载平衡, 容错, 消息编码, 中间件, 监视等卸载到单个位置. 独立运行它或与服务一起运行.
- **registry:** 注册表提供服务发现以查找其他服务, 存储功能丰富的元数据和终结点信息. 它是一个服务资源管理器, 允许您在运行时集中和动态地存储此信息.
- **store:** 有状态是任何系统的必然需求. 我们提供密钥值存储, 提供简单的状态存储, 可在服务之间共享或长期卸载m 以保持微服务无状态和水平可扩展.
- **web:** Web 仪表板允许您浏览服务, 描述其终结点, 请求和响应格式, 甚至直接查询它们. 仪表板还包括内置 CLI 的体验, 适用于希望动态进入终端的开发人员.

此外, Micro 还提供了 Go 开发框架:

- **go-micro:** 利用强大的 Go Micro 框架轻松, 快速地开发微服务. Go Micro 抽象了分布式系统的复杂性, 并提供了更简单的抽象来构建高度可扩展的微服务.

开始

- **框架** - 使用 go-micro 框架开始编写服务.
- **运行时** - 使用 micro 运行时管理您的服务.
- **网络** - 在网络上 公开共享和托管您的服务.
- **示例** - 通过实际代码 示例 示例来学习
- **社区** - 加入社区并在 slack 开始协作

概述

常见问题

常见问题

常见问题应该提供大多数问题的的快速解答.

什么是 Micro?

Micro 使开发人员能够构建, 共享和协作分布式系统, 即微服务.

- Micro 是一个 框架
- Micro 是一个 运行时
- Micro 是一个 网络
- Micro 是一个 社区

开源

Micro 构建为开源库和工具, 以帮助微服务开发.

- **框架** - 编写微服务的 Go 框架; 服务发现, 远程过程调用, 发布/订阅等.
- **运行时** - 微服务运行时环境; API 网关, cli, slackbot, service proxy, 等.
- **插件** - 框架和运行时的插件, 包括 etcd, kubernets, nats, grpc, 等.

可以在 github.com/micro 找到所有工具.

网络

Micro 为共享和运行服务提供了开放的全球始终在线的云环境. 任何人都可以加入和使用 网络. 这是在任何组织或团队之外开启公共协作的好方法.

探索 网络.

社区

有一个 slack 的社区, 有成千上万的成员。

邀请你自己加入 micro.mu/slack/

我该从哪里开始?

从 go-micro 开始. readme 提供了一个简单的示例微服务.

通过阅读 入门 指南或检出 示例 来了解更多信息.

使用 micro 工具包通过 cli, web ui, slack 或 api 网关访问微服务.

谁在使用 Micro?

查看 用户 页面, 其中列出了使用 Micro 的公司(但请注意, 更新存在滞后所以可能不是最新的).

还有更多人也在使用它, 但尚未公开. 如果您使用的是 Micro, 请随时添加您的公司.

如何使用 Micro?

很简单

1. 使用 go-micro 编写服务.
2. 通过 micro 工具包访问它们.
3. 利润.

查看完整的 问候者 示例.

零依赖的服务发现

多播 DNS 是内置注册的用于零配置的服务发现机制.

你不需要做任何事情! 它刚刚内置, 默认情况下已启用.

我可以使用某些内容而不是 MDNS 吗?

是的! 服务发现注册表是完全可插拔的. Etcd 也包含在默认插件中.

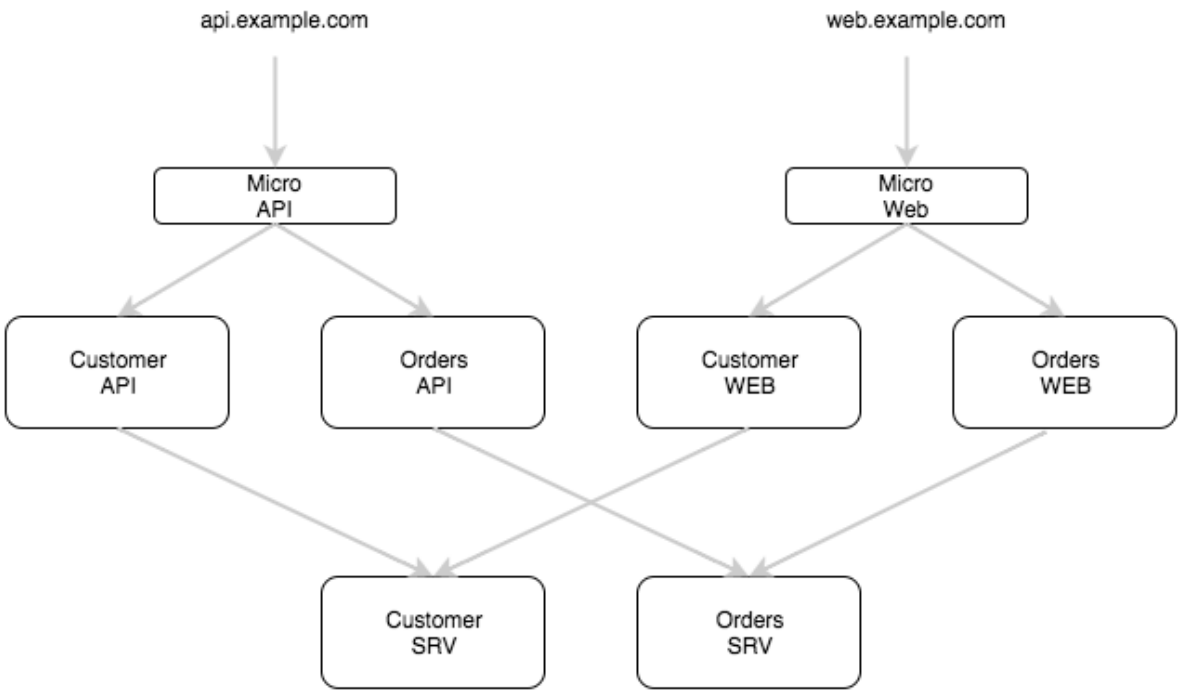
```
MICRO_REGISTRY=etcd MICRO_REGISTRY_ADDRESS=127.0.0.1:2379 myservice
```

我在哪里可以运行 Micro?

Micro 是运行时无感知的. 你可以在你喜欢的任何地方运行它. 在裸机上, 在 AWS 上, 使用 Google 云. 在你最喜欢的容器编排系统上, 如 Mesos 或 Kubernetes.

事实上, 在 Kubernetes 有 Micro 的演示配置. 可参阅此仓库 github.com/micro/kubernetes

API, Web 和 SRV 服务之间的区别是什么?



作为 micro 工具包的一部分, 我们尝试通过分离 API, Web 仪表板和后端服务 (SRV） 的关注点, 为可扩展体系结构定义一组设计模式.

API 服务

API 服务是一个单独的服务层, 用于为 http/json API 提供服务. 这可能是您的客户面向公众的 api 或网络. API 服务由 micro api 提供, 并符合 api 网关模式. 它们编写为标准 Go Micro 服务, 但使用命名空间 `go.micro.api` 在逻辑上分离自身.

Web 服务

Web 服务由默认命名空间 go.micro.web 的 micro web 提供服务. 我们相信 Web 应用程序是微服务领域的一等公民, 因此将 Web 仪表板构建为微服务. Micro Web 是反向代理, 将根据服务解析路径将 HTTP 请求转发到相应的 Web 应用.

想要了解更多, 可参阅 这里

SRV 服务

SRV 服务基本上是标准的 RPC 服务, 即您通常编写的服务类型. 我们通常称他们为 RPC 或后端服务, 因为它们主要应该是后端体系结构的一部分, 并且永远不会面向公共. 默认情况下, 我们使用命名空间 go.micro.srv 进行这些操作, 但您应该使用域 com.example.srv.

性能如何?

性能不是 Micro 的当前焦点, 但始终是我们努力的目标. 我们让更广泛的 Go 生态系统处理繁重的工作, 并首先关注开发人员的工作效率. 这意味着我们在底层使用 gRPC 在和 NATS 进行消息传递.

如果默认插件的性能不够, 您可以使用 go-plugins 切换到您选择的工具.

Micro 支持 gRPC 吗?

是的. 在 v2 中, 默认情况下使用 gRPC.

Micro vs Go-Kit

有很多人问过这个问题. micro 和 go-kit 之间的区别是什么?

Go-kit 将自己描述为微服务的标准库. 与 Go 一样, go-kit 还为您提供可用于构建应用程序的单个包. Go-kit 非常适合完全控制如何定义服务.

Go Micro 是微服务的可插拔 RPC 框架. 这是一个有意见的框架, 它试图简化分布式系统的通信方面, 以便您可以专注于业务逻辑本身. Go-micro 非常适合快速启动和运行, 同时拥有可插入的内容, 无需代码更改即可切换出基础结构.

Micro 是一个微服务工具包. 它就像一把用于微服务的瑞士军刀, 它建立在 go-micro 的基础上, 提供传统的入口点, 如 http api 网关, web ui, cli, slack bot, 等. Micro 使用工具来指导体系结构中关注点的逻辑分离, 推动您为公共 API 创建微服务的 API 层, 并单独为 Web UI 创建 WEB 微服务层.

在需要完全控制的地方使用 go-kit. 在需要有意见的框架的地方使用 go-micro.

我在哪里可以了解更多信息?

- 加入 slack 社区 - micro.mu/slack
- 阅读博客 - micro.mu/blog

特性

特性

Micro 的生态系统发展迅速, 但仍有很多工作要做.

功能页保持对最重要或值得注意功能的跟踪.

- 框架 用来编写服务
- 运行时 用来管理服务
- 网络 用来共享服务
- 示例 用来学习
- 插件 用来扩展工具

框架

Go Micro 是微服务开发的框架

- 抽象远离分布式系统
- 服务发现, 远程过程调用, 发布订阅, 消息编码
- 带超时, 重试和负载平衡的容错
- 通过包装器扩展功能
- 可交换后端技术的可插拔接口

运行

Micro 是管理服务的运行时环境

- API 网关作为单个 http 入口点
- Web 仪表板用于可视化服务
- 用于命令行访问的 CLI
- 通过 Slack 或 HipChat 查询的机器人
- 全新模板生成可快速启动

网络

网络 是共享和协作服务的地方

- 开放全球服务网络
- 多云和多区域
- 无麻烦, 无基础架构管理

插件

Go 插件 是社区驱动插件的场所

- 用于 go-micro/micro 的插件
- 包括最流行的后端技术
- grpc, kubernetes, 等, kafka, 等
- 已在生产中测试过的实践

部署

随时随地运行服务

- 本地
- Docker
- Kubernetes

用户

用户

如果你使用 Micro, 可随时添加您的公司到这里

公司

- AdCamie - 社交媒体广告解决方案
- Agora.io – 实时视频, 音频和实时交互式广播通信 API 解决方案
- Ankr - 分布式云计算网络
- Arex - 自动应收账款结算
- BetMakers - 赛马和体育信息的数据聚合
- Bottos - 分散的 AI 网络
- Capital One - 金融服务
- CESNET - 捷克教育和研究网络
- CodersRank - 开源贡献的实时配置文件
- Coesia - 工业和包装解决方案
- Dark Cubed - 网络安全服务
- Elumium - 租户通信
- Glue - 数字开发机构
- Hightech - 高科技媒体平台
- Honeywell - 生产商业和消费品, 工程服务和航空航天系统
- HuanLian（焕链科技） - 区块链技术
- Human Ventures - 总部位于纽约市的创业工作室
- Imgur - 社交形象共享平台
- Jumia - 食品配送服务
- Karhoo - 城市移动市场
- Kazoup - 文件管理平台
- Kyperion - 健康与健身训练平台
- Kytra - 投资简化
- Mohu - 区块链融资解决方案

- MTS Bank - 俄罗斯排名前 20 的银行
- Neds - 在线投注网站
- OnSky - 智能家居和办公室安全
- OpenSDS - 开放式自主数据平台
- ownCloud - 个人云协作平台
- PaySuper - 用于游戏开发者的支付
- Popsa - 照片书印刷服务
- Profects - 软件工作室
- Pydio - 文件共享平台
- Reserve - 餐厅预订服务
- Ridygo - 为您短途旅行的实时拼车
- Riverbed - 软件定义的网络
- Routable - 车队管理 API
- Sipsynergy - 电信服务
- Sixt - 汽车租赁企业
- Stillwater Supercomputing - 知识处理系统
- Tencent
- TodayTix - 机票预订服务
- Ulule - 众筹平台
- Unity Labs - 区块链开发公司
- VCC Live - 云联系中心软件
- Vimeo - 视频共享平台
- Volvo Cars - 汽车制造商
- VXLab - 可视化编程平台
- Wonderbly - 儿童图书公司

资源

资源

外部资源, 视频, 教程和博客文章列表如下. 随时作出贡献.

视频

- Asim Aslam 的 简化微服务
- Brian Ketelsen 的 使用 Micro 的微服务

播客

- 在 GoTime.fm 与 Asim, Brian, Erik 和 Carlisia 一起

博客

- Ewan Valentine 的 用 Go 的微服务
- Che Dan 的 Micro 实战
  - 第 1 部分: 入门
  - 第 2 部分: 开始的最佳指南
  - 第 3 部分: 调用服务
  - @dche423/micro-in-action-part4-pub-sub-564f3b054ecd"target="_blank">
    第 4 部分: 发布/订阅
  - 第 5 部分: 消息代理
  - 第 6 部分: 服务发现
  - 第 7 部分: 断路器和速率限制器
  - 结尾: 分布式调度作业

开始

概述

概述

> 前言: 微型入门指南

什么是 Micro

Micro 是一个用于构建和管理分布式系统的系统.

Micro 的主要组件如下:

- 运行时: 用于管理服务包括 auth, config, discovery, networking 的运行时环境.
- 框架: 用于编写要在运行时运行的服务的 Go 框架.
- 客户端: 多语言客户端, 使其他程序能够访问微服务.

安装

使用 Go:

```
go install github.com/micro/micro/v2
```

或者通过下载二进制文件

```
# MacOS

curl -fsSL https://raw.githubusercontent.com/micro/micro/master/scripts/install.sh | /bin/bash

# Linux
wget -q  https://raw.githubusercontent.com/micro/micro/master/scripts/install.sh -O - | /bin/bash

# Windows
powershell -Command "iwr -useb https://raw.githubusercontent.com/micro/micro/master/scripts/install.ps1 | iex"
```

运行服务

在深入了解编写服务之前, 让我们运行一个现有的服务, 因为它只是几个命令!

首先, 我们必须开始. 执行此操作的命令是:

```
micro server
```

如果一切顺利, 您将看到各种服务初始化的日志输出; 此终端将继续输出日志, 因为我们通过本教程的其余部分, 所以保持它运行.

要与此服务器交互, 我们只需告诉 Micro CLI 解决我们的服务器问题, 而不是使用默认实现 - micro 也可以在没有服务器的情况下工作, 但稍后将对此进行更多介绍.

以下命令告诉 CLI 与我们的服务器交互:

```
micro env set server
```

非常好! 我们继续. 只是为了验证一切是否井然有序, 让我们看看正在运行的服务:

```
$ micro list services
go.micro.api
go.micro.auth
go.micro.bot
go.micro.broker
go.micro.config
go.micro.debug
go.micro.network
go.micro.proxy
go.micro.registry
go.micro.router
go.micro.runtime
go.micro.server
go.micro.web
```

所有这些服务都是由我们的启动的. 这很酷, 但它仍然不是我们这节要说的重点! 让我们开始一个服务, 我们实际上可以信任的存在. 如果我们去 github.com/micro/services, 我们看到一堆由 mirco 作者编写的服务. 其中之一是 helloworld. 让我们来试试, 好吗?

运行服务的命令是 `micro run` . 此命令可能需要一段时间才能从 GitHub 检出存储库. (@todo这实际上当前失败, 修复)

```
micro run github.com/micro/services/helloworld
```

如果我们查看下正在运行的 `micro server` ，我们应该看到类似如下内容

```
Creating service helloworld version latest source /tmp/github.com-micro-services/helloworld
Processing create event helloworld:latest
```

我们还可以查看服务的日志，以验证其正在运行.

```
$ micro logs helloworld
Starting [service] go.micro.service.helloworld
Server [grpc] Listening on [::]:36577
Registry [service] Registering node: go.micro.service.helloworld-213b807a-15c2-496f-93ac-7949ad38aadf
```

因此，由于我们的服务运行的很好，让我们尝试调用它！这就是该服务存在的原因.

调用服务

我们有几个选项可以调用我们在运行的服务 `micro server`

通过 CLI 调用服务

最简单的可能是通过 CLI:

```
$ micro call go.micro.service.helloworld Helloworld.Call '{"name":"Jane"}'
{
    "msg": "Hello Jane"
}
```

成功了！如果我们想知道服务具有哪些终结点，我们可以运行以下命令:

```
micro get service go.micro.service.helloworld
```

否则，最好的地方看是 proto 定义. 您还可以在 http://localhost:8082 浏览 UI 以查看实时信息.

使用 Go Micro 调用服务

让我们编写一个小客户端，我们可以用它来调用 helloworld 服务. 通常，您将在另一个服务内进行服务调用，因此这只是您可以编写的函数的示例. 我们将学习如何尽快编写一份全面的服务.

让我们采用以下文件和代码:

```
package main

import (
    "context"
    "fmt"

    "github.com/micro/go-micro/v2"
    proto "github.com/micro/services/helloworld/proto"
)

func main() {
    // create and initialise a new service
    service := micro.NewService()
    service.Init()

    // create the proto client for helloworld
    client := proto.NewHelloworldService("go.micro.service.helloworld", service.
Client())

    // call an endpoint on the service
    rsp, err := client.Call(context.Background(), &proto.Request{
        Name: "John",
    })
    if err != nil {
        fmt.Println("Error calling helloworld: ", err)
        return
    }

    // print the response
    fmt.Println("Response: ", rsp.Msg)

    // let's delay the process for exiting for reasons you'll see below
    time.Sleep(time.Second * 5)
}
```

在本地保存示例. 为了便于遵循本指南, 可以命名文件夹为 `example-service` . 执行 `example-service && go mod init examplemicro run` 命令后, 我们准备使用 `micro run` 命令运行此服务:

```
micro run .
```

另一个有用的命令，以查看正在运行的内容，是 `micro status` . 此时，我们应该有两个服务正在运行:

```
$ micro status
NAME            VERSION    SOURCE                                          STATUS
BUILD    UPDATED        METADATA
example-service    latest    /home/username/example-service                  start
ing    n/a        4s ago        owner=n/a, group=n/a
helloworld        latest    /tmp/github.com-micro-services/helloworld    running
n/a        6m5s ago    owner=n/a, group=n/a
```

现在，由于我们的示例服务客户端也在运行，我们应该能够看到它的日志:

```
$ micro logs example-service
# some go build output here
Response:  Hello John
```

非常好! 这种响应内容直接来自我们之前开始的 helloworld 服务!

从其他语言

在 客户端仓库 中，有用于各种语言和框架的 micro 客户端. 它们旨在轻松连接到实时 micro 环境或本地环境，但稍后将提供更多有关环境.

编写服务

要建立新服务的脚手架，可以使用该命令 `micro new` . 它应会输出类似如下的内容:

```
$ micro new foobar
Creating service go.micro.service.foobar in foobar

.
├── main.go
├── generate.go
├── plugin.go
├── handler
│   └── foobar.go
├── subscriber
│   └── foobar.go
├── proto/foobar
│   └── foobar.proto
├── Dockerfile
├── Makefile
```

```
├──── README.md
├──── .gitignore
└──── go.mod


download protobuf for micro:

brew install protobuf
go get -u github.com/golang/protobuf/proto
go get -u github.com/golang/protobuf/protoc-gen-go
go get github.com/micro/micro/v2/cmd/protoc-gen-micro@master

compile the proto file foobar.proto:

cd foobar
protoc --proto_path=.:$GOPATH/src --go_out=. --micro_out=. proto/foobar/foobar.p
roto
```

从上面的输出可以看出，在构建第一个服务之前，必须安装以下工具：

- protoc
- protobuf/proto
- protoc-gen-micro

它们都需要将原文件转换为实际的 Go 代码. Protos 的存在是为了提供一种与语言无关的方法来描述服务终结点，其输入和输出类型，并具有高效的序列化格式.

目前，Micro 是 Go 重点 (除了前面提到的客户端)，但这种情况很快就会改变.

因此，一旦安装了所有工具，位于服务根中，我们可以发出以下命令，从 protos 生成 Go 代码：

```
protoc --proto_path=.:$GOPATH/src --go_out=. --micro_out=. proto/foobar/foobar.p
roto
```

生成的代码必须提交到源代码管理，以使其他服务在进行服务调用时导入 proto (请参阅上一节调用服务).

现在，我们知道如何编写服务，运行服务以及调用其他服务. 我们的一切都触手可及，但仍有一些缺失的部件来编写应用程序. 其中一部分是存储接口，它有助于持久数据存储，即使没有数据库.

存储

在许多其他有用的内置服务中，Micro 包括用于存储数据的持久存储服务.

接口作为构建基块

快速侧记. Micro（server/CLI）和 Go Micro (框架) 以强定义的接口为中心, 这些接口是可插入的, 并为底层分布式系统概念提供了抽象. 这是什么意思?

让我们以我们目前的 存储接口 为例. 它旨在启用具有几个不同实现的服务编写器数据存储:

- 在内存中
- 文件存储（micro server 运行时默认存储实现)
- cockroachdb

同样, 允许您以完全运行时无关的方式运行服务的 运行时 接口具有一些实现:

- 本地, 只是运行实际流程 - 旨在本地发展
- kubernetes - 以高可用和分布式方式运行容器

这是跨 Micro 接口的反复出现的主题. `micro server` 运行时, 让我们看一下默认的存储.

使用存储

将存储与 CLI 一起使用

首先, 让我们来介绍一下最基本的存储 CLI 命令.

要保存值, 我们使用写入命令:

```
micro store write key1 value1
```

UNIX 风格中没有输出意味着它被愉快地保存. 读它怎么样?

```
$ micro store read key1
val1
```

或者, 为了以更奇特的方式显示它, 我们可以使用 `--verbose` 或 `-v` 标志:

```
KEY    VALUE    EXPIRY
key1   val1     None
```

当我们使用 `--prefix` 或 `-p` 标志时, 此显示特别有用, 它允许我们搜索具有某些前缀的键的条目.

为了演示首先让我们保存一个其他值:

```
micro store write key2 val2
```

在此之后，我们可以列出两个键 `key1` 和 `key2` 键，因为它们都共享通用前缀：

```
$ micro store read --prefix --verbose key
KEY    VALUE    EXPIRY
key1   val1     None
key2   val2     None
```

存储里还有更多的东西，但是这些知识已经让我们很危险了！

将存储与 Go-Micro 一起使用

从 Go Micro 服务访问我们刚刚操作的相同数据不会更容易. 首先，让我们创建一个条目，我们的服务可以读取. 这次我们也用 `micro store write` 指定该命令的表，因为每个服务在存储中都有自己的表：

```
micro store write --table go.micro.service.example mykey "Hi there"
```

让我们修改我们以前编写的示例服务，以便它不是调用服务，而是从存储中读取上述值.

```
package main

import (
    "fmt"
    "time"

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

func main() {
    service := micro.NewService()

    service.Init(micro.Name("go.micro.service.example"))

    records, err := service.Options().Store.Read("mykey")
    if err != nil {
        fmt.Println("Error reading from store: ", err)
    }

    if len(records) == 0 {
        fmt.Println("No records")
    }
```

```
    for _, record := range records {
        fmt.Printf("key: %v, value: %v\n", record.Key, string(record.Value))
    }

    time.Sleep(1 * time.Hour)
}
```

我们快完成了！但首先，我们必须学习如何更新服务.

更新和终止服务

现在，由于示例服务正在运行（可以通过 `micro status` 轻松验证），我们不应该使用 `micro run` ，而应该部署它.

我们可以简单地发出更新命令（记住先切换回示例服务的根目录）

```
micro update .
```

并验证两者与微服务器输出：

```
Updating service example-service version latest source /home/username/example-service
Processing update event example-service:latest in namespace default
```

和微状态：

```
$ micro status example-service
NAME            VERSION    SOURCE                              STATUS        BUILD    UPDATED        METADATA
example-service    latest    /home/username/example-service    starting    n/a    10s ago        owner=n/a,group=n/a
```

服务将被更新.

如果由于某种原因的事情进展不太顺利，我们可以尝试时间测试“关闭并重新打开”解决方案，并完成：

```
micro kill .
micro run .
```

从干净的状态重新开始.

因此, 一旦我们更新了示例服务, 我们就应在日志中看到以下内容:

```
$ micro logs example-service
key: mykey, value: Hi there
```

好! 示例服务从存储中成功读取值.

客户端

除此之外, 我们还在多语言客户端上工作, 您可以在 github.com/micro/clients 在 github 上找到这些客户端并作出贡献. 我们很想进一步讨论这个问题, 但还没有完全准备好.

进一步阅读

这只是一个简短的入门指南, 用于快速启动和运行 Micro. 当本指南不断升级时, 不时回来了解更多信息. 如果您有兴趣学习更多 micro 魔法, 请查看以下内容:

- 阅读 文档
- 按 示例 学习
- 来加入我们的 slack 和提问

安装

安装

Go Micro 是 Go 中开发微服务的 RPC 框架

依赖

生成代码需要 protoc-gen-micro

- protoc-gen-micro

导入

确保导入 go-micro v2

```
import "github.com/micro/go-micro/v2"
```

运行

Micro 提供了访问和管理微服务的运行时

安装

来自源

```
go get github.com/micro/micro/v2
```

Docker 映像

```
docker pull micro/micro
```

最新版本二进制文件

```
# MacOS
curl -fsSL https://raw.githubusercontent.com/micro/micro/master/scripts/install.sh | /bin/bash
```

```
# Linux
wget -q  https://raw.githubusercontent.com/micro/micro/master/scripts/install.sh -O - | /bin/bash

# Windows
powershell -Command "iwr -useb https://raw.githubusercontent.com/micro/micro/master/scripts/install.ps1 | iex"
```

使用

启动服务器

```
micro server
```

运行问候器服务

```
micro run github.com/micro/examples/greeter/srv
```

列出服务

```
micro list services
```

获取服务

```
micro get service go.micro.srv.greeter
```

输出

```
service  go.micro.srv.greeter

version 2019.11.09.10.34

ID      Address Metadata
go.micro.srv.greeter-e25a5edd-0936-4d32-b4d7-e62bf454d5f7       172.17.0.1:33031 broker=http,protocol=mucp,registry=mdns,server=mucp,transport=http

Endpoint: Say.Hello

Request: {
        name string
}
```

```
Response: {
        msg string
}
```

调用服务

```
micro call go.micro.srv.greeter Say.Hello '{"name": "John"}'
```

输出

```
{
    "msg": "Hello John"
}
```

Hello World

Hello World

概述

这是使用 Go Micro 的 helloworld 示例. 我们将翻阅完整的内容.

编写服务

使用 Go Micro 编写服务非常简单. 它提供了快速移动的框架, 而无需首先了解所有内容. 下面是一个简单的问候器服务示例, 我们将贯穿其中.

在 examples/service 上查找完整的代码示例.

服务协议

微服务的关键要求之一是强定义接口. Micro 使用 protobuf 来实现这一目标.

在这里我们使用 Hello 方法定义 Greeter 处理程序. 它使用一个字符串参数来获取请求和响应.

```
syntax = "proto3";

service Greeter {
    rpc Hello(Request) returns (Response) {}
}

message Request {
    string name = 1;
}

message Response {
    string greeting = 2;
}
```

生成原型

编写原型定义后, 我们必须使用带有 micro plugin 的 protoc 编译它.

```
protoc --proto_path=$GOPATH/src:. --micro_out=. --go_out=. path/to/greeter.proto
```

实现服务

现在，我们已经定义了实现服务所需的服务接口.

下面是问候器服务的代码. 它执行以下操作:

1. 实现为 Greeter 处理程序定义的接口
2. 初始化一个 micro.Service
3. 注册 Greeter 处理器
4. 运行服务

```
package main

import (
    "context"
    "fmt"

    micro "github.com/micro/go-micro/v2"
    proto "github.com/micro/examples/service/proto"
)

type Greeter struct{}

func (g *Greeter) Hello(ctx context.Context, req *proto.Request, rsp *proto.Resp
onse) error {
    rsp.Greeting = "Hello " + req.Name
    return nil
}

func main() {
    // Create a new service. Optionally include some options here.
    service := micro.NewService(
        micro.Name("greeter"),
    )

    // Init will parse the command line flags.
    service.Init()

    // Register handler
    proto.RegisterGreeterHandler(service.Server(), new(Greeter))

    // Run the server
    if err := service.Run(); err != nil {
```

```
        fmt.Println(err)
    }
}
```

运行服务

现在使用 Go 运行示例. 如果您正在使用示例代码, 请执行以下操作:

```
go run examples/service/main.go
```

这应该会输出类似如下的内容

```
2019-11-13 21:39:38.327452 I | Transport [http] Listening on [::]:32945
2019-11-13 21:39:38.327548 I | Broker [http] Connected to [::]:38955
2019-11-13 21:39:38.328095 I | Registry [mdns] Registering node: greeter-3937310
7-5ae7-42a2-b5e2-ebebf7eafbd9
```

编写客户端

一旦我们有了服务, 我们实际上需要一种方法来查询它. 这是微服务的核心, 因为我们不仅服务, 而且消耗其他服务. 下面是查询问候器服务的客户端代码.

生成的原型包括一个问候器客户端, 以减少样板代码.

```
package main

import (
    "context"
    "fmt"

    micro "github.com/micro/go-micro/v2"
    proto "github.com/micro/examples/service/proto"
)

func main() {
    // Create a new service
    service := micro.NewService(micro.Name("greeter.client"))
    // Initialise the client and parse command line flags
    service.Init()

    // Create new greeter client
    greeter := proto.NewGreeterService("greeter", service.Client())
```

```
    // Call the greeter
    rsp, err := greeter.Hello(context.TODO(), &proto.Request{Name: "John"})
    if err != nil {
        fmt.Println(err)
    }

    // Print response
    fmt.Println(rsp.Greeting)
}
```

现在运行客户端

```
go run client.go
```

输出应该会简单地打印响应的内容

```
Hello John
```

任意内容作为微服务

任意内容作为微服务

将任意内容转换为微服务. Micro 提供了一种封装任何内容以成为服务的方法.

概述

Micro 是管理微服务的运行时. 命令行 `micro service` 封装了任何应用程序或服务, 使其在 micro 生态系统中可访问. 下面的示例用于基本 http 应用.

HTTP 应用程序

这里有一个简单的 http hello world 应用程序

```
package main

import (
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte(`hello world`))
    })
    http.ListenAndServe(":9090", nil)
}
```

使用 micro 启动服务

```
micro service --name helloworld --endpoint http://localhost:9090 go run main.go
```

通过 cli 查询服务

```
micro call -o raw helloworld /
```

文件服务器

将文件发回给调用方

文件 /tmp/helloworld.txt

```
helloworld
```

运行服务

```
micro service --name helloworld --endpoint file:///tmp/helloworld.txt
```

获取文件

```
micro call -o raw helloworld .
```

执行脚本

远程执行脚本或命令

```
#!bin/bash

echo `date` hello world
```

```
micro service --name helloworld --endpoint exec:///tmp/hellworld.sh
```

```
micro call -o raw helloworld .
```

部署

本地

本地

安装

来自源

```
go get github.com/micro/micro/v2
```

发行二进制

```
# MacOS
curl -fsSL https://raw.githubusercontent.com/micro/micro/master/scripts/install.sh | /bin/bash

# Linux
wget -q  https://raw.githubusercontent.com/micro/micro/master/scripts/install.sh -O - | /bin/bash

# Windows
powershell -Command "iwr -useb https://raw.githubusercontent.com/micro/micro/master/scripts/install.ps1 | iex"
```

运行

运行 micro 像输入 `micro` 一样简单.

```
# Display the commands
micro --help
```

运行服务器

```
micro server
```

Docker

Docker

安装 Micro

```
docker pull micro/micro
```

Compose

使用 Docker Compose 运行本地服务器

```
server:
  command: server
  build: .
  ports:
  - "8080:8080"
  - "8081:8081"
  - "8082:8082"
```

从头开始构建

仓库中包括 Docker 文件

```
## checkout the repo
git clone https://github.com/micro/micro

## build the image
cd micro && docker build -t micro .
```

Kubernetes

Kubernetes

本文档提供了在 kubernetes 上运行 micro 的指南.

开始

依赖

在 kubernetes 我们建议运行 etcd 和 nats.

- etcd 用于高度可扩展的服务发现
- NATS 用于异步消息传递

安装 etcd (指南)

```
helm install --name my-release --set customResources.createEtcdClusterCRD=true s
table/etcd-operator
```

安装 nats (指南)

```
helm install my-release stable/nats
```

您现在应该具有所需的依赖项.

部署

下面是 micro 服务的 k8s 部署示例

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  namespace: default
  name: greeter
spec:
  replicas: 1
  selector:
    matchLabels:
      name: greeter-srv
      micro: service
  template:
    metadata:
      labels:
        name: greeter-srv
        micro: service
    spec:
      containers:
        - name: greeter
          command: [
        "/greeter-srv",
      ]
          image: micro/go-micro
          imagePullPolicy: Always
          ports:
          - containerPort: 8080
            name: greeter-port
          env:
          - name: MICRO_SERVER_ADDRESS
            value: "0.0.0.0:8080"
          - name: MICRO_BROKER
            value: "nats"
          - name: MICRO_BROKER_ADDRESS
            value: "nats-cluster"
          - name: MICRO_REGISTRY
            value: "etcd"
          - name: MICRO_REGISTRY_ADDRESS
            value: "etcd-cluster-client"
```

使用 kubectl 部署

```
kubectl apply -f greeter.yaml
```

Micro API

要部署 micro api, 请使用以下配置. 注意默认情况下 ENABLE_ACME 环境变量需要我们 Let’s Encrypt SSL 加密措施.

创建 api 服务

```
apiVersion: v1
kind: Service
metadata:
  name: micro-api
  namespace: default
  labels:
    name: micro-api
    micro: service
spec:
  ports:
  - name: https
    port: 443
    targetPort: 443
  selector:
    name: micro-api
    micro: service
  type: LoadBalancer
```

创建部署

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: default
  name: micro-api
  labels:
    micro: service
spec:
  replicas: 3
  selector:
    matchLabels:
      name: micro-api
      micro: service
  strategy:
    rollingUpdate:
      maxSurge: 0
      maxUnavailable: 1
  template:
    metadata:
```

```
      labels:
        name: micro-api
        micro: service
    spec:
      containers:
      - name: api
        env:
        - name: MICRO_ENABLE_STATS
          value: "true"
        - name: MICRO_BROKER
          value: "nats"
        - name: MICRO_BROKER_ADDRESS
          value: "nats-cluster"
        - name: MICRO_REGISTRY
          value: "etcd"
        - name: MICRO_REGISTRY_ADDRESS
          value: "etcd-cluster-client"
        - name: MICRO_REGISTER_TTL
          value: "60"
        - name: MICRO_REGISTER_INTERVAL
          value: "30"
        - name: MICRO_ENABLE_ACME
          value: "true"
        args:
        - api
        image: micro/micro
        imagePullPolicy: Always
        ports:
        - containerPort: 443
          name: api-port
```

Micro Web

要部署 micro Web, 请使用以下配置 注意默认情况下 ENABLE_ACME 环境变量使用 Let’s Encrypt SSL 加密措施.

创建服务

```
apiVersion: v1
kind: Service
metadata:
  name: micro-web
  namespace: default
  labels:
```

```
    name: micro-web
    micro: service
spec:
  ports:
  - name: https
    port: 443
    targetPort: 443
  selector:
    name: micro-web
    micro: service
  type: LoadBalancer
```

创建部署

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: default
  name: micro-web
  labels:
    micro: service
spec:
  replicas: 1
  selector:
    matchLabels:
      name: micro-web
      micro: service
  template:
    metadata:
      labels:
        name: micro-web
        micro: service
    spec:
      containers:
      - name: web
        env:
        - name: MICRO_BROKER
          value: "nats"
        - name: MICRO_BROKER_ADDRESS
          value: "nats-cluster"
        - name: MICRO_ENABLE_STATS
          value: "true"
        - name: MICRO_REGISTRY
          value: "etcd"
        - name: MICRO_REGISTRY_ADDRESS
```

```
          value: "etcd-cluster-client"
        - name: MICRO_ENABLE_ACME
          value: "true"
        args:
        - web
        image: micro/micro
        imagePullPolicy: Always
        ports:
        - containerPort: 443
          name: web-port
```

框架

概述

概述

> 前言: Go Micro 是一个微服务开发的框架

概述

Go Micro 为分布式系统开发 (包括 RPC 和事件驱动通信) 提供了核心要求. micro 的理念是具有可插拔架构的默认实践. 我们提供默认值, 让您快速入门, 但一切都可以轻松置换.

特性

Go Micro 抽象了分布式系统的详细信息. 以下是主要功能.

- **服务发现** - 自动服务注册和名称解析. 服务发现是微服务开发的核心. 当服务 A 需要与服务 B 说话时, 它需要该服务的位置. 默认发现机制是多播 DNS (mdns), 这是一个零配置的发现系统.
- **负载平衡** - 基于服务发现构建的客户端负载平衡. 一旦我们拥有服务任意数量的实例的地址, 我们现在需要一种方法来决定要路由到哪个节点. 我们使用随机哈希负载平衡来跨服务提供均匀分布, 并在出现问题时重试其他节点.
- **消息编码** - 基于内容类型的动态消息编码. 客户端和服务器将使用编解码器以及内容类型来无缝编码和解码 Go 类型. 可以编码和从不同的客户端发送任何种类的消息. 默认情况下, 客户端和服务器处理此情况. 默认情况下, 这包括 protobuf 和 json.
- **请求/响应** - 基于 RPC 的请求/响应, 支持双向流式处理. 我们为同步通信提供抽象. 对服务发出的请求将自动解析, 负载平衡, 拨号和流式传输. 默认传输为 gRPC。
- **异步消息** - PubSub 是作为异步通信和事件驱动架构的一等公民而构建的. 事件通知是微服务开发的核心模式. 默认的消息系统是嵌入式 NATS 服务器.
- **可插拔接口**- Go Micro 为每个分布式系统抽象使用 Go 接口. 因此这些接口是可插拔的, 并允许 Go Micro 与运行时无关. 您可以插入任何基础技术. 在 github.com/micro/go-plugins 中查找插件.

开始

依赖

默认情况下, Go Micro 会使用原型. 因此我们可以编写生成样板代码的代码, 并提供有效的线格式, 以便在服务之间来回传输数据.

我们还需要某种形式的服务发现, 以便将服务名称解析为其地址以及元数据和终结点信息. 有关详细信息, 请参阅下文.

Protobuf

您需要安装 protobuf 以生成 API 接口的代码:

- protoc-gen-micro

发现

服务发现用于将服务名称解析为地址. 默认情况下, 我们使用多播 DNS 提供零配置的发现系统. 这是在大多数操作系统上是内置的. 如果您需要更具弹性和多主机的东西, 请使用 etcd.

Etcd

etcd 可用作替代服务发现系统.

- 下载并运行 etdc
- 传递给任何命令或环境变量 `--registry=etcd` MICRO_REGISTRY=etcd`

```
MICRO_REGISTRY=etcd go run main.go
```

服务发现是可插拔的. 可在 micro/go-plugins 仓库中寻到 consul, kubernetes, zookeeper 和更多插件.

安装

Go Micro 是基于 Go 的开发的框架. 您可以使用 go 工具链轻松获取此功能.

在服务中导入 go-micro

```
import "github.com/micro/go-micro/v2"
```

我们提供发布标签，并建议坚持使用最新的稳定版本. 使用 go 模块将启用此功能.

```
# enable go modules
export GO111MODULE=on
# initialise go modules in your app
go mod init
# now go get
go get ./...
```

编写服务

可以从检出 hello world 的例子开始

配置

配置

Go Config 是一个可插拔的动态配置库

概述

应用程序中的大多数配置都是静态配置的或者包括从多个源加载的复杂逻辑. Go-config 使此变得简单, 可插拔和可合并. 你再也不必以旧的的方式处理配置了.

特性

- **动态加载** - 在需要时从多个源加载配置. Go Config 管理在后台监视配置源, 并自动合并和更新内存视图中的配置文件源.
- **可插拔的源** - 从任意数量的源中进行选择以加载和合并配置. 后端源被抽象为内部使用并通过编码器解码的标准格式. 源可以是 env vars, flags, etcd, k8s configmap, 等.
- **可合并配置** - 如果指定多个配置源, 无论格式如何, 它们都将合并并在单个视图中显示. 这极大地简化了基于环境的优先级顺序加载和更改.
- **观察更改** - 可选地观察配置, 以观察对特定值的更改. 使用 Go Config 的观察程序进行热重新加载你的应用. 不必处理临时关机重新加载或其他任何内容, 只需继续读取配置并监视需要通知的更改.
- **安全恢复** - 如果配置负载严重或由于未知原因完全擦除, 则可以在直接访问任何配置值时指定回退值. 这可确保在发生问题时始终读取某些合理的默认值.

开始

- 源 - 配置从中加载的后端
- 编码器 - 处理编码/解码源配置
- 读取器 - 将多个编码源合并为单一格式
- 配置 - 管理多个源的配置管理器
- 使用 - go-config 的示例用法
- 常见问题 - 常见问题和回答

- 待办 - 待办任务/特性

源

`Source` 是从加载配置的后端. 可以同时使用多个源.

支持以下源:

- cli - 从 CLI 标志读取
- consul - 从 consul 读取
- env - 从环境变量读取
- etcd - 从 etcd v3 读取
- file - 从文件读取
- flag - 从标志读取
- memory - 从内存读取

还有一些社区支持的插件, 支持以下源:

- configmap - 从 k8s configmap 读取
- grpc - 从 grpc 服务器读取
- runtimevar - 从 Go Cloud Development Kit 运行时变量读取
- url - 从 url 读取
- vault - 从 Vault 服务器读取

待办:

- git url

更改集

源将配置作为一个 ChangeSet 返回. 这是多个后端的单一内部抽象.

```
type ChangeSet struct {
    // Raw encoded config data
    Data      []byte
    // MD5 checksum of the data
    Checksum  string
    // Encoding format e.g json, yaml, toml, xml
    Format    string
    // Source of the config e.g file, consul, etcd
```

```
    Source    string
    // Time of loading or update
    Timestamp time.Time
}
```

编码器

`Encoder` 处理源配置编码/解码. 后端源可能以许多不同的格式存储配置. 编码器使我们能够处理任何格式. 如果未指定编码器, 则默认为 json.

支持以下编码格式:

- json
- yaml
- toml
- xml
- hcl

读取器

`Reader` 将多个更改集表示为单个合并和可查询的值集.

```
type Reader interface {
    // Merge multiple changeset into a single format
    Merge(...*source.ChangeSet) (*source.ChangeSet, error)
    // Return return Go assertable values
    Values(*source.ChangeSet) (Values, error)
    // Name of the reader e.g a json reader
    String() string
}
```

读取器使用编码器将更改集解码为 `map[string]interface{}` Values`, 然后将它们合并到单个更改集中. 它通过 Format 字段以确定编码器. 然后更改集表示为一组, 具有重新备份 Go 类型和无法加载值的回退功能.

```

// Values is returned by the reader
type Values interface {
    // Return raw data
        Bytes() []byte
    // Retrieve a value
```

```
        Get(path ...string) Value
    // Return values as a map
        Map() map[string]interface{}
    // Scan config into a Go type
        Scan(v interface{}) error
}
```

`Value` 接口允许强制转换/类型断言转到具有回退默认值的类型.

```
type Value interface {
    Bool(def bool) bool
    Int(def int) int
    String(def string) string
    Float64(def float64) float64
    Duration(def time.Duration) time.Duration
    StringSlice(def []string) []string
    StringMap(def map[string]string) map[string]string
    Scan(val interface{}) error
    Bytes() []byte
}
```

Config

`Config` 管理所有配置, 抽象掉源, 编码器和读取器.

它管理从多个后端源读取, 同步, 监视, 并将它们表示为单个合并和可查询源.

```
// Config is an interface abstraction for dynamic configuration
type Config interface {
    // provide the reader.Values interface
    reader.Values
    // Stop the config loader/watcher
    Close() error
    // Load config sources
    Load(source ...source.Source) error
    // Force a source changeset sync
    Sync() error
    // Watch a value for changes
    Watch(path ...string) (Watcher, error)
}
```

使用

示例配置

只要我们有一个编码器来支持它，配置文件就可以是任何格式的.

json 配置示例:

```
{
    "hosts": {
        "database": {
            "address": "10.0.0.1",
            "port": 3306
        },
        "cache": {
            "address": "10.0.0.2",
            "port": 6379
        }
    }
}
```

新配置

创建一个新配置 (或者仅使用默认实例)

```
import "github.com/micro/go-micro/v2/config"

conf := config.NewConfig()
```

加载文件

从文件源加载配置. 这使用文件扩展名来确定配置格式.

```
import (
    "github.com/micro/go-micro/v2/config"
)

// 加载 json 配置文件
config.LoadFile("/tmp/config.json")
```

通过指定具有相应文件扩展名的文件来加载 yaml, toml 或 xml 文件

```
// 加载 yaml 配置文件
config.LoadFile("/tmp/config.yaml")
```

如果扩展不存在, 可指定编码器

```
import (
    "github.com/micro/go-micro/v2/config"
    "github.com/micro/go-micro/v2/config/source/file"
)

enc := toml.NewEncoder()

// 通过编码器加载 toml 文件
config.Load(file.NewSource(
        file.WithPath("/tmp/config"),
    source.WithEncoder(enc),
))
```

读取配置

读取整个配置作为一个集合

```
// retrieve map[string]interface{}
conf := config.Map()

// map[cache:map[address:10.0.0.2 port:6379] database:map[address:10.0.0.1 port:
3306]]
fmt.Println(conf["hosts"])
```

扫描配置到结构体

```
type Host struct {
        Address string `json:"address"`
        Port int `json:"port"`
}

type Config struct{
    Hosts map[string]Host `json:"hosts"`
}

var conf Config

config.Scan(&conf)

// 10.0.0.1 3306
fmt.Println(conf.Hosts["database"].Address, conf.Hosts["database"].Port)
```

读取值

从配置扫描值到结构体

```
type Host struct {
    Address string `json:"address"`
    Port int `json:"port"`
}

var host Host

config.Get("hosts", "database").Scan(&host)

// 10.0.0.1 3306
fmt.Println(host.Address, host.Port)
```

读取单个值作为 Go 类型

```
// Get address. Set default to localhost as fallback
address := config.Get("hosts", "database", "address").String("localhost")

// Get port. Set default to 3000 as fallback
port := config.Get("hosts", "database", "port").Int(3000)
```

观察路径

观察路径以进行更改. 当文件更改时, 新值将可用.

```
w, err := config.Watch("hosts", "database")
if err != nil {
    // do something
}

// wait for next value
v, err := w.Next()
if err != nil {
    // do something
}

var host Host

v.Scan(&host)
```

多源

可以加载和合并多个源. 合并优先级顺序相反.

```
config.Load(
    // base config from env
    env.NewSource(),
    // override env with flags
    flag.NewSource(),
    // override flags with file
    file.NewSource(
        file.WithPath("/tmp/config.json"),
    ),
)
```

设置源编码器

源要求编码器对数据进行编码/解码并指定更改集格式.

默认的编码器是 json. 要更改编码器可调整对应的类型选项.

```
e := yaml.NewEncoder()

s := consul.NewSource(
    source.WithEncoder(e),
)
```

添加读取器编码器

读取器使用编码器从不同格式的源解码数据.

默认的读取器支持 json, yaml, xml, toml 和 hcl. 它将合并的配置表示为 json.

可以通过指定的选项来添加一个新的编码器.

```
e := yaml.NewEncoder()

r := json.NewReader(
    reader.WithEncoder(e),
)
```

常见问题

跟 Viper 有什么不一样?

Viper 和 go-config 都在解决相同的问题. Go-config 提供一个不一样的接口是大型 micro 生态系统工具中的一部分.

编码器和读取器之间有什么不一样?

后端源使用编码器对数据进行编码/解码. 读取器使用编码器解码来自不同格式的多个源的数据, 然后将它们合并为单个编码格式.

对于文件源, 我们使用文件扩展名来确定配置格式 以便使用不同编码.

对于 consul, etcd 或 类似键值的源, 我们可能会从包含多个键的前缀加载, 这意味着源需要了解编码, 以便它可以返回单个更改集.

在有环境变量和标志的情况下, 我们还需要一种方法将值编码为字节并指定格式, 以便以后可以由读取器合并.

为什么更改集数据不表示为 map[string]interface{}?

在某些情况下, 源数据实际上可能不是键值, 因此表示为字节更容易将其将解码及随后到读取器.

错误

错误

> 前言: Go Micro 产生的错误处理和错误

Go Micro 为分布式系统中发生的大多数事物包括错误提供了抽象和类型。通过提供一组核心错误和定义详细错误类型的能力, 我们可以始终如一地了解典型 Go 错误字符串之外发生的情况.

概述

我们定义以下错误类型:

```
type Error struct {
    Id     string `json:"id"`
    Code   int32  `json:"code"`
    Detail string `json:"detail"`
    Status string `json:"status"`
}
```

在系统中, 要求您从处理程序返回错误或从客户端接收错误的任何位置, 都应假定其为 go-micro 错误, 或者应该生成错误. 默认情况下, 我们返回 errors.InternalServerError, 其中某些问题在内部出错及发现超时错误 errors.Timeout.

使用

让我们假设处理程序中发生了一些错误. 然后, 您应该决定返回哪种错误, 并执行以下操作.

假设提供的某些数据无效

```
return errors.BadRequest("com.example.srv.service", "invalid field")
```

如果发生内部错误

```
if err != nil {
    return errors.InternalServerError("com.example.srv.service", "failed to read db: %v", err.Error())
}
```

现在，假设您从客户端收到一些错误

```
pbClient := pb.NewGreeterService("go.micro.srv.greeter", service.Client())
rsp, err := pb.Client(context, req)
if err != nil {
    // parse out the error
    e := errors.Parse(err.Error())

    // inspect the value
    if e.Code == 401 {
        // unauthorised...
    }
}
```

例子

例子

> 前言: 使用 Go Micro 的例子

概述

我们在 github 上有一个广泛的示例存储库. 检出 github.com/micro/examples 随时参与或进行更改和更新.

Hello World

一个 helloworld 示例服务是入门的最佳地方.

上述指南的示例服务可在 **exmples/service** 中找到.

端到端

如果想要一个从端到端的合成, 可参阅 问候器 服务. 这包括后端服务, api 和 web 应用.它还将向您展示如何运行 micro api 和 web 服务.

API 处理器

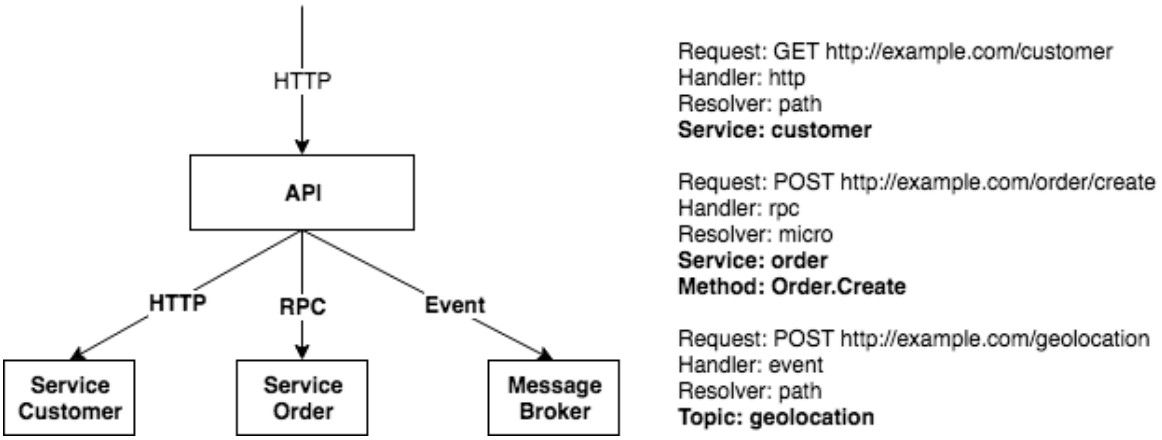
API 处理器

> 前言: 用于定义 api 路由和处理程序的包

概述

Go API 是一个可插拔的 API 框架, 由服务发现驱动, 可帮助构建强大的公共 API 网关.

Go API 库提供 api 网关路由功能. 微服务体系结构将应用程序逻辑分离到单独的服务中. api 网关提供单个入口点, 以将这些服务合并到统一 api 中. Go API 使用在服务发现元数据中定义的路由来生成路由规则并服务 http 请求.



micro api 基于 GO API.

处理器

处理程序是用于处理请求的 http 处理程序. 它使用 `http.Handler` 模式更方便.

- `api` - 处理任何 HTTP 请求. 通过 RPC 完全控制 http 请求/响应.
- `broker` - 实现 go-micro 代理接口的 http 处理程序
- `cloudevents` - 处理云事件并发布到消息总线.
- `event` - 处理任何 HTTP 请求并发布到消息总线.
- `http` - 处理任何 HTTP 请求, 并作为反向代理转发.

- `registry` - 实现 go-micro 注册表接口的 http 处理程序
- `rpc` - 处理 json 和原式 POST 请求. 转发为 RPC.
- `web` - 包含 web 套接字支持的 HTTP 处理程序.

API 处理程序

API 处理程序是默认处理程序. 它提供任何 HTTP 请求, 并作为具有特定格式的 RPC 请求转发.

- Content-Type：任何
- Body： 任何
- Forward Format: api.Request/api.Response
- Path: `/[service]/[method]`
- Resolver: 路径用于解析服务和方法的路径

代理处理程序

代理处理程序是一个 http 处理程序, 它服务于 go-micro 代理接口

- Content-Type: 任何
- Body: 任何
- Forward Format: HTTP
- PATH: `/`
- Resolver: 指定为查询参数的主题

发布请求并将发布

云事件处理程序

CloudEvents 处理程序使用在消息总线上提供 HTTP 并将请求作为 CloudEvents 消息转发的 go-micro/client.Publish 发布方法.

- Content-Type:任何
- Body: 任何
- Forward Format: 请求格式转为 云事件 消息
- Path: `/[topic]`
- Resolver: 用于解析主题路径

事件处理程序

事件处理程序使用在消息总线上提供 HTTP 并将请求作为消息转发的 go-micro/client.Publish 发布方法.

- Content-Type: Any
- Body: Any
- Forward Format: 请求格式为 go-api/proto.Event
- Path: `/[topic]/[event]`
- Resolver: 用于解析主题和事件名称路径

HTTP 处理程序

http 处理程序是具有内置服务发现的 http 反向代理.

- Content-Type: 任何
- Body: 任何
- Forward Format: HTTP 反向代理
- Path: `/[service]`
- Resolver: 用于解析服务名称路径

注册表处理程序

注册表处理程序是一个 http 处理程序, 它为 go-micro 注册表接口提供服务

- Content-Type: 任何
- Body: JSON
- Forward Format: HTTP
- Path: `/`
- Resolver: 用于获取服务, 注册或注销的 GET, POST, DELETE 方法

RPC 处理程序

RPC 处理程序为 json 或 protobuf HTTP POST 请求提供服务, 并作为 RPC 请求转发.

- Content-Type: `application/json` 或 `application/protobuf`
- Body: JSON 或 Protobuf

- Forward Format: 基于内容的 json-rpc 或 proto-rpc
- Path: `/[service]/[method]`
- Resolver: 用于解析服务和方法路径

Web 处理程序

Web 处理程序是一个 http 反向代理, 内置服务发现和 web 套接字支持.

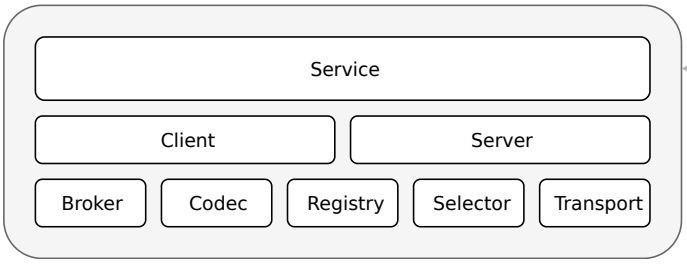
- Content-Type: 任何
- Body: 任何
- Forward Format: HTTP 反向代理, 包括 web 套接字
- Path: `/[service]`
- Resolver: 用于解析服务名称路径

接口

接口

> 前言: go-micro 接口的描述

Go Micro 是一个可插拔的框架, 它利用接口进行抽象和构建基块. 这使我们能够为基础分布式系统概念创建定义强的抽象, 并可置换实现.




```go
package main

import (
        "github.com/micro/go-micro"
)

func main() {
        service := micro.NewService(
                micro.Name("helloworld"),
        )
        service.Run()
}
```

接口

Go micro 由以下接口列表组成:

- **auth** - 用于身份验证和授权
- **broker** - 异步消息传递
- **client** - 用于高级别请求/响应和通知
- **config** - 用于动态配置的
- **codec** - 用于消息编码的
- **debug** - 调式日志, 跟踪, 统计信息
- **netwrok** - 多云的网络
- **registry** - 服务发现的注册表
- **runtime** - 用于运行服务
- **selector** - 用于负载平衡
- **server** - 用于处理请求和通知
- **store** - 用于数据存储
- **sync** - 用于同步, 锁和选举领导
- **transport** - 用于同步通信
- **tunnel** - 用于 vpn 隧道

有关每个接口的说明, 请参阅下文

待办: 填补空白

代理

代理为异步 pub/sub 子通信提供消息代理的接口. 这是事件驱动体系结构和微服务的基本要求之一. 默认情况下, 我们使用收件箱样式点来点 HTTP 系统, 以尽量减少开始所需的依赖项数. 但是在 go-plugins 中有许多消息代理实现, 例如 RabbitMQ, NATS, NSQ, Google Cloud Pub Sub.

客户端

客户端提供一个接口来向服务发出请求. 同样与服务器一样, 它构建在其他包上, 提供统一接口, 用于使用注册表按名称查找服务, 使用选择器进行负载平衡, 使用代理使用传输和异步消息传递进行同步请求.

上述组件在微作为 **服务** 的顶级级别进行组合.

我们还为分布式系统开发提供了一些其他组件:

编解码器

编解码器用于编码和解码消息, 然后再通过导线传输消息. 数据可能是 json, protobuf, beson, msgpack 等. 这与大多数其他编解码器的不同之处在于, 我们实际上也支持此处的 RPC 格式. 因此们有 JSON-RPC, PROTO-RPC, BSON-RPC 等. 它将编码与客户端/服务器分离, 并提供集成其他系统 (如 gRPC, Vandium 等) 的强大方法.

配置

配置是一个接口, 用于从任意数量的源进行动态配置加载, 这些源是可以合并的. 大多数系统都主动要求独立于代码进行更改的配置. 拥有一个配置接口, 可以根据需要动态加载这些值是强大的. 它还支持许多不同的配置格式.

服务器

服务器是编写服务的构建基块. 在这里, 您可以命名您的服务, 注册请求处理程序, 添加中间件等. 该服务基于上述包, 为服务请求提供统一接口. 内置服务器是 RPC 系统. 将来可能还会有其他实现. 服务器还允许您定义多个编解码器以服务不同的编码消息.

存储

存储是一个简单的键值存储接口, 用于抽象掉轻量级数据存储. 我们不是试图实现一个完整的 sql 方言或存储, 只是仅仅能够保持状态, 否则将丢失在无状态的服务. 它们存储接口将成为未来

所有形式的存储的构建基块.

注册表

注册表提供一种服务发现机制, 用于将名称解析为地址. 它可以由 consul, etcd zookeeper, dns, gossip 等支持. 服务应在启动时使用注册表进行注册, 并在关闭时取消注册. 服务可以选择提供过期的 TTL 并在间隔内重新注册, 以确保服务在失效时进行清理.

选择器

选择器是一个负载平衡抽象, 它建立在注册表上. 它允许使用筛选器函数”筛选”服务, 并使用随机, 循环, 最小等算法选择”选择”. 客户在发出请求时利用选择器. 客户端将使用选择器而不是注册表, 因为它提供了内置的负载平衡机制.

传输

传输是服务之间同步请求/响应通信的接口. 它类似于 golang 网络包, 但提供了一个更高级别的抽象, 允许我们切换通信机制, 例如 http, rabbitmq, websocket, NATS. 传输还支持双向流. 这对客户端推送到服务器功能非常强大.

选项

选项

> 前言: 设置和使用 Go Micro 选项

Go Micro 使用可变选项模型来设计包的创建和初始化传递参数, 以及作为方法的可选参数. 这为扩展跨插件的选项使用提供了灵活性.

概述

创建新服务时, 您可以选择传递其他参数, 例如设置名称, 版本, 消息代理, 注册表或存储, 以便与所有其他内部一起使用.

选项通常定义如下

```
type Options struct {
  Name string
  Version string
  Broker broker.Broker
  Registry registry.Registry
}

type Option func(*Options)

// set the name
func Name(n string) Option {
    return func(o *Options) {
        o.Name = n
    }
}

// set the broker
func Broker(b broker.Broker) Option {
    return func(o *Options) {
        o.Broker = b
    }
}
```

然后可以按照如下方式设置这些设置

```
service := micro.NewService(
    micro.Name("foobar"),
    micro.Broker(broker),
)
```

服务选项

在 Go Micro 中，我们有许多选项可以设置，包括将用于身份验证，配置和存储等内容的基础包. 您可以使用 `service.Options()` 来访问这些.

身份验证，配置，注册表，存储等包将默认为我们的零依赖插件. 在要通过环境变量或标志配置它们的位置，可以指定 `service.Init()` 来分析它们。

例如，如果要使用文件存储来替换内存存储，可以按照以下方式完成

```
## as an env var
MICRO_STORE=file go run main.go

## or as a flag
go run main.go --store=file
```

然后在内部，可以通过选项访问存储

```
service := micro.NewService(
    micro.Name("foobar"),
)

service.Init()

store := service.Options().Store
```

插件

插件

> 前言: Go Micro 是一个可插拔的框架

概述

Go Micro 构建于 Go 接口之上. 因此这些接口的实现是可插拔的.

默认情况下, go-micro 只提供核心上每个接口的几个实现, 但它是完全可插拔的. 已经有几十个插件, 在这里 github.com/micro/go-plugins 欢迎贡献! 插件可确保 Go Micro 服务在技术发展后长期保持生存。

添加插件

如果要集成插件, 只需将它们链接到单独的文件中并重新生成

创建 plugins.go 文件并导入所需的插件:

```
package main

import (
        // consul registry
        _ "github.com/micro/go-plugins/registry/consul"
        // rabbitmq transport
        _ "github.com/micro/go-plugins/transport/rabbitmq"
        // kafka broker
        _ "github.com/micro/go-plugins/broker/kafka"
)
```

通过包含插件文件来构建应用程序:

```
# assuming files main.go and plugins.go are in the top level dir

# For local use
go build -o service *.go
```

插件的标志用法:

```
service --registry=etcdv3 --transport=nats --broker=kafka
```

或者首选的是使用环境变量来处理 12 因素的应用程序

```
MICRO_REGISTRY=consul
MICRO_TRANSPORT=rabbitmq
MICRO_BROKER=kafka
service
```

插件选项

或者您可以将插件设置为直接在代码中服务的选项

```
package main

import (
        "github.com/micro/go-micro/v2"
        // consul registry
        "github.com/micro/go-plugins/registry/consul"
        // rabbitmq transport
        "github.com/micro/go-plugins/transport/rabbitmq"
        // kafka broker
        "github.com/micro/go-plugins/broker/kafka"
)

func main() {
    registry := consul.NewRegistry()
    broker := kafka.NewBroker()
    transport := rabbitmq.NewTransport()

        service := micro.NewService(
                micro.Name("greeter"),
                micro.Registry(registry),
                micro.Broker(broker),
                micro.Transport(transport),
        )

    service.Init()
    service.Run()
}
```

编写插件

插件是一个建立在 Go 接口上的概念. 每个包都维护一个高级接口抽象. 只需实现接口并将其作为服务选项传递给它即可.

服务发现接口称为 Registry. 实现此接口的任何内容都可以用作注册表. 这同样适用于其他包.

```
type Registry interface {
    Register(*Service, ...RegisterOption) error
    Deregister(*Service) error
    GetService(string) ([]*Service, error)
    ListServices() ([]*Service, error)
    Watch() (Watcher, error)
    String() string
}
```

参阅 go-plugins 更好地了解实现细节.

发布订阅

发布订阅

> 前言: 用 Go Micro 的发布订阅构建事件驱动的微服务

概述

微服务是一种事件驱动的体系结构模式, 因此 Go Micro 使用消息代理接口构建异步消息的概念. 它可为您无缝地运行 protobuf 类型. 自动编码和解码消息, 因为它们从代理发送和接收.

默认情况下, go-micro 包括点对点 http 代理, 但可以通过 go-plugins 置换实现.

发布消息

使用 `topic` 名称和服务客户端创建一个新的发布者

```
p := micro.NewEvent("events", service.Client())
```

发布 proto 消息

```
p.Publish(context.TODO(), &proto.Event{Name: "event"})
```

订阅

创建消息处理程序. 它的签名应该是 `func(context.Context, v interface{}) error`

```
func ProcessEvent(ctx context.Context, event *proto.Event) error {
    fmt.Printf("Got event %+v\n", event)
    return nil
}
```

使用 `topic` 注册消息处理程序

```
micro.RegisterSubscriber("events", ProcessEvent)
```

有关完整示例, 可参阅 examples/pubsub

网络服务

网络服务

> 前言: Go Web 为 micro web 应用提供了一个接口

概述

Go Web 提供了一个很小的 HTTP Web 服务器库, 利用 go-micro 作为微服务世界中的一等公民创建 micro web 服务. 它包装了 go-micro,为您提供服务发现, 检测信号以及将 web 应用创建为微服务的能力.

特性

- **服务发现** - 服务在启动时自动注册在服务发现中. Go Web 包括一个具有预初始化往返器的 http.Client, 它利用服务发现, 以便您可以使用服务名称.
- **健康检测** - Go Web 应用将定期使用服务发现进行检测信号, 以提供实时更新. 如果服务失败, 将在预定义的过期时间后将其从注册表中删除.
- **自定义处理程序** - 指定您自己的 http 路由器以处理请求. 这允许您完全控制要路由到内部处理程序的方式.
- **静态服务** - Go Web 自动检测本地静态 `html` 目录, 如果未指定路由处理程序, 则提供文件. 对于那些希望将 JS Web 应用程序编写为微服务的用户的快速解决方案.

开始

依赖

Go Web 使用 Go Micro, 这意味着它需要服务发现

有关安装说明, 可参阅 go-micro

使用

```
service := web.NewService(
    web.Name("example.com"),
)

service.HandleFunc("/foo", fooHandler)

if err := service.Init(); err != nil {
    log.Fatal(err)
}

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

设置处理程序

您可能具有对 HTTP 处理程序的首选项, 因此请使用其他操作. 这失去了在发现中注册终结点的能力, 但我们会很快修复它.

```
import "github.com/gorilla/mux"

r := mux.NewRouter()
r.HandleFunc("/", indexHandler)
r.HandleFunc("/objects/{object}", objectHandler)

service := web.NewService(
    web.Handler(r)
)
```

调用服务

Go-web 包括一个具有自定义 http.RoundTripper 并使用服务发现 http.Client

```
c := service.Client()

rsp, err := c.Get("http://example.com/foo")
```

这将查找 `example.com` 服务发现以及路由到其中一个可用节点。

静态文件

Go Web 总是意味着注册 Web 应用， 其中大多数代码将用 JS 编写。 默认情况下， 如果 “/“ 上没有注册任何处理程序, 并且我们发现本地的 “html” 目录, 则将提供静态文件.

您将看到这样的日志输出.

```
2019/05/12 14:55:47 Enabling static file serving from /tmp/foo/html
```

如果要手动设置此路径, 请使用 StaticDir 选项. 如果指定了相对路径, 我们将使用 os.Getwd（）和前缀.

```
service := web.NewService(
    web.Name("example.com"),
    web.StaticDir("/tmp/example.com/html"),
)
```

包装器

包装器

> 前言: 包装器是 Go Micro 的中间件

概述

包装器是 Go Micro 的中间件. 我们希望创建一个可扩展的框架, 其中包含钩子, 以添加非核心要求的额外功能. 很多时候, 你需要执行的东西, 如验证, 跟踪等, 所以这提供了这样做的能力.

为此我们使用了“修饰器模式”.

使用

这里有一些示例用法和真实代码 examples/wrapper.

你可以在这里 go-plugins/wrapper 中找到一系列的包装器.

处理器

下面是一个示例服务处理程序包装器, 它记录传入的请求

```
// 实现 server.HandlerWrapper
func logWrapper(fn server.HandlerFunc) server.HandlerFunc {
    return func(ctx context.Context, req server.Request, rsp interface{}) error
 {
        fmt.Printf("[%v] server request: %s", time.Now(), req.Endpoint())
        return fn(ctx, req, rsp)
    }
}
```

创建服务时可以初始化

```
service := micro.NewService(
    micro.Name("greeter"),
    // wrap the handler
```

```
    micro.WrapHandler(logWrapper),
)
```

客户端

下面是客户端包装器的示例，该包装器记录发出的请求

```
type logWrapper struct {
    client.Client
}

func (l *logWrapper) Call(ctx context.Context, req client.Request, rsp interface
{}, opts ...client.CallOption) error {
    fmt.Printf("[wrapper] client request to service: %s endpoint: %s\n", req.Ser
vice(), req.Endpoint())
    return l.Client.Call(ctx, req, rsp)
}

// 实现 client.Wrapper 作为日志包装器 logWrapper
func logWrap(c client.Client) client.Client {
    return &logWrapper{c}
}
```

创建服务时可以初始化

```
service := micro.NewService(
    micro.Name("greeter"),
    // wrap the client
    micro.WrapClient(logWrap),
)
```

编写服务

编写服务

这是编写服务及其相关的内部内容的更详细的指南. 顶级 服务 接口是构建服务的主要组件. 它将 Go Micro 的所有基础包包装到一个便捷的接口中.

```
type Service interface {
    Init(...Option)
    Options() Options
    Client() client.Client
    Server() server.Server
    Run() error
    String() string
}
```

1. 初始化

像这样使用 `micro.NewService` 创建服务。

```
import "github.com/micro/go-micro/v2"

service := micro.NewService()
```

选项可以在创建期间传入.

```
service := micro.NewService(
        micro.Name("greeter"),
        micro.Version("latest"),
)
```

所有可用的选项都可以 在这里 找到.

Go Micro 还提供一种使用 `micro.Flags` 设置命令行标志的方法.

```
import (
        "github.com/micro/cli"
        "github.com/micro/go-micro/v2"
)
```

```
service := micro.NewService(
        micro.Flags(
                cli.StringFlag{
                        Name:  "environment",
                        Usage: "The environment",
                },
        )
)
```

要分析标志请使用 `service.Init` micro.Action` 此外访问标志使用该选项.

```
service.Init(
        micro.Action(func(c *cli.Context) {
                env := c.StringFlag("environment")
                if len(env) > 0 {
                        fmt.Println("Environment set to", env)
                }
        }),
)
```

Go Micro 提供预定义的标志, 如果调用 `service.Init` 这些标志将设置和解析. 在此处查看 所有标志

2. 定义 API

我们使用原型文件来定义服务 API 接口. 这是一种非常方便的方法来严格定义 API 并为服务器和客户端提供具体类型.

下面是一个示例定义。

greeter.proto

```
syntax = "proto3";

service Greeter {
    rpc Hello(Request) returns (Response) {}
}

message Request {
    string name = 1;
}

message Response {
    string greeting = 2;
}
```

在这里我们定义一个服务处理程序称为 Greeter 与 Hello 的方法, 它采取参数请求类型并返回响应.

3. 生成 API 接口

您需要以下规则来生成原型代码

- protoc
- protoc-gen-go
- protoc-gen-micro

我们使用 protoc, protoc-gen-go, protoc-gen-micro 来生成此定义的具体实现.

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

```
go get github.com/micro/micro/v2/cmd/protoc-gen-micro@master
```

```
protoc --proto_path=$GOPATH/src:. --micro_out=. --go_out=. greeter.proto
```

现在在发出请求时, 可以在服务器或客户端的 **handler** 中导入和使用生成的类型.

下面是生成的代码的一部分.

```
type Request struct {
    Name string `protobuf:"bytes,1,opt,name=name" json:"name,omitempty"`
}

type Response struct {
    Greeting string `protobuf:"bytes,2,opt,name=greeting" json:"greeting,omitempty"`
}

// Client API for Greeter service

type GreeterClient interface {
    Hello(ctx context.Context, in *Request, opts ...client.CallOption) (*Response, error)
}

type greeterClient struct {
    c           client.Client
```

```
    serviceName string
}

func NewGreeterClient(serviceName string, c client.Client) GreeterClient {
    if c == nil {
        c = client.NewClient()
    }
    if len(serviceName) == 0 {
        serviceName = "greeter"
    }
    return &greeterClient{
        c:           c,
        serviceName: serviceName,
    }
}

func (c *greeterClient) Hello(ctx context.Context, in *Request, opts ...client.C
allOption) (*Response, error) {
    req := c.c.NewRequest(c.serviceName, "Greeter.Hello", in)
    out := new(Response)
    err := c.c.Call(ctx, req, out, opts...)
    if err != nil {
        return nil, err
    }
    return out, nil
}

// Server API for Greeter service

type GreeterHandler interface {
    Hello(context.Context, *Request, *Response) error
}

func RegisterGreeterHandler(s server.Server, hdlr GreeterHandler) {
    s.Handle(s.NewHandler(&Greeter{hdlr}))
}
```

4. 实现处理程序

服务器需要注册 **处理程序** 以服务请求. 处理程序是一种公共类型, 具有符合签名 `func(ctx context.Context, req interface{}, rsp interface{}) error` 的公共方法.

如上所示, Greeter 接口的处理程序签名如下所示.

```
type GreeterHandler interface {
        Hello(context.Context, *Request, *Response) error
}
```

下面是 Greeter 处理程序的实现.

```
import proto "github.com/micro/examples/service/proto"

type Greeter struct{}

func (g *Greeter) Hello(ctx context.Context, req *pb.Request, rsp *pb.Response)
 error {
    rsp.Greeting = "Hello " + req.Name
    return nil
}
```

处理程序在服务中注册, 就像 http.Handler 一样.

```
service := micro.NewService(
    micro.Name("greeter"),
)

pb.RegisterGreeterHandler(service.Server(), new(Greeter))
```

您还可以创建双向流式处理器, 但我们会将该处理程序留给另一天.

5. 运行服务

该服务可以通过调用 `server.Run` 来运行. 这将导致服务绑定到配置中的地址 (默认为找到的第一个 RFC1918 接口和一个随机端口), 并侦听请求.

这还将在启动时向注册表注册服务, 并在发出终止信号时注销.

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

6. 完整的服务

greeter.go

```
package main

import (
        "log"

        "github.com/micro/go-micro/v2"
        pb "github.com/micro/examples/service/proto"

        "golang.org/x/net/context"
)

type Greeter struct{}

func (g *Greeter) Hello(ctx context.Context, req *pb.Request, rsp *pb.Response)
 error {
        rsp.Greeting = "Hello " + req.Name
        return nil
}

func main() {
        service := micro.NewService(
                micro.Name("greeter"),
        )

        service.Init()

        pb.RegisterGreeterHandler(service.Server(), new(Greeter))

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

注意服务发现机制需要运行，以便服务可以注册以被客户端和其他服务发现. 一个快速开始 在这里.

编写客户端

客户端 包用于查询服务. 创建服务时，将包含与服务器使用的初始包匹配的客户端.

查询上述服务非常简单.

```
// create the greeter client using the service name and client
greeter := pb.NewGreeterService("greeter", service.Client())
```

```
// request the Hello method on the Greeter handler
rsp, err := greeter.Hello(context.TODO(), &pb.Request{
    Name: "John",
})
if err != nil {
    fmt.Println(err)
    return
}

fmt.Println(rsp.Greeting)
```

`pb.NewGreeterClient` 获取用于发出请求的服务名称和客户端.

完整的例子可以在这里 go-micro/examples/service 中找到.

平台

概述

概述

> 前言: M3O 是一个无服务器微服务平台

获取访问权限

当我们处于封闭的测试阶段, 您必须通过邀请才能使用该平台. 请加入 等待列表, 并在 Slack 的 #platform 频道中的提醒我们, 以便推送到列表的顶部.

设置

在开始之前, 让我们确保在本地安装了最新版本的 Micro. 为此, 请在终端中运行以下命令:

```
rm $GOPATH/bin/micro
go get github.com/micro/micro/v2@master
```

接下来, 让我们转到 帐户 页面以创建新帐户. 拥有帐户后, 您将被重定向到 M3O 门户, 您可以在其中访问 API 令牌并开始使用 CLI. 从您的帐户设置复制令牌, 并在 CLI 上登录.

```
micro login --token $token
```

如果登录成功, 您将看到以下消息: `You have been logged in` .

编写您的第一个服务

如上所述, 虽然 M30 处于封闭测试版中, 但唯一可以部署的服务必须位于 github.com/micro/services 仓库中. 让我们关闭此仓库, 使用无检出标志来加快该过程.

```
git clone https://github.com/micro/services && cd services
```

接下来, 让我们创建第一个服务 (可以使用您觉得更好的名字)

```
micro new foobar && cd foobar
```

至此您已准备好新的微服务进行部署. 在部署之前, 我们只需构建 protobuf. 我们可以使用 Make 命令执行此操作:

```
make build
```

部署第一个服务

当您指示 M30 运行服务时, 它将提取平台仓库的最新源代码并运行您指定的任何服务. 首先, 让我们将更改推送到 GitHub:

```
git add . && git commit -m "Initialising service" && git push
```

接下来, 让我们使用 `micro run` 命令运行服务. 请注意, foobar 必须是要部署的服务的目录.

```
micro run --platform foobar
```

如果成功, 您将看到以下消息: `[Platform] Service foobar:latest created` . 我们可以通过运行以下功能检查部署的进度:

```
micro ps --platform
```

现在您将看到服务列表, 包括:

```
NAME        VERSION    SOURCE                  STATUS        BUILD    METADATA
foobar        latest     github.com/micro/services    running        n/a    owner
=n/a, group=n/a
```

与您的第一个服务交互

现在, 我们已经部署了第一个服务, 让我们跟它进行交互. 我们可以通过 平台 来实现.

入门

入门

> 前言: M3O 是一个无服务器微服务平台

概述

M3O 即将发布无服务器平台, 用于云内外的微服务开发. 通过提供一个完全托管的 Micro 服务平台, 它将消除管理基础架构的麻烦和云原生的复杂性.

特性

M3O 平台将包括以下功能:

- **托管服务**- 服务将在云中运行, 默认情况下进行管理.
- **自动配置** - 服务将自动配置与所需的注册表, 代理等
- **简化的调试** - 对于每个服务, 将以简单, 干净的方式提供统计数据, 日志和跟踪
- **从任何位置连接** - 能够连接和扩展平台以外的网络. 在本地或您自己的云中运行服务.

入门

在 slack 加入 #platform 频道来讨论更多. 我们很快就会透露细节.

服务

服务

> 前言: M3O 平台提供的服务

概述

M3O 平台是一个微服务开发的服务平台. 它建立在开源项目 Micro 和框架 Go Micro 之上.

该平台提供全局自动部署和配置, 简化调试和从任何位置进行连接. 它包括了 Micro 和 Go Micro 所有你喜爱的功能.

服务

该平台提供以下服务

- **Runtime** - 服务部署
- **Registry** - 服务发现
- **Broker** - 异步发布订阅消息
- **Proxy** - gRPC 通信服务代理
- **Store** - 分布式密钥值存储
- **Debug** - 用于调试的统计信息、日志和跟踪
- **Auth** - 用户和服务身份验证和授权
- **Events** - 事件流和时间序列存储

使用

使用 Go Micro 框架时, 将自动配置微服务以将平台与上述所有服务一起使用. 只需导入所需的包, 即可使用零代码更改.

```
示例待补充...
```

运行时

概述

概述

> 前言: Micro 是微服务开发的运行时

概述

Micro 解决了构建可扩展系统的关键要求. 它采用微服务体系结构模式并将其转换为一组工具, 作为平台的构建基块. Micro 处理分布式系统的复杂性, 并提供开发人员已经理解的简单抽象.

技术在不断发展. 基础结构堆栈总是在变化. Micro 是一个可插拔的工具包, 可以解决这些问题. 插入任何堆栈或底层技术. 使用 micro 构建面向未来的系统.

特性

运行时由以下功能组成:

- **API Gateway:** 使用具有服务发现的动态请求路由的单个 http 入口点. API 网关允许您在后端构建可扩展的微服务体系结构, 并在前端合并公共接口. micro api 通过发现和可插拔处理程序提供强大的路由为 http, grpc, websocket, publish events 等提供服务.
- **Interactive CLI:** 一个用于描述, 查询和直接从终端与您的平台和服务进行交互的命令行接口. CLI 为您提供了所有预期的命令, 以了解您的微服务发生了什么. 它还包括一个交互式模式.
- **Service Proxy:** 建立在 Go Micro 和 MUCP 协议之上的透明代理. 将服务发现, 负载平衡, 消息编码, 中间件, 传输和代理插件放置到一起. 可独立或与服务一起运行.
- **Service Templates:** 生成新的服务模板以快速入门. Micro 提供用于编写微服务的预定义模板. 始终以相同的方式启动, 构建相同的服务以提高工作效率.
- **SlackOps Bot:** 在您的平台上运行的机器人, 允许您从 Slack 本身管理应用程序. micro bot 支持 ChatOps 使您能够通过消息传送与您的团队一起完成所有事情. 它还包括创建可动态发现的 slack 命令服务力.
- **Web Dashboard:** Web 仪表板允许您浏览服务, 描述其终结点, 请求和响应格式, 甚至直接查询它们. 仪表板还包括内置命令行接口的体验, 适用于希望动态进入终端的开发人员.

安装 Micro

```
go get github.com/micro/micro/v2
```

或通过 Docker

```
docker pull micro/micro
```

最新版本二进制文件

```
# MacOS
curl -fsSL https://raw.githubusercontent.com/micro/micro/master/scripts/install.sh | /bin/bash

# Linux
wget -q  https://raw.githubusercontent.com/micro/micro/master/scripts/install.sh -O - | /bin/bash

# Windows
powershell -Command "iwr -useb https://raw.githubusercontent.com/micro/micro/master/scripts/install.ps1 | iex"
```

依赖

Micro 运行时有两个依赖项:

- 服务发现 - 用于命名解析
- 原型缓冲 - 用于代码生成

服务发现

服务发现用于名称解析, 路由和集中元数据. 所有服务都在发现中注册, 以便其他服务都可以找到它们.

Micro 使用 go-micro 注册表接口进行服务发现. 多播 DNS 是默认实现. 启用了零配置, 因此您不必为本地开发调整任何内容. 如果您使用的是 docker, kubernetes 或需要更有弹性的东西, 我们的建议是 etcd.

Etcd

Etcd 是一种高可用的服务发现机制

```
# install
brew install etcd

# run
etcd
```

通过任意命令输入 `--registry=etcd` 或设置环境变量 `MICRO_REGISTRY=consul`

```
# Use flag
micro --registry=etcd list services

# Use env var
MICRO_REGISTRY=etcd micro list services`
```

当不在同一主机上时可以通过以下命令指定 etcd 服务的地址

```
MICRO_REGISTRY_ADDRESS=10.0.0.1:2379
```

有关更多服务发现插件, 可参阅 go-plugins.

Protobuf

Protobuf 用于代码生成. 它减少了需要编写的样板代码量.

```
# 安装 protobuf
brew install protobuf

# 安装 protoc-gen-go
go get github.com/golang/protobuf/{proto,protoc-gen-go}

# 安装 protoc-gen-micro
go get github.com/micro/micro/v2/cmd/protoc-gen-micro@master
```

可参阅 protoc-gen-micro 查看更多信息

编写服务

Micro 包括新的模板生成, 以加快编写应用程序

可参阅 **go-micro** 获取更多的编写服务的内容

生成模板

这里我们将使用 `micro new`

指定 $GOPATH 的相对路径

```
micro new github.com/micro/example
```

该命令将输出

```
example/
    Dockerfile    # A template docker file
    README.md    # A readme with command used
    handler/    # Example rpc handler
    main.go        # The main Go program
    proto/        # Protobuf directory
    subscriber/    # Example pubsub Subscriber
```

使用 `protoc 编译 protobuf 代码`

```
protoc --proto_path=. --micro_out=. --go_out=. proto/example/example.proto
```

现在可以像其他任何应用程序一样运行

```
# 运行 micro service
micro server

# 进入到你的服务目录
cd github.com/micro/example

# 运行服务器
micro run --server .
```

示例

现在我们有一个使用 `micro new` 模板生成的应用程序，让我们测试一下.

- 列出服务
- 获取服务
- 调用服务
- 运行 API

- 调用 API

列出服务

每个服务都注册了服务发现，因此我们可以找到它.

```
micro list services
```

输出

```
etcd
go.micro.srv.example
topic:topic.go.micro.srv.example
```

示例应用注册的完全限定名为 `go.micro.srv.example`

获取服务

每个服务都使用唯一编号，地址和元数据进行注册.

```
micro get service go.micro.srv.example
```

输出

```
service  go.micro.srv.example

version latest

ID    Address    Port    Metadata
go.micro.srv.example-437d1277-303b-11e8-9be9-f40f242f6897    192.168.1.65    535
45    transport=http,broker=http,server=rpc,registry=etcd

Endpoint: Example.Call
Metadata: stream=false

Request: {
    name string
}

Response: {
    msg string
}
```

```
Endpoint: Example.PingPong
Metadata: stream=true

Request: {}

Response: {}

Endpoint: Example.Stream
Metadata: stream=true

Request: {}

Response: {}

Endpoint: Func
Metadata: subscriber=true,topic=topic.go.micro.srv.example

Request: {
    say string
}

Response: {}

Endpoint: Example.Handle
Metadata: subscriber=true,topic=topic.go.micro.srv.example

Request: {
    say string
}

Response: {}
```

调用服务

通过 CLI 进行 RPC 调用. 查询作为 json 发送.

```
micro call go.micro.srv.example Example.Call '{"name": "John"}'
```

输出

```
{
    "msg": "Hello John"
}
```

可参阅 cli 文档 查看更多详细信息.

现在让我们测试一下通过 HTTP 调用服务.

运行 API

micro api 是一个 http 网关, 可动态路由到后端服务

让我们运行它, 以便我们可以查询示例服务.

```
MICRO_API_HANDLER=rpc
MICRO_API_NAMESPACE=go.micro.srv
micro api
```

一些信息:

- `MICRO_API_HANDLER` 设置 http 处理程序
- `MICRO_API_NAMESPACE` 设置服务命名空间

调用 API

使用 json 向 api 发出 POST 请求

```
curl -XPOST -H 'Content-Type: application/json' -d '{"name": "John"}' http://loc
alhost:8080/example/call
```

输出

```
{"msg":"Hello John"}
```

可参阅 api 文档 查看更多详细信息.

插件

Micro 建立在 go-micro 之上, 使其成为可插拔的工具包.

Go-micro 为分布式系统基础结构提供抽象, 可以置换这些基础结构.

可插拔功能

可插拔的 micro 功能有:

- broker - 发布订阅消息经纪人
- registry - 服务发现
- selector - 客户端负载均衡
- transport - 请求-响应或双向流
- client - 管理上述功能的客户端
- server - 管理上述功能的服务器

可在 go-plugins 找到更多插件

使用插件

只需将它们链接到单独的文件中, 即可集成 go-micro 插件

首先创建 plugins.go 文件

```
import (
    // etcd v3 registry
    _ "github.com/micro/go-plugins/registry/etcdv3"
    // nats transport
    _ "github.com/micro/go-plugins/transport/nats"
    // kafka broker
    _ "github.com/micro/go-plugins/broker/kafka"
)
```

构建二进制

使用 Go 工具链重建 micro 的二进制可执行文件

```
# 本地使用
go build -i -o micro ./main.go ./plugins.go

# 用于 docker 镜像
CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -ldflags '-w' -i -o micr
o ./main.go ./plugins.go
```

启用插件

使用命令行标志或环境变量启用插件

```
# 变量标志
micro --registry=etcdv3 --transport=nats --broker=kafka [command]
```

```
# 环境变量
MICRO_REGISTRY=etcdv3 MICRO_TRANSPORT=nats MICRO_BROKER=kafka micro [command]
```

架构

架构

Micro 为微服务提供了基本构建基块. 它的目标是简化分布式系统开发. 由于微服务是一种体系架构模式, 因此 Micro 看起来可以通过工具在逻辑上分离职责.

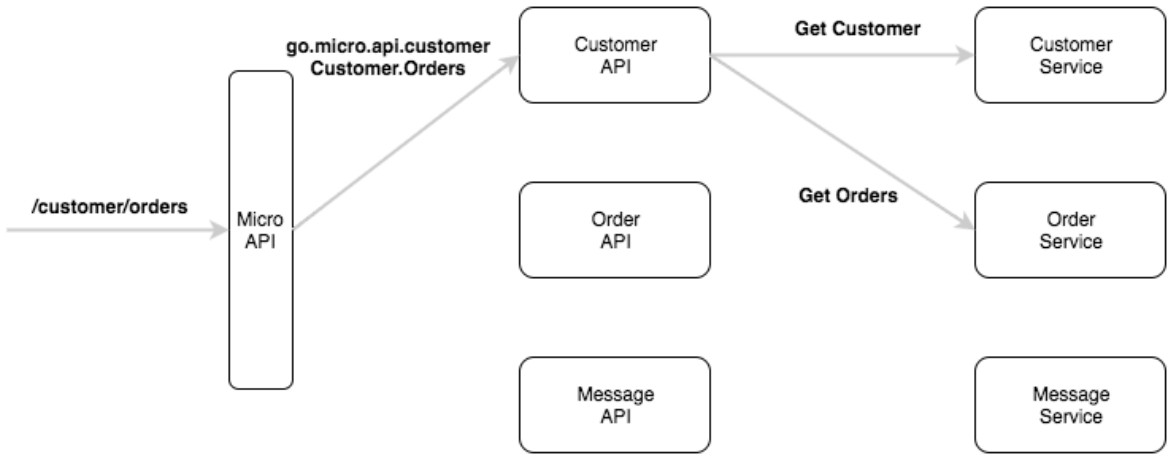
可参阅 micro 架构相关的博客文章 https://micro.mu/blog/2016/04/18/micro-architecture.html, 了解更多详细信息.

本节会详细介绍 micro 如何构建各种库/仓库如何相互关联.

运行时

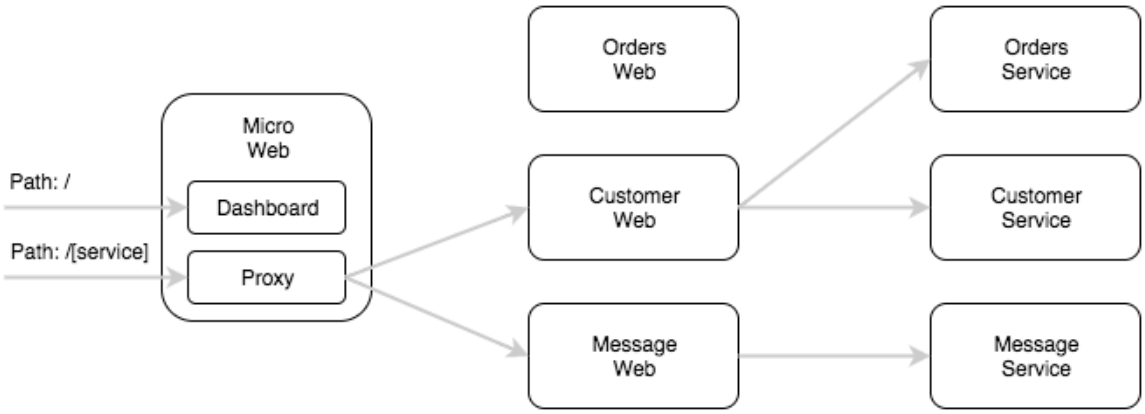
API

API 充当网关或代理, 以启用用于访问微服务的单个入口点. 它应在基础结构的边缘运行. 它将 HTTP 请求转换为 RPC 并转发到相应的服务.



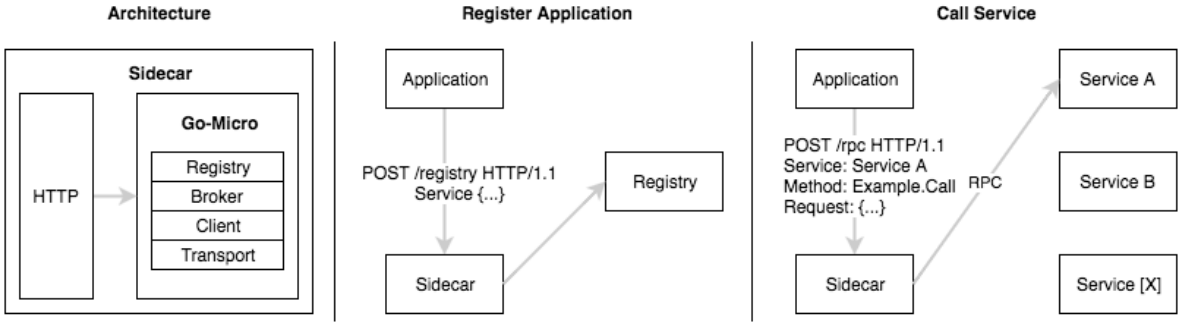
Web

这是 Go-micro 的 Web UI 版本, 允许视觉交互到环境. 将来它也将是一种聚合 micro web 服务的方式. 它包括一种代理 Web 应用的方法. /[name] 将路由到注册表中的服务. Web UI 添加了 “go.micro.web” 的命名前缀（可以配置）到名称, 在注册表中查找它, 然后反转对它的代理.



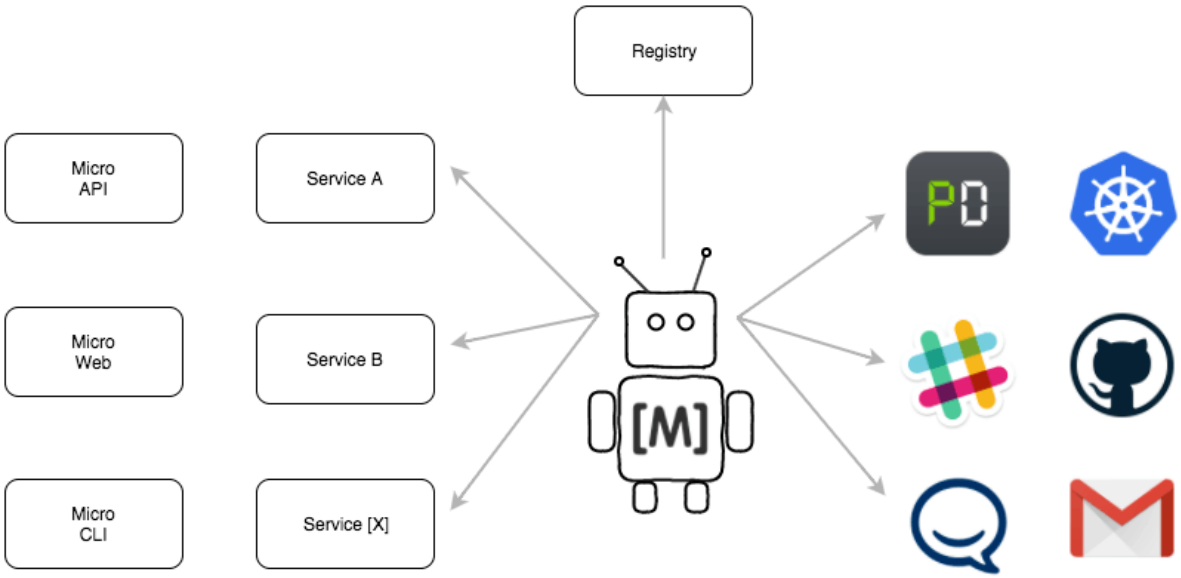
代理

代理是远程环境的一个命令行接口代理.



Bot

Bot A Hubot 风格的机器人位于您的微服务平台内, 可以通过 Slack, HipChat, XMPP, 等进行交互. 它通过消息传递提供了命令行接口的功能. 可以添加其他命令以自动执行常见操作任务.



CLI

Micro CLI 是 go-micro 的命令行版本, 它提供了一种观察和与正在运行的环境交互的方法.

插件

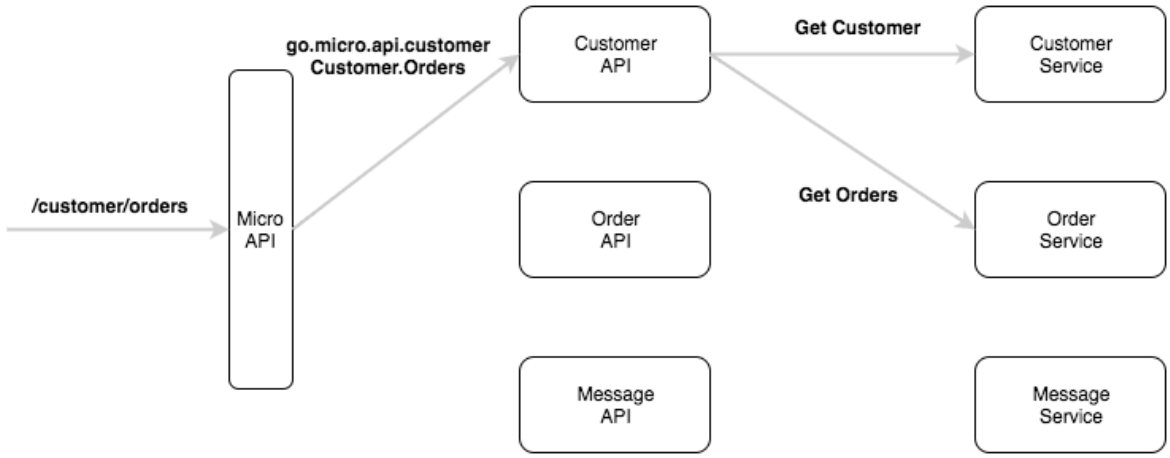
插件是向运行时添加其他功能的一种方式. 可参阅 概述.

接口网关

接口网关

> 前言: micro api 是一种接口网关

使用 API 网关 模式 为服务提供单个公共入口点. micro api 使用服务发现提供 HTTP 和动态路由.



概述

micro api 是一个 HTTP api. 对 API 的请求通过 HTTP 提供, 并通过服务发现进行路由. 它建立在 go-micro 的基础上, 利用它进行服务发现, 负载平衡, 编解码以及基于 RPC 的通信.

由于 micro api 在内部使用 go-micro, 这也使得它具备可插拔特性. 可参阅 go-plugins 了解 gRPC, kubernetes, etcd, nats, rabbitmq 以及更多支持. 此外它使用了允许配置处理程序的 go-api.

安装

```
go get -u github.com/micro/micro/v2
```

运行

```
# Default port 8080
micro api
```

使用 ACME

默认情况下，通过 **“Let’s Encrypt”** 使用 ACME 提供的安全服务

```
MICRO_ENABLE_ACME=true micro api
```

可指定主机白名单选项

```
MICRO_ENABLE_ACME=true
MICRO_ACME_HOSTS=example.com,api.example.com
micro api
```

设置 TLS 证书

API 支持使用 TLS 证书安全地提供服务

```
MICRO_ENABLE_TLS=true
MICRO_TLS_CERT_FILE=/path/to/cert
MICRO_TLS_KEY_FILE=/path/to/key
micro api
```

设置命名空间

API 使用命名空间在逻辑上分隔后端和面向公共的服务. 命名空间和 http 路径用于解析服务名称/方法，例如 `GET /foo HTTP/1.1` 路由到服务名称 `go.micro.api.foo` .

默认命名空间是 `go.micro.api` 也可以像下面这样调整

```
MICRO_NAMESPACE=com.example.api micro api
```

要禁用命名空间，请将其设置为空格. 这是一个特殊手段，后续我们将会修复.

```
MICRO_NAMESPACE=' '
```

示例

里我们有一个 3 层体系结构的示例

- `micro api` :(本地主机:8080) - 作为 http 入口点

- `api service`:(go.micro.api.greeter) - 作为一个公开的接口
- `backend service`:(go.micro.srv.greeter) - 内部后端服务

完整的示例是 示例/问候器

运行示例

```
# 下载示例
git clone https://github.com/micro/examples

# 启动服务
go run examples/greeter/srv/main.go

# 启动接口
go run examples/greeter/api/api.go

# 启动 micro api 接口网关
micro api
```

查询

通过 micro api 进行 HTTP 调用

```
curl "http://localhost:8080/greeter/say/hello?name=John"
```

HTTP 路径 /greet/say/hello 映射到服务 go.micro.api.greeter 的 Say.Hello 方法

绕过 api 服务直接通过 /rpc 调用后端接口服务

```
curl -d 'service=go.micro.srv.greeter'
     -d 'method=Say.Hello'
     -d 'request={"name": "John"}'
     http://localhost:8080/rpc
```

完全与 JSON 相同的调用

```
curl -H 'Content-Type: application/json'
     -d '{"service": "go.micro.srv.greeter", "method": "Say.Hello", "request":
 {"name": "John"}}'
     http://localhost:8080/rpc
```

接口

micro api 提供以下 HTTP api

```
- /[service]/[method]    # HTTP paths are dynamically mapped to services
- /rpc                   # Explicitly call a backend service by name and method
```

有关示例, 见下文

处理器

处理程序是管理请求路由的 HTTP 处理程序.

默认处理程序使用注册表中的终结点元数据来确定服务路由. 如果未找到路由匹配项, 它将回退到 “rpc” 处理程序. 您可以使用 go-api 在注册时配置路由.

API 具有以下可配置的请求处理程序.

- `api` - 处理任何 HTTP 请求. 通过 RPC 完全控制 http 请求/响应.
- `rpc` - 处理 json 和协议缓冲的 POST 请求. 转发为 RPC.
- `proxy` - 作为反向代理处理 HTTP 并转发.
- `event` - 处理任何 HTTP 请求并发布到消息总线.
- `web` - HTTP 反向代理, 包括 web sockets.

可使用 `/rpc` 终结点选项绕过处理程序

API 处理程序

API 处理程序提供任何 HTTP 请求, 并将转发为具有特定格式的 RPC 请求.

- Content-Type：任何
- Body: 任何
- Forward Format: api.Request/api.Response
- Path: `/[service]/[method]`
- Resolver: 路径用于解析服务和方法
- Configure：标记 `--handler=api` 或环境变量 `MICRO_API_HANDLER=api`

RPC 处理程序

RPC 处理程序为 json 或 protobuf HTTP POST 请求提供服务, 并作为 RPC 请求转发.

- Content-Type: `application/json` 或 `application/protobuf`
- Body: JSON 或 Protobuf
- Forward Format: 基于内容的 json-rpc 或 proto-rpc
- Path: `/[service]/[method]`
- Resolver: 路径用于解析服务和方法
- Configure：标记 `--handler=rpc` 或环境变量 `MICRO_API_HANDLER=rpc`
- 未指定处理程序时的默认处理程序

代理处理程序

代理处理程序是具有内置服务发现的 http 反向代理.

- Content-Type: 任何
- Body: 任何
- Forward Format: HTTP 反向代理
- Path: `/[service]`
- Resolver: 路径用于解析服务名称
- Configure: 标记 `--handler=proxy` 或环境变量 `MICRO_API_HANDLER=proxy`
- REST 可作为微服务在 API 的后端实现

事件处理程序

事件处理程序使用 go-micro 代理在消息总线上提供 HTTP 并将请求作为消息转发.

- Content-Type: 任何
- Body: 任何
- Forward Format: 格式为 go-api/proto.Event 的请求
- Path: `/[topic]/[event]`
- Resolver: 路径用于解决主题和事件名称
- Configure：标记 `--handler=event` 或环境变量 `MICRO_API_HANDLER=event`

Web 处理程序

Web 处理程序是一个 http 保留代理, 具有内置服务发现和 web 套接字支持.

- Content-Type: 任何
- Body: 任何

- Forward Format: HTTP 反向代理, 包括 web 套接字
- Path: `/[service]`
- Resolver: 路径用于解析服务名称
- Configure: 标记 `--handler=web` 或环境变量 `MICRO_API_HANDLER=web`

RPC 终结点

/rpc 终结点允许您绕过主处理程序直接与任何服务通信

- 请求参数
  - `service` - 设置服务名称
  - `method` - 设置服务方法
  - `request` - 请求正文
  - `address` - 可选地将主机地址指定为目标

调用示例:

```
curl -d 'service=go.micro.srv.greeter'
     -d 'method=Say.Hello'
     -d 'request={"name": "Bob"}'
     http://localhost:8080/rpc
```

可在 github.com/micro/examples/api 中查找工作示例

解析器

使用命名空间值和 HTTP 路径的微动态路由到服务.

默认命名空间为 `go.micro.api` . 通过 `--namespace` 或 `MICRO_NAMESPACE=` 设置命名空间.

下面将介绍所使用的解析器.

RPC 解析器

RPC 服务具有名称 (go.micro.api.greet) 和方法 (Greeter.Hello).

网址解析如下:

路径	服务	方法

路径	服务	方法
/foo/bar	go.micro.api.foo	Foo.Bar
/foo/bar/baz	go.micro.api.foo	Bar.Baz
/foo/bar/baz/cat	go.micro.api.foo.bar	Baz.Cat

可版本化 API URL 可轻松映射到服务名称:

路径	服务	方法
/foo/bar	go.micro.api.foo	Foo.Bar
/v1/foo/bar	go.micro.api.v1.foo	Foo.Bar
/v1/foo/bar/baz	go.micro.api.v1.foo	Bar.Baz
/v2/foo/bar	go.micro.api.v2.foo	Foo.Bar
/v2/foo/bar/baz	go.micro.api.v2.foo	Bar.Baz

代理解析器

使用代理处理程序, 我们只需要处理解析服务名称. 因此 RPC 解析器略有不同.

URLS 解析如下：

路径	服务	服务路径
/foo	go.micro.api.foo	/foo
/foo/bar	go.micro.api.foo	/foo/bar
/greeter	go.micro.api.greeter	/greeter
/greeter/:name	go.micro.api.greeter	/greeter/:name

Web 仪表盘

Web 仪表盘

> 前言: Micro Web 提供了一个用于可视化和浏览服务的仪表板

Web 仪表板提供了一个可视化工具, 用于探索服务, 并为基于 web 的 micro 服务提供内置 web 代理.

使用

```
micro web
```

浏览地址 localhost:8082 看看

使用 ACME

micro web 仪表板通过 “Let’s Encrypt” 支持 ACME. 它可以自动获取你域名的 TLS 证书.

```
micro --enable_acme web
```

可选地指定主机白名单

```
micro --enable_acme --acme_hosts=example.com,api.example.com web
```

设置 TLS 证书

仪表板支持使用 TLS 证书安全地提供服务

```
micro --enable_tls --tls_cert_file=/path/to/cert --tls_key_file=/path/to/key web
```

Web 服务

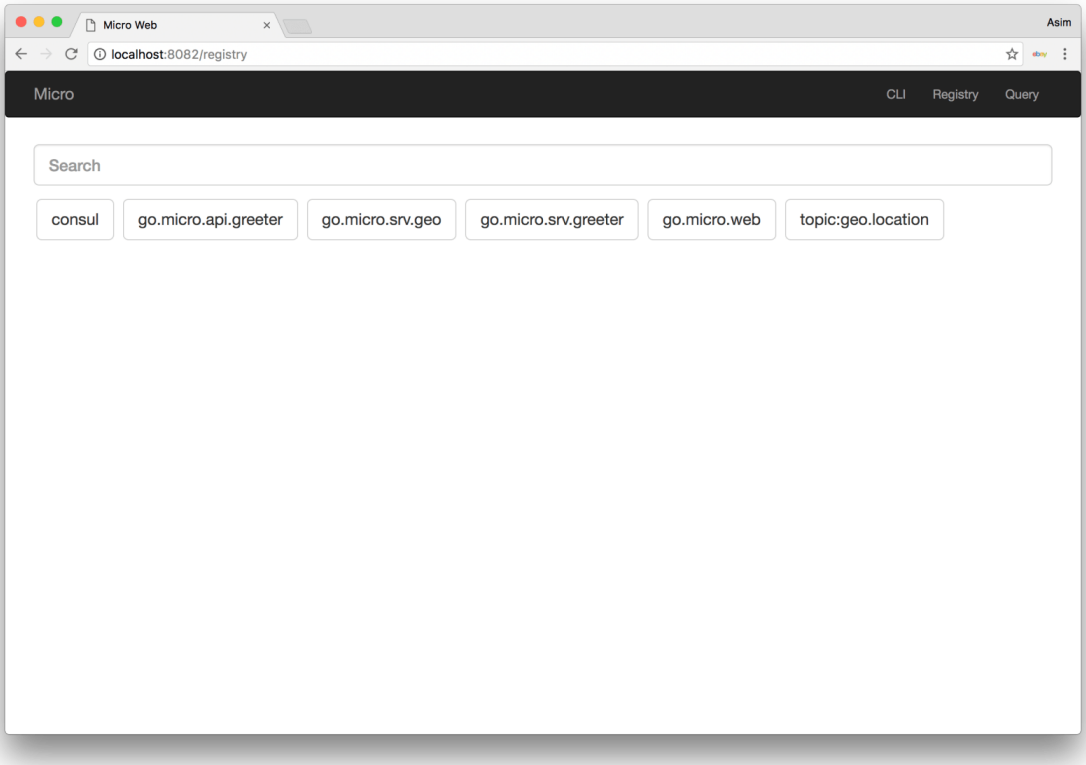
Web 仪表板具有 web 服务的内置代理. 这是将 Web 应用程序构建为微服务的想法, 您可以通过 go-micro/web 包进行.

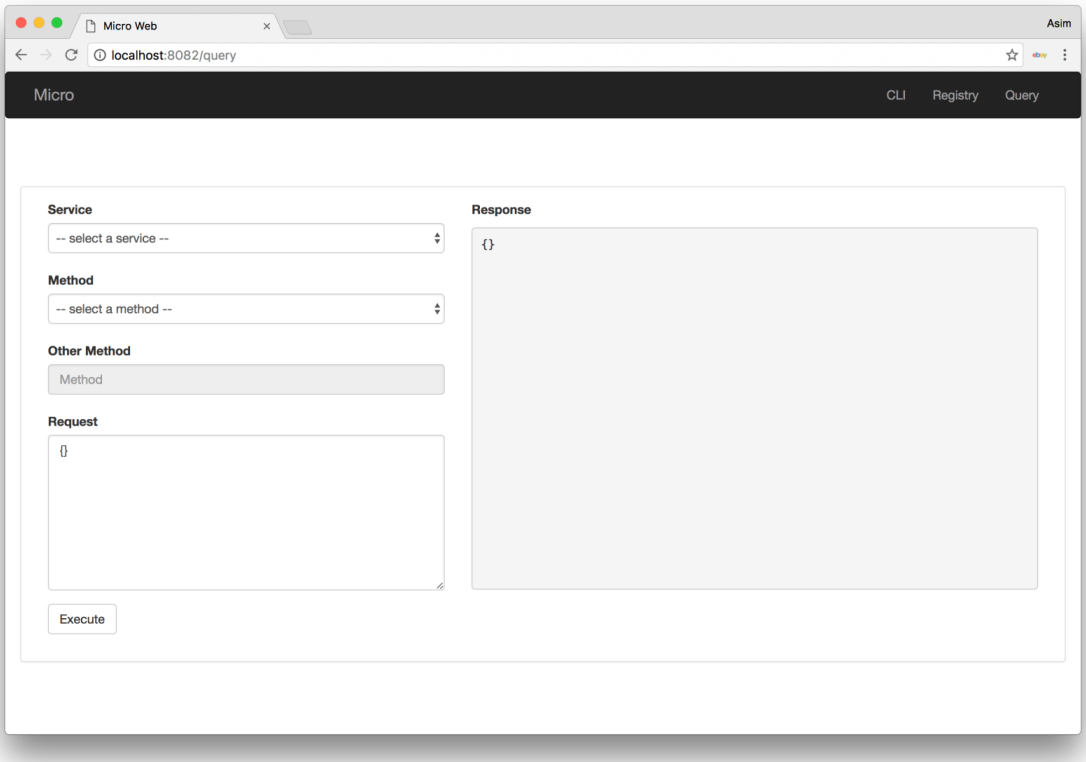
路由

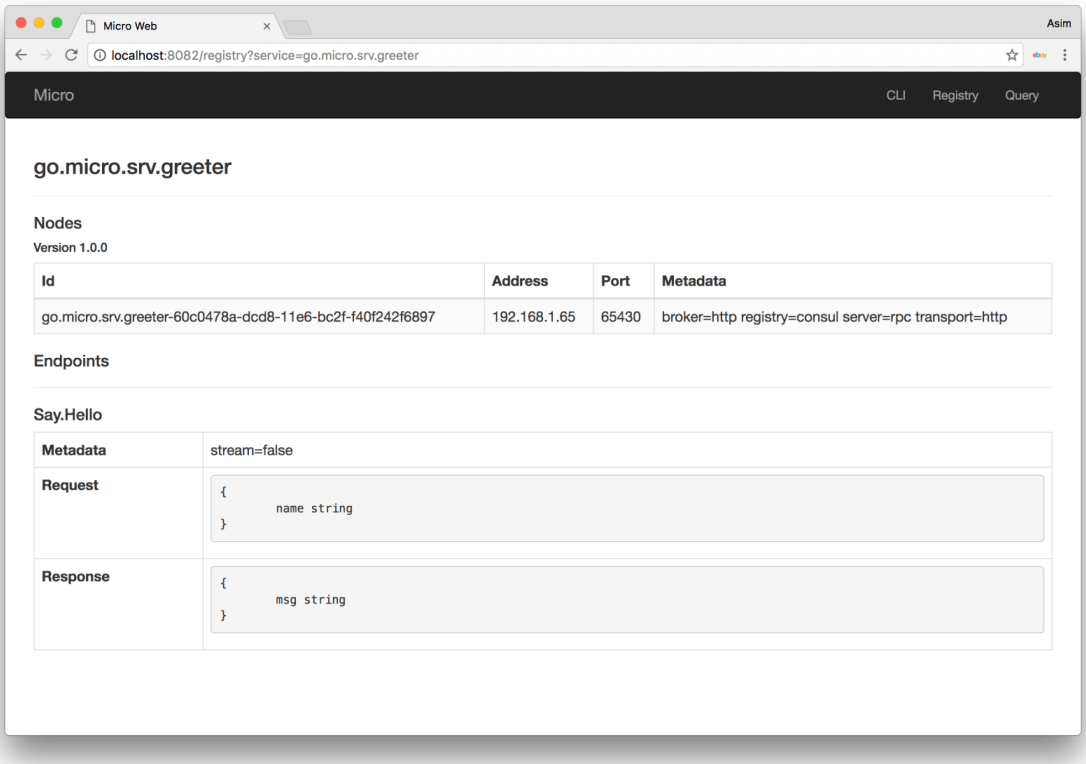
Web 服务与 API 服务非常类似, 因为它们的命令是一致的. 默认命名空间为 “go.micro.web”.

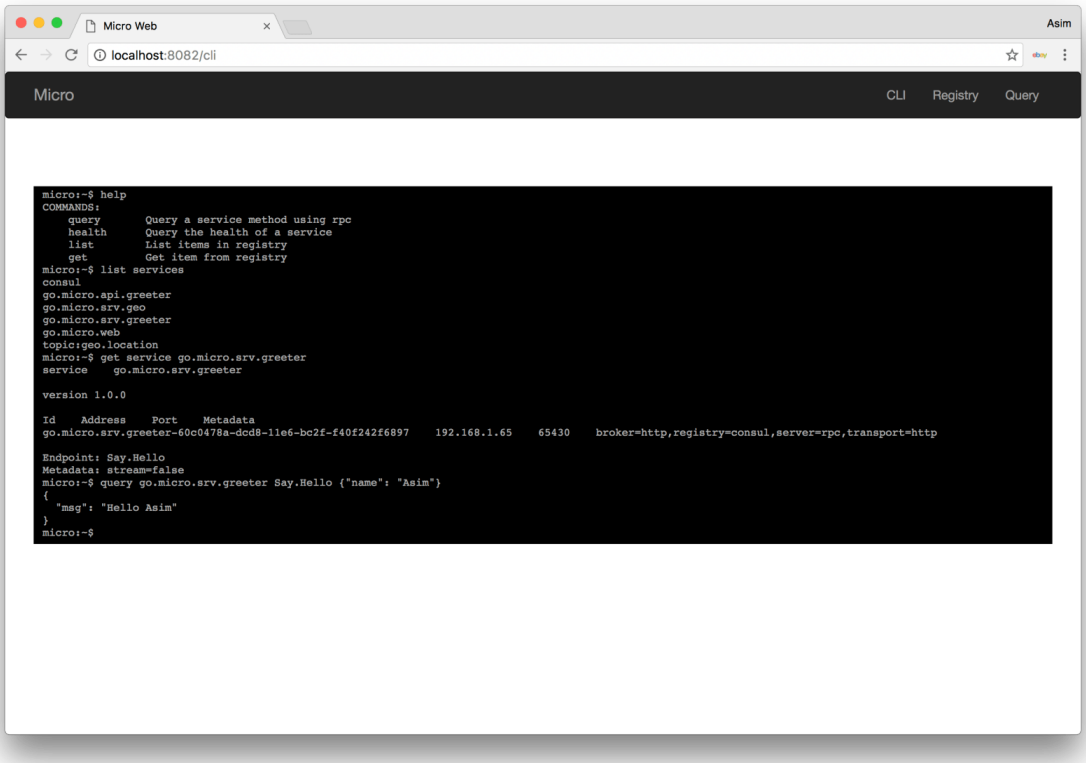
当请求例如 `/foo` 命中 web 代理时, 它将路由到具体服务 `go.micro.web.foo` . 这就是您的服务应该调用的; `namespace + path`

截图




Micro Web
Asim
localhost:8082/query
Micro
CLI
Registry
Query
Service
-- select a service --
Method
-- select a method --
Other Method
Method
Request
{}
Execute
Response
{}


Micro Web
Asim
localhost:8082/registry?service=go.micro.srv.greeter
Micro
CLI
Registry
Query
go.micro.srv.greeter
Nodes
Version 1.0.0
Id
Address
Port
Metadata
go.micro.srv.greeter-60c0478a-dcd8-11e6-bc2f-f40f242f6897
192.168.1.65
65430
broker=http registry=consul server=rpc transport=http
Endpoints
Say.Hello
Metadata
stream=false
Request
{
name string
}
Response
{
msg string
}


Micro Web
Asim
localhost:8082/cli
Micro
CLI
Registry
Query
micro:~$ help
COMMANDS:
query Query a service method using rpc
health Query the health of a service
list List items in registry
get Get item from registry
micro:~$ list services
consul
go.micro.api.greeter
go.micro.srv.geo
go.micro.srv.greeter
go.micro.web
topic:geo.location
micro:~$ get service go.micro.srv.greeter
service go.micro.srv.greeter
version 1.0.0
Id Address Port Metadata
go.micro.srv.greeter-60c0478a-dcd8-11e6-bc2f-f40f242f6897 192.168.1.65 65430 broker=http,registry=consul,server=rpc,transport=http
Endpoint: Say.Hello
Metadata: stream=false
micro:~$ query go.micro.srv.greeter Say.Hello {"name": "Asim"}
{
"msg": "Hello Asim"
}
micro:~$

调试

调试

要进行 micro 或 go-micro 的调试非常简单, 只需启用以下环境变量即可.

日志

启用调试日志记录

```
MICRO_LOG_LEVEL=debug
```

支持的日志级别是

```
trace
debug
error
info
```

查看服务日志

```
micro log [service]
```

分析

通过 pprof 启用分析

```
MICRO_DEBUG_PROFILE=http
```

这将在 :6060 端口上启动一个 http 服务端

统计

查看当前运行时统计信息

```
micro stats [service]
```

健康

查看服务是否正在运行并响应 RPC 查询

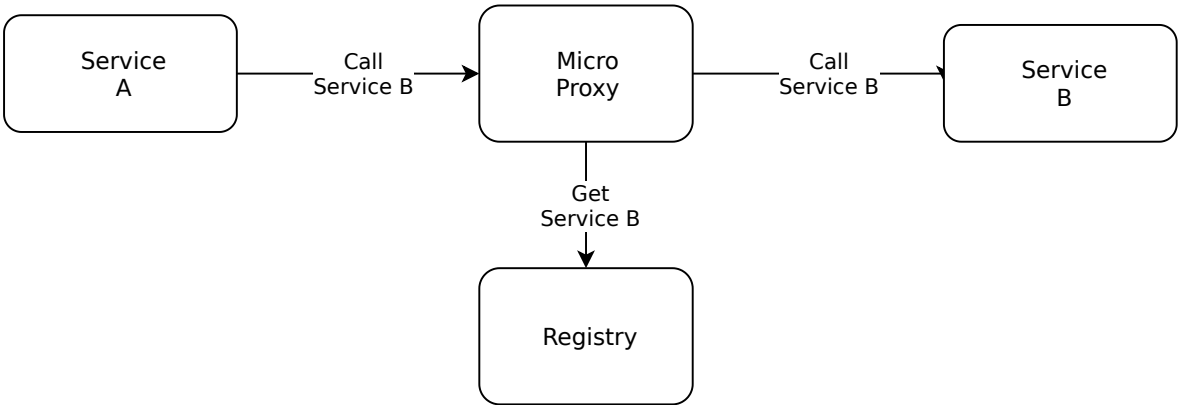
```
micro health [service]
```

服务代理

服务代理

> 前言: micro proxy 是一个服务到服务的代理.

服务代理是充当从一个服务到另一个服务的请求的中介服务器.



概述

微代理提供了 go-micro 框架的代理实现. 这将 go-micro 功能整合到一起, 从而允许将服务发现, 负载均衡, 容错, 插件, 包装器等整合到代理本身. 与其针对基础结构级别问题更新每个 Go Micro 应用, 不如将其放入代理中. 它还允许将任何语言与客户端集成, 而不必实现所有功能.

运行代理

启动代理

```
micro proxy
```

服务器地址是动态的, 但可以通过如下配置指定.

```
MICRO_SERVER_ADDRESS=localhost:9090 micro proxy
```

代理服务

现在代理正在运行, 您可以很简单地通过它代理请求.

现在像下面这样启动你的 micro 应用程序

```
MICRO_PROXY=go.micro.proxy go run main.go
```

您的服务将在发现中查找代理，然后使用它路由任何请求. 如果存在多个代理，它将平衡分发它们的请求. 它还将在本地缓存代理地址.

如果希望通过单个代理发送请求，请指定其地址，如下所示.

```
MICRO_PROXY=localhost:9090 go run main.go
```

确保代理在指定的地址上运行.

```
MICRO_SERVER_ADDRESS=localhost:9090 micro proxy
```

单个终结点

将代理用作单个终结点的前代理

```
MICRO_SERVER_NAME=helloworld
MICRO_PROXY_ENDPOINT=localhost:10001
micro proxy
```

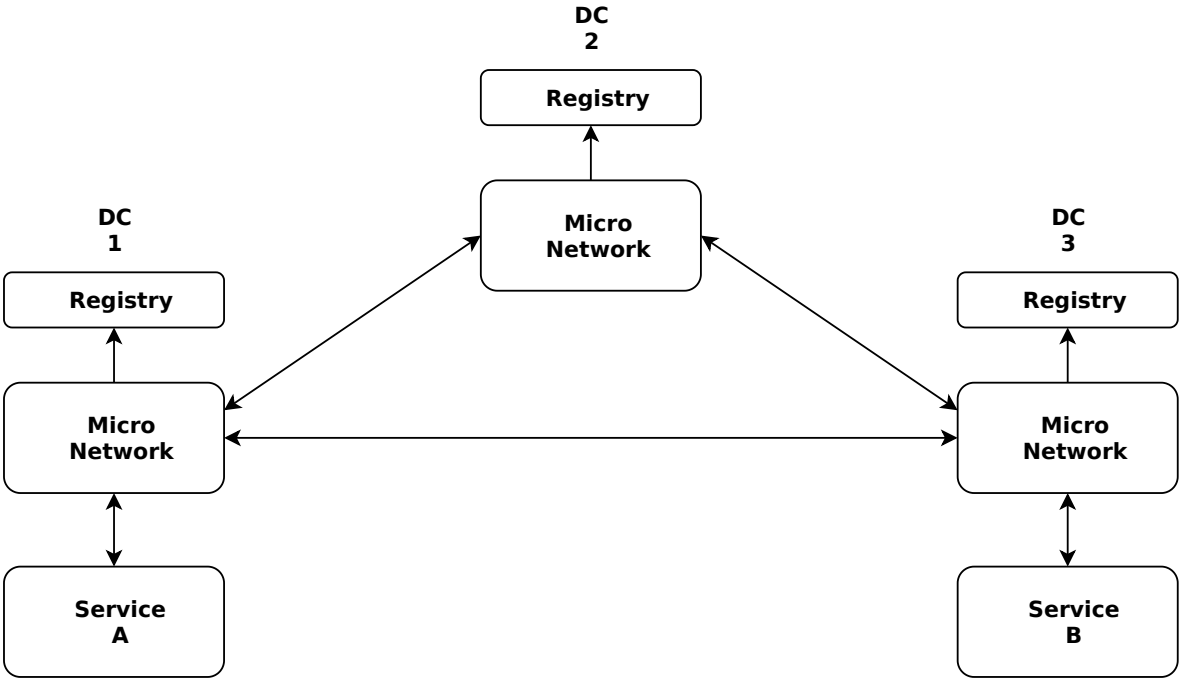
对 helloworld 的所有请求都将发送到后端地址 localhost:10001

服务网络

服务网络

> 前言: micro network 服务是一种多云服务网络解决方案

micro netwrok 服务是一种多云服务网络解决方案, 适用于公有云及私有云环境.



概述

micro network 提供多云服务功能, 并构建了一个大规模的扁平网络, 所有服务都可以通过该网络相互通信. 它利用我们的代理, 路由器, 隧道和网络包在 go-micro 中生成跨任何环境的全局路由.

网络基于本地服务注册表生成路由表, 并在节点之间共享该表. 它在路由器和代理中生成, 因此对任何网络节点发出的任何请求都可以通过全局网络路由. 它首先优先处理本地路由, 如果需要 可以在网络链中跳跃最多 3 个跃点.

运行网络

启动网络种子节点 (在 :8085 端口上运行)

```
micro network
```

在连接到第一个节点的不同环境中启动下一个节点 (假设其在 10.0.0.1:8085 运行)

```
micro network --nodes=10.0.0.1:8085
```

网络服务

现在您可以列出节点, 路由, 服务和图形

```
# 列出节点
micro network nodes

# 列出路由
micro network routes

# 列出服务
micro network services

# 打印图表
micro network graph
```

现在通过网络发出的任何请求都将接近另一端的服务.

将代理设置为使用网络

```
MICRO_PROXY=go.micro.network go run main.go
```

您的服务将引导所有流量通过网络.

认证

指定网络令牌以限制对网络的访问.

```
MICRO_NETWORK_TOKEN=foobar micro network
```

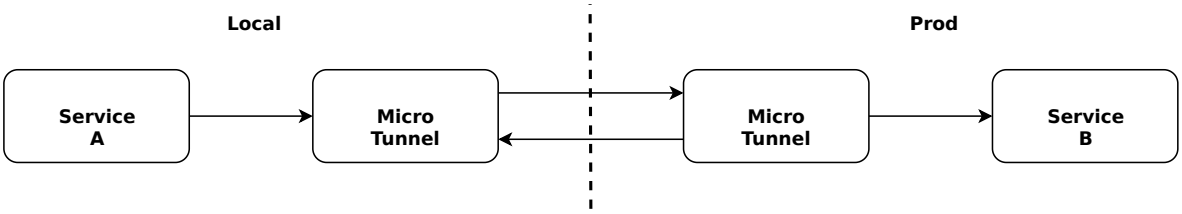
节点必须提供有效且匹配的令牌才能加入网络. 默认令牌为 “go.micro.tunnel”, 它允许任何节点在它们之间联接和通信.

服务隧道

服务隧道

> 前言: micro tunnel 是一种点对点隧道.

服务隧道是一种用于访问远程环境中的服务的点对点隧道.



概述

micro tunnel 提供了一种跨远程环境访问服务的方法. 这是非常有用当你想打通到 staging, prod 或公开本地服务到外部, 而无需使用任何其他的东西, 如 openvpn 或 wireguard , 这将暴露你网络中的所有东西.

运行隧道

启动隧道服务器 (在 :8083 端口上运行)

```
micro tunnel
```

隧道服务

现在隧道正在运行, 您可以通过本地隧道连接它.

```
micro tunnel --server=remote.env:8083
```

现在通过隧道发出的任何请求都将链接到另一边的服务.

将代理设置为使用隧道

```
MICRO_PROXY=go.micro.tunnel go run main.go
```

您的服务将引导所有流量通过隧道.

认证

指定隧道令牌以限制可以隧道进入环境的人员的访问. 令牌必须在隧道客户端和服务器之间匹配, 否则连接将被拒绝.

```
MICRO_TUNNEL_TOKEN=foobar go run main.go
```

默认情况下令牌 “micro” 允许任何人通过隧道进行连接.

命令行

命令行

micro cli

micro cli 是 micro 工具包中的一个命令行接口.

开始

安装

```
go get github.com/micro/micro/v2
```

交互模式

使用 cli 作为交互式提示符

```
micro cli
```

在交互模式中使用以下命令删除 `micro`

示例用法

列出服务

```
micro list services
```

列出服务

```
micro get service go.micro.srv.example
```

输出

```
go.micro.srv.example

go.micro.srv.example-fccbb6fb-0301-11e5-9f1f-68a86d0d36b6    [::]    62421
```

调用服务

```
micro call go.micro.srv.example Example.Call '{"name": "John"}'
```

输出

```
{
    "msg": "go.micro.srv.example-fccbb6fb-0301-11e5-9f1f-68a86d0d36b6: Hello John"
}
```

服务运行状况

```
micro health go.micro.srv.example
```

输出

```
node          address:port          status
go.micro.srv.example-fccbb6fb-0301-11e5-9f1f-68a86d0d36b6          [::]:62421          ok
```

注册/注销

```
micro register service '{"name": "foo", "version": "bar", "nodes": [{"id": "foo-1", "address": "127.0.0.1", "port": 8080}]}'
```

```
micro deregister service '{"name": "foo", "version": "bar", "nodes": [{"id": "foo-1", "address": "127.0.0.1", "port": 8080}]}'
```

代理远程环境

使用 `micro proxy`

在针对远程环境进行开发时，您可能无法直接访问服务发现，这使得使用 CLI 变得困难. `micro proxy` 为此诸如此类问题提供了 http 代理解决方案.

在远程环境中运行代理

```
micro proxy
```

设置环境变量 `MICRO_PROXY_ADDRESS` ，以便 cli 知道使用代理

```
MICRO_PROXY_ADDRESS=staging.micro.mu:8081 micro list services
```

使用

```
NAME:
   micro - A cloud-native toolkit

USAGE:
   micro [global options] command [command options] [arguments...]

VERSION:
   0.8.0

COMMANDS:
    api         Run the micro API
    bot         Run the micro bot
    registry    Query registry
    call    Call a service or function
    query    Deprecated: Use call instead
    stream    Create a service or function stream
    health    Query the health of a service
    stats    Query the stats of a service
    list    List items in registry
    register    Register an item in the registry
    deregister    Deregister an item in the registry
    get         Get item from registry
```

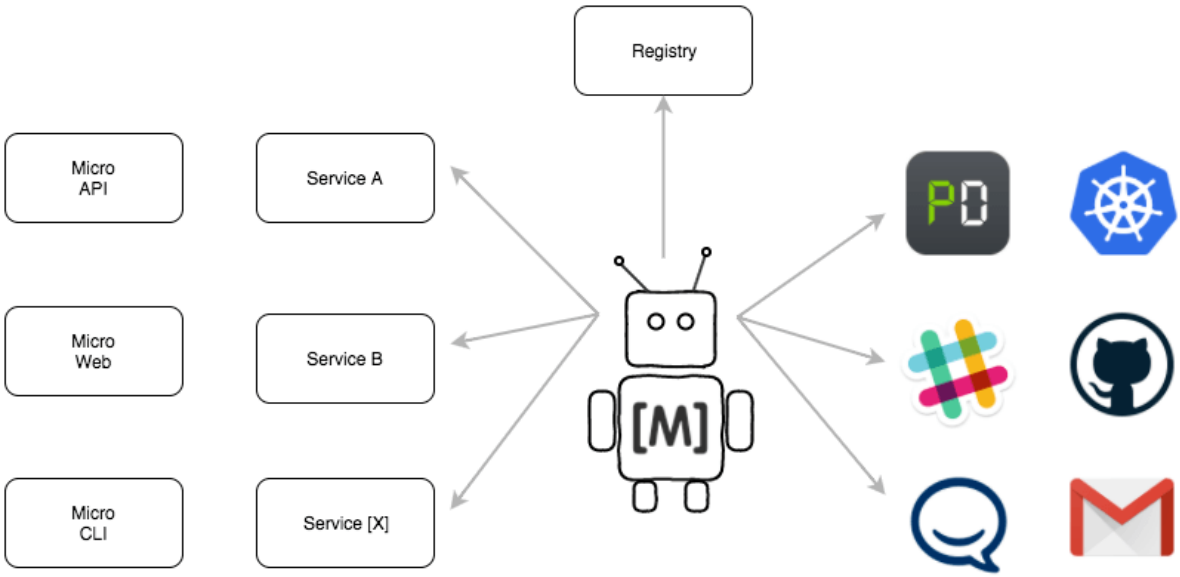
```
    proxy    Run the micro proxy
    new        Create a new micro service by specifying a directory path relativ
e to your $GOPATH
    web        Run the micro web app
```

Slack 机器人

Slack 机器人

micro bot

micro bot 是一个机器人, 位于您的微服务环境中, 您可以通过 Slack, HipChat, XMPP 等进行交互. 它通过消息模拟 CLI 的功能.



输入支持

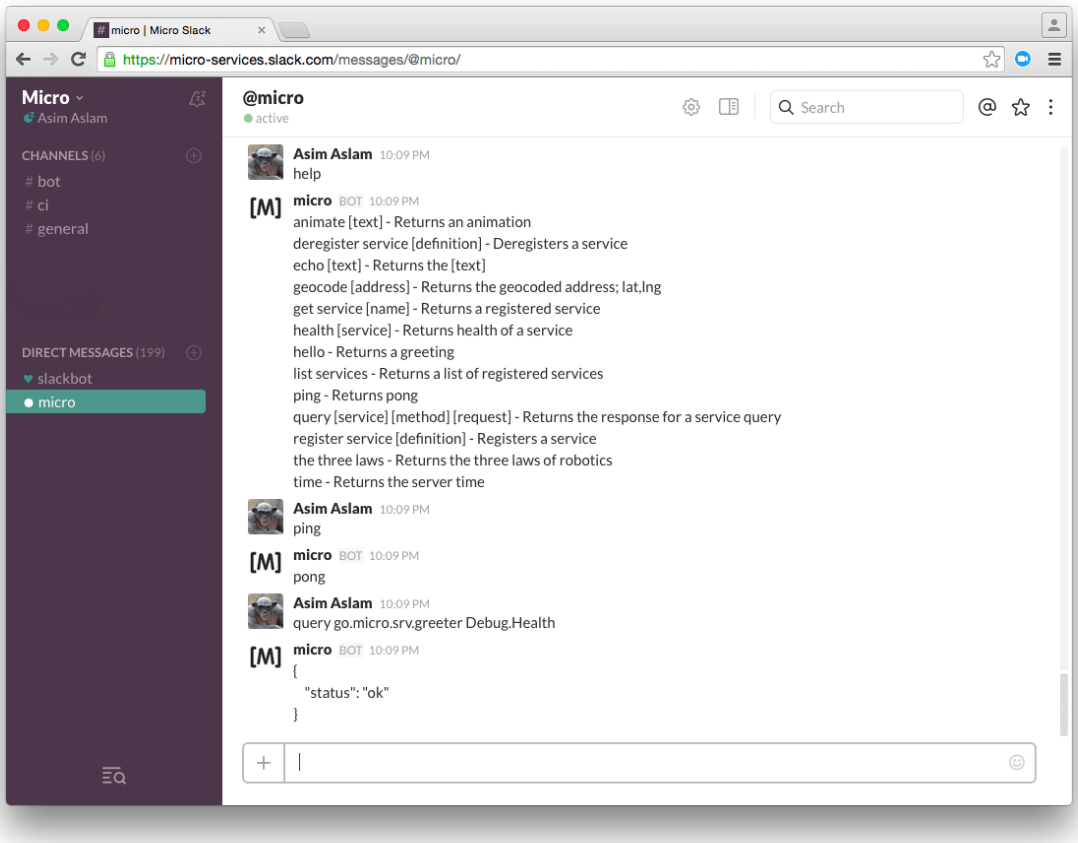
- Discord
- Slack
- Telegram

入门

使用 Slack

运行并使用 slack 输入参数

```
micro bot --inputs=slack --slack_token=SLACK_TOKEN
```



多个输入

通过指定逗号分隔列表使用多个输入

```
micro bot --inputs=discord,slack --slack_token=SLACK_TOKEN --discord_token=DISCORD_TOKEN
```

帮助

在 slack 中

```
micro help

deregister service [definition] - Deregisters a service
echo [text] - Returns the [text]
get service [name] - Returns a registered service
health [service] - Returns health of a service
hello - Returns a greeting
list services - Returns a list of registered services
ping - Returns pong
query [service] [method] [request] - Returns the response for a service query
```

```
register service [definition] - Registers a service
the three laws - Returns the three laws of robotics
time - Returns the server time
```

添加新命令

命令是自动程序基于基于文本的模式匹配执行的函数.

编写命令

```
import "github.com/micro/go-micro/v2/agent/command"

func Ping() command.Command {
    usage := "ping"
    description := "Returns pong"

    return command.NewCommand("ping", usage, desc, func(args ...string) ([]byte,
error) {
        return []byte("pong"), nil
    })
}
```

注册命令

可使用由 golang/regexp.Match 匹配的模式键将命令添加到 Commands 集合 .

```
import "github.com/micro/go-micro/v2/agent/command"

func init() {
    command.Commands["^ping$"] = Ping()
}
```

重新构建 micro

生成二进制

```
cd github.com/micro/micro

// For local use
go build -i -o micro ./main.go

// For docker image
```

```
CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -ldflags '-w' -i -o micr
o ./main.go
```

添加新输入

输入是用于通信的插件, 例如 Slack, HipChat, XMPP, IRC, SMTP 等等.

可用以下方式添加新输入.

编写输入

编写实现了输入接口的输入.

```
type Input interface {
    // Provide cli flags
    Flags() []cli.Flag
    // Initialise input using cli context
    Init(*cli.Context) error
    // Stream events from the input
    Stream() (Conn, error)
    // Start the input
    Start() error
    // Stop the input
    Stop() error
    // name of the input
    String() string
}
```

注册输入

将输入添加到 Inputs 集合.

```
import "github.com/micro/go-micro/v2/agent/input"

func init() {
    input.Inputs["name"] = MyInput
}
```

重新构建 Micro

生成二进制可执行文件

```
cd github.com/micro/micro

// For local use
go build -i -o micro ./main.go

// For docker image
CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -ldflags '-w' -i -o micr
o ./main.go
```

命令即服务

micro bot 支持将命令创建为微服务的能力.

它是如何工作的?

机器人监视服务注册表的服务与它的命名空间. 默认命名空间为 `go.micro.bot` . 此命名空间中的任何服务将自动添加到可用命令列表中. 当执行命令时, 自动程序将使用 `Command.Exec` 方法调用服务. 它还希望该 `Command.Help` 方法存在使用情况信息.

服务接口如下所示, 可在这里找到 go-bot/proto

```
syntax = "proto3";

package go.micro.bot;

service Command {
    rpc Help(HelpRequest) returns (HelpResponse) {};
    rpc Exec(ExecRequest) returns (ExecResponse) {};
}

message HelpRequest {
}

message HelpResponse {
    string usage = 1;
    string description = 2;
}

message ExecRequest {
    repeated string args = 1;
}

message ExecResponse {
    bytes result = 1;
```

```
    string error = 2;
}
```

示例

这是一个命令即服务的示例

```
package main

import (
    "fmt"
    "strings"

    "github.com/micro/go-micro/v2"
    "golang.org/x/net/context"

    proto "github.com/micro/go-micro/v2/agent/proto"
)

type Command struct{}

// Help returns the command usage
func (c *Command) Help(ctx context.Context, req *proto.HelpRequest, rsp *proto.H
elpResponse) error {
    // Usage should include the name of the command
    rsp.Usage = "echo"
    rsp.Description = "This is an example bot command as a micro service which e
chos the message"
    return nil
}

// Exec executes the command
func (c *Command) Exec(ctx context.Context, req *proto.ExecRequest, rsp *proto.E
xecResponse) error {
    rsp.Result = []byte(strings.Join(req.Args, " "))
    // rsp.Error could be set to return an error instead
    // the function error would only be used for service level issues
    return nil
}

func main() {
    service := micro.NewService(
        micro.Name("go.micro.bot.echo"),
    )
```

```
    service.Init()

    proto.RegisterCommandHandler(service.Server(), new(Command))

    if err := service.Run(); err != nil {
        fmt.Println(err)
    }
}
```

新项目模板

新项目模板

micro new [service]

micro new 命令是生成 micro services 样板模板的便捷方法.

使用

通过指定相对于 $GOPATH 的目录路径创建新的服务

```
micro new github.com/micro/foo
```

随后会自动创建一些内容

```
micro new github.com/micro/foo

creating service go.micro.srv.foo
creating /Users/asim/checkouts/src/github.com/micro/foo
creating /Users/asim/checkouts/src/github.com/micro/foo/main.go
creating /Users/asim/checkouts/src/github.com/micro/foo/handler
creating /Users/asim/checkouts/src/github.com/micro/foo/handler/example.go
creating /Users/asim/checkouts/src/github.com/micro/foo/subscriber
creating /Users/asim/checkouts/src/github.com/micro/foo/subscriber/example.go
creating /Users/asim/checkouts/src/github.com/micro/foo/proto/example
creating /Users/asim/checkouts/src/github.com/micro/foo/proto/example/example.pr
oto
creating /Users/asim/checkouts/src/github.com/micro/foo/Dockerfile
creating /Users/asim/checkouts/src/github.com/micro/foo/README.md

download protobuf for micro:

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

compile the proto file example.proto:

protoc -I/Users/asim/checkouts/src
```

```
    --go_out=plugins=micro:/Users/asim/checkouts/src
    /Users/asim/checkouts/src/github.com/micro/foo/proto/example/example.proto
```

选项

也可以指定更多选项，如命名空间，类型，fqdn 和别名

```
micro new --fqdn com.example.srv.foo github.com/micro/foo
```

帮助

```
NAME:
   micro new - Create a new micro service

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

OPTIONS:
   --namespace "go.micro"    Namespace for the service e.g com.example
   --type "srv"              Type of service e.g api, srv, web
   --fqdn                FQDN of service e.g com.example.srv.service (defaults to n
amespace.type.alias)
   --alias               Alias is the short name used as part of combined name if
 specified
```

运行服务

运行服务

> 前言: 使用 micro runtime 运行一个服务

micro runtime 提供了一种管理服务生命周期的方法, 而无需业务流程系统的复杂性. 默认情况下, 它提供了一个简单的本地流程管理器, 用于启动服务, 监视更改并根据需要进行重建.

您可以使用以下命令启动服务

```
micro run service
```

使用

要在本地运行和管理服务, 请执行以下操作.

```
# cd to your service directory e.g examples/greeter/srv
cd examples/greeter/srv

# 运行此服务
micro run service

# 编辑文件
sed -i '1 i\// Package main' main.go
```

在服务启动时, 查看文件更改, 重新生成并重新启动.

待办

后续我们将添加将命令发送到 `micro runtime` 服务以及查询状态, 终止服务等功能.

插件

概述

概述

Micro 为所有工具提供可插拔的体系结构. 这意味着可以基础底层实现.

Go Micro 和 Micro 工具包包括不同类型的插件. 从侧边栏导航以了解有关每个功能的更多.

概述

在高级别插件中, 插件提供了置换基础基础结构和依赖关系的机会. 这意味着微服务可以单向编写, 并以零依赖项在本地运行, 但在使用分布式系统支持其使用时, 在云中同样作为高弹性系统运行.

使用

默认情况下, go-micro 只提供核心上每个接口的几个实现, 但它是完全可插拔的. 已经有几十个插件, 可在 github.com/micro/go-plugins 欢迎贡献! 插件可确保 Go Micro 服务在技术发展后长期保持生存.

如果要集成插件,只需将它们链接到单独的文件中并重新生成.

创建一个 plugins.go 文件并导入所需的插件:

```
package main

import (
        // consul registry
        _ "github.com/micro/go-plugins/registry/consul"
        // rabbitmq transport
        _ "github.com/micro/go-plugins/transport/rabbitmq"
        // kafka broker
        _ "github.com/micro/go-plugins/broker/kafka"
)
```

编写

插件是一个建立在 Go 接口之上的概念. 每个包都维护一个高级接口抽象. 只需实现接口并将其作为服务选项传递给它即可.

服务发现接口称为 Registry. 实现此接口的任何内容都可以用作注册表. 这同样适用于其他包.

```
type Registry interface {
    Register(*Service, ...RegisterOption) error
    Deregister(*Service) error
    GetService(string) ([]*Service, error)
    ListServices() ([]*Service, error)
    Watch() (Watcher, error)
    String() string
}
```

可参阅 go-plugins 了解更多的实现细节.

例子

框架

- Consul Registry- 使用 Consul 的发现服务
- K8s Registry- 使用 Kubernetes 的服务发现
- Kafka Broker- Kafka 消息代理

运行时

- Router - 可配置的 http 路由和代理
- AWS X-Ray - AWS X-Ray 的跟踪集成
- IP Whitelist - 白名单 IP 地址访问

仓库

开源插件可以在此仓库 github.com/micro/go-plugins 找到.

框架

框架

Micro 是一个可插拔的工具包和框架. 内部功能可以与任何 插件 置换.

该工具包具有单独的插件接口. 可参阅 micro/plugin 了解更多信息.

以下是有关 go-micro 插件使用情况的信息.

使用

插件可以通过以下方式添加到 go-micro. 通过这样做它们可以通过命令行参数或环境变量进行设置.

导入 Go 程序中的插件, 然后调用服务. 初始化以分析命令行和环境变量.

```
import (
    "github.com/micro/go-micro/v2"
    _ "github.com/micro/go-plugins/broker/rabbitmq"
    _ "github.com/micro/go-plugins/registry/kubernetes"
    _ "github.com/micro/go-plugins/transport/nats"
)

func main() {
    service := micro.NewService(
        // Set service name
        micro.Name("my.service"),
    )

    // Parse CLI flags
    service.Init()
}
```

标志

将插件指定为标志

```
go run service.go --broker=rabbitmq --registry=kubernetes --transport=nats
```

环境变量

使用环境变量指定插件

```
MICRO_BROKER=rabbitmq
MICRO_REGISTRY=kubernetes \
MICRO_TRANSPORT=nats \
go run service.go
```

选项

创建新服务时，导入并设置为选项

```
import (
    "github.com/micro/go-micro/v2"
    "github.com/micro/go-plugins/registry/kubernetes"
)

func main() {
    registry := kubernetes.NewRegistry() //a default to using env vars for master API

    service := micro.NewService(
        // Set service name
        micro.Name("my.service"),
        // Set service registry
        micro.Registry(registry),
    )
}
```

构建

修改 `main.go` 文件来包含插件是一种反模式的做法. 最佳做法建议将插件包含在单独的文件中，并与其一起重新生成. 这允许构建插件的自动化和问题之间的干净分离.

创建文件 plugins.go

```
package main

import (
    _ "github.com/micro/go-plugins/broker/rabbitmq"
    _ "github.com/micro/go-plugins/registry/kubernetes"
```

```
    _ "github.com/micro/go-plugins/transport/nats"
)
```

使用插件构建.

```
go build -o service main.go plugins.go
```

使用插件运行

```
MICRO_BROKER=rabbitmq
MICRO_REGISTRY=kubernetes
MICRO_TRANSPORT=nats
service
```

重建工具包

如果要集成插件，只需将它们链接到单独的文件中并重新生成

创建 plugins.go 文件

```
import (
        // etcd v3 registry
        _ "github.com/micro/go-plugins/registry/etcdv3"
        // nats transport
        _ "github.com/micro/go-plugins/transport/nats"
        // kafka broker
        _ "github.com/micro/go-plugins/broker/kafka"
)
```

生成二进制

```
// For local use
go build -i -o micro ./main.go ./plugins.go

// For docker image
CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -ldflags '-w' -i -o micr
o ./main.go ./plugins.go
```

插件的标记用法

```
micro --registry=etcdv3 --transport=nats --broker=kafka
```

仓库

go-micro 的插件可以在此仓库 github.com/micro/go-plugins 找到.

运行时

运行时

插件是将外部代码集成到 micro 工具包中的一种方式. 这与 go-micro 插件完全分离. 使用插件可以在此处添加其他标志, 命令和 HTTP 处理程序到工具包.

工作原理

在 micro/plugin 下有一个全局插件管理器, 它由将在整个工具包中使用的插件组成. 插件可以通过调用进行注册. 每个组件 (api, web, proxy, cli, bot) 都有一个单独的插件管理器, 用于注册插件, 该插件应仅作为该组件的一部分添加. 它们可以通过调用 `plugin.Register` , `api.Register` , `web.Register` 等方式使用.

下面是接口

```
// Plugin is the interface for plugins to micro. It differs from go-micro in tha
t it's for
// the micro API, Web, Proxy, CLI. It's a method of building middleware for the
 HTTP side.
type Plugin interface {
    // Global Flags
    Flags() []cli.Flag
    // Sub-commands
    Commands() []cli.Command
    // Handle is the middleware handler for HTTP requests. We pass in
    // the existing handler so it can be wrapped to create a call chain.
    Handler() Handler
    // Init called when command line args are parsed.
    // The initialised cli.Context is passed in.
    Init(*cli.Context) error
    // Name of the plugin
    String() string
}

// Manager is the plugin manager which stores plugins and allows them to be retr
ieved.
// This is used by all the components of micro.
type Manager interface {
        Plugins() map[string]Plugin
        Register(name string, plugin Plugin) error
```

```
}

// Handler is the plugin middleware handler which wraps an existing http.Handler
passed in.
// Its the responsibility of the Handler to call the next http.Handler in the ch
ain.
type Handler func(http.Handler) http.Handler
```

如何使用

下面是一个简单的插件示例，该插件添加一个标志，然后打印值

插件

在顶层目录中创建 plugins.go 文件

```
package main

import (
    "log"
    "github.com/micro/cli"
    "github.com/micro/micro/plugin"
)

func init() {
    plugin.Register(plugin.NewPlugin(
        plugin.WithName("example"),
        plugin.WithFlag(cli.StringFlag{
            Name:   "example_flag",
            Usage:  "This is an example plugin flag",
            EnvVar: "EXAMPLE_FLAG",
            Value: "avalue",
        }),
        plugin.WithInit(func(ctx *cli.Context) error {
            log.Println("Got value for example_flag", ctx.String("example_flag"
))
            return nil
        }),
    ))
}
```

构建代码

只需使用插件构建 micro

```
go build -o micro ./main.go ./plugin.go
```

仓库

工具包的插件可以在此仓库 github.com/micro/go-plugins/micro 中找到.