Table of Contents

go-micro 微服务开发中文手册

相关连接

- 官方文档
- 官方内源

下载

- pdf
- mobi
- epub

目录

概述

Micro文档

概述：Micro是一个简化分布式开发的微服务生态系统。它为开发分布式应用程序提供了基本的构建模块。这篇Micro文档做为采用Micro的参考指南。

介绍

Micro是一个微服务生态系统。目标是简化分布式系统开发。

技术正在迅速发展。现在云计算能够给我们几乎是无限的scale能力，但是采用现有工具来使用scale能力仍然是很困难的。Micro试图去解决这个问题，开发人员首先关注。

Micro的核心是简单易用，任何人都可以轻松开始编写微服务。随着您扩展到数百种服务，Micro将提供管理微服务环境所需的基本工具

开始使用

如果你想开始写微服务，直接去go-micro仓库。

概述

提供的主要软件是Micro，一个微服务工具包。

该工具包由以下组件组成：

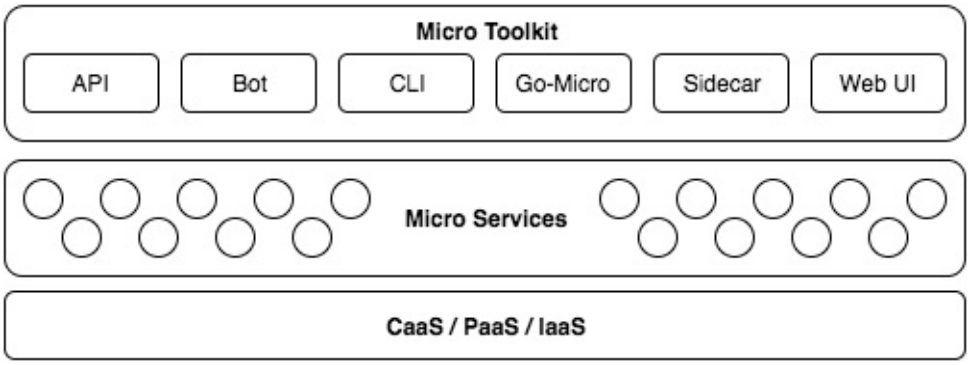
- **Go Micro** - 用于在Go中编写微服务的插件式RPC框架。它提供了用于服务发现，客户端负载平衡，编码，同步和异步通信库。
- **API** - 提供并将HTTP请求路由到相应微服务的API网关。它充当单个入口点，可以用作反向代理或将HTTP请求转换为RPC。
- **Sidecar** - 一种对语言透明的RPC代理，具有go-micro作为HTTP端点的所有功能。虽然Go是构建微服务的伟大语言，但您也可能希望使用其他语言，因此Sidecar提供了一种将其他应用程序集成到Micro世界的方法。

- **Web** - 用于Micro Web应用程序的仪表板和反向代理。我们认为应该基于微服务建立web应用，因此被视为微服务领域的一等公民。它的行为非常像API反向代理，但也包括对web sockets的支持。
- **CLI** - 一个直接的命令行界面来与你的微服务进行交互。它还使您可以利用Sidecar作为代理，您可能不想直接连接到服务注册表。
- **Bot** - Hubot风格的bot，位于您的微服务平台中，可以通过Slack，HipChat，XMPP等进行交互。它通过消息传递提供CLI的功能。可以添加其他命令来自动执行常见的操作任务。

注意：*Go-micro*是一个独立的库，可以独立于其他工具包使用。

运行时

该工具包是可插入式并运行时不感知。在笔记本电脑基于docker，使用kubernetes上运行micro或者AWS等等。



了解更多

浏览此文档以了解更多信息，查看下面的资源或尝试一些示例。

资源

- 阅读博客，深入了解Micro和更广泛的微服务理念。
- 在Golang UK Conf 2016上观看Micro简化微服务的视频。
- 查看演讲台上各种演示的幻灯片。

赞助商

Micro开源开发是由  赞助

微服务

我们来谈谈软件开发的未来。

变化正在进行中。我们正在越来越多地走向每个企业核心技术驱动的世界。在这个时代保持竞争优势变得困难。当企业试图通过低效的平台，流程和结构进行扩展时，组织的执行能力可能会停滞不前。十年前的技术公司已经经历了这些伸缩的痛苦，大多数已经使用相同的方法来克服这些挑战。

是时候把世界上最成功的公司的竞争优势带给其他人了。所以，我们来谈谈微服务，这是一种创造竞争优势的方式。

什么是微服务？

微服务是一种软件架构模式，用于将大型整体应用程序分解为更小的可管理独立服务，这些独立服务通过跨语言的协议进行通信，每个服务都专注于做好一件事情。

来自行业专家的微服务的定义。

1. 松散耦合的面向服务的体系结构
   Adrian Cockcroft
2. 一种将单个应用程序开发为一套小型服务的方法，每个小型服务都运行在自己的进程中，并与轻量级机制进行通信
   Martin Fowler

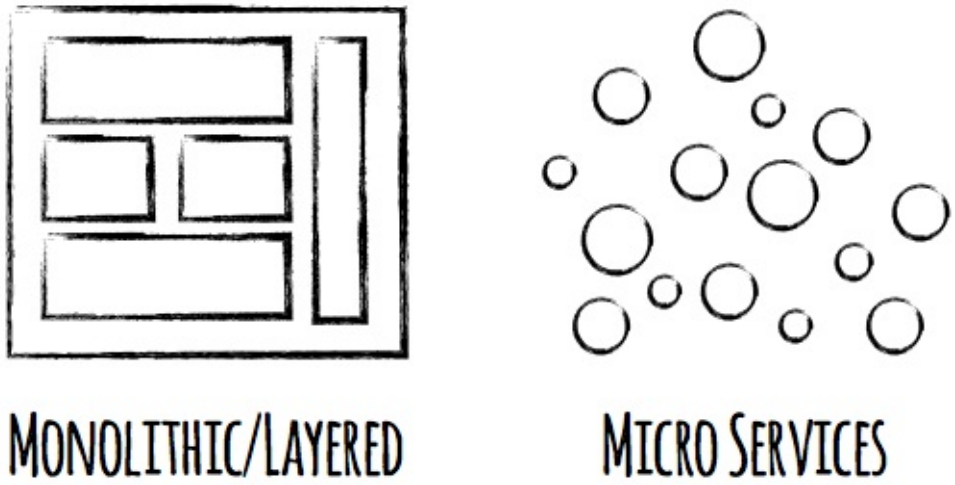
微服务的概念并不新鲜，这是对面向服务的体系结构的重新构想，而是采用了一种更加全面的方式与unix进程和管道对齐的方法。

微服务架构的理念：

- 这些服务是小的 - 作为一个单一的商业目的的细粒度，类似unix哲学的“做一件事，做得好”
- 组织文化应该包含部署和测试的自动化。这减轻了管理和运营的负担。
- 文化和设计原则应该包含失败和错误，类似于抗脆弱的系统。

为什么需要微服务？

随着组织规模技术和人数的增长，管理单一代码库变得更加困难。我们都已经习惯了在一段时间内整个Twitter失败，因为他们试图用一个单一的系统来扩展他们的用户群和产品功能集。微服务使Twitter可以将他们的应用程序分解成更小的服务，这些服务可以由许多不同的团队分别管理。每个团队负责由许多可独立于其他团队部署的微服务组成的业务功能。



我们已经看到了第一手的经验，微服务系统可以缩短开发周期，提高生产力和优越的可扩展系统。

我们来谈谈一些好处：

1. 更容易扩展开发 - 团队围绕不同的业务需求组织管理自己的服务。
2. 更容易理解 - 微服务要小得多，通常为1000 LOC或更少。
3. 更容易频繁地部署新版本的服务 - 可以部署，缩放和独立管理服务。
4. 改进的容错和隔离 - 关注分离可以最大限度地减少一个服务中的问题对另一个服务的影响。
5. 提高执行速度 - 团队通过独立开发，部署和管理微服务来更快地实现业务需求。
6. 可重复使用的服务和快速原型 - 微服务中的unix理念使您能够重用现有服务，并更快地构建全新的功能。

什么是**Micro**？

Micro是一个微服务生态系统，致力于提供产品，服务和解决方案，以实现现代软件驱动型企业的创新。我们计划成为任何与微服务相关的事实资源，并期待公司能够利用这项技术为自己的业务。从早期的原型开始一直到大规模的生产部署。

我们已经看到行业发生根本性转变。摩尔定律是有效的，我们每天都能获得越来越多的计算能力。但是，我们无法完全获取这种新的能力。现有的工具和开发实践在这个新时代并没有scale。没有提供开发人员从单一代码库转向更高效的设计模式的工具。大多数公司不可避免地以单一设计达到收益递减，必须进行大规模的研发再造。 Netfix，Twitter，Gilt和Hailo都是最好的例子。所有最终都建立了自己的微服务平台。

我们的愿景是提供基本的构建模块，使任何人都可以轻松采用微服务。

Micro文档

架构

Micro为微服务提供了基本的构建模块。目标是简化分布式系统开发。因为微服务是一种架构模式，所以Micro通过工具在逻辑上拆分。

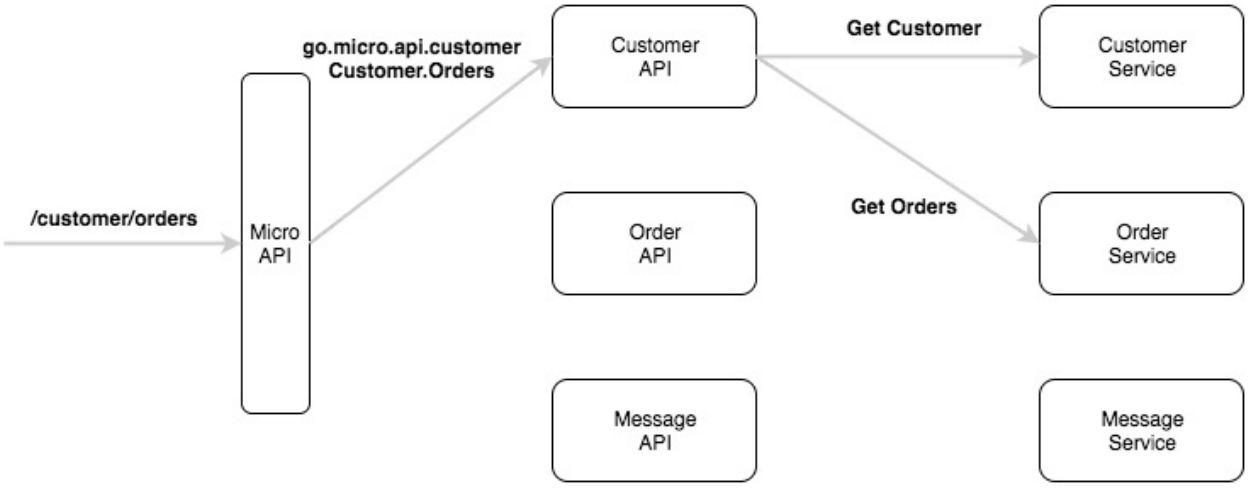
查看体系结构上的博客文章，获取详细的概述。

这部分应该很多详解，解释Micro是如何构建的，以及各种lib/仓库之间是如何相互关联的。

工具包

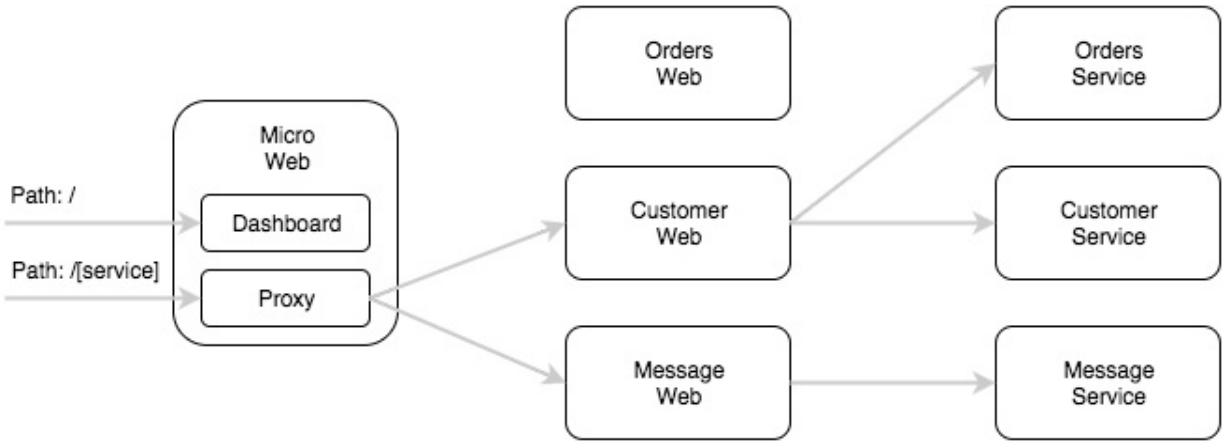
API

启用API作为一个网关或代理，来作为微服务访问的单一入口。它应该在您的基础架构的前端运行。它将HTTP请求转换为RPC并转发给相应的服务。



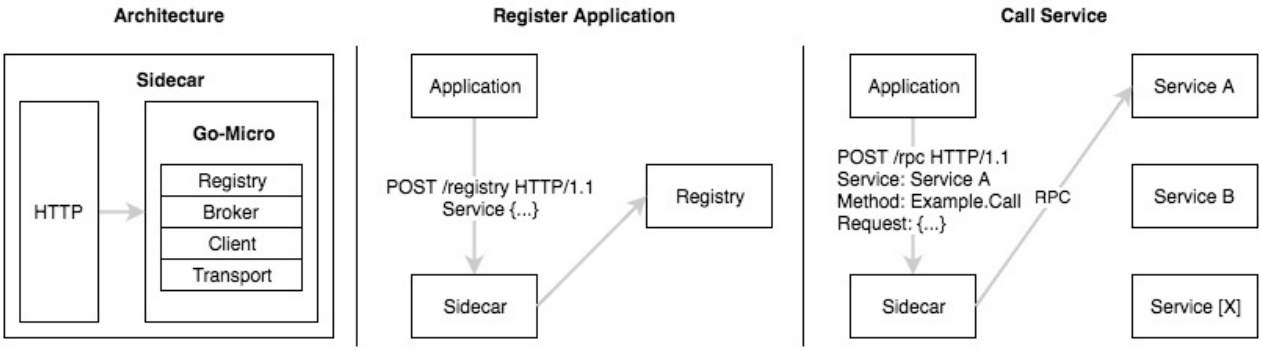
Web

UI是go-micro的web版本，允许通过UI交互访问环境。在未来，它也将是一种聚合Micro Web服务的方式。它包含一种Web应用程序的代理方式。将 `/[name]` 通过注册表路由到相应的服务。Web UI将前缀“go.micro.web。”（可以配置）添加到名称中，在注册表中查找它，然后将进行反向代理。



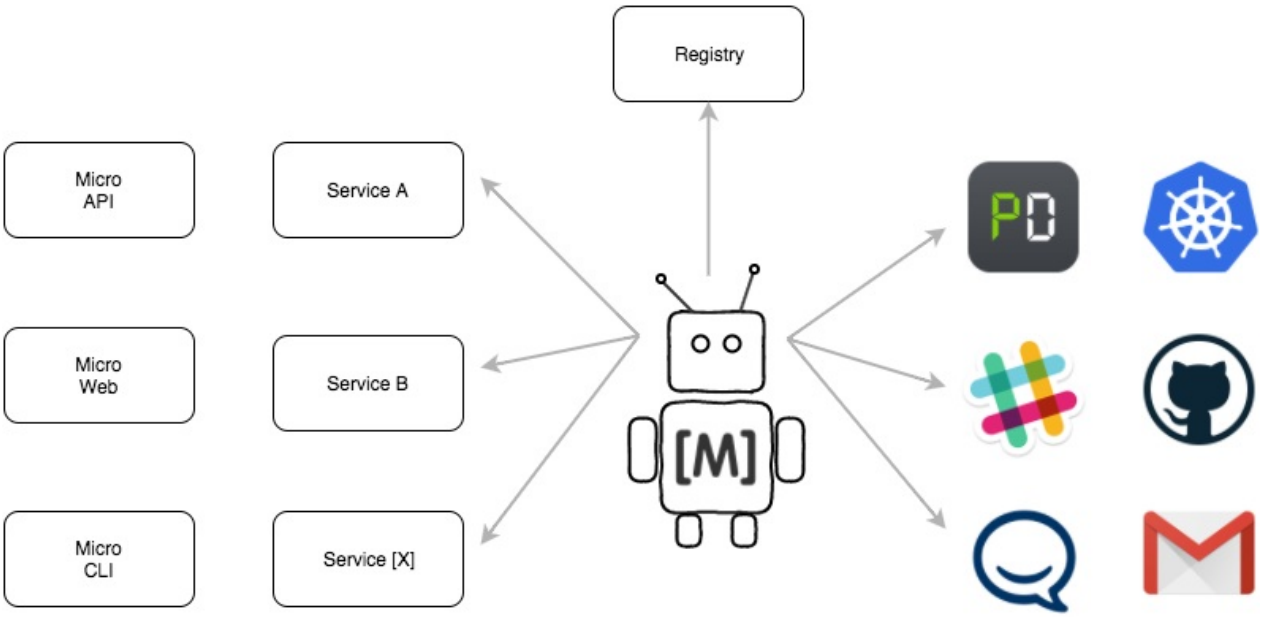
Sidecar

该Sidecar是go-micro的HTTP接口版本。这是将非Go应用程序集成到Micro环境中的一种方式。



Bot

Bot是一个Hubot风格的工具，位于您的微服务平台中，可以通过Slack，HipChat，XMPP等进行交互。它通过消息传递提供CLI的功能。可以添加其他命令来自动执行常用操作任务。

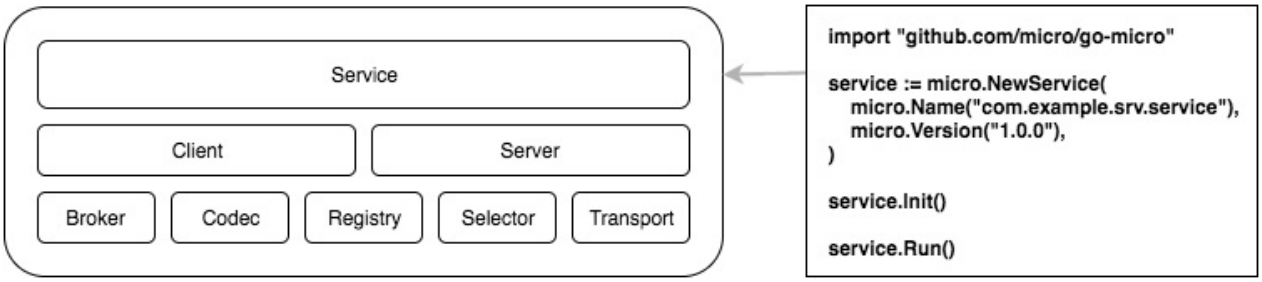


CLI

Micro CLI是go-micro的命令行版本，它提供了一种观察和与运行环境交互的方式。

Go Micro

Go-micro是微服务的独立RPC框架。它是该工具包的核心，并受到上述所有组件的影响。在这里，我们将看看go-micro的每个特征。



Registry

注册表提供可插入的服务发现库，来查找正在运行的服务。当前的实现是consul，etcd，内存和kubernetes。如果您的喜欢不一样，该接口很容易实现。

Selector

选择器通过选择提供负载均衡机制。当客户端向服务器发出请求时，它将首先查询服务的注册表。这通常会返回一个表示服务的正在运行的节点列表。选择器将选择这些节点中的一个用于查询。多次调用选择器将允许使用平衡算法。目前的方法是循环法，随机哈希和黑名单。

Broker

Broker是发布和订阅的可插入接口。微服务是一个事件驱动的架构，发布和订阅事件应该是一流的公民。目前的实现包括nats，rabbitmq和http（用于开发）。

Transport

传输是通过点对点传输消息的可插拔接口。目前的实现是http，rabbitmq和nats。通过提供这种抽象，运输可以无缝地换出。

Client

客户端提供了一种制作RPC查询的方法。它结合了注册表，选择器，代理和传输。它还提供重试，超时，使用上下文等。

Server

服务器是构建正在运行的微服务的接口。它提供了一种提供RPC请求的方法。

Plugins

提供go-micro的micro/go-plugins插件。

常问问题

常见问题解答应该为最常见的问题提供快速解答。

什么是Micro？

Micro是一个专注于简化分布式系统开发的微服务生态系统。

- Micro是一个框架
- Micro是一个工具包
- Micro是一个社区
- Micro是一个生态系统

开源

Micro由开放源码库和工具组成，以帮助微服务开发。

- **go-micro** - 用于编写微服务的可插入Go RPC框架; 服务发现，客户端/服务器rpc，pub/sub等。
- **go-plugins** - go-micro的插件，包括etcd，kubernetes，nats，rabbitmq，grpc等
- **micro** - 一个包含传统入口点的微服务工具包; API网关，CLI，Slack Bot，Sidecar和Web UI。

其他各种库和服务可以在github.com/micro找到。

社区

有一个有千名会员的松散社区。

在slack.micro.mu邀请你自己。

生态系统

Micro跨越单一组织。开源工具和服务正在由社区自己提供。

在micro.mu/explore/上探索生态系统。

我从哪里开始？

从go-micro开始。自述文件提供了一个微服务示例。

阅读入门指南或查看示例，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？

在用户页面查看使用Micro的公司列表，（但注意它可能已过时）。

还有很多人也在使用它，但尚未公开列出。如果您使用Micro，请随时添加您的公司。

我如何使用Micro？

这很简单。

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

检查完整的greeter示例。

我可以替代Consul吗？

可以! 服务发现注册表与其他所有软件包一样，是完全可插入的。由于其特点和简单性，Consul被用作默认值。

ETCD

举个例子。如果您想使用etcd，请导入插件并在二进制文件中设置命令行标志。

```
import (
        _ "github.com/micro/go-plugins/registry/etcd"
)
```

```
service --registry=etcd --registry_address=127.0.0.1:2379
```

零依赖

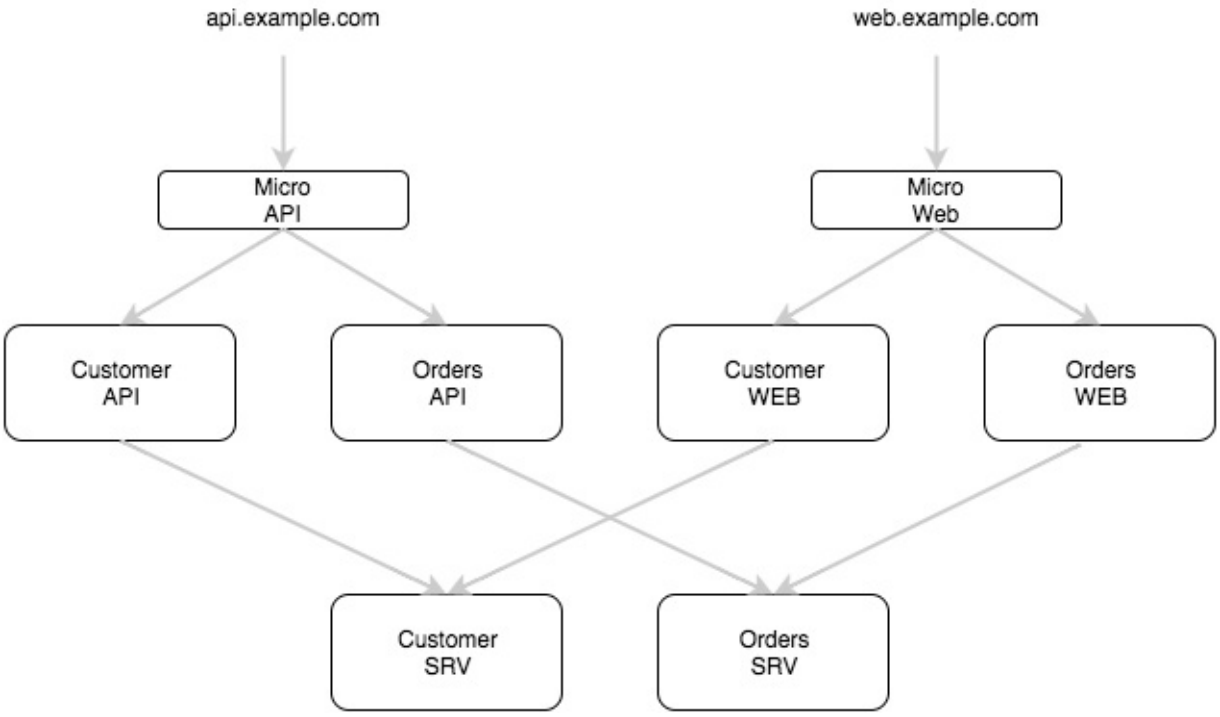
有一个内置的零依赖的Multicast DNS服务注册表配置。在启动时将 `--registry=mdns` 或 `MICRO_REGISTRY=mdns` 传递给您的应用程序即可。

我可以在哪里运行Micro？

Micro是运行时不感知的。你可以在任何你喜欢的地方运行它。裸机上AWS，谷歌云。在你最喜欢的容器编排系统，如Mesos或Kubernetes。

事实上，在Kubernetes上有Micro的演示配置。查看https://github.com/micro/kubernetes。

API，Web和SRV服务有什么区别？



作为Micro工具包的一部分，我们尝试通过分离API，Web仪表盘和后端服务（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服务由Micro Api提供，默认命名空间为go.micro.api。micro API符合API网关模式。

点击此处了解详情

Web服务

Web服务由Micro Web提供，默认名称空间为go.micro.web。我们相信web应用程序是微服务世界中的一等公民，因此可以将web仪表板作为微服务来构建。Micro网络是一个反向代理，它会根据服务解析的路径将HTTP请求转发到相应的Web应用程序。

点击此处了解详情

SRV服务

SRV服务基本上是标准的RPC服务，通常这是你写的服务。我们通常称它们为RPC或后端服务，因为它们主要应该是后端架构的一部分，并且永远不会面向公众。默认情况下，我们使用命名空间go.micro.srv，但是您应该使用您的域com.example.srv。

它性能如何？

性能不是Micro的当前焦点。尽管代码编写为最佳并避免了开销，但基准测试并没有花费太多时间。与net/http或其他web框架进行比较是没有意义的。Micro为包括服务发现，负载平衡，消息编码等的微服务提供了更高级别的要求。为了比较，您需要将所有这些功能添加进来。

如果你仍然关心性能。提取最大值的最简单方法是简单地通过运行以下标志：

```
--selector=cache # enables in memory caching of discovered nodes
--client_pool_size=10 # enables the client side connection pool
```

Micro是否支持gRPC？

是的。在micro/go-plugins中有传输，客户端和服务器的插件。

如果你想快速入门，只需使用micro/go-grpc。

Micro与Go-Kit

这个问题出现了很多。micro和go-kit有什么区别？

Go-kit将自己描述为微服务的标准库。像Go一样，go-kit为您提供可用于构建应用程序的单独包。如果您想完全控制您定义服务的方式，Go-kit非常棒。

Go-micro是微服务的可插入RPC框架。这是一个自发的框架，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非常适合您快速启动和运行，同时拥有可插拔的基础架构而无需更改代码。

Micro是一个微服务工具包。这就像微型服务的瑞士军刀一样，微型服务以go-micro为基础，提供诸如http api gateway，web ui，cli，slack bot等传统入口点。Micro使用工具来指导逻辑上分离您的架构中的关注点，从而推动您为公共API创建一个微服务的API层，并为Web UI分别创建一个微服务的WEB层。

在想要完全控制的地方使用go-kit。你想要一个自用的框架使用go-micro。

我在哪里可以了解更多？

- 加入松散社区 - slack.micro.mu
- 阅读博客 - micro.mu/blog
- 如果你想谈谈 - contact@micro.mu

特性

micro生态系统正在迅速发展，但仍有许多工作要做。

特性页面保持跟踪最重要或最值得注意的功能。

- 编写服务的框架
- 工具包来访问服务
- 示例显示使用情况
- Explorer来搜索OSS贡献

Micro

Micro是查询和访问微服务的工具包

- http到rpc的API网关
- Web微服务的Web代理
- Sidecar作为服务网格
- 用于命令行访问的CLI
- Bot通过Slack进行查询

Go Micro

Go-micro可以帮助您编写微服务

- 分布式系统抽象
- 服务发现，rpc，pub/sub，消息编码
- 容错超时，重试和负载均衡
- 通过封装扩展功能
- 可替换后端的插件接口

Go Plugins

- go-micro/micro插件
- 包括最流行的后端技术

- grpc，kubernetes，etcd，kafka等
- 生产测试

部署

- Kubernetes
- Docker Compose

路标

路标显示更大的里程碑，或仅显示发生了什么事情

项目

我们正在使用GitHub项目来管理路标 https://github.com/micro/micro/projects/1

当前

- [ ] HTTP2传输
- [ ] Sidecar作为服务网格

长期：

- [x] 调用策略{分片，传播，分散收集}
- [x] 加密支持
- [x] 事件处理程序
- [ ] Sidecar HTTP/gRPC/Thrift
- [ ] 数据库
- [ ] HTTP2传输
- [ ] GraphQL Handler

资源

这里有一些帮助开始

文章

- Micro博客
- 微服务之旅
- 微服务 作者：Martin Fowler
- 微服务引起共鸣的四个原因 作者：Neal Ford

幻灯片

- 微服务世界之旅 作者： Matt Heath（幻灯片）
- 微服务：分解应用程序以实现可部署和可伸缩性 作者：Chris Richardson（幻灯片）

影片

- Golang UK 2016上的简化微服务（视频）（幻灯片）

用户

如果您使用Micro，请随时添加您的公司

公司

- Sixt - 跨国汽车租赁公司在100多个国家以及Micro赞助商。
- Lostmy.name - 为儿童个性化的故事。国际畅销书。
- Pie Mapping - 为物流和运输公司管理他们的车辆和为他们的运营提供动力的按需解决方案。
- Dark Cubed - 企业级，完全匿名，不可归属的近实时网络安全信息共享平台。
- Kyperion - Kyperion是一个训练平台（跑步，自行车，游泳和铁人三项）。我们加入教练和业余运动员，以最大限度地提高时间和成果
- Kazoup - 征服你的数据 - 分析.搜索.存档。

指南

安装指南

依赖

我们需要服务发现，所以让我们启动Consul（默认），或者通过go-plugins替换。

Consul

```
brew install consul
consul agent -dev
```

或者

```
docker run consul
```

Multicast DNS

我们可以使用Multicast DNS进行零依赖的服务发现

将 `--registry=mdns` 传递给任何命令，例如 `micro --registry = mdns list services`

Go Micro

Go Micro是Go开发微服务的RPC框架

安装

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

Protobuf

如果您使用代码生成，您还需要使用protoc-gen-go

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

访问github.com/micro/go-micro了解更多。

工具包

Micro工具包提供了访问微服务的各种方法

安装

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

Docker

可用预制docker images

```
docker pull microhq/micro
```

尝试CLI

运行greeter服务

```
go get github.com/micro/examples/greeter/srv && srv
```

服务清单

```
$ micro list services
consul
go.micro.srv.greeter
```

获取服务

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

version 1.0.0

Id    Address    Port    Metadata
go.micro.srv.greeter-34c55534-368b-11e6-b732-68a86d0d36b6    192
.168.1.66    62525    server=rpc,registry=consul,transport=http,
broker=http

Endpoint: Say.Hello
Metadata: stream=false

Request: {
    name string
}

Response: {
    msg string
}
```

查询服务

```
$ micro query go.micro.srv.greeter Say.Hello '{"name": "John"}'
{
    "msg": "Hello John"
}
```

GRPC网关

本指南有助于将grpc网关与go-micro服务结合使用。

grpc-gateway是protoc的插件。它读取gRPC服务定义，并生成将RESTful JSON API转换为gRPC的反向代理服务器。

我们使用go-grpc编写后端服务。Go-GRPC是围绕go-micro和用于客户端和服务器的grpc插件的简单包装。当调用grpc.NewService时，它返回一个micro.Service。

code

在examples/grpc找到示例代码。

前提

这些是一些先决条件

安装protobuf

```
mkdir tmp
cd tmp
git clone https://github.com/google/protobuf
cd protobuf
./autogen.sh
./configure
make
make check
sudo make install
```

安装插件

```
go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-grpc
-gateway
go get -u github.com/micro/protobuf/protoc-gen-go
```

Greeter服务

在这个例子中，我们使用go-grpc创建了一个Greeter的微服务。 这项服务非常简单。

原型如下：

```
syntax = "proto3";

package go.micro.srv.greeter;

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

message Request {
    string name = 1;
}

message Response {
    string msg = 1;
}
```

服务如下：

```
package main

import (
    "log"
    "time"

    hello "github.com/micro/examples/greeter/srv/proto/hello"
    "github.com/micro/go-grpc"
```

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

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

type Say struct{}

func (s *Say) Hello(ctx context.Context, req *hello.Request, rsp
 *hello.Response) error {
    log.Print("Received Say.Hello request")
    rsp.Msg = "Hello " + req.Name
    return nil
}

func main() {
    service := grpc.NewService(
        micro.Name("go.micro.srv.greeter"),
        micro.RegisterTTL(time.Second*30),
        micro.RegisterInterval(time.Second*10),
    )

    // optionally setup command line usage
    service.Init()

    // Register Handlers
    hello.RegisterSayHandler(service.Server(), new(Say))

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

GRPC网关

grpc网关使用与服务相同的协议，并增加一个http选项

```
syntax = "proto3";

package greeter;

import "google/api/annotations.proto";

service Say {
    rpc Hello(Request) returns (Response) {
        option (google.api.http) = {
            post: "/greeter/hello"
            body: "*"
        };
    }
}

message Request {
    string name = 1;
}

message Response {
    string msg = 1;
}
```

proto使用以下命令生成grpc stub和反向代理

```
protoc -I/usr/local/include -I. \
  -I$GOPATH/src \
  -I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/third_par
ty/googleapis \
  --go_out=plugins=grpc:. \
  path/to/your_service.proto
```

```
protoc -I/usr/local/include -I. \
  -I$GOPATH/src \
  -I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/third_par
ty/googleapis \
  --grpc-gateway_out=logtostderr=true:. \
  path/to/your_service.proto
```

我们使用下面的代码创建了greeter服务的示例api。将写入类似的代码来注册其他端点。请注意，网关需要greeter服务的端口地址。

```
package main

import (
    "flag"
    "net/http"

    "github.com/golang/glog"
    "github.com/grpc-ecosystem/grpc-gateway/runtime"
    "golang.org/x/net/context"
    "google.golang.org/grpc"

    hello "github.com/micro/examples/grpc/gateway/proto/hello"
)

var (
    // the go.micro.srv.greeter address
    endpoint = flag.String("endpoint", "localhost:9090", "go.mic
ro.srv.greeter address")
)

func run() error {
    ctx := context.Background()
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()

    mux := runtime.NewServeMux()
    opts := []grpc.DialOption{grpc.WithInsecure()}

    err := hello.RegisterSayHandlerFromEndpoint(ctx, mux, *endpo
```

```
int, opts)
    if err != nil {
        return err
    }

    return http.ListenAndServe(":8080", mux)
}

func main() {
    flag.Parse()

    defer glog.Flush()

    if err := run(); err != nil {
        glog.Fatal(err)
    }
}
```

运行示例

运行greeter服务。指定mdns，因为我们不需要发现。

```
go run examples/grpc/greeter/srv/main.go --registry=mdns --serve
r_address=localhost:9090
```

运行网关。它将默认为端点localhost:9090的greeter服务。

```
go run examples/grpc/gateway/main.go
```

使用curl在（localhost:8080）网关上发起请求。

```
curl -d '{"name": "john"}' http://localhost:8080/greeter/hello
```

限制

grpc网关的例子需要提供服务地址，而我们自己的micro api使用服务发现，动态路由和负载均衡。这使得grpc网关的集成性稍差一些。

访问github.com/micro/micro了解更多信息

写一个Go服务

这是go-micro入门指南。

如果您首先喜欢更高级别的工具包概览，请查看介绍性博客文章 https://micro.mu/blog/2016/03/20/micro.html。

写一个服务

顶级服务接口是构建服务的主要组件。它将Go Micro的所有底层软件包整合到一个简单的接口中。

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

1.初始化

像使用 `micro.NewService` 一样创建一个服务。

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

service := micro.NewService()
```

选项可以在创建过程中传入。

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

所有可用的选项可以在这里找到。

Go Micro还提供了使用 `micro.Flags` 设置命令行标志的方法。

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

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

解析标志使用 `service.Init` 。另外访问标志使用 `micro.Action` 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提供了预定义的标志，如果调用 `service.Init` ，它将被设置和解析。看到这里的所有标志

2.定义API

我们使用protobuf文件来定义服务API接口。这是一种非常方便的方式来严格定义API并为服务器和客户端提供具体的类型。

这是一个示例定义。

greeter.proto

```
syntax = "proto3";

service Greeter {
    rpc Hello(HelloRequest) returns (HelloResponse) {}
}

message HelloRequest {
    string name = 1;
}

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

在这里，我们定义了一个名为Greeter的服务处理程序，其中的方法Hello使用参数 `HelloRequest` 类型并返回 `HelloResponse` 。

3.生成API接口

我们使用protoc和protoc-gen-go为这个定义生成具体的go实现。

Go-micro使用代码生成来提供客户端桩方法来减少代码编写，就像gRPC一样。这是通过一个protobuf插件完成的，它需要一个golang/protobuf分支，可以在这里找到github.com/micro/protobuf。

```
go get github.com/micro/protobuf/{proto,protoc-gen-go}
protoc --go_out=plugins=micro:. greeter.proto
```

生成的类型现在可以在请求时在服务器或客户端的处理程序中导入和使用。

这是生成代码的一部分。

```
type HelloRequest struct {
    Name string `protobuf:"bytes,1,opt,name=name" json:"name,omi
tempty"`
}

type HelloResponse 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 *HelloRequest, opts ...client.
CallOption) (*HelloResponse, error)
}

type greeterClient struct {
    c           client.Client
    serviceName string
}

func NewGreeterClient(serviceName string, c client.Client) Greet
erClient {
    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 *HelloRequ
est, opts ...client.CallOption) (*HelloResponse, error) {
```

```
    req := c.c.NewRequest(c.serviceName, "Greeter.Hello", in)
    out := new(HelloResponse)
    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, *HelloRequest, *HelloResponse) 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来处理请求。处理程序是具有符合签名

```
func(ctx context.Context, req interface{}, rsp interface{}) erro
r
```

错误的公共方法的公共类型。正如你在上面看到的，Greeter接口的处理器签名看起来像这样。

```
type GreeterHandler interface {
        Hello(context.Context, *HelloRequest, *HelloResponse) er
ror
}
```

以下是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 *proto.HelloReq
uest, rsp *proto.HelloResponse) error {
    rsp.Greeting = "Hello " + req.Name
    return nil
}
```

该处理程序在您的服务中注册很像一个http.Handler。

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

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

您也可以创建一个双向流媒体处理程序。

5. 运行服务

该服务可以通过调用 `server.Run` 来运行。这导致服务绑定到配置中的地址（默认为第一个RFC1918接口和随机端口）并侦听请求。

这将另外注册服务注册表启动和注销时发出一个kill信号。

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

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

type Greeter struct{}

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

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

        service.Init()

        proto.RegisterGreeterHandler(service.Server(), new(Greet
er))

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

注意：服务发现机制将需要运行，以便服务可以注册被客户和其他服务发现。快速入门就在这里。

写一个客户端

客户端软件包用于查询服务。在创建服务时，将包含一个与服务器使用的初始化包相匹配的客户端。

查询以上服务如下所示。

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

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

fmt.Println(rsp.Greeter)
```

`proto.NewGreeterClient` 获取服务名称和用于发出请求的客户端。

完整的例子可以在go-micro/examples/service中找到。

编写一个Go函数

这是开始使用go-micro函数的指南。函数是一次执行服务。

如果您首先喜欢更高级别的工具包概述，请查看介绍的博客文章
https://micro.mu/blog/2016/03/20/micro.html

编写一个函数

顶层函数接口是go-micro中函数编程模型的主要组件。它封装了Service接口，同时提供一次执行。

```
// Function is a one time executing Service
type Function interface {
    // Inherits Service interface
    Service
    // Done signals to complete execution
    Done() error
    // Handle registers an RPC handler
    Handle(v interface{}) error
    // Subscribe registers a subscriber
    Subscribe(topic string, v interface{}) error
}
```

1.初始化

一个函数就像使用 `micro.NewFunction` 一样创建。

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

function := micro.NewFunction()
```

选项可以在创建过程中传入。

```
function := micro.NewFunction(
        micro.Name("greeter"),
        micro.Version("latest"),
)
```

所有可用的选项可以在这里找到。

Go Micro还提供了使用 `micro.Flags` 设置命令行标志的方法。

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

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

解析标志使用 `function.Init` 。另外访问标志使用 `micro.Action` 选项。

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

Go Micro提供了预定义的标志，如果调用了 `function.Init` ，它将被设置和解析。看到这里的所有标志。

2.定义API

我们使用protobuf文件来定义API接口。这是严格定义API并提供服务端和客户端的具体类型的一种非常方便的方式。

这是一个示例定义。

greeter.proto

```
syntax = "proto3";

service Greeter {
    rpc Hello(HelloRequest) returns (HelloResponse) {}
}

message HelloRequest {
    string name = 1;
}

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

在这里我们定义了一个名为Greeter的函数处理程序，其中的方法Hello使用参数 `HelloRequest` 类型并返回 `HelloResponse` 。

3.生成API接口

我们使用protoc和protoc-gen-go为这个定义生成具体的实现。

Go-micro使用代码生成来提供客户端桩方法来减少代码编写，就像gRPC一样。这是通过一个protobuf插件完成的，它需要一个golang/protobuf分支，可以在这里找到github.com/micro/protobuf。

```
go get github.com/micro/protobuf/{proto,protoc-gen-go}
protoc --go_out=plugins=micro:. greeter.proto
```

生成的类型现在可以在请求时在服务端或客户端的处理程序中导入和使用。

这是生成的代码的一部分。

```
type HelloRequest struct {
    Name string `protobuf:"bytes,1,opt,name=name" json:"name,omi
tempty"`
}

type HelloResponse 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 *HelloRequest, opts ...client.
CallOption) (*HelloResponse, error)
}

type greeterClient struct {
    c           client.Client
    serviceName string
}

func NewGreeterClient(serviceName string, c client.Client) Greet
erClient {
    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 *HelloRequ
est, opts ...client.CallOption) (*HelloResponse, error) {
```

```
    req := c.c.NewRequest(c.serviceName, "Greeter.Hello", in)
    out := new(HelloResponse)
    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, *HelloRequest, *HelloResponse) error
}

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

4.实现处理程序

服务器要求注册处理程序来处理请求。处理程序是一种公共方法，符合签名

```
func(ctx context.Context, req interface{}, rsp interface{}) erro
r
```

正如你上面看到的，Greeter接口的处理器签名看起来像这样。

```
type GreeterHandler interface {
        Hello(context.Context, *HelloRequest, *HelloResponse) er
ror
}
```

这是一个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 *proto.HelloReq
uest, rsp *proto.HelloResponse) error {
    rsp.Greeting = "Hello " + req.Name
    return nil
}
```

处理程序的注册很像一个 `http.Handler` 。

```
function := micro.NewFunction(
    micro.Name("greeter"),
)

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

或者，函数接口提供了一个更简单的注册模式。

```
function := micro.NewFunction(
        micro.Name("greeter"),
)

function.Handle(new(Greeter))
```

您也可以使用Subscribe方法注册一个异步订阅方法。

5.运行功能

该函数可以通过调用function.Run来运行。这将导致它绑定到配置中的地址（默认是第一个RFC1918接口和随机端口）并监听请求。

这将另外注册功能与注册表启动和注销时发出一个kill信号。

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

一旦发出请求，函数将退出。您可以使用micro run管理功能的生命周期。一个完整的例子可以在examples/function中找到。

6.完整的功能

greeter.go

```
package main

import (
        "log"

        "github.com/micro/go-micro"
        proto "github.com/micro/examples/function/proto"

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

type Greeter struct{}

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

func main() {
        function := micro.NewFunction(
                micro.Name("greeter"),
                micro.Version("latest"),
        )

        function.Init()

    function.Handle(new(Greeter))

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

注意：服务发现机制将需要运行，以便函数可以注册以供希望查询的人发现。快速入门就在这里。

写一个客户端

客户端软件包用于查询功能和服务。当您创建一个函数时，将包含一个与服务器使用的初始化包相匹配的客户端。

查询上述功能就像下面这样简单。

```
// create the greeter client using the service name and client
greeter := proto.NewGreeterClient("greeter", function.Client())

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

fmt.Println(rsp.Greeter)
```

`proto.NewGreeterClient` 接受函数名称和用于发出请求的客户端。

完整的例子可以在go-micro/examples/function中找到。

容错

在分布式系统中，世界各地的故障一直在发生。Micro试图用一些容错的最佳实践来解决这个问题。本文档介绍了一些可以配置的方法。

心跳

心跳是服务发现中刷新注册的机制。

基本原理

服务在启动时注册服务发现，并在关闭时取消注册。有时这些服务可能会意外死亡，被强行杀死或面临暂时的网络问题。在这些情况下，遗留的节点将存在服务发现中。如果服务被自动删除，这将是理想的。

解决办法

由于这个确切原因，Micro支持注册的TTL选项和间隔。TTL指定在发现之后注册的信息存在多长时间，然后过期并被删除。时间间隔是服务应该重新注册的时间，以保留其在服务发现中的注册信息。

这些选项可以用micro工具包中提供的标志。

用法

对于micro工具包，只需使用内置标志来设置ttl和间隔。

```
micro --register_ttl=30 --register_interval=15 api
```

以上这个例子，我们设置了一个30秒的ttl，重新注册间隔为15秒。

对于go-micro，当声明一个微服务时，你可以通过time.duration传递选项。

```
service := micro.NewService(
        micro.Name("com.example.srv.foo"),
        micro.RegisterTTL(time.Second*30),
        micro.RegisterInterval(time.Second*15),
)
```

负载均衡

负载均衡是传输请求负载或维持高可用性的一种方式

基本原理

任何单个流程应用程序的可用性和扩展都存在限制。如果应用程序因任何原因死亡，您将无法再处理请求。如果有足够的请求负载发送到应用程序，它可能会开始缓慢响应或根本不响应。通过发送请求至多个应用程序副本可以解决这些问题。

解决办法

Micro通过选择器接口进行客户端负载平衡，以在任意数量的服务节点上传播请求。当服务启动时，它将服务发现注册为具有唯一地址和ID的服务节点。在发出请求时，Micro客户端使用选择器来决定向哪个节点发出请求。选择器使用服务注册表查找服务的节点，然后使用负载均衡策略（例如：随机哈希或循环法）选择要发送请求的节点。

用法

客户端负载平衡内置于go-micro客户端。这是自动完成的。

重试

重试是一种在不成功时重试请求的方法

基本原理

由于许多原因，请求可能会失败; 网络错误，请求加载，应用程序死亡。在这些情况下，如果我们可以针对不同的应用程序副本重试请求以获得成功的响应，那将是理想的。

解决办法

微客户端包括一个重试请求的机制。选择器（如上所述）返回一个Next函数，该函数在执行时使用负载平衡策略从列表中返回一个节点。Next函数可以执行多次，根据负载均衡策略返回一个新节点。如果设置了重试次数，如果请求失败，则会执行Next函数，并且将针对新节点重试请求。

用法

重试可以设置为客户端的标志或选项。它默认为1，意味着1次尝试请求。

通过标志进行更改。

```
micro --client_retries=3
```

通过选项进行设置。

```
client.Init(
    client.Retries(3),
)
```

缓存发现

发现缓存是服务发现信息的客户端缓存

基本原理

服务发现是微服务的核心依赖，但如果架构不正确，也可能成为单点故障。每个发现系统都有自己的扩展性和高可用性。在事件发生下降的情况下，由于服务不知道如何解析名称到地址，系统的其余部分将无法使用。如果为系统中的每个请求执行查找，发现也可能成为瓶颈。

解决办法

客户端缓存是一种消除服务发现瓶颈和单点故障的方法。Micro包含一个选择器（客户端负载平衡器），它在服务发现信息的内存缓存中维护与之相关的信息。如果发生缓存未命中，选择器将使用服务注册表进行查找并缓存数据。缓存也会周期性地被TTL掉，以确保陈旧的数据不会存在。

用法

高速缓存选择器可以通过标志或在创建新服务时进行设置。

作为工具包的标志执行以下操作

```
micro --selector=cache api
```

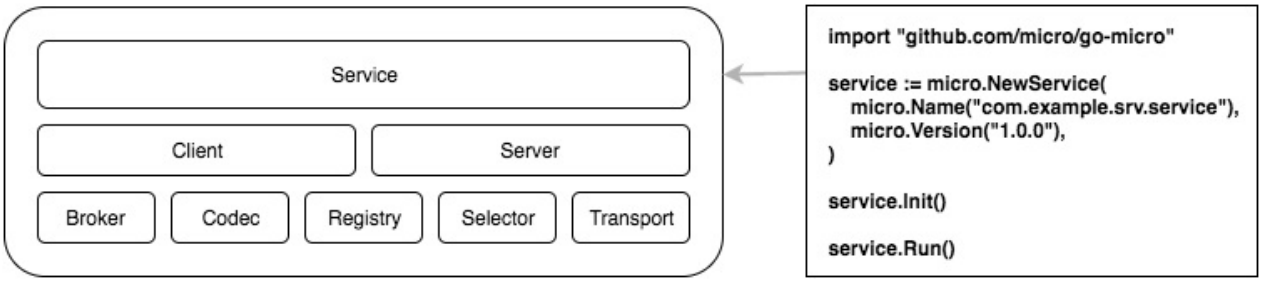
如果调用Init方法，Go-micro服务也可以使用相同的标志。或者，选择器可以用代码设置。

```
import (
    "github.com/micro/go-micro/client"
    "github.com/micro/go-micro/selector/cache"
)

service := micro.NewService(
    micro.Name("com.example.srv.foo"),
)

service.Client().Init(cache.NewSelector())
```

组件

Go Micro

Go Micro是一个插件式的RPC框架。它用于分布式系统开发。



特性

Go Micro抽象出分布式系统的细节。以下是主要功能。

- 服务发现 - 通过服务发现自动注册和名称解析
- 负载平衡 - 基于发现构建的服务的智能客户端负载平衡
- 同步通信 - 基于RPC的通信，支持双向流
- 异步通信 - 为事件驱动架构内置的Pub/Sub接口
- 消息编码 - 基于带有protobuf和json的内容类型的动态编码
- 服务接口 - 所有功能都打包在一个简单的高级界面中，用于开发微服务

Go Micro支持服务和功能编程模型。请继续阅读以了解更多信息。

插件

Go-micro使用Go界面进行抽象。由于这个原因，底层实现可以被换出。

我们提供了开箱即用的默认设置。

- Consul或mDNS服务发现
- 随机散列客户端负载平衡
- JSON-RPC 1.0和PROTO-RPC消息编码
- HTTP通信机制

示例

在示例/函数以及示例/服务可以找到示例服务。

示例目录包含使用诸如中间件/封装器，选择器过滤器，pub/sub，grpc，插件等的示例。对于完整的greeter示例，请查看示例/greeter程序。其他示例可以在整个GitHub存储库中找到。

观看Golang UK Conf 2016视频以获取高级概述。

软件包

Go micro由多个软件包组成。

- 传输同步消息
- 代理异步消息
- 用于消息编码的编解码器
- 服务发现注册表
- 选择器进行负载平衡
- 客户端提出请求
- 服务器来处理请求

更多细节如下

注册机制

注册机制提供了一个服务发现机制来将名称解析为地址。它可以由consul，etcd，zookeeper，dns，gossip等提供支持。服务应该在启动时使用注册机制进行注册，并在关闭时取消注册。服务可以选择提供一个到期的TTL并在一段时间内重新注册以确保活跃，并且如果服务死亡，服务将被清除。

选择器

选择器是建立在注册表上的负载平衡抽象。它允许使用过滤器功能对“服务”进行“过滤”，并使用诸如random，roundrobin，leastconn等算法选择“选择”服务。客户端在请求时利用选择器。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

传输是用于服务之间的同步请求/响应通信的接口。它类似于golang网络包，但提供更高层次的抽象，允许我们切换通信机制，例如http，rabbitmq，websockets，NATS。该传输也支持双向流式传输。这对客户端推送到服务器提供强大的能力。

代理

代理为消息代理提供异步发布/订阅通信的接口。这是事件驱动架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指向HTTP系统，来最小化基本所需的依赖的数量。但是，在go-plugins中有许多消息代理实现可用，例如RabbitMQ，NATS，NSQ，Google Cloud Pub Sub。

编解码器

编解码器用于编码和解码消息，然后再通过线路传输消息。这可能是json，protobuf，bson，msgpack等。这与其他大多数编解码器不同之处在于我们实际上也支持RPC格式。所以我们有JSON-RPC，PROTO-RPC，BSON-RPC等。它将客户端/服务器的编码分离出来，并提供了一个强大的方法来集成其他系统，如gRPC，Vanadium等。

服务器

服务器是编写服务的构建模块。在这里，您可以命名您的服务，注册请求处理程序，添加middeware等。该服务构建在上述软件包上，为服务请求提供统一接口。内置一个RPC系统的服务器。将来可能会有其他的实现。该服务器还允许您定义多个编解码器以提供不同的编码消息。

客户端

客户端提供一个接口来向服务发出请求。就像服务器一样，它建立在其他软件包上，以提供一个统一的界面，用于使用注册机制，根据名称查找服务，使用选择器进行负载平衡，使用代理进行传输和异步消息传输的同步请求。

上述组件被组合在微服务的顶层。

内部实现

以下是关于核心部件的内部工作的原理。

service.Run()

Go-micro服务通过调用service.Run()来启动，

1. 在启动功能之前执行

```
 for _, fn := range s.opts.BeforeStart {
         if err := fn(); err != nil {
                 return err
         }
 }
```

2. 启动服务

```
if err := s.opts.Server.Start(); err != nil {
         return err
}
```

3. 注册和服务发现

```
 if err := s.opts.Server.Register(); err != nil {
         return err
 }
```

4. 在启动功能后执行

```
for _, fn := range s.opts.AfterStart {
        if err := fn(); err != nil {
                return err
        }
}
```

server.Start()

server.Start由service.Run调用

1. 调用transport.Listen监听连接

```
ts, err := config.Transport.Listen(config.Address)
if err != nil {
        return err
}
```

2. 调用transport.Accept开始接收链路

```
go ts.Accept(s.accept)
```

3. 调用broker.Connect开始处理链路消息

```
config.Broker.Connect()
```

4. 等待退出信号，关闭运输并断开代理

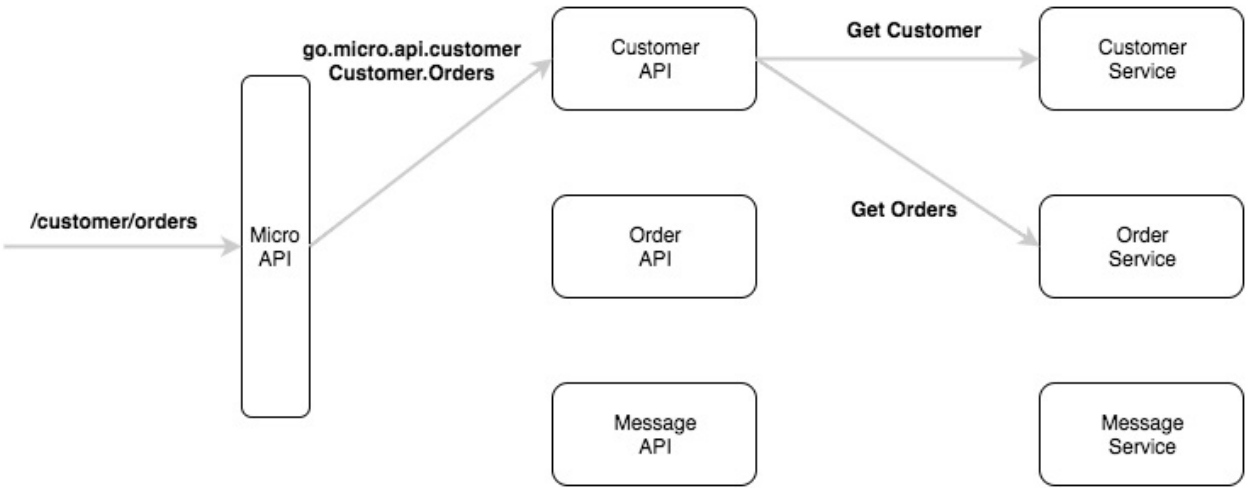
```
go func() {
        // wait for exit
        ch := <-s.exit

        // wait for requests to finish
        if wait(s.opts.Context) {
                s.wg.Wait()
        }

        // close transport listener
        ch <- ts.Close()

        // disconnect the broker
        config.Broker.Disconnect()
}()
```

编写服务

查看入门文档

Micro API

micro api是微服务的API网关。使用API网关模式为您的服务提供一个入口点。
micro api提供HTTP并动态路由到适当的后端服务。



如何工作的

micro api构建在go-micro上，利用它进行服务发现，负载平衡，编码和基于RPC的通信。对API的请求通过HTTP提供，并通过RPC进行内部路由。

由于micro api在内部使用go-micro，因此它也支持插件，因此可以随时切换为kubernetes api的consul服务发现或gRPC的RPC。

API

micro api提供了以下HTTP API

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

见下面的例子

Handler

Handler是管理请求路由的HTTP处理程序。

默认Handler使用注册表中的端口元数据来确定服务路由。如果未找到所匹配的路由，它将回退到API处理程序。您可以使用go-api配置注册路由。

该API有三种可配置的请求Handler。

1. API Handler： `/[service]/[method]`
    - 请求/响应： `api.Request/api.Response`
    - 该路径用于解析服务和方法
    - 请求通过API服务处理，API服务采用请求 `api.Request` 和响应 `api.Response` 类型
    - 请求/响应的定义可以在go-api/proto中找到
    - 请求/响应主体的内容类型可以是任何东西
    - 路由不可用的默认回退处理程序
    - 通过 `--handler=api` 设置
2. RPC Handler： `/[service]/[method]`
    - 请求/响应： `json/protobuf`
    - 使用go-micro客户端将请求主体转发为RPC请求的默认处理程序的替代方案
    - 允许使用具体的Go类型定义API处理程序。
    - 在不需要完全控制标题或请求/响应的情况下很有用
    - 可以用来运行单层后端服务，而不是其他API服务
    - 支持的内容类型 `application/json` 和 `application/protobuf`
    - 通过 `--handler=rpc` 设置
3. 反向代理： `/[service]`
    - 请求/响应：http
    - 该请求经过反向代理到服务的路径的第一个处理
    - 这允许REST在API后面实现
    - 通过 `--handler=proxy` 设置
4. Event Handler： `/[topic]/[event]`
    - 异步处理程序向消息代理发布请求作为事件
    - 请求被格式化为go-api/proto.Event

- 通过 `--handler=event` 进行设置

或者，使用 `/rpc` 端口直接与任何服务通话 - 期望参数： `service` ， `method` ， `request` ，可选接受 `address` ，以指定特定主机。

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

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

API Handler 请求/响应原型

API Handler是一个默认处理原型，服务也是基于该原型使用特定的请求和响应处理，可在go-api/proto中获得。这允许micro api将HTTP请求解析为RPC并返回到HTTP。

入门

安装

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

运行

```
micro api
```

通过ACME加密

通过使用letsencrypt的ACME，提供默认安全服务

```
micro --enable_acme api
```

可以指定一个主机白名单

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

提供安全的TLS

该API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 api
```

设置命名空间

该API默认为服务名称空间 `go.micro.api` 。命名空间和请求路径的组合用于解析发送查询的API服务和方法。

```
micro api --namespace=com.example.api
```

例子

这里我们有一个3层架构的例子

- micro api（localhost：8080） - 作为http入口点
- api服务（go.micro.api.greeter） - 为面向公众提供服务
- 后端服务（go.micro.srv.greeter） - 内部范围服务

完整的工作示例在这里

运行示例

先决条件：确保您正在运行服务发现，例如consul agent -dev

获取示例

```
git clone https://github.com/micro/examples
```

启动服务go.micro.srv.greeter

```
go run examples/greeter/srv/main.go
```

启动API服务go.micro.api.greeter

```
go run examples/greeter/api/api.go
```

开始 Micro API

```
micro api
```

查询

通过micro API进行HTTP调用

```
curl "http://localhost:8080/greeter/say/hello?name=Asim+Aslam"
```

HTTP `path/greeter/say/hello` 映射到服务 `go.micro.api.greeter` 方法 `Say.Hello`

绕过api服务并通过 `/rpc` 直接调用后端

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

与JSON完全相同的调用

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

请求映射

Micro使用固定的命名空间和HTTP路径动态地路由到服务。

这些服务的默认命名空间是 `go.micro.api` ，但可以通过 `--namespace` 标志设置命名空间。

每个服务的API

我们提倡为面向公众的流量创建每个后端服务的API服务模式。这在逻辑上将服务API前端和后端服务的分开。

RPC映射

URLs映射如下：

Path	Service	Method
/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

版本化的API URL可以很容易地映射到服务名称：

Path	Service	Method
/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

REST映射

您可以使用API作为反向代理并使用诸如go-restful之类的库实现RESTful路径，从而为RESTful API提供服务。REST API服务的一个例子可以在greeter/api/rest找到。

使用 `--handler=proxy` 运行micro API会将代理请求反转为API名称空间内的服务。

Path	Service	Service Path
/foo/bar	go.micro.api.foo	/foo/bar
/greeter	go.micro.api.greeter	/greeter
/greeter/:name	go.micro.api.greeter	/greeter/:name

使用这个处理程序意味着直接与后端服务通话，忽略任何go-micro传输插件。

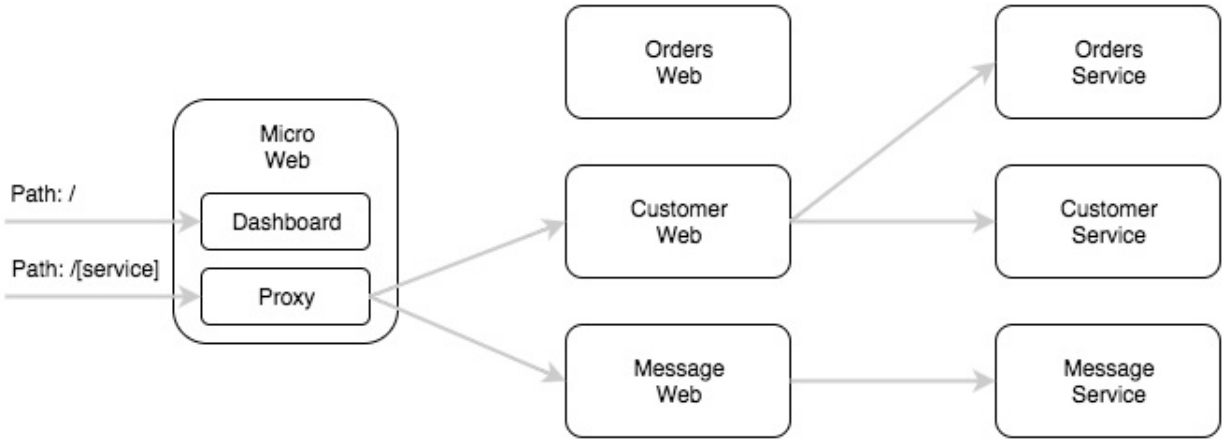
统计仪表板

通过 `--enable_stats` 标志启用统计信息显示板。它将暴露在 `/stats` 上。

```
micro --enable_stats api
```

Micro Web

Micro Web提供了一个用于查看和查询服务的仪表板，以及一个用于为Micro Web应用程序提供服务的反向代理。我们相信web应用程序是微服务世界中的一等公民。



API

```
- / (UI)
- /[service]
- /rpc
```

特性

Feature	Description
UI	用于查看和查询正在运行的服务的仪表板
Proxy	Micro Web服务的反向代理（包括websocket支持）

Proxy

Micro Web为网络应用程序提供内置的HTTP反向代理。这基本上允许您将Web应用程序视为微服务环境中的一等公民。代理将使用 `/[service]` 以及命名空间（默认：go.micro.web）在服务发现中查找服务。它将服务名称组成

为 `[namespace].[name]` 。

该代理将从请求中除去 `/[service]` ，并将URL路径的其余部分转发给Web应用程序。它还会将头部"X-Micro-Web-Base-Path"设置为已删除的路径，因此您需要使用它才能构建URL等某种原因。

示例翻译

Path	Service	Service Path	Header: X-Micro-Web-Base-Path
/foo	go.micro.web.foo	/	/foo
/foo/bar	go.micro.web.foo	/bar	/foo

注意：*Web*代理使用*HTTP*请求服务。没有其他传输能力。

入门

安装

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

运行Web UI/Proxy

```
micro web
```

通过浏览器访问 `localhost:8082`

通过ACME使能加密

通过ACME提供默认安全服务

```
micro --enable_acme web
```

可以指定一个主机白名单

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

提供**TLS**安全

Web代理支持使用TLS证书提供安全服务

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

设置命名空间

Web默认服务名称空间为**go.micro.web**。名称空间和请求路径的组合用于解析服务以反向代理。

```
micro web --namespace=com.example.web
```

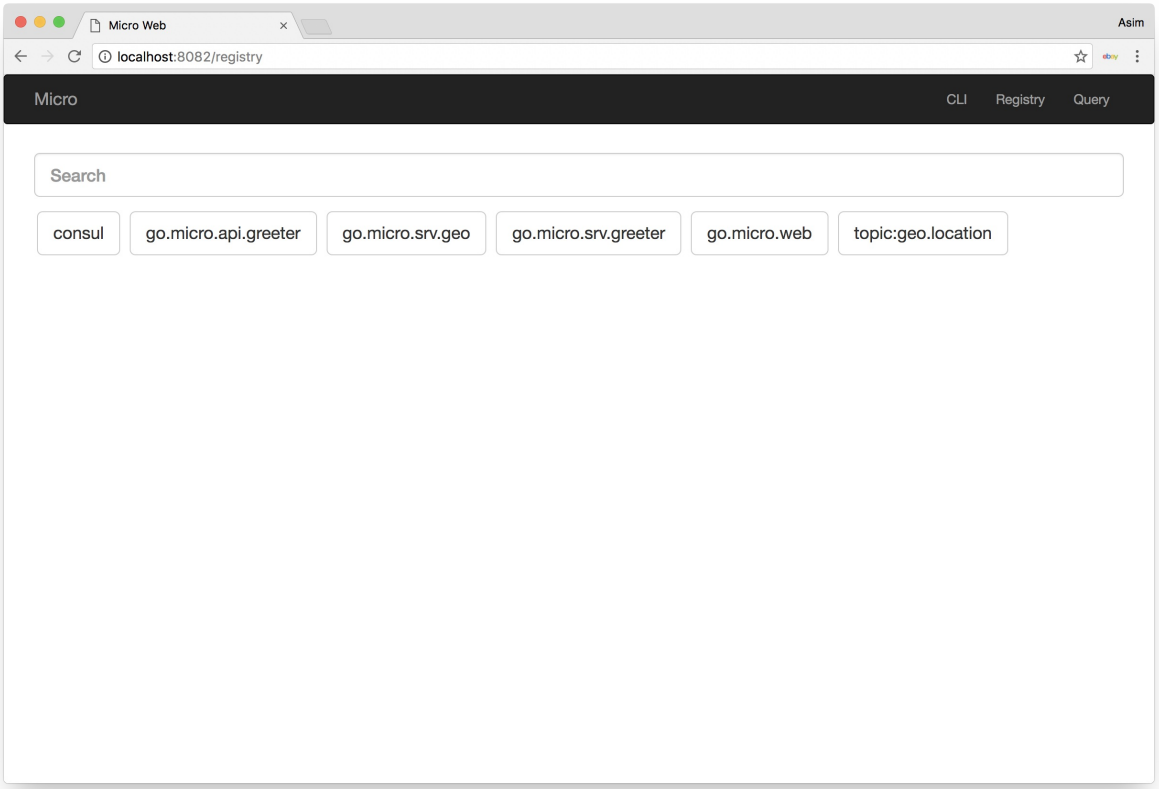
统计

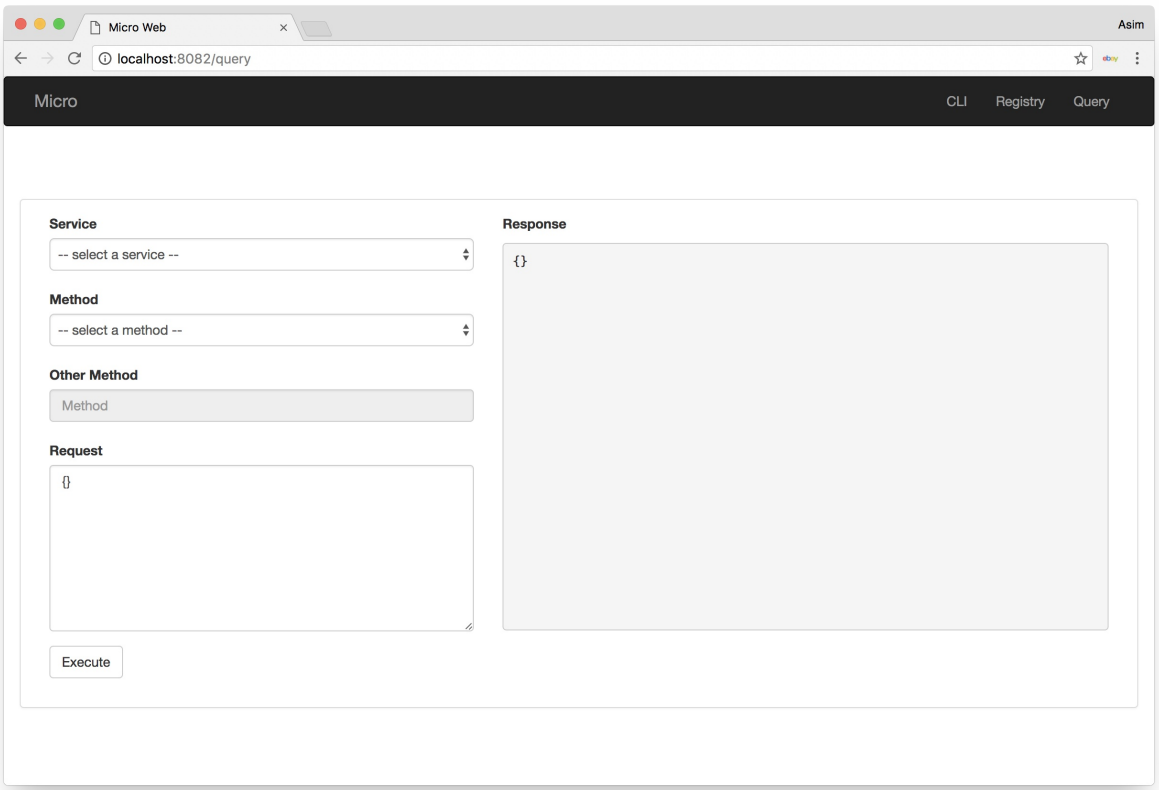
您可以通过 `--enable_stats` 标志启用统计信息显示板。它将暴露在 `/stats` 上。

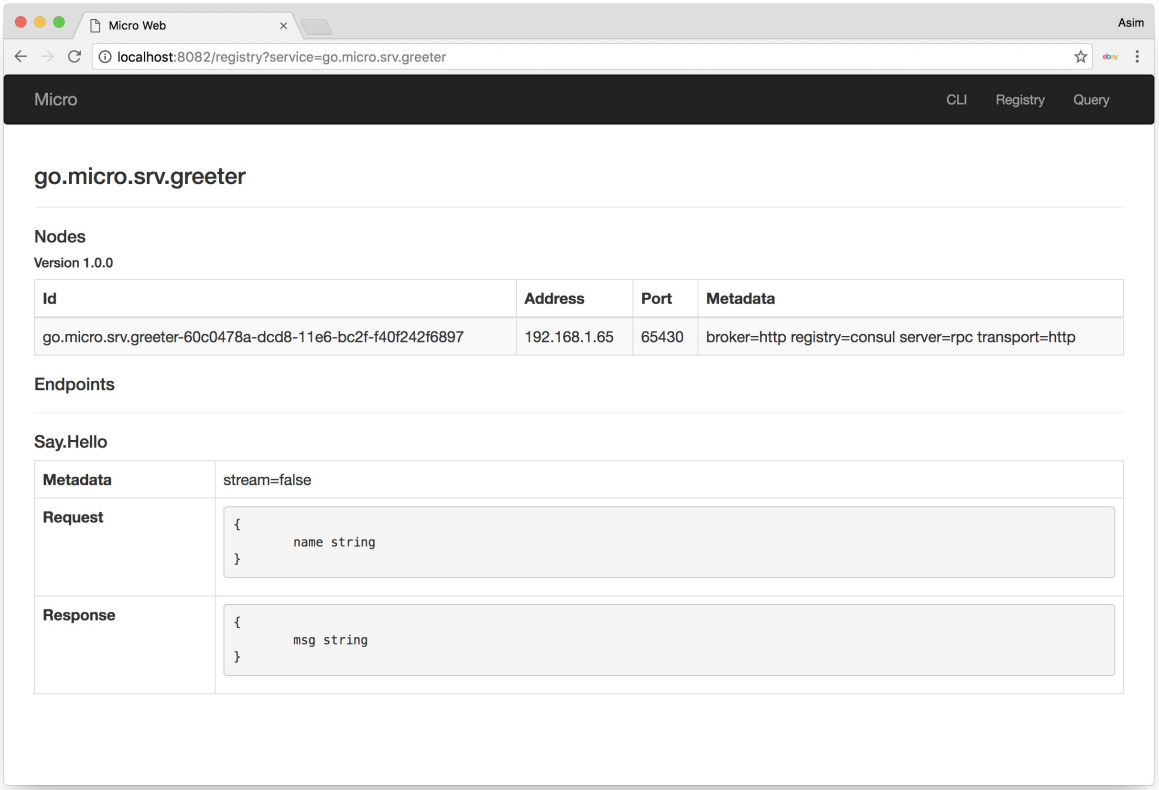
```
micro --enable_stats web
```

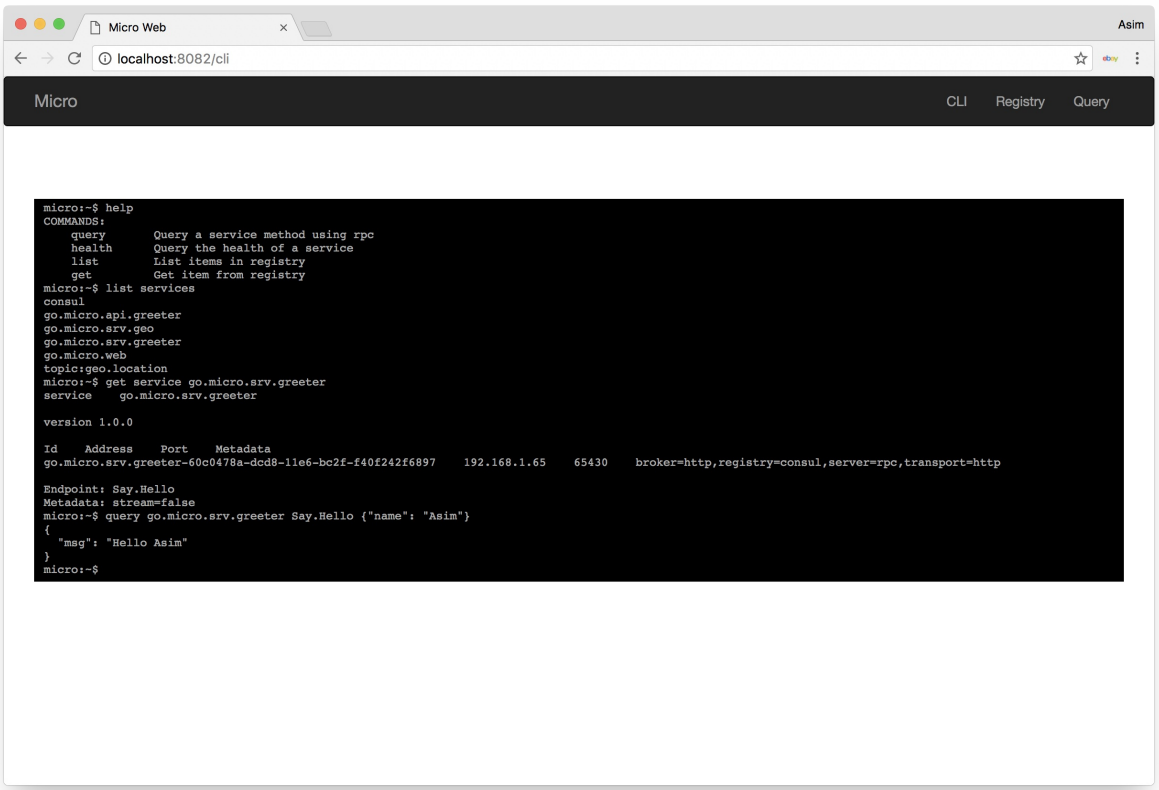

Micro
Last updated Mon, 29 Feb 2016 21:16:28 GMT
Info
Started
Mon, 29 Feb 2016 21:15:24 GMT
Uptime
64.907s
Memory
1.06mb
Threads
17
GC
5.493ms
Requests
Total
1133
20x
525
40x
284
50x
324
Request Load
50
40
30
20
10
0
09:15:30 PM
09:15:50 PM
09:16:10 PM
09:16:30
updates every 5 secs
20x 36 40x 20 50x 14
CanvasJS.com

截图


Micro Web
Asim
localhost:8082/registry
Micro
CLI
Registry
Query
Search
consul
go.micro.api.greeter
go.micro.srv.geo
go.micro.srv.greeter
go.micro.web
topic:geo.location


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
}

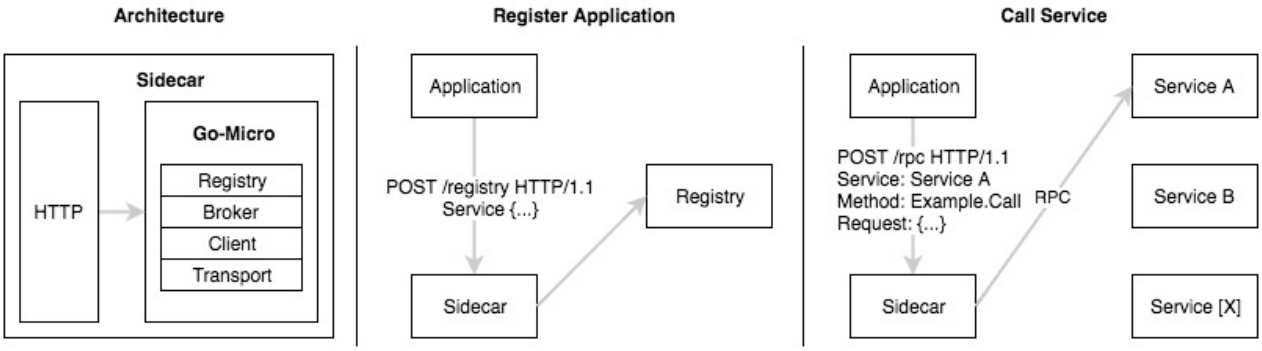

Micro Web
Asim
localhost:8082/cli
Micro
CLI
Registry
Query
micro:~$ help
COMMANDS:
query Query a service method using rpc
health Query the health of a service
list List items in registry
get Get item from registry
micro:~$ list services
consul
go.micro.api.greeter
go.micro.srv.geo
go.micro.srv.greeter
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:~$

Micro Sidecar

Micro Sidecar是用于构建高度可用和容错微服务的服务网格。

它与Netflix的Prana，Buoyant的RPC Proxy Linkerd或Lyft的Envoy类似。

Micro Sidecar采用go-micro，具有相同的默认设置和可插拔性。



可以在examples/sidecar找到许多语言的用法示例。

API

该sidecar具有以下HTTP API。

```
- /[service]/[method]
- /broker
- /registry
- /rpc
```

特征

sidecar具有go-micro的所有功能。这是最相关的。

- 服务发现
- 消息总线
- RPC和代理处理程序
- 负载平衡，重试，超时
- 健康检测

- 统计界面
- 可通过go-micro插入

入门

安装

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

依赖

Sidecar使用go-micro，这意味着它有一个默认依赖关系，用于服务发现的Consul。

```
brew install consul
consul agent -dev
```

运行

默认情况下，在端口8081上运行Micro Sidecar。

启动Sidecar

```
micro sidecar
```

如果要在启动时自动注册应用程序，请指定应用程序服务名和地址。

```
micro sidecar --server_name=foo --server_address=127.0.0.1:9090
```

通过**ACME**使能加密

通过使用ACME提供安全服务

```
micro --enable_acme sidecar
```

可以指定一个主机白名单

```
micro --enable_acme --acme_hosts=example.com,proxy.example.com s
idecar
```

提供TLS安全

Sidecar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 sidecar
```

自动健康检查

用“-healthcheck_url=”启动微型边车以启用健康检查器

它执行以下操作：

- 自动服务注册
- 定期HTTP健康检查
- 通过非200响应取消注册

```
micro sidecar --server_name=foo --server_address=127.0.0.1:9090
\
    --healthcheck_url=http://127.0.0.1:9090/health
```

注册

注册服务

```
// specify ttl as a param to expire the registration
// units ns|us|ms|s|m|h
// http://127.0.0.1:8081/registry?ttl=10s

curl -H 'Content-Type: application/json' http://127.0.0.1:8081/r
egistry -d
{
    "Name": "foo.bar",
    "Nodes": [{
        "Port": 9091,
        "Address": "127.0.0.1",
        "Id": "foo.bar-017da09a-734f-11e5-8136-68a86d0d36b6"
    }]
}
```

取消注册

```
curl -X "DELETE" -H 'Content-Type: application/json' http://127.
0.0.1:8081/registry -d
{
    "Name": "foo.bar",
    "Nodes": [{
        "Port": 9091,
        "Address": "127.0.0.1",
        "Id": "foo.bar-017da09a-734f-11e5-8136-68a86d0d36b6"
    }]
}
```

查询服务

```
curl http://127.0.0.1:8081/registry?service=go.micro.srv.example
{
    "name":"go.micro.srv.example",
    "nodes":[{
        "id":"go.micro.srv.example-c5718d29-da2a-11e4-be11-68a86
d0d36b6",
        "address":"[::]","port":60728
    }]
}
```

Handlers

RPC

使用json或protobuf查询微服务。对后端的请求将使用go-micro RPC客户端进行。

使用 `/[service]/[method]`

所调用服务的默认名称空间是 `go.micro.srv`

```
curl -H 'Content-Type: application/json' -d '{"name": "John"}' h
ttp://127.0.0.1:8081/example/call
```

使用 `/rpc` 端口

```
curl -d 'service=go.micro.srv.example' \
    -d 'method=Example.Call' \
    -d 'request={"name": "John"}' http://127.0.0.1:8081/rpc
```

Proxy

与api和web服务器一样，sidecar可以提供完整的http代理。

在命令行上启用代理处理程序。

```
micro sidecar --handler=proxy
```

URL路径中的第一个元素将与名称空间一起用作要路由到的服务。

请求映射

URL路径映射与Micro API相同

URL的映射如下：

Path	Service	Method
/foo/bar	go.micro.srv.foo	Foo.Bar
/foo/bar/baz	go.micro.srv.foo	Bar.Baz
/foo/bar/baz/cat	go.micro.srv.foo.bar	Baz.Cat

版本化的API URL可以很容易地映射到服务名称：

Path	Service	Method
/foo/bar	go.micro.srv.foo	Foo.Bar
/v1/foo/bar	go.micro.srv.v1.foo	Foo.Bar
/v1/foo/bar/baz	go.micro.srv.v1.foo	Bar.Baz
/v2/foo/bar	go.micro.srv.v2.foo	Foo.Bar
/v2/foo/bar/baz	go.micro.srv.v2.foo	Bar.Baz

事件

发布

```
curl -XPOST \
        -H "Timestamp: 1499951537" \
        -d "Hello World!" \
        "http://localhost:8081/broker?topic=foo"
```

订阅

```
conn, _, _ := websocket.DefaultDialer.Dial("ws://127.0.0.1:8081/
broker?topic=foo", make(http.Header))

// optionally specify "queue=[queue name]" param to distribute t
raffic amongst subscribers
// websocket.DefaultDialer.Dial("ws://127.0.0.1:8081/broker?topi
c=foo&queue=group-1", make(http.Header))

for {
    // Read message
    _, p, err := conn.ReadMessage()
    if err != nil {
        return
    }

    // Unmarshal into broker.Message
    var msg *broker.Message
    json.Unmarshal(p, &msg)

    // Print message body
    fmt.Println(msg.Body)
}
```

CLI代理

该sidecar还充当CLI访问远程环境的代理。

```
$ micro --proxy_address=127.0.0.1:8081 list services
go.micro.srv.greeter
```

统计仪表板

通过 `--enable_stats` 标志启用统计信息显示板。它将暴露在 `/stats` 上。

```
micro --enable_stats sidecar
```

Micro

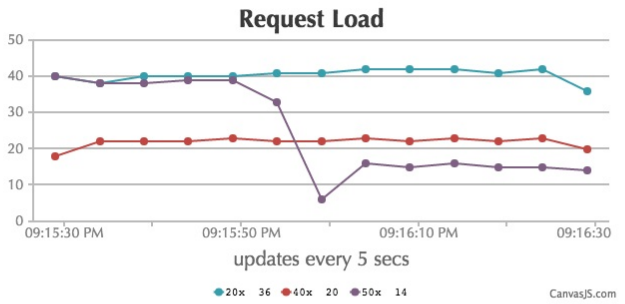
Last updated Mon, 29 Feb 2016 21:16:28 GMT

Info

Started	Mon, 29 Feb 2016 21:15:24 GMT
Uptime	64.907s
Memory	1.06mb
Threads	17
GC	5.493ms

Requests

Total	1133
20x	525
40x	284
50x	324



Micro cli

Micro cli是microservices toolkit micro的命令行界面。

入门

安装

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

服务列表

```
micro list services

go.micro.srv.example
```

获取服务

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

go.micro.srv.example

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

查询服务

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

{
    "msg": "go.micro.srv.example-fccbb6fb-0301-11e5-9f1f-68a86d0
d36b6: Hello John"
}
```

查询服务健康

```
micro health go.micro.sv.example

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

通过**CLI**注册/去注册

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

```
micro get service foo

service  foo

version  bar

Id    Address    Port    Metadata
foo-1    127.0.0.1    8080
```

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

```
micro get service foo

Service not found
```

运行API

```
micro api
```

运行SideCar

```
micro sidecar --server_name=foo --server_address=127.0.0.1:9090
--healthcheck_url=http://127.0.0.1:9090/_status/health
```

通过Sidecar代理CLI

该sidecar可以用作远程环境的代理。

```
micro --proxy_address=proxy.micro.pm list services

go.micro.srv.example
```

用法

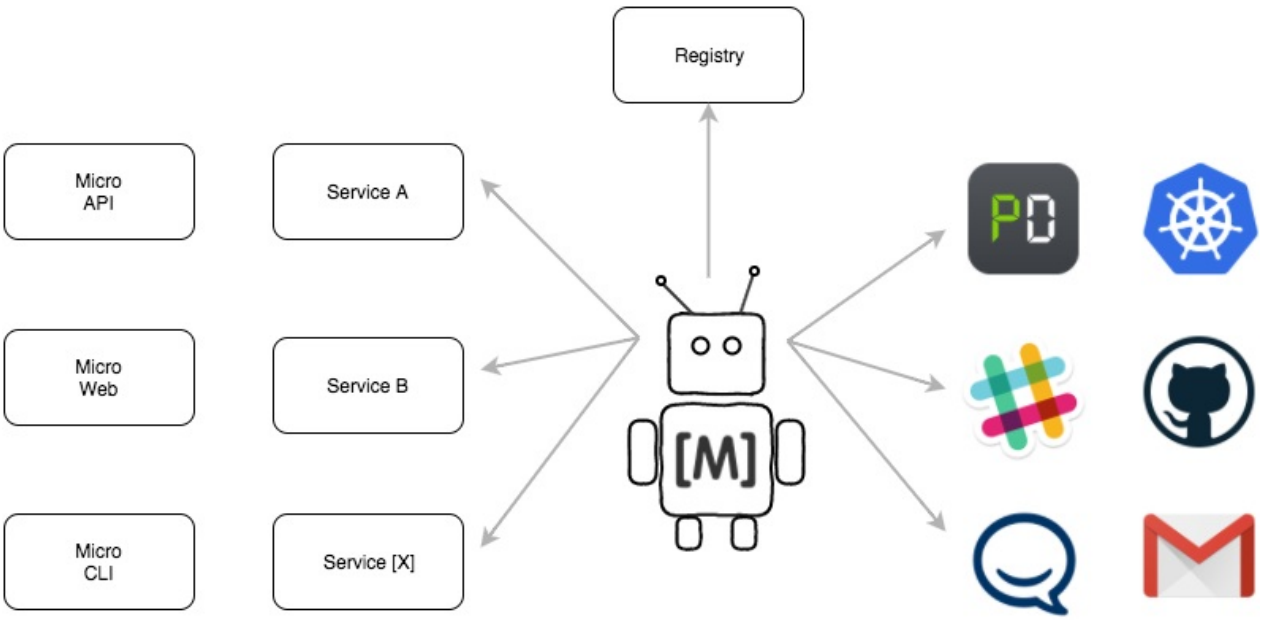
```
NAME:
   micro - A microservices toolkit

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

VERSION:
   latest
```

```
COMMANDS:
   api        Run the micro API
   bot        Run the micro bot
   registry    Query registry
   query    Query a service method using rpc
   stream    Query a service method using streaming rpc
   health    Query the health 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
   sidecar    Run the micro sidecar
   web        Run the micro web app
   help, h    Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --server_name                                    Name of the ser
ver. go.micro.srv.example [$MICRO_SERVER_NAME]
   --server_version                                   Version of t
he server. 1.1.0 [$MICRO_SERVER_VERSION]
   --server_id                                     Id of the ser
ver. Auto-generated if not specified [$MICRO_SERVER_ID]
   --server_address                                  Bind address
 for the server. 127.0.0.1:8080 [$MICRO_SERVER_ADDRESS]
   --server_advertise                                  Used inste
ad of the server_address when registering with discovery. 127.0.
0.1:8080 [$MICRO_SERVER_ADVERTISE]
   --server_metadata [--server_metadata option --server_metadata
 option]    A list of key-value pairs defining metadata. version
=1.0.0 [$MICRO_SERVER_METADATA]
   --broker                                    Broker for pub/s
ub. http, nats, rabbitmq [$MICRO_BROKER]
   --broker_address                                  Comma-separa
ted list of broker addresses [$MICRO_BROKER_ADDRESS]
   --registry                                   Registry for d
iscovery. memory, consul, etcd, kubernetes [$MICRO_REGISTRY]
   --registry_address                                  Comma-sepa
rated list of registry addresses [$MICRO_REGISTRY_ADDRESS]
   --selector                                   Selector used
to pick nodes for querying. random, roundrobin, blacklist [$MICR
```

```
O_SELECTOR]
   --transport                                    Transport mec
hanism used; http, rabbitmq, nats [$MICRO_TRANSPORT]
   --transport_address                              Comma-sep
arated list of transport addresses [$MICRO_TRANSPORT_ADDRESS]
   --enable_tls                                   Enable TLS [$
MICRO_ENABLE_TLS]
   --tls_cert_file                                TLS Certifica
te file [$MICRO_TLS_CERT_File]
   --tls_key_file                                TLS Key file [
$MICRO_TLS_KEY_File]
   --api_address                                Set the api add
ress e.g 0.0.0.0:8080 [$MICRO_API_ADDRESS]
   --proxy_address                                Proxy request
s via the HTTP address specified [$MICRO_PROXY_ADDRESS]
   --sidecar_address                                Set the sid
ecar address e.g 0.0.0.0:8081 [$MICRO_SIDECAR_ADDRESS]
   --web_address                                Set the web UI
address e.g 0.0.0.0:8082 [$MICRO_WEB_ADDRESS]
   --register_ttl "0"                               Register TT
L in seconds [$MICRO_REGISTER_TTL]
   --register_interval "0"                           Register i
nterval in seconds [$MICRO_REGISTER_INTERVAL]
   --api_handler                                Specify the req
uest handler to be used for mapping HTTP requests to services. e
.g api, proxy [$MICRO_API_HANDLER]
   --api_namespace                                Set the names
pace used by the API e.g. com.example.api [$MICRO_API_NAMESPACE]
   --web_namespace                                Set the names
pace used by the Web proxy e.g. com.example.web [$MICRO_WEB_NAME
SPACE]
   --api_cors                                    Comma separate
d whitelist of allowed origins for CORS [$MICRO_API_CORS]
   --web_cors                                    Comma separate
d whitelist of allowed origins for CORS [$MICRO_WEB_CORS]
   --sidecar_cors                                Comma separate
d whitelist of allowed origins for CORS [$MICRO_SIDECAR_CORS]
   --enable_stats                               Enable stats [$
MICRO_ENABLE_STATS]
   --help, -h                                   show help
```

微型机器人

微型机器人是一个位于微服务环境中的机器人，您可以通过Slack，HipChat，XMPP等进行交互。它通过消息传递模拟CLI的功能。



支持的输入

- Slack
- HipChat

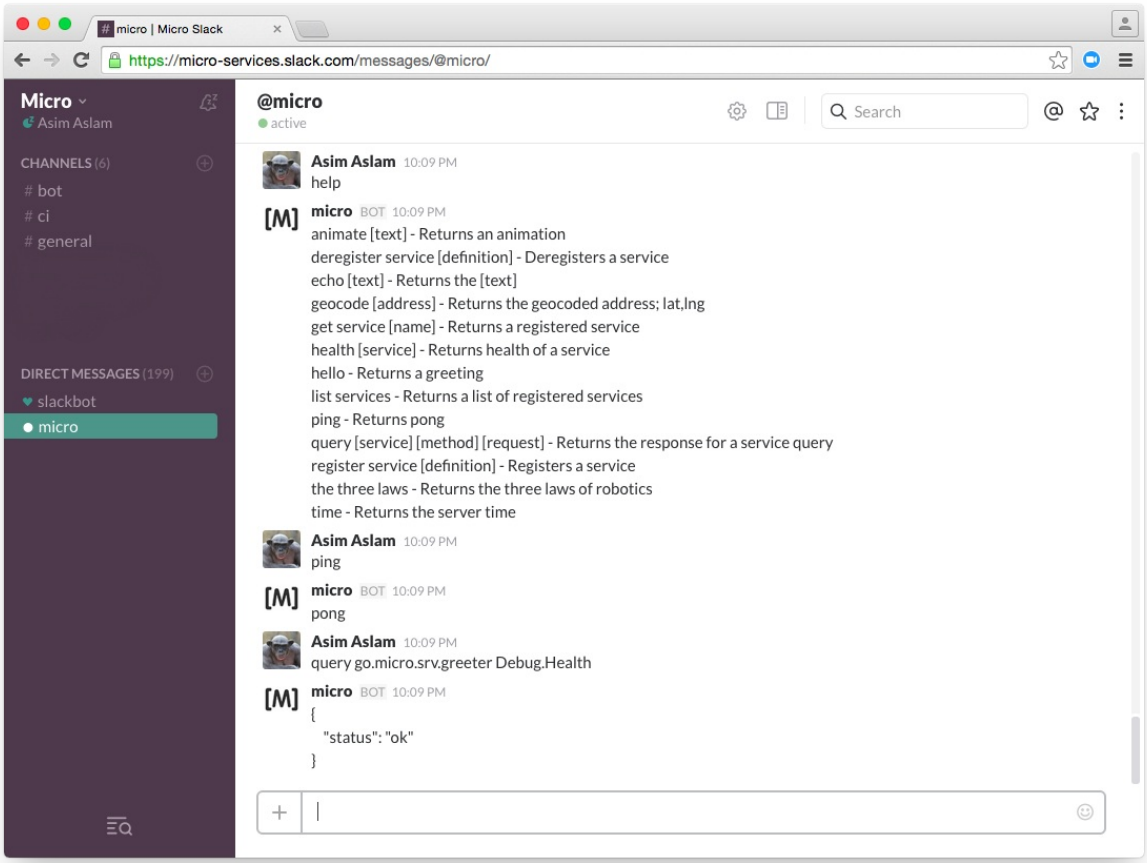
入门

安装Micro

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

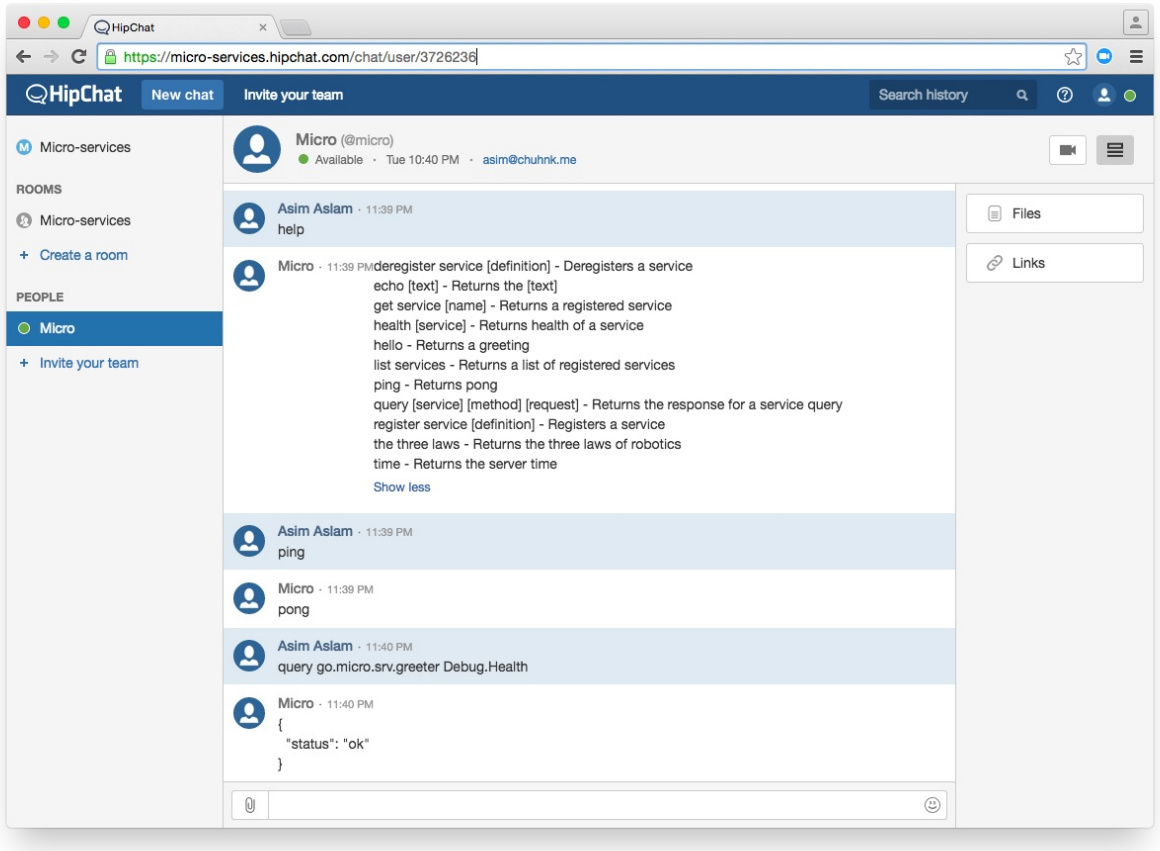
用Slack运行

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



用HipChat运行

```
micro bot --inputs=hipchat --hipchat_username=XMPP_USER --hipcha
t_password=XMPP_PASSWORD
```



通过使用逗号分隔列表来指定多个输入

```
micro bot --inputs=hipchat,slack --slack_token=SLACK_TOKEN --hip
chat_username=XMPP_USER --hipchat_password=XMPP_PASSWORD
```

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
```

添加新的命令

由机器人执行的命令和功能是基于文本的匹配模式。

写一个命令

```
import "github.com/micro/go-bot/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
    })
}
Registe
```

注册该命令

使用可以通过 `golang/regexp.Match` 匹配的模式键将命令添加到Commands映射表。

```
import "github.com/micro/go-bot/command"

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

重新构建Micro

构建二进制

```
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 micro ./main.go
```

添加新的输入

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

可以通过以下方式添加新的输入。

编写一个输入

编写满足输入接口的输入。

```
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

将输入添加到输入映射。

```
import "github.com/micro/go-bot/input"

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

重新构建Micro

构建二进制

```
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 micro ./main.go
```

作为服务的命令

微型机器人支持作为微服务创建命令的能力。

它是如何工作的？

机器人使用它的命名空间监视服务注册中心的服务。默认名称空间是 `go.micro.bot` 。该名称空间内的任何服务都将自动添加到可用命令列表中。执行命令时，机器人将使用 `Command.Exec` 方法调用该服务。它也希望方法 `Command.Help` 存在于使用信息中。

服务接口如下，可以在go-bot/proto中找到。

```
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

这里有一个echo命令作为微服务的示例。

```
package main

import (
    "fmt"
    "strings"

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

```
    proto "github.com/micro/go-bot/proto"
)

type Command struct{}

// Help returns the command usage
func (c *Command) Help(ctx context.Context, req *proto.HelpReque
st, rsp *proto.HelpResponse) 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 echos the message"
    return nil
}

// Exec executes the command
func (c *Command) Exec(ctx context.Context, req *proto.ExecReque
st, rsp *proto.ExecResponse) 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 i
ssues
    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)
    }
}
```

Micro New[service]

micro new命令是为微服务生成示例模板的快捷方式。

用法

通过指定相对于$GOPATH的目录路径来创建一个新服务。

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

运行的示例。

```
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/subscrib
er
creating /Users/asim/checkouts/src/github.com/micro/foo/subscrib
er/example.go
creating /Users/asim/checkouts/src/github.com/micro/foo/proto/ex
ample
creating /Users/asim/checkouts/src/github.com/micro/foo/proto/ex
ample/example.proto
creating /Users/asim/checkouts/src/github.com/micro/foo/Dockerfi
le
creating /Users/asim/checkouts/src/github.com/micro/foo/README.m
d

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

指定更多选项，如名称空间，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.e
xample
   --type "srv"            Type of service e.g api, srv, web
   --fqdn              FQDN of service e.g com.example.srv.servic
e (defaults to namespace.type.alias)
   --alias             Alias is the short name used as part of c
ombined name if specified
```

Micro Run

micro run命令管理微服务的生命周期。它获取源代码，构建二进制文件并执行它。这是一个可用于本地开发的简单工具。如果没有指定参数，则微运行作为可以管理其他服务的服务来运行。

注意：默认运行时（*Go*）需要设置*PATH*和*GOPATH*中的*Go*二进制文件。

概述

Run

```
micro run github.com/service/foo
```

Status

```
micro run -s github.com/service/foo
```

Kill

```
micro run -k github.com/service/foo
```

运行服务管理

```
micro run
```

推迟运行服务管理

```
micro run -x github.com/service/foo
```

运行并重新启动

```
micro run -r github.com/service/foo
```

运行并更新源代码

```
micro run -u github.com/service/foo
```

使用帮助

```
NAME:
   micro run - Run the micro runtime

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

OPTIONS:
   -k    Kill service
   -r    Restart if dies. Default: false
   -u    Update the source. Default: false
   -x    Defer run to service. Default: false
   -s    Get service status
```

TODO

- [ ]支持接受args和env变量的服务
- [ ]添加服务接口go-run
- [ ]支持Go以外的可配置运行时
- [ ]插件支持重建
- [ ]守护进程？
- [ ]监控内存消耗并kill？
- [ ]chroot的进程？

部署

- Docker
- Kubernetes

Docker部署

Micro易于在docker容器中运行

预置的镜像

Docker Hub上提供了预置的镜像

安装Micro

```
docker pull microhq/micro
```

Compose

使用docker compose运行本地部署

```
consul:
  command: -server -bootstrap -rejoin
  image: progrium/consul:latest
  hostname: "registry"
  ports:
  - "8300:8300"
  - "8400:8400"
  - "8500:8500"
  - "8600:53/udp"
api:
  command: --registry_address=registry:8500 --register_interval=
5 --register_ttl=10 api
  build: .
  links:
  - consul
  ports:
  - "8080:8080"
sidecar:
  command: --registry_address=registry:8500 --register_interval=
5 --register_ttl=10 sidecar
  build: .
  links:
  - consul
  ports:
  - "8081:8081"
web:
  command: --registry_address=registry:8500 --register_interval=
5 --register_ttl=10 web
  build: .
  links:
  - consul
  ports:
  - "8082:8082"
```

从头开始构建

Dockerfile包含在仓库中。

```
FROM alpine:3.2
RUN apk add --update ca-certificates && \
    rm -rf /var/cache/apk/* /tmp/*
ADD micro /micro
WORKDIR /
ENTRYPOINT [ "/micro" ]
```

构建二进制

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

构建镜像

```
docker build -t micro .
```

Kubernetes部署

Micro可以在kubernetes上轻松运行，甚至可以使用kubernetes API进行服务发现。

配置

在github.com/micro/kubernetes仓库为Kubernetes提供了Micro示例配置。

```
git clone https://github.com/micro/kubernetes
```

仓库中有什么？

目前配置为在Kubernetes上运行Micro（在Google Container Engine上测试）。

- 微型API，Web UI和Sidecar（旋转GCE负载平衡器）
- 服务 - 一些示例微服务
- run.sh - 使用kubectl启动/停止服务的简单shell脚本

入门

以下是我开始使用的步骤。

运行Kubernetes

GKE是运行托管kubernetes集群的最简单方法。什么更好？免费赠送300美元在60天期限内。

1. 让自己免费试用Google Container Engine
2. 访问快速入门指南以创建群集

运行Micro

确保kubectl命令行在path中。

```
./run.sh start
```

插件

插件

Micro为所有工具提供可插拔架构。这意味着底层的实现可以被换出。

Go-Micro和Micro工具包包含不同类型的插件。从侧边栏导航以了解更多信息。

例子

Go Micro

- Etcd Registry - 使用Etcd进行服务发现
- K8s Registry - 使用Kubernetes进行服务发现
- Kafka Broker - Kafka消息总线

工具包

- 路由器 - 可配置的http路由和代理
- AWS X-Ray - 跟踪AWS X-Ray的集成
- IP Whitelite - 白名单IP地址访问

知识库

开源插件可以在github.com/micro/go-plugins上找到。

Go Micro Plugins

Micro是一个可插拔的工具包和框架。内部功能都可以通过go-plugins进行替换。

该工具包有一个单独的插件界面。micro/plugin了解更多。

以下是关于go-micro插件的使用情况。

用法

可以通过以下方式将插件添加到go-micro中。通过这样做，他们可以通过命令行参数或环境变量进行设置。

导入插件

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

func main() {
    // Parse CLI flags
    cmd.Init()
}
```

调用 `service.Init` 时也是一样

```
import (
    "github.com/micro/go-micro"
    _ "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()
}
```

通过CLI标志使用

通过CLI标志激活

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

直接使用插件

CLI标志提供了初始化插件的简单方法，但您可以自己做同样的事情。

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

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

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

构建模式

您可能想要使用自动化替换插件或将插件添加到micro工具包。一个简单的方法是通过为插件导入维护一个单独的文件并在构建过程中包含它。

创建plugins.go文件

```
package main

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

构建plugins.go

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

运行plugins

```
shell service --broker=rabbitmq --registry=kubernetes --transpor
t=nats
```

用插件重建工具包

如果你想集成插件，只需将它们链接到一个单独的文件并重建。

创建一个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"
)
```

构建二进制文件

```
// 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 micro ./main.go ./plugins.go
```

插件标记的用法

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

仓库

go-micro插件可以在这里找到github.com/micro/go-plugins。

工具包插件

插件是将外部代码集成到Micro工具箱的一种方式。这与Micro插件完全分开。这里使用的插件，允许您在工具包中添加额外的标志，命令和HTTP处理程序。

怎么运行的？

在 `micro/plugin` 下有一个全局插件管理器，它包含整个工具包中使用的插件。可以通过调用 `plugin.Register` 来注册插件。每个组件（api，web，sidecar，cli，bot）都有一个单独的插件管理器，用于注册只作为该组件一部分的添加插件。它们可以通过称为 `api.Register` ， `web.Register` 等以相同的方式使用。

这是接口

```
// Plugin is the interface for plugins to micro. It differs from
 go-micro in that it's for
// the micro API, Web, Sidecar, CLI. It's a method of building m
iddleware 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 pa
ss in
    // the existing handler so it can be wrapped to create a cal
l 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 retrieved.
// 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 exist
ing http.Handler passed in.
// Its the responsibility of the Handler to call the next http.H
andler in the chain.
type Handler func(http.Handler) http.Handler
```

如何使用它

这里有一个简单的插件示例，它添加一个标志，然后打印该值。

插件

在顶级目录中创建一个plugin.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

只需使用该插件构建micro。

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

知识库

该工具包的插件可以在中找到github.com/micro/go-plugins/micro。

NATS

这篇文档着眼于nats，将nats消息系统集成到micro工具包中。它讨论包括围绕服务发现，微服务的同步和异步通信。

什么是NATS？

NATS是一个开源的原生云消息系统或更简单的消息总线。NATS由Apcera的创始人Derek Collison创建。它起源于VMWare，开始基于ruby的一个系统。基于Go重写了很长时间，当前正在稳步进行，并且获得那些寻求高度扩展和高性能消息传递系统的人们所采用。

如果您想了解有关NATS的更多信息，请访问nats.io或加入社区。

为什么选择NATS？

为什么不NATS？过去曾与许多消息总线合作过，很快就清楚NATS是分开的。多年来，消息总线被誉为企业的救星，导致系统试图成为所有人的一切。这导致了很多虚假的承诺，显着的性能膨胀和高成本技术，产生了比解决问题更多的问题。

相比之下，NATS采取非常专注的方式，解决性能和可用性问题，同时保持令人难以置信的精益。它提到“永远在线并且可用”，并且使用“消防和忘记”消息模式。它的简单性，重点和轻量级特性使其成为微服务生态系统的主要候选者。我们相信它很快将成为服务间消息传递的主要候选人。

NATS提供了什么：

- 高性能和可扩展性
- 高可用性
- 极其轻巧
- 最多一次交付

什么NATS不提供：

- 持久化
- 事务处理

- 增强交付模式
- 企业级队列

这简要介绍了选择NATS的原因。那么它如何适应Micro？来！我们讨论一下。

Micro on NATS

Micro是一个微服务工具包，采用可插拔的体系结构，允许将底层的依赖关系以最小的更改进行替换。Go-Micro框架的每个接口都为微服务提供了构建块；用于服务发现的注册表，用于同步通信的传输，用于异步消息总线的代理等等。

为每个组件创建插件并实现接口一样简单。我们将花更多时间详细说明，如何在未来的博客文章中撰写插件。如果您想查看NATS或任何其他系统的插件，例如etcd discovery，kafka broker，rabbitmq transport，您可以在这里找到它们 github.com/micro/go-plugins。

基于NATS的Micro本质上是一组go-micro插件，可用于与NATS消息总线系统集成。通过为go-micro的各种接口提供插件，我们创建了许多可以选择的架构模式集成。

根据我们的经验，一种尺寸不适合所有情况，而Micro NATS的灵活性允许您定义适合您和您的团队模型。

下面我们将讨论传输，代理和注册中心的NATS插件的实现。

传输

传输是同步通信的Micro接口。它使用相当普通的Socket语义，与 `Listen` ， `Dial` 和 `Accept` 类似。这些概念和模式对于使用tcp，http等的同步通信很好理解，但适应消息总线可能有些困难。与消息总线建立连接，而不是与服务本身建立连接。为了解决这个问题，我们使用与topics和channels的伪连接的概念。

这是它的工作原理。

服务使用 `transport.Listen` 来侦听消息。这将创建一个到NATS的连接。
当 `transport.Accept` 被调用时，一个独特的topic被创建和订阅。这个独特的topic将被用作go-micro注册表中的服务地址。接收到的每条消息将被用作伪套接字/连接的基础。如果现有连接具有相同的回复地址，我们只需将该消息放入该连接的缓存中。

想要与此服务通信的客户端将使用 `transport.Dial` 创建与服务的连接。 这将连接到NATS，创建它自己独特的topic并订阅它。该topic用于服务的响应。每当客户端向服务发送消息时，它都会将回复地址设置为该topic。

当任何一方想要关闭连接时，他们只需调用 `transport.Close` 即可终止与NATS的连接。

使用传输插件

导入传输插件

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

从传输标志开始

```
go run main.go --transport=nats --transport_address=127.0.0.1:42
22
```

或者直接使用传输工具

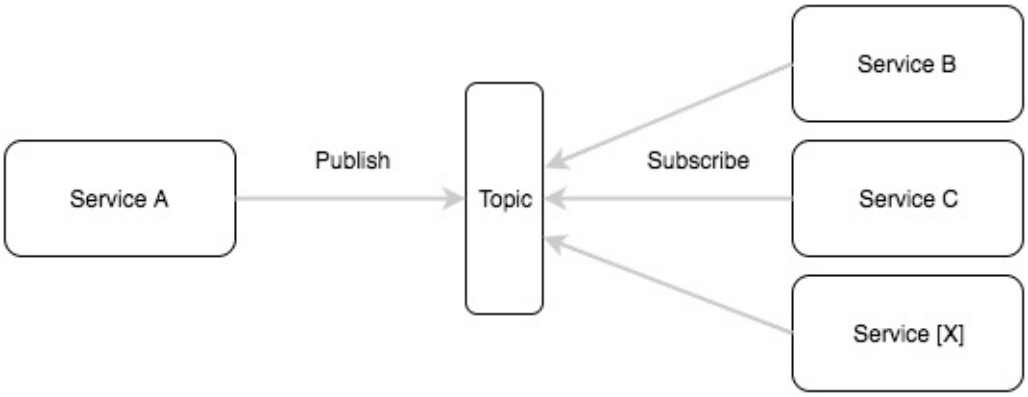
```
transport := nats.NewTransport()
```

go-micro传输接口

```
type Transport interface {
    Dial(addr string, opts ...DialOption) (Client, error)
    Listen(addr string, opts ...ListenOption) (Listener, error)
    String() string
}
```

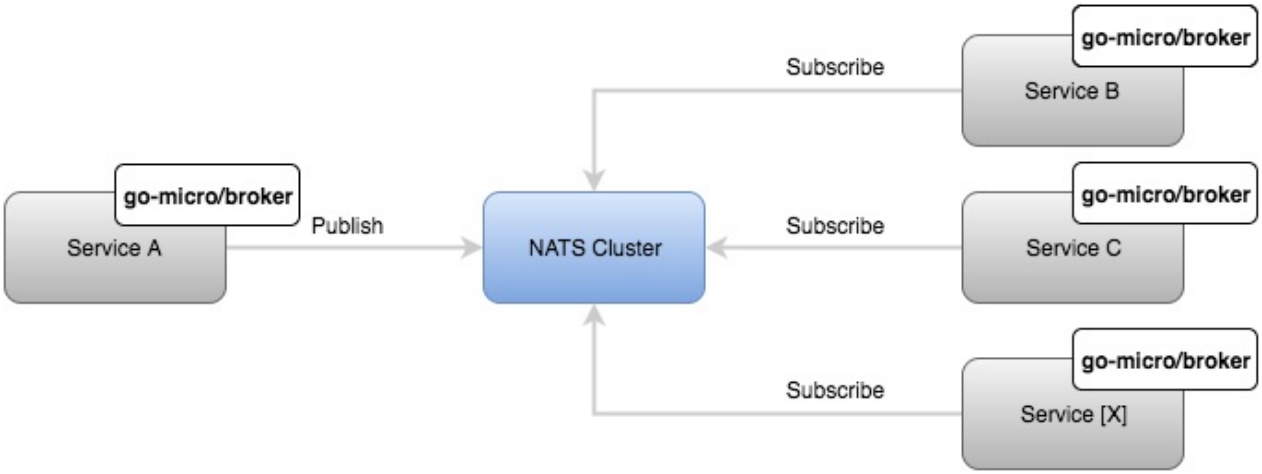
Transport

Broker



代理是异步消息的转发micro接口。它提供了适用于大多数邮件代理的高级通用实现。就其本质而言，NATS是一个异步消息传输系统，它被用作消息代理。只有一个警告，NATS不持久化消息。虽然这对一些人来说可能并不理想，但我们仍然认为NATS可以也应该作为一个代理来使用go-micro。在不需要持久性的情况下，它允许高度可扩展的pub/sub子体系结构。

NATS提供了一个非常直接的发布和订阅机制，包括topic，channel等概念。在这里没有真正的花哨工作来使它工作。消息可以以异步火灾和遗忘的方式发布。使用相同channel名称的订阅者在NATS中形成一个队列组，然后它将允许消息自动均匀分布在订阅者中。



使用代理插件

导入代理插件

```
import _ "github.com/micro/go-plugins/broker/nats"
```

从代理标志开始

```
go run main.go --broker=nats --broker_address=127.0.0.1:4222
```

或者直接使用代理

```
broker := nats.NewBroker()
```

go-micro 代理接口：

```
type Broker interface {
    Options() Options
    Address() string
    Connect() error
    Disconnect() error
    Init(...Option) error
    Publish(string, *Message, ...PublishOption) error
    Subscribe(string, Handler, ...SubscribeOption) (Subscriber,
error)
    String() string
}
```

Broker

注册表

注册表是服务发现的Go-Micro界面。你可能会想。使用消息总线进行服务发现？这能工作吗？事实上它确实并且相当好。许多使用消息总线进行传输的人，会避免使用任何单独的发现机制。这是因为消息总线本身可以处理通过topic和channel的路由。定义为服务名称的topic可以用作路由key，在订阅该topic的服务的实例之间自动进行负载平衡。

Go-micro将服务发现和传输机制视为两个不同的问题。无论何时，客户端向另一个服务器发出请求（在封面下方），它会按名称在注册表中查找服务，选择节点的地址，然后通过传输器与其进行通信。

通常存储服务发现信息的最常用方式是通过 一个分布式的键值存储，如zookeeper，etcd或类似的东西。正如你可能已经意识到的那样，NATS不是一个分布式的键值存储，所以我们要做一些有点不同的事情.....

广播查询！

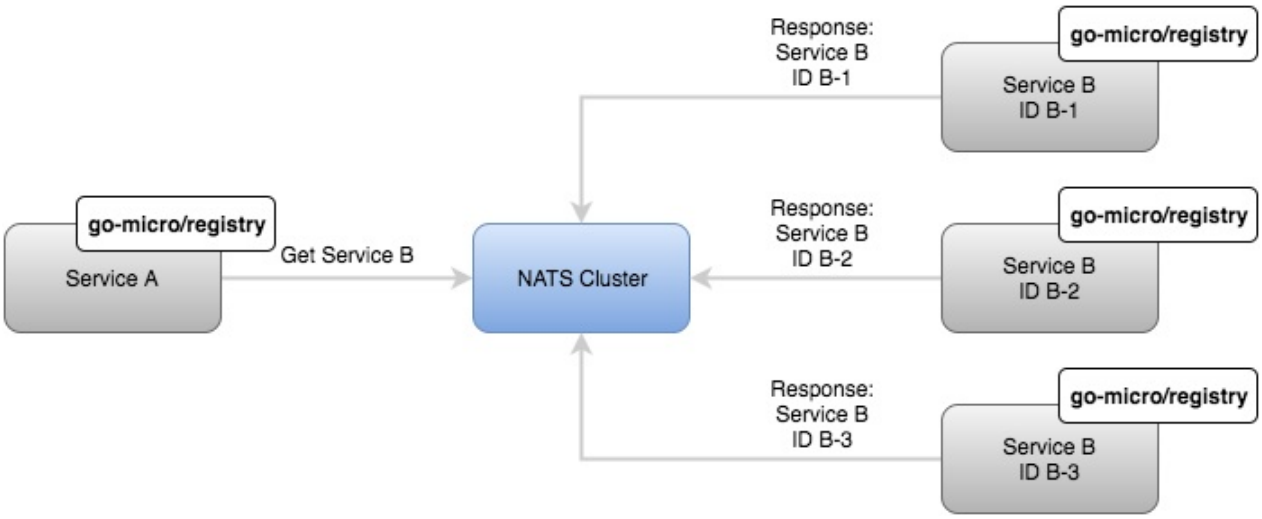
广播查询与您想象的一样。 服务监听我们认为用于广播查询的特定主题。 任何想要获得服务发现信息的人都会首先创建一个它所订阅的回复主题，然后用他们的回复地址在广播主题上进行查询。

因为我们实际上并不知道有多少服务实例正在运行，或者有多少响应将被返回，所以我们设定了一个我们愿意等待响应的时间的上限。 这是发现分散聚集的粗糙机制，但由于NAT的可扩展和高性能特性，它实际上运行得非常好。 它也间接提供了一个非常简单的过滤服务的方法，并且响应时间更长。 将来，我们将着眼于改进底层实现。

总结它的工作原理：

1. 创建回复topic并订阅
2. 用广播地址发送广播topic的查询
3. 听取回复并在时间限制后取消订阅

4. 合并响应和返回结果



使用注册表插件

导入注册表插件

```
import _ "github.com/micro/go-plugins/registry/nats"
```

从注册表标志开始

```
go run main.go --registry=nats --registry_address=127.0.0.1:4222
```

或者直接使用注册表

```
registry := nats.NewRegistry()
```

go-micro注册表接口：

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

Registry

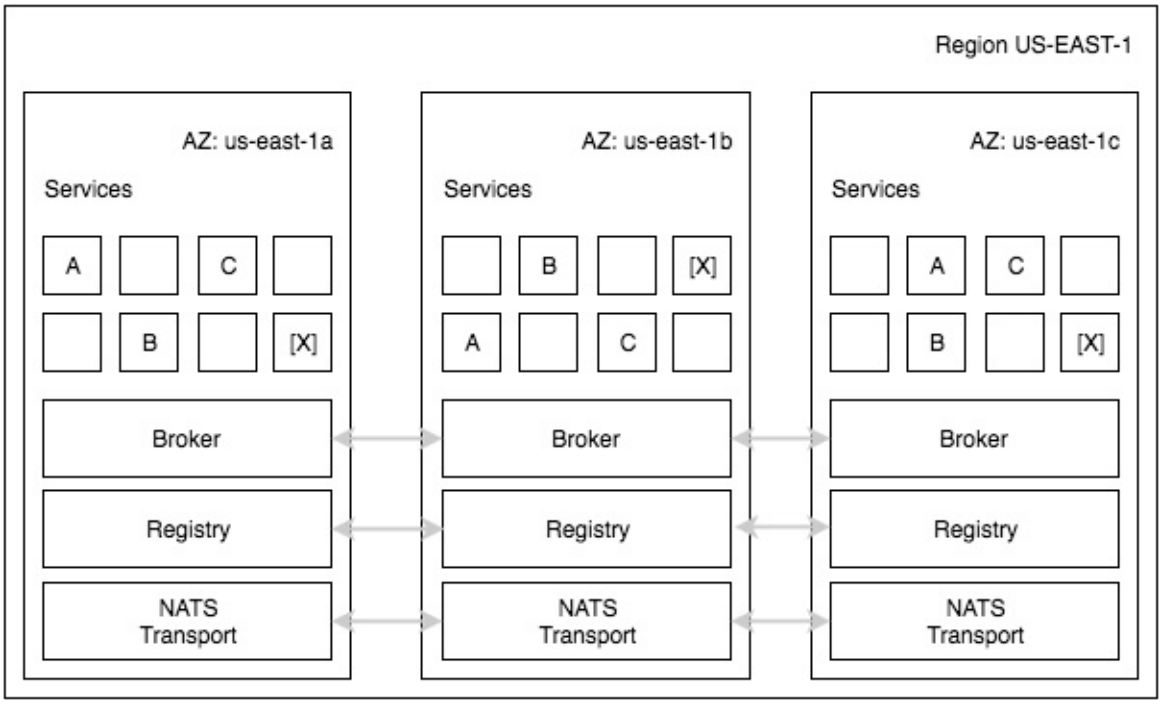
在NATS上伸缩Micro服务

在上面的例子中，我们只在本地主机上指定了一个NATS服务器，但我们推荐的实际使用方法是，设置一个NATS群集以获得高可用性和容错性。要了解有关NAT群集的更多信息，请查看此处的NATS文档。

Micro接受逗号分隔的地址列表作为上面提到的标志或可选地使用环境变量。如果您直接使用客户端库，它还允许一组可变主机作为注册表，传输和代理的初始化选项。

对于云原生应用的世界架构而言，我们过去的经验表明，每个AZ或每个地区的群集都是理想的。大多数云提供商在AZ之间具有相对较低（3-5ms）的延迟，这允许区域集群没有问题。在运行高可用性配置时，确保您的系统能够容忍AZ故障并且在更成熟的配置下，可以承受整个区域故障很重要。我们不建议跨地区集群。理想情况下，应该使用更高级别的工具来管理多群集和多区域系统。

Micro是一个令人难以置信的灵活的，运行时不感知的微服务系统。它可以在任何地方和配置下运行。它的世界是由服务注册机制引导。服务集群可以完全基于您提供访问权限的服务注册中心，AZ或区域池中进行本地化和命名空间。结合NATS集群，您可以构建高度可用的体系结构以满足您的需求。



概要

NATS是一个可扩展的高性能消息中间件系统，我们认为它非常适合微服务生态系统。它与Micro配合具有非常好竞争力，我们已经证明它可以用作注册表，传输或代理的插件。我们已经实施了所有三项，以突出NATS的灵活性。

Micro on NATS，是Micro强大的可插拔架构的一个典型例子。每个go-micro软件包都可以通过最小的更改来实现和替换。将来查看更多关于[X]的Micro示例。接下来最有可能是Kubernetes上的Micro。

希望这会激励你尝试使用NAT上的Micro，甚至为其他系统编写一些插件并回馈给社区。

在github.com/micro/go-plugins找到NATS插件的来源。