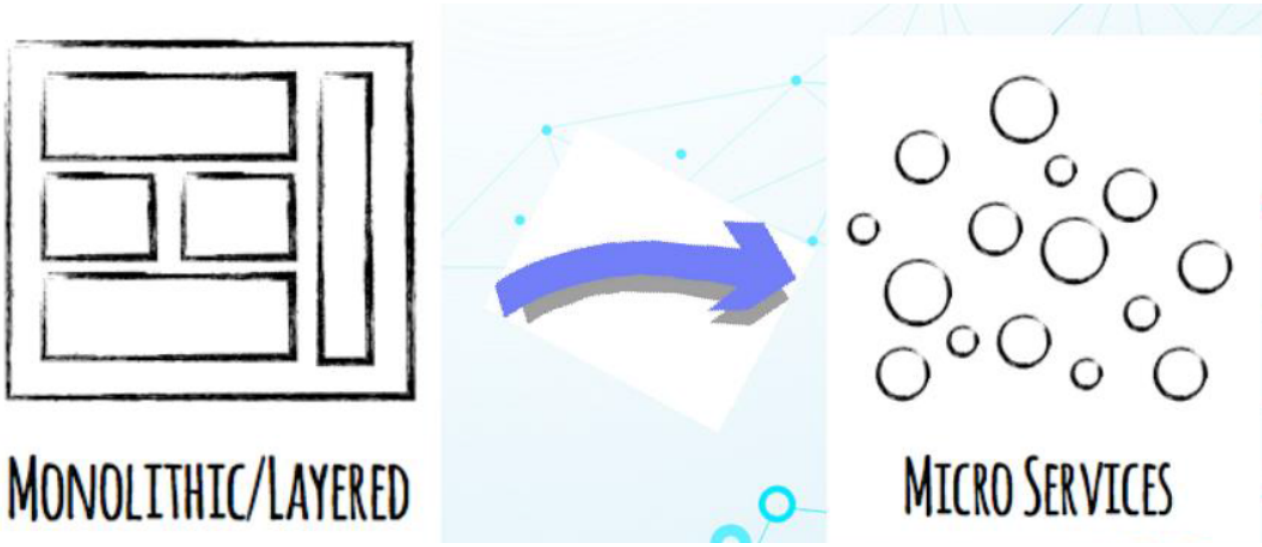
微服务



一、微服务（microservices）

近几年,微服这个词闯入了我们的实线范围。在百度与谷歌中随便搜一搜也有几千万条的结果。那么，什么是微服务呢？微服务的概念是怎么产生的呢？ 我们就来了解一下Go语言与微服务的千丝万缕与来龙去脉。

什么是微服务?

在介绍微服务时，首先得先理解什么是微服务，顾名思义，微服务得从两个方面去理解，什么是"微"、什么是"服务"？ **微（micro）** 狭义来讲就是体积小，著名的"2 pizza 团队"很好的诠释了这一解释（2 pizza 团队最早是亚马逊 CEO Bezos提出来的，意思是说单个服务的设计，所有参与人从设计、开发、测试、运维所有人加起来 只需要2个披萨就够了）。 **服务（service）** 一定要区别于系统，服务一个或者一组相对较小且独立的功能单元，是用户可以感知最小功能集。

那么广义上来讲，微服务是一种分布式系统解决方案，推动细粒度服务的使用，这些服务协同工作。

微服务这个概念的由来?

据说，早在2011年5月，在威尼斯附近的软件架构师讨论会上，就有人提出了微服务架构设计的概念，用它来描述与会者所见的一种通用的架构设计风格。时隔一年之后，在同一个讨论会上，大家决定将这种架构设计风格用微服务架构来表示。 起初，对微服务的概念，没有一个明确的定义，大家只能从各自的角度说出了微服务的理解和看法。 有人把微服务理解为一种细粒度SOA（service-oriented Architecture，面向服务架构），一种轻量级的组件化的小型SOA。 在2014年3月，詹姆斯·刘易斯（James Lewis）与马丁·福勒（Martin Fowler）所发表的一篇博客中，总结了微服务架构设计的一些共同特点，这应该是一个对微服务比较全面的描述。

```
1 原文链接    https://martinfowler.com/articles/microservices.html
```

这篇文章中认为：“简而言之，微服务架构风格是将单个应用程序作为一组小型服务开发的方法，每个服务程序都在自己的进程中运行，并与轻量级机制（通常是HTTP资源API）进行通信。这些服务是围绕业务功能构建的。可以通过全自动部署机器独立部署。这些服务器可以用不同的编程语言编写，使用不同的数据存储技术，并尽量不用集中式方式进行管理”

微服务与微服务框架

在这里我们可能混淆了一个点，那就是微服务和微服务架构，这应该是两个不同的概念，而我们平时说道的微服务可能就已经包含了这两个概念了，所以我们要把它们说清楚以免我们很纠结。微服务架构是一种设计方法，而微服务这是应该指使用这种方法而设计的一个应用。所以我们必要对微服务的概念做出一个比较明确的定义。

微服务框架是将复杂的系统使用组件化的方式进行拆分，并使用轻量级通讯方式进行整合的一种设计方法。

微服务是通过这种架构设计方法拆分出来的一个独立的组件化的小应用。

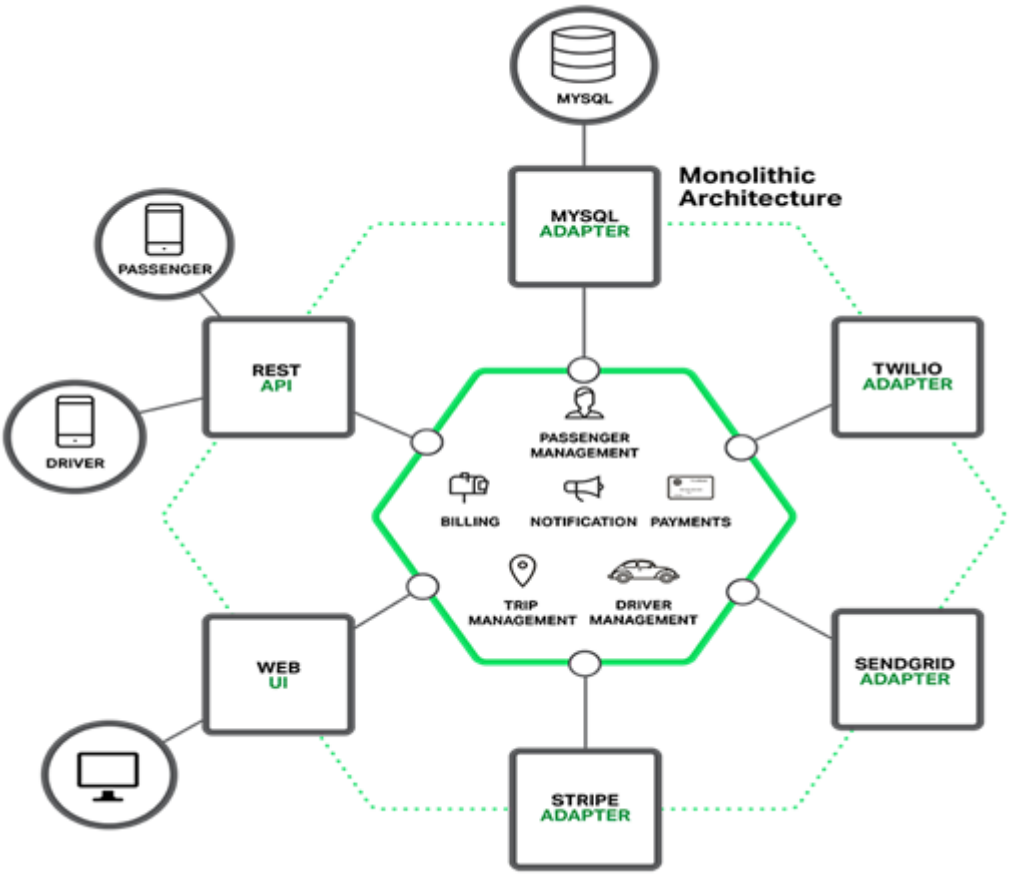
微服务架构定义的精髓，可以用一句话来描述，那就是“分而治之，合而用之”。

将复杂的系统进行拆分的方法，就是“分而治之”。分而治之，可以让复杂的事情变的简单，这很符合我们平时处理问题的方法。

使用轻量级通讯等方式进行整合的设计，就是“合而用之”的方法，合而用之可以让微小的力量变动强大。

微服务架构和整体式架构的区别?

开发单体式（整体式）应用的不足之处



三层架构（MVC）的具体内容如下：

表示层（view）： 用户使用应用程序时，看到的、听见的、输入的或者交互的部分。 **业务逻辑层（controller）**： 根据用户输入的信息，进行逻辑计算或者业务处理的部分。 **数据访问层（model）**： 关注有效地操作原始数据的部分，如将数据存储到存储介质（如数据库、文件系统）及从存储介质中读取数据等。 虽然现在程序被分成了三层，但只是逻辑上的分层，并不是物理上的分层。也就是说，对不同层的代码而言，经过编译、打包和部署后，所有的代码最终还是运行在同一个进程中。而这，就是所谓的单块架构。

单体架构在规模比较小的情况下工作情况良好，但是随着系统规模的扩大，它暴露出来的问题也越来越多，主要有以下几点：

复杂性逐渐变高

比如有的项目有几十万行代码，各个模块之间区别比较模糊，逻辑比较混乱，代码越多复杂性越高，越难解决遇到的问题。

技术债务逐渐上升

公司的人员流动是再正常不过的事情，有的员工在离职之前，疏于代码质量的自我管束，导致留下来很多坑，由于单体项目代码量庞大的惊人，留下的坑很难被发觉，这就给新来的员工带来很大的烦恼，人员流动越大所留下的坑越多，也就是所谓的技术债务越来越多。

维护成本大

当应用程序的功能越来越多、团队越来越大时，沟通成本、管理成本显著增加。当出现 bug 时，可能引起 bug 的原因组合越来越多，导致分析、定位和修复的成本增加；并且在对全局功能缺乏深度理解的情况下，容易在修复 bug 时引入新的 bug。

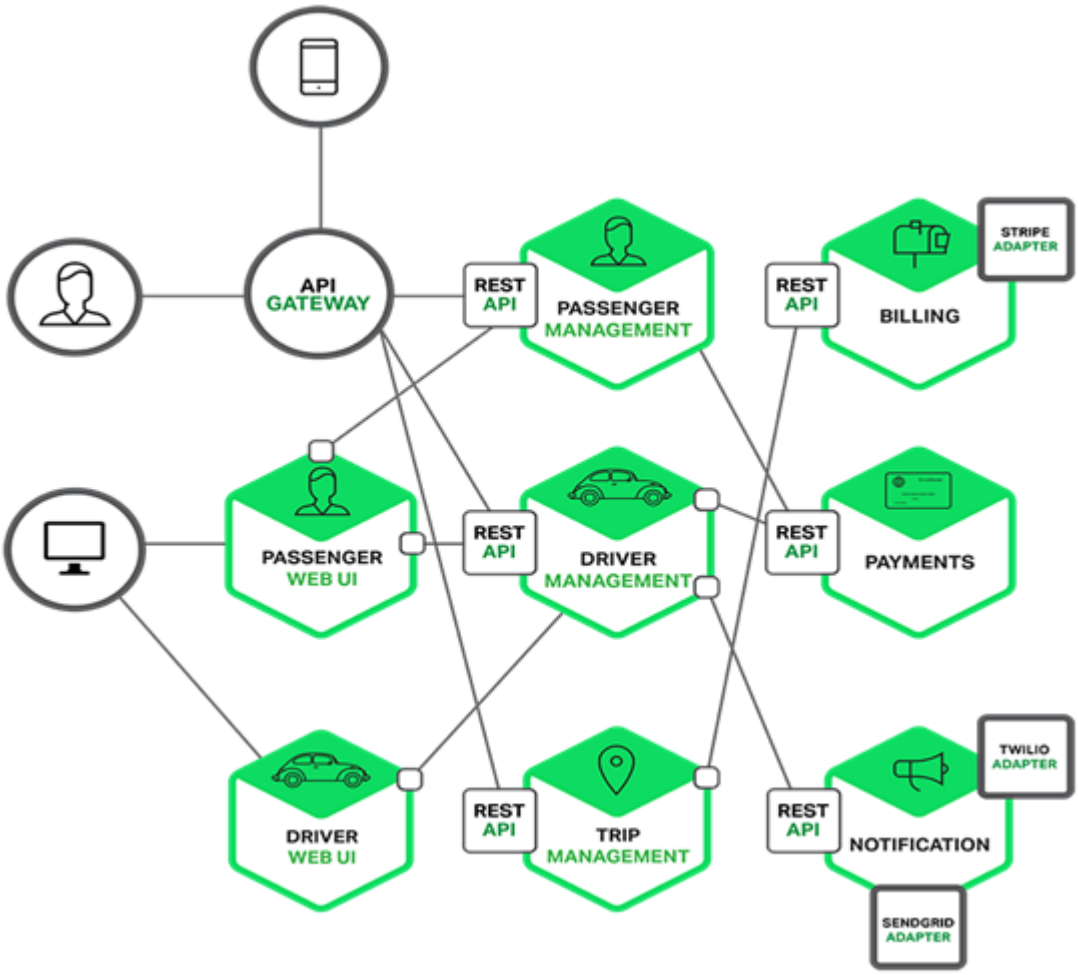
持续交付周期长

构建和部署时间会随着功能的增多而增加，任何细微的修改都会触发部署流水线。新人培养周期长：新成员了解背景、熟悉业务和配置环境的时间越来越长。 技术选型成本高 单块架构倾向于采用统一的技术平台或方案来解决所有问题，如果后续想引入新的技术或框架，成本和风险都很大。

可扩展性差

随着功能的增加，垂直扩展的成本将会越来越大；而对于水平扩展而言，因为所有代码都运行在同一个进程，没办法做到针对应用程序的部分功能做独立的扩展。

微服务架构的特性

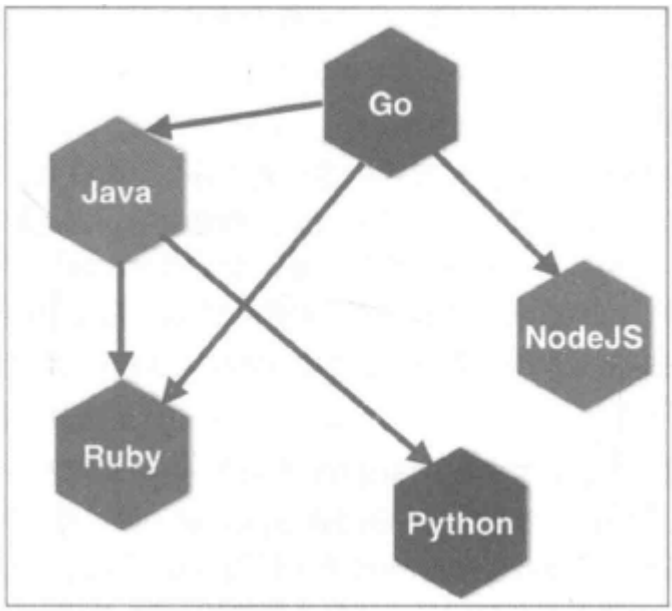


单一职责

微服务架构中的每个服务，都是具有业务逻辑的，符合高内聚、低耦合原则以及单一职责原则的单元，不同的服务通过“管道”的方式灵活组合，从而构建出庞大的系统。

轻量级通信

服务之间通过轻量级的通信机制实现互通互联，而所谓的轻量级，通常指语言无关、平台无关的交互方式。



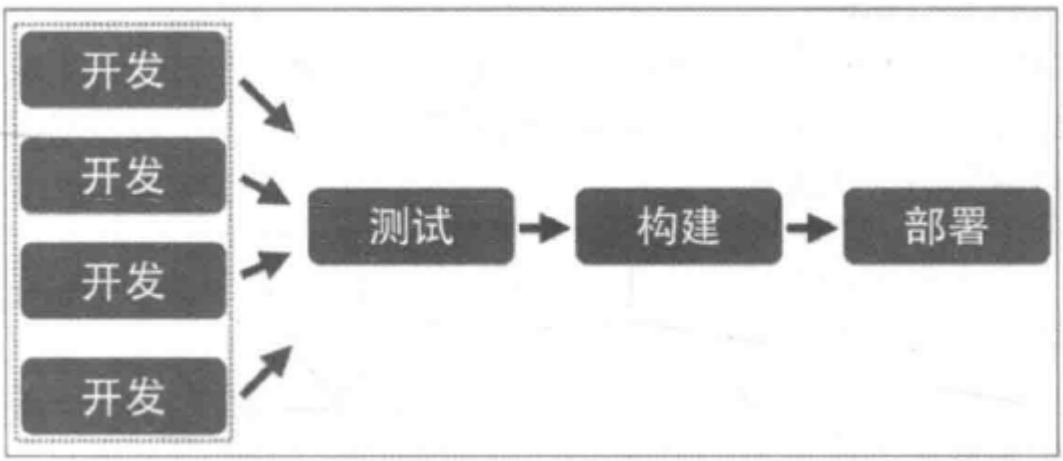
对于轻量级通信的格式而言，我们熟悉的 XML 和 JSON，它们是语言无关、平台无关的；对于通信的协议而言，通常基于 HTTP，能让服务间的通信变得标准化、无状态化。目前大家熟悉的 REST（Representational State Transfer）是实现服务间互相协作的轻量级通信机制之一。使用轻量级通信机制，可以让团队选择更适合的语言、工具或者平台来开发服务本身。

问：REST是什么和restful一样吗?

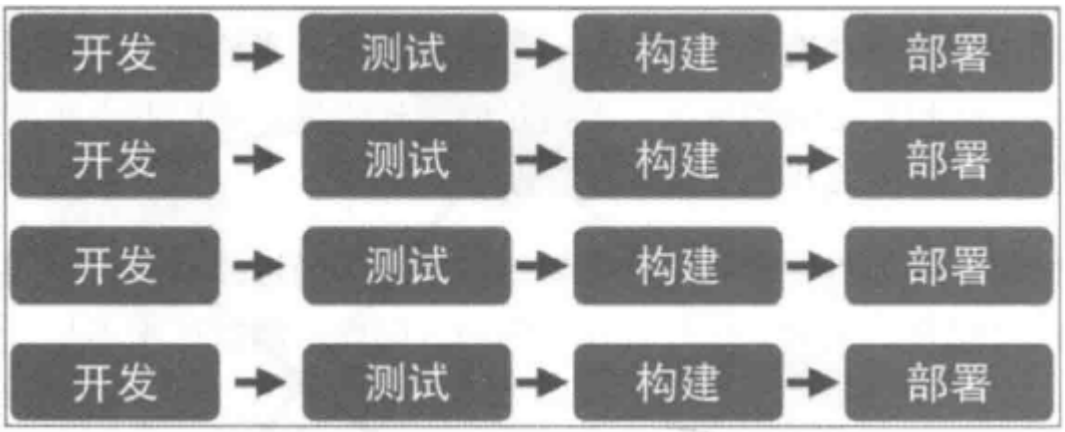
答：REST 指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。

独立性

每个服务在应用交付过程中，独立地开发、测试和部署。 在单块架构中所有功能都在同一个代码库，功能的开发不具有独立性；当不同小组完成多个功能后，需要经过集成和回归测试，测试过程也不具有独立性；当测试完成后，应用被构建成一个包，如果某个功能存在 bug，将导致整个部署失败或者回滚。



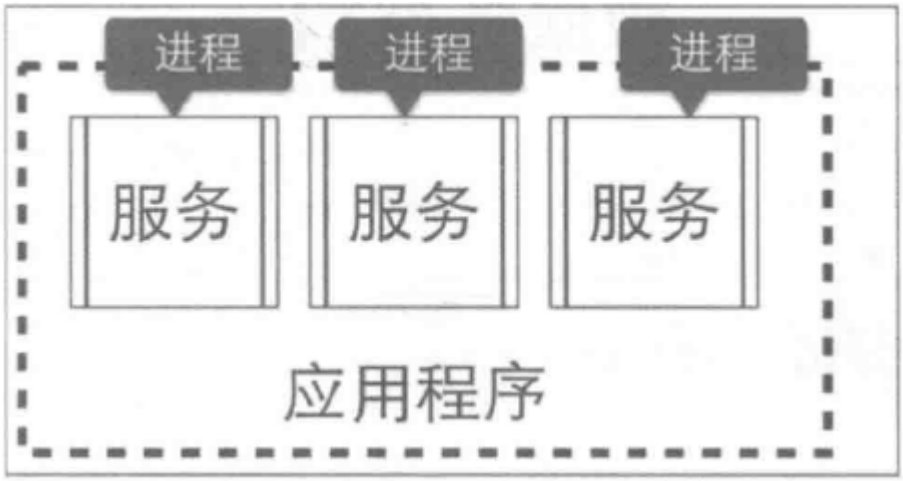
在微服务架构中，每个服务都是独立的业务单元，与其他服务高度解耦，只需要改变当前服务本身，就可以完成独立的开发、测试和部署。



进程隔离

单块架构中，整个系统运行在同一个进程中，当应用进行部署时，必须停掉当前正在运行的应用，部署完成后再重启进程，无法做到独立部署。 有时候我们会将重复的代码抽取出来封装成组件，在单块架构中，组件通常的形态叫做共享库（如 jar 包或者 DLL），但是当程序运行时，所有组件最终也会被加载到同一进程中运行。

在微服务架构中，应用程序由多个服务组成，每个服务都是高度自治的独立业务实体，可以运行在独立的进程中，不同的服务能非常容易地部署到不同的主机上。



微服务架构的缺点

运维要求较高

对于单体架构来讲，我们只需要维护好这一个项目就可以了，但是对于微服务架构来讲，由于项目是由多个微服务构成的，每个模块出现问题都会造成整个项目运行出现异常，想要知道是哪个模块造成的问题往往是不容易的，因为我们无法一步一步通过debug的方式来跟踪，这就对运维人员提出了很高的要求。

分布式的复杂性

对于单体架构来讲，我们可以不使用分布式，但是对于微服务架构来说，分布式几乎是必会用的技术，由于分布式本身的复杂性，导致微服务架构也变得复杂起来。

接口调整成本高

比如，用户微服务是要被订单微服务和电影微服务所调用的，一旦用户微服务的接口发生大的变动，那么所有依赖它的微服务都要做相应的调整，由于微服务可能非常多，那么调整接口所造成的成本将会明显提高。

重复劳动

对于单体架构来讲，如果某段业务被多个模块所共同使用，我们便可以抽象成一个工具类，被所有模块直接调用，但是微服务却无法这样做，因为这个微服务的工具类是不能被其它微服务所直接调用的，从而我们便不得不在每个微服务上都建这么一个工具类，从而导致代码的重复。

	传统单体架构	分布式微服务化架构
新功能开发	需要时间	容易开发和实线
部署	不经常而且容易部署	经常发布，部署复杂
隔离性	故障影响范围大	故障影响范围小
架构设计	初期设计选型难度大	设计逻辑难度大
系统性能	相应时间快，吞吐量小	相应时间慢，吞吐量大
系统运维	运维简单	运维复杂
新人上手	学习曲线大（应用逻辑）	学习曲线大（架构逻辑）
技术	技术单一而且封闭	技术多样而且开发
测试和差错	简单	复杂（每个服务都要进行单独测试，还需要集群测试）
系统扩展性	扩展性差	扩展性好
系统管理	重点在于开发成本	重点在于服务治理和调度

为什么使用微服务架构

开发简单

微服务架构将复杂系统进行拆分之后，让每个微服务应用都开放变得非常简单，没有太多的累赘。对于每一个开发者来说，这无疑是一种解脱，因为再也不用进行繁重的劳动了，每天都在一种轻松愉快的氛围中工作，其效率也会整备地提高

快速响应需求变化

一般的需求变化都来自于局部功能的改变，这种变化将落实到每个微服务上，二每个微服务的功能相对来说都非常简单，更改起来非常容易，所以微服务非常是和敏捷开发方法，能够快速的影响业务的需求变化。

随时随地更新

一方面，微服务的部署和更新并不会影响全局系统的正常运行；另一方面，使用多实例的部署方法，可以做到一个服务的重启和更新在不易察觉的情况下进行。所以每个服务任何时候都可以进行更新部署。

系统更加稳定可靠

微服务运行在一个高可用的分布式环境之中，有配套的监控和调度管理机制，并且还可以提供自由伸缩的管理，充分保障了系统的稳定可靠性

二、Protobuf

protobuf是google旗下的一款平台无关，语言无关，可扩展的序列化结构数据格式。所以很适合用做数据存储和作为不同应用，不同语言之间相互通信的数据交换格式，只要实现相同的协议格式即同一 proto文件被编译成不同的语言版本，加入到各自的工程中去。这样不同语言就可以解析其他语言通过 protobuf序列化的数据。目前官网提供了 C++,Python,JAVA,GO等语言的支持。google在2008年7月7号将其作为开源项目对外公布。

protoBuf简介

Google Protocol Buffer(简称 Protobuf)是一种轻便高效的结构化数据存储格式，平台无关、语言无关、可扩展，可用于通讯协议和数据存储等领域。

数据交互的格式比较

数据交互xml、json、protobuf格式比较

1、json: 一般的web项目中，最流行的主要还是json。因为浏览器对于json数据支持非常好，有很多内建的函数支持。

2、xml: 在webservice中应用最为广泛，但是相比于json，它的数据更加冗余，因为需要成对的闭合标签。json使用了键值对的方式，不仅压缩了一定的数据空间，同时也具有可读性。

3、protobuf:是后起之秀，是谷歌开源的一种数据格式，适合高性能，对响应速度有要求的数据传输场景。因为profobuf是二进制数据格式，需要编码和解码。数据本身不具有可读性。因此只能反序列化之后得到真正可读的数据。

相对于其它protobuf更具有优势

1：序列化后体积相比Json和XML很小，适合网络传输

2：支持跨平台多语言

3：消息格式升级和兼容性还不错

4：序列化反序列化速度很快，快于Json的处理速速

protoBuf的优点

Protobuf 有如 XML，不过它更小、更快、也更简单。你可以定义自己的数据结构，然后使用代码生成器生成的代码来读写这个数据结构。你甚至可以在无需重新部署程序的情况下更新数据结构。只需使用 Protobuf 对数据结构进行一次描述，即可利用各种不同语言或从各种不同数据流中对你的结构化数据轻松读写。 它有一个非常棒的特性，即“向后”兼容性好，人们不必破坏已部署的、依靠“老”数据格式的程序就可以对数据结构进行升级。 Protobuf 语义更清晰，无需类似 XML 解析器的东西（因为 Protobuf 编译器会将 .proto 文件编译生成对应的数据访问类以对 Protobuf 数据进行序列化、反序列化操作）。使用 Protobuf 无需学习复杂的文档对象模型，Protobuf 的编程模式比较友好，简单易学，同时它拥有良好的文档和示例，对于喜欢简单事物的人们而言，Protobuf 比其他的技术更加有吸引力。

ProtoBuf 的不足

Protobuf 与 XML 相比也有不足之处。它功能简单，无法用来表示复杂的概念。 XML 已经成为多种行业标准的编写工具，Protobuf 只是 Google 公司内部使用的工具，在通用性上还差很多。 由于文本并不适合用来描述数据结构，所以 Protobuf 也不适合用来对基于文本的标记文档（如 HTML）建模。另外，由于 XML 具有某种程度上的自解释性，它可以被人直接读取编辑，在这一点上 Protobuf 不行，它以二进制的方式存储，除非你有 .proto 定义，否则你没法直接读出 Protobuf 的任何内容。

Protobuf安装

安装protoBuf

```
1
2  #下载 protoBuf:
3  $ git clone https://github.com/protocolbuffers/protobuf.git
4  #或者直接将压缩包拖入后解压
5  unzip protobuf.zip
6
7
8  #安装依赖库
9  $ sudo apt-get install autoconf  automake  libtool  curl  make  g++  unzip libffi-
   dev -y
10 #安装
11 $ cd protobuf/
12 $ ./autogen.sh
13 $ ./configure
14 $ make
15 $ sudo make install
16 $ sudo ldconfig    # 刷新共享库 很重要的一步啊
17 #安装的时候会比较卡
18 #成功后需要使用命令测试
19 $ protoc -h
```

```
20
```

获取 proto包

```
#Go语言的proto API接口
$ go get  -v -u github.com/golang/protobuf/proto

```

安装protoc-gen-go插件

它是一个 go程序，编译它之后将可执行文件复制到\bin目录。

```
#安装
$ go get -v -u github.com/golang/protobuf/protoc-gen-go
#编译
$ cd $GOPATH/src/github.com/golang/protobuf/protoc-gen-go/
$ go build
#将生成的 protoc-gen-go可执行文件，放在/bin目录下
$ sudo cp protoc-gen-go /bin/
```

protobuf的语法

要想使用 protobuf必须得先定义 proto文件。所以得先熟悉 protobuf的消息定义的相关语法。

定义一个消息类型

```
syntax = "proto3";

message PandaRequest {
  string name = 1;
  int32 shengao = 2;
  repeated int32 tizhong = 3;
}
```

PandaRequest消息格式有3个字段，在消息中承载的数据分别对应于每一个字段。其中每个字段都有一个名字和一种类型。 文件的第一行指定了你正在使用proto3语法：如果你没有指定这个，编译器会使用proto2。这个指定语法行必须是文件的非空非注释的第一个行。 在上面的例子中，所有字段都是标量类型：两个整型（shengao和tizhong），一个string类型（name）。 **Repeated** 关键字表示重复的那么在go语言中用切片进行代表 正如上述文件格式，在消息定义中，每个字段都有唯一的一个标识符。

添加更多消息类型

在一个.proto文件中可以定义多个消息类型。在定义多个相关的消息的时候，这一点特别有用——例如，如果想定义与SearchResponse消息类型对应的回复消息格式的话，你可以将它添加到相同的.proto文件中

```
syntax = "proto3";

message PandaRequest {
  string name = 1;
  int32 shengao = 2;
  int32 tizhong = 3;
}

message PandaResponse {
 ...
}
```

添加注释

向.proto文件添加注释，可以使用C/C++/java/Go风格的双斜杠（//） 语法格式，如：

```
syntax = "proto3";
message PandaRequest {
  string name = 1;          //姓名
  int32 shengao = 2;        //身高
  int32 tizhong = 3;        //体重
}
message PandaResponse {
 ...
}
```

从.proto文件生成了什么?

当用protocol buffer编译器来运行.proto文件时，编译器将生成所选择语言的代码，这些代码可以操作在.proto文件中定义的消息类型，包括获取、设置字段值，将消息序列化到一个输出流中，以及从一个输入流中解析消息。

对C++来说，编译器会为每个.proto文件生成一个.h文件和一个.cc文件，.proto文件中的每一个消息有一个对应的类。 对Python来说，有点不太一样——Python编译器为.proto文件中的每个消息类型生成一个含有静态描述符的模块，，该模块与一个元类（metaclass）在运行时（runtime）被用来创建所需的Python数据访问类。 对go来说，编译器会为每个消息类型生成了一个.pd.go文件。

标准数据类型

一个标量消息字段可以含有一个如下的类型——该表格展示了定义于.proto文件中的类型，以及与之对应的、在自动生成的访问类中定义的类型：

.proto Type	Notes	C++ Type	Python Type	Go Type
double		double	float	float64
float		float	float	float32
int32	使用变长编码，对于负值的效率很低，如果你的域有可能有负值，请使用sint64替代	int32	int	int32
uint32	使用变长编码	uint32	int/long	uint32
uint64	使用变长编码	uint64	int/long	uint64
sint32	使用变长编码，这些编码在负值时比int32高效的多	int32	int	int32
sint64	使用变长编码，有符号的整型值。编码时比通常的int64高效。	int64	int/long	int64
fixed32	总是4个字节，如果数值总是比总是比228大的话，这个类型会比uint32高效。	uint32	int	uint32
fixed64	总是8个字节，如果数值总是比总是比256大的话，这个类型会比uint64高效。	uint64	int/long	uint64
sfixed32	总是4个字节	int32	int	int32
sfixed32	总是4个字节	int32	int	int32
sfixed64	总是8个字节	int64	int/long	int64
bool		bool	bool	bool
string	一个字符串必须是UTF-8编码或者7-bit ASCII编码的文本。	string	str/unicode	string
bytes	可能包含任意顺序的字节数据。	string	str	[]byte

默认值

当一个消息被解析的时候，如果被编码的信息不包含一个特定的元素，被解析的对象锁对应的域被设置位一个默认值，对于不同类型指定如下： 对于strings，默认是一个空string 对于bytes，默认是一个空的bytes 对于bools，默认是false 对于数值类型，默认是0

使用其他消息类型

你可以将其他消息类型用作字段类型。例如，假设在每一个PersonInfo消息中包含Person消息，此时可以在相同的.proto文件中定义一个Result消息类型，然后在PersonInfo消息中指定一个Person类型的字段

```
message PersonInfo {
  repeated Person info = 1;
}
message Person {
  string name = 1;
  int32 shengao = 2;
  repeated int32 tizhong = 3;
}
```

使用proto2消息类型

在你的proto3消息中导入proto2的消息类型也是可以的，反之亦然，然后proto2枚举不可以直接在proto3的标识符中使用（如果仅仅在proto2消息中使用是可以的）。

嵌套类型

你可以在其他消息类型中定义、使用消息类型，在下面的例子中，Person消息就定义在PersonInfo消息内，如：

```
message PersonInfo {
  message Person {
    string name = 1;
    int32 shengao = 2;
    repeated int32 tizhong = 3;
  }
  repeated Person info = 1;
}
```

如果你想在它的父消息类型的外部重用这个消息类型，你需要以PersonInfo.Person的形式使用它，如：

```
message PersonMessage {
  PersonInfo.Person info = 1;
}
```

当然，你也可以将消息嵌套任意多层，如：

```
message Grandpa {                  // Level 0
  message Father {  // Level 1
    message son {   // Level 2
      string name = 1;
      int32  age = 2;
    }
  }
  message Uncle {  // Level 1
    message Son {   // Level 2
      string name = 1;
      int32  age = 2;
    }
  }
}
```

定义服务(Service)

如果想要将消息类型用在RPC(远程方法调用)系统中，可以在.proto文件中定义一个RPC服务接口，protocol buffer编译器将会根据所选择的不同语言生成服务接口代码及存根。如，想要定义一个RPC服务并具有一个方法，该方法能够接收 SearchRequest并返回一个SearchResponse，此时可以在.proto文件中进行如下定义：

```
service SearchService {
    //rpc 服务的函数名 （传入参数）返回（返回参数）
  rpc Search (SearchRequest) returns (SearchResponse);
}
```

最直观的使用protocol buffer的RPC系统是gRPC一个由谷歌开发的语言和平台中的开源的RPC系统，gRPC在使用protocl buffer时非常有效，如果使用特殊的protocol buffer插件可以直接为您从.proto文件中产生相关的RPC代码。

如果你不想使用gRPC，也可以使用protocol buffer用于自己的RPC实现，你可以从proto2语言指南中找到更多信息

生成访问类（了解）

可以通过定义好的.proto文件来生成Java,Python,C++, Ruby, JavaNano, Objective-C,或者C# 代码，需要基于.proto文件运行protocol buffer编译器protoc。如果你没有安装编译器，下载安装包并遵照README安装。对于Go,你还需要安装一个特殊的代码生成器插件。你可以通过GitHub上的protobuf库找到安装过程

通过如下方式调用protocol编译器：

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR  --python_out=DST_DIR --
go_out=DST_DIR   path/to/file.proto
```

IMPORT_PATH声明了一个.proto文件所在的解析import具体目录。如果忽略该值，则使用当前目录。如果有多个目录则可以多次调用--proto_path，它们将会顺序的被访问并执行导入。-I=IMPORT_PATH是--proto_path的简化形式。

当然也可以提供一个或多个输出路径： --cpp_out 在目标目录DST_DIR中产生C++代码，可以在C++代码生成参考中查看更多。 --python_out 在目标目录 DST_DIR 中产生Python代码，可以在Python代码生成参考中查看更多。 **--go_out 在目标目录 DST_DIR 中产生Go代码，可以在GO代码生成参考中查看更多。** 作为一个方便的拓展，如果DST_DIR以.zip或者.jar结尾，编译器会将输出写到一个ZIP格式文件或者符合JAR标准的.jar文件中。注意如果输出已经存在则会被覆盖，编译器还没有智能到可以追加文件。 - 你必须提议一个或多个.proto文件作为输入，多个.proto文件可以只指定一次。虽然文件路径是相对于当前目录的，每个文件必须位于其IMPORT_PATH下，以便每个文件可以确定其规范的名称。

测试

protobuf的使用方法是将数据结构写入到 .proto文件中，使用 protoc编译器编译(间接使用了插件）得到一个新的go包，里面包含 go中可以使用的数据结构和一些辅助方法。

编写 test.proto文件

1.$GOPATH/src/创建 myproto文件夹

```
$ cd $GOPATH/src/
$ make myproto
```

2.myproto文件夹中创建 test.proto文件 (protobuf协议文件

```
$ vim test.proto
```

文件内容

```
syntax = "proto3";
package myproto;

message Test {
  string name  = 1;
  int32 stature = 2 ;
  repeated int64 weight  = 3;
  string motto = 4;
}
```

3.编译 :执行

```
$ protoc --go_out=./ *.proto
```

生成 test.pb.go文件 4.使用 protobuf做数据格式转换

```
package main

import (
    "fmt"
    "github.com/golang/protobuf/proto"
    "myproto"
)

func main() {
    test := &myproto.Test{
        Name :   "panda",
        Stature : 180,
        Weight : []int64{120,125,198,180,150,180},
        Motto : "天行健，地势坤",
    }
    //将Struct test 转换成 protobuf
    data,err:= proto.Marshal(test)
    if err!=nil{
    fmt.Println("转码失败",err)
    }
    //得到一个新的Test结构体 newTest
    newtest:= &myproto.Test{}
```

```
    //将data转换为test结构体
    err = proto.Unmarshal(data,newtest)
    if err!=nil {
    fmt.Println("转码失败",err)
    }
    fmt.Println(newtest.String())
    //得到name字段
    fmt.Println("newtest->name",newtest.GetName())
    fmt.Println("newtest->Stature",newtest.GetStature())
    fmt.Println("newtest->Weight",newtest.GetWeight())
    fmt.Println("newtest->Motto",newtest.GetMotto())
}
```

三、GRPC

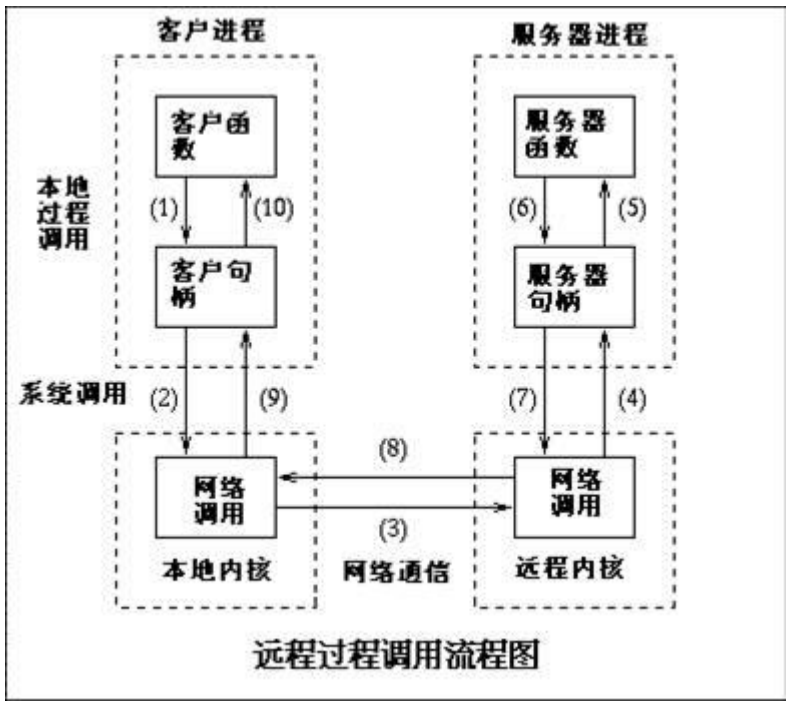
gRPC 是一个高性能、开源和通用的 RPC 框架，面向移动和 HTTP/2 设计。 gRPC基于 HTTP/2标准设计，带来诸如双向流、流控、头部压缩、单 TCP连接上的多复用请求等特。这些特性使得其在移动设备上表现更好，更省电和节省空间占用。

RPC

RPC（Remote Procedure Call Protocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

简单来说，就是跟远程访问或者web请求差不多，都是一个client向远端服务器请求服务返回结果，但是web请求使用的网络协议是http高层协议，而rpc所使用的协议多为TCP，是网络层协议，减少了信息的包装，加快了处理速度。

golang本身有rpc包，可以方便的使用，来构建自己的rpc服务，下边是一个简单是实例，可以加深我们的理解



1.调用客户端句柄；执行传送参数 2.调用本地系统内核发送网络消息 3.消息传送到远程主机 4.服务器句柄得到消息并取得参数 5.执行远程过程 6.执行的过程将结果返回服务器句柄 7.服务器句柄返回结果，调用远程系统内核 8.消息传回本地主机 9.客户句柄由内核接收消息 10.客户接收句柄返回的数据

服务端

```
package main

import (
    "net/http"
    "net/rpc"
    "net"
    "github.com/astaxie/beego"

    "io"
)

//- 方法是导出的
//- 方法有两个参数，都是导出类型或内建类型
//- 方法的第二个参数是指针
//- 方法只有一个error接口类型的返回值
//
//func (t *T) MethodName(argType T1, replyType *T2) error

type Panda int;

func (this *Panda)Getinfo(argType int, replyType *int) error {

    beego.Info(argType)
    *replyType  =1 +argType

```

```
    return nil
}

func main() {

    //注册1个页面请求
    http.HandleFunc("/panda",pandatext)

    //new 一个对象
    pd :=new(Panda)
    //注册服务
    //Register在默认服务中注册并公布 接收服务 pd对象 的方法
    rpc.Register(pd)

    rpc.HandleHTTP()
    //建立网络监听
    ln , err :=net.Listen("tcp","127.0.0.1:10086")
    if err != nil{
        beego.Info("网络连接失败")
    }

    beego.Info("正在监听10086")
    //service接受侦听器l上传入的HTTP连接,
    http.Serve(ln,nil)

}
//用来现实网页的web函数
func pandatext(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w,"panda")
}
```

客户端

```
package main

import (
    "net/rpc"
    "github.com/astaxie/beego"
)

func main() {
    //rpc的与服务端建立网络连接
    cli,err := rpc.DialHTTP("tcp","127.0.0.1:10086")
    if err !=nil {
        beego.Info("网络连接失败")
    }

    var val int
    //远程调用函数（被调用的方法，传入的参数 ，返回的参数）
    err =cli.Call("Panda.Getinfo",123,&val)
    if err!=nil{
```

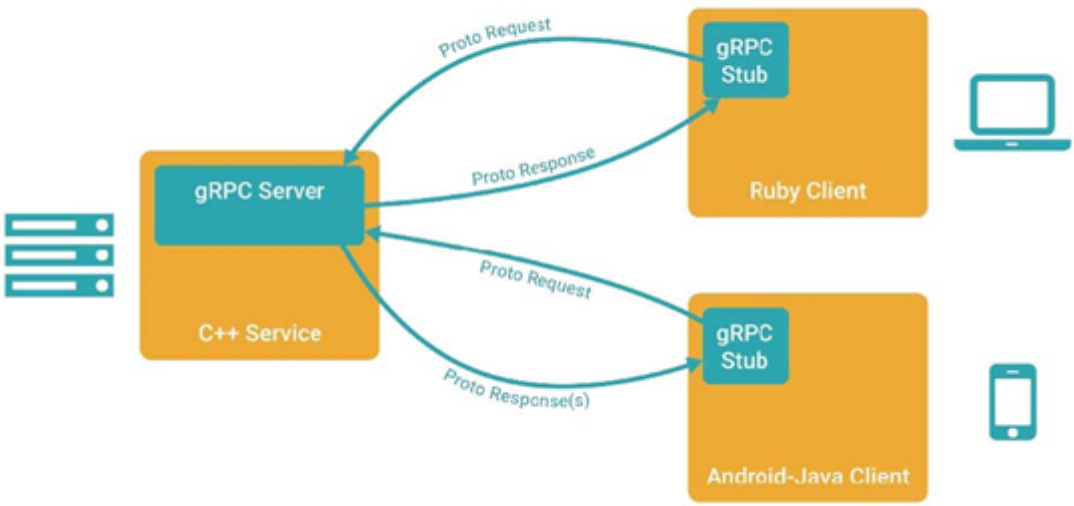
```
19          beego.Info("打call失败")
20      }
21      beego.Info("返回结果",val)
22
23  }
24
```

GRPC是什么?

在 gRPC里客户端应用可以像调用本地对象一样直接调用另一台不同的机器上服务端应用的方法，使得您能够更容易地创建分布式应用和服务。与许多 RPC系统类似， gRPC也是基于以下理念：定义一个服务，指定其能够被远程调用的方法（包含参数和返回类型）。在服务端实现这个接口，并运行一个 gRPC服务器来处理客户端调用。在客户端拥有一个存根能够像服务端一样的方法。 gRPC客户端和服务端可以在多种环境中运行和交互 -从 google内部的服务器到你自己的笔记本，并且可以用任何 gRPC支持的语言 来编写。所以，你可以很容易地用 Java创建一个 gRPC服务端，用 Go、 Python、Ruby来创建客户端。此外， Google最新 API将有 gRPC版本的接口，使你很容易地将 Google的功能集成到你的应用里。

GRPC使用 protocol buffers

gRPC默认使用protoBuf，这是 Google开源的一套成熟的结构数据序列化机制（当然也可以使用其他数据格式如 JSON）。正如你将在下方例子里所看到的，你用 proto files创建 gRPC服务，用 protoBuf消息类型来定义方法参数和返回类型。你可以在 Protocol Buffers文档找到更多关于 protoBuf的资料。 虽然你可以使用 proto2 (当前默认的 protocol buffers版本)，我们通常建议你在 gRPC里使用 proto3，因为这样你可以使用 gRPC支持全部范围的的语言，并且能避免 proto2客户端与 proto3服务端交互时出现的兼容性问题，反之亦然。



你好 gRPC

现在你已经对 gRPC有所了解，了解其工作机制最简单的方法是看一个简单的例子。 Hello World将带领你创建一个简单的客户端 —— 服务端应用，向你展示： 通过一个protoBuf模式，定义一个简单的带有 Hello World方法的RPC服务。 用你最喜欢的语言 (如果可用的话)来创建一个实现了这个接口的服务端。 用你最喜欢的 (或者其他你愿意的)语言来访问你的服务端。

这个例子完整的代码在我们 GitHub源码库的 examples目录下。我们使用 Git版本系统来进行源码管理，但是除了如何安装和运行一些 Git命令外，你没必要知道其他关于 Git的任何事情。需要注意的是，并不是所有 gRPC支持的语言都可以编写我们例子的服务端代码，比如 PHP和 Objective-C仅支持创建客户端。比起针对于特定语言的复杂教程，这更像是一个介绍性的例子。你可以在本站找到更有深度的教程， gRPC支持的语言的参考文档很快就会全部开放。

环境搭建

```
#将x.zip 解压到 $GOPATH/src/golang.org/x 目录下
$ unzip x.zip -d /GOPATH/src/golang.org/x
#-d 是指定解压目录地址
#/home/itcast/go/src/golang.org
#文件名为x

#将google.golang.org.zip 解压到 $GOPATH/src/google.golang.org 目录下

```

启动服务端

```
$ cd $GOPATH/src/google.golang.org/grpc/examples/helloworld/greeter_server
$ go run main.go
```

启动客户端

```
$ cd $GOPATH/src/google.golang.org/grpc/examples/helloworld/greeter_client
$ go run main.go
```

客户端代码介绍

```
package main

import (
    "log"
```

```
    "os"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    pb "google.golang.org/grpc/examples/helloworld/helloworld"
    //这是引用编译好的protobuf
)

const (
    address     = "localhost:50051"
    defaultName = "world"
)

func main() {
    // 建立到服务器的连接。
    conn, err := grpc.Dial(address, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    //延迟关闭连接
    defer conn.Close()
    //调用protobuf的函数创建客户端连接句柄
    c := pb.NewGreeterClient(conn)

    // 联系服务器并打印它的响应。
    name := defaultName
    if len(os.Args) > 1 {
        name = os.Args[1]
    }
    //调用protobuf的sayhello函数
    r, err := c.SayHello(context.Background(), &pb.HelloRequest{Name: name})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    //打印结果
    log.Printf("Greeting: %s", r.Message)
}
```

服务端代码介绍

```
package main

import (
    "log"
    "net"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    pb "google.golang.org/grpc/examples/helloworld/helloworld"
    "google.golang.org/grpc/reflection"
)

```

```go
const (
    port = ":50051"
)

// 服务器用于实现helloworld.GreeterServer。
type server struct{}

// SayHello实现helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest)
(*pb.HelloReply, error) {
    return &pb.HelloReply{Message: "Hello " + in.Name}, nil
}

func main() {
    //监听
    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    //new服务对象
    s := grpc.NewServer()
    //注册服务
    pb.RegisterGreeterServer(s, &server{})
    // 在gRPC服务器上注册反射服务。
    reflection.Register(s)
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

go语言实现GRPC远程调用

protobuf协议定义

创建一个 protobuf package,如： my_rpc_proto;

在$GOPATH/src/下创建 /my_grpc_proto/文件夹

里面创建 protobuf协议文件 helloServer.proto

```
#到工作目录
$ CD $GOPATH/src/
#创建目录
$ mkdir grpc/myproto
#进入目录
$ cd  grpc/myproto
#创建proto文件
$ vim helloServer.proto
```

文件内容

```
syntax = "proto3";

package my_grpc_proto;

service HelloServer{
//    创建第一个接口
    rpc SayHello(HelloRequest)returns(HelloReplay){}
//    创建第二个接口
    rpc GetHelloMsg(HelloRequest)returns(HelloMessage){}
}

message HelloRequest{
    string name = 1 ;
}
message HelloReplay{
    string message = 1;
}

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

在当前文件下，编译 helloServer.proto文件

```
$ protoc --go_out=./ *.proto #不加grpc插件
$ protoc --go_out=plugins=grpc:./ *.proto #添加grpc插件
#对比发现内容增加
#得到 helloServer.pb.go文件
```

gRPC-Server编写

```
package main

import (
    "net"
    "fmt"
    "google.golang.org/grpc"
    pt "demo/grpc/proto"
```

```
    "context"
)

const (
    post  = "127.0.0.1:18881"
)
//对象要和proto内定义的服务一样
type server struct{}

//实现RPC SayHello 接口
func(this *server)SayHello(ctx context.Context,in *pt.HelloRequest)(*pt.HelloReplay
, error){
    return  &pt.HelloReplay{Message:"hello"+in.Name},nil
}
//实现RPC GetHelloMsg 接口
func (this *server) GetHelloMsg(ctx context.Context, in *pt.HelloRequest)
(*pt.HelloMessage, error) {
    return &pt.HelloMessage{Msg: "this is from server HAHA!"}, nil
}

func main() {
    //监听网络
    ln ,err :=net.Listen("tcp",post)
    if err!=nil {
        fmt.Println("网络异常",err)
    }

//  创建一个grpc的句柄
    srv:= grpc.NewServer()
    //将server结构体注册到 grpc服务中
    pt.RegisterHelloServerServer(srv,&server{})

    //监听grpc服务
    err= srv.Serve(ln)
    if err!=nil {
        fmt.Println("网络启动异常",err)
    }

}
```

gRPC-Client编写

```
package main

import (
    "google.golang.org/grpc"
    pt "demo/grpc/proto"
    "fmt"
    "context"
```

```
)

const (
    post  = "127.0.0.1:18881"
)

func main() {

//  客户端连接服务器
    conn,err:=grpc.Dial(post,grpc.WithInsecure())
    if err!=nil {
        fmt.Println("连接服务器失败",err)
    }

    defer conn.Close()

    //获得grpc句柄
    c:=pt.NewHelloServerClient(conn)

    //  远程调用 SayHello接口

    //远程调用 SayHello接口
    r1, err := c.SayHello(context.Background(), &pt.HelloRequest{Name: "panda"})
    if err != nil {
        fmt.Println("cloud not get Hello server ..", err)
        return
    }
    fmt.Println("HelloServer resp: ", r1.Message)

    //远程调用 GetHelloMsg接口
    r2, err := c.GetHelloMsg(context.Background(), &pt.HelloRequest{Name: "panda"})
    if err != nil {
        fmt.Println("cloud not get hello msg ..", err)
        return
    }
    fmt.Println("HelloServer resp: ", r2.Msg)

}
```

运行

```
#先运行 server, 后运行 client

#得到以下输出结果
HelloServer resp:  hellopanda
HelloServer resp:  this is from server HAHA!

#如果反之则会报错
```

四、Consul

为什么要学习consul服务发现?

因为一套微服务架构中有很多个服务需要管理，也就是说会有很多对grpc。

如果一一对应的进行管理会很繁琐所以我们需要有一个管理发现的机制

Consul的介绍

Consul是什么

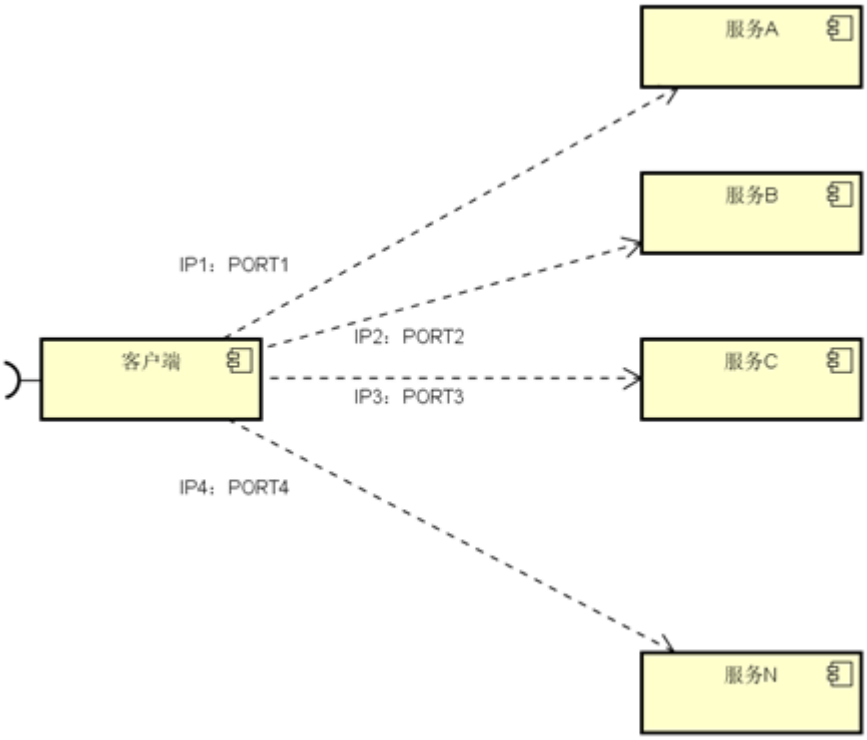
Consul是HashiCorp公司推出的开源工具，用于实现分布式系统的服务发现与配置。 Consul是分布式的、高可用的、可横向扩展的。它具备以下特性：service discovery：consul通过DNS或者HTTP接口使服务注册和服务发现变的很容易，一些外部服务，例如saas提供的也可以一样注册。 health checking：健康检测使consul可以快速的告警在集群中的操作。和服务发现的集成，可以防止服务转发到故障的服务上面。 key/value storage：一个用来存储动态配置的系统。提供简单的HTTP接口，可以在任何地方操作。 multi-datacenter：无需复杂的配置，即可支持任意数量的区域。

下面的例子有助于我们理解服务发现的形式：

例如邮递员去某公司一栋大楼投递快件，向门卫询问员工甲在哪一个房间，门卫拿起桌上的通讯录查询，告知邮递员员工甲在具体什么位置。假如公司来了一个员工乙，他想让邮递员送过来，就要先让门卫知道自己在哪一个房间，需要去门卫那边登记，员工乙登记后，当邮递员向门卫询问时，门卫就可以告诉邮递员员工乙的具体位置。门卫知道员工乙的具体位置的过程就是服务发现，员工乙的位置信息可以被看作服务信息，门卫的通讯录就是上文中提到的数据交换格式，此例中员工乙就是上文的已方，门卫就是服务发现的提供者。

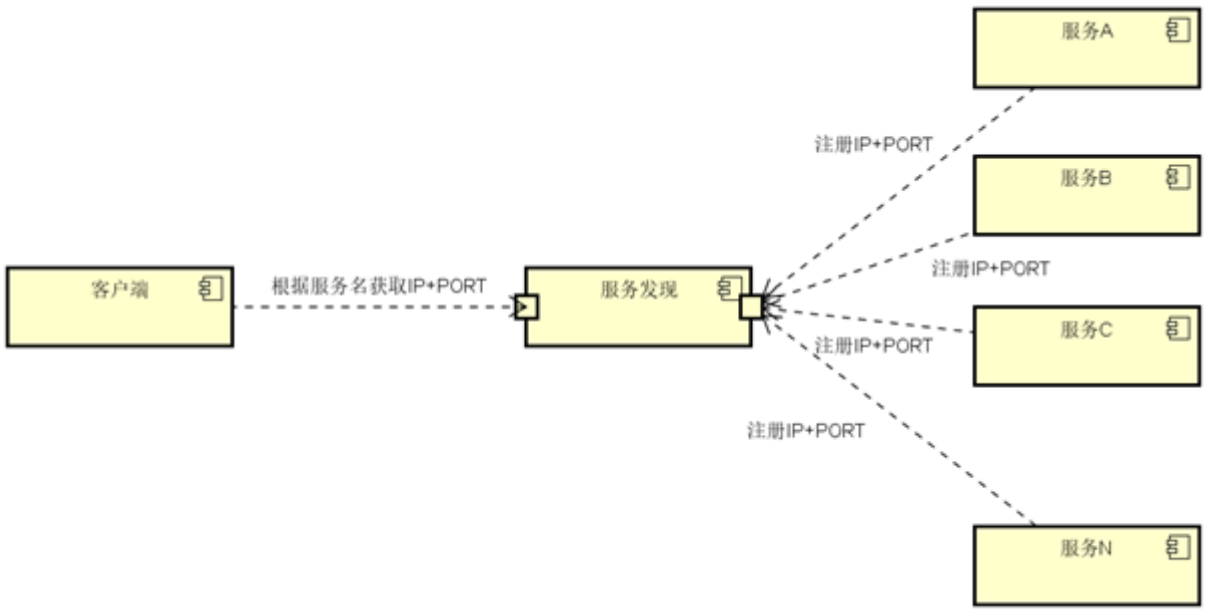
什么是服务发现

微服务的框架体系中，服务发现是不能不提的一个模块。我相信了解或者熟悉微服务的童鞋应该都知道它的重要性。这里我只是简单的提一下，毕竟这不是我们的重点。我们看下面的一幅图片：



图中，客户端的一个接口，需要调用服务A-N。客户端必须要知道所有服务的网络位置的，以往的做法是配置是配置文件中，或者有些配置在数据库中。这里就带出几个问题： 需要配置N个服务的网络位置，加大配置的复杂性 服务的网络位置变化，都需要改变每个调用者的配置 集群的情况下，难以做负载（反向代理的方式除外）

总结起来一句话：服务多了，配置很麻烦，问题多多 既然有这些问题，那么服务发现就是解决这些问题的。话说，怎么解决呢？我们再看一张图：



与之前一张不同的是，加了个服务发现模块。图比较简单，这边文字描述下。服务A-N把当前自己的网络位置注册到服务发现模块（这里注册的意思就是告诉），服务发现就以K-V的方式记录下，K一般是服务名，V就是IP:PORT。服务发现模块定时的轮询查看这些服务能不能访问的了（这就是健康检查）。客户端在调用服务A-N的时候，就跑去服务发现模块问下它们的网络位置，然后再调用它们的服务。这样的方式是不是就可以解决上面的问题了呢？客户端完全不需要记录这些服务网络位置，客户端和服务端完全解耦！

> 这个过程大体是这样，当然服务发现模块没这么简单。里面包含的东西还很多。这样表述只是方便理解。

Consul的安装

Consul用Golang实现，因此具有天然可移植性 (支持 Linux、windows和macOS)。安装包仅包含一个可执行文件。 Consul安装非常简单，只需要下载对应系统的软件包并解压后就可使用。

下载安装

```
# 这里以 Linux系统为例:
$ wget https://releases.hashicorp.com/consul/1.2.0/consul_1.2.0_linux_amd64.zip

$ unzip consul_1.2.0_linux_amd64.zip
$ mv consul /usr/local/bin/
```

其它系统版本可在这里下载： https://www.consul.io/downloads.html

验证安装

安装 Consul后，通过执行 consul命令，你可以看到命令列表的输出

```
$ consul
```

```
itcast@itcast-virtual-machine:~$ consul
Usage: consul [--version] [--help] <command> [<args>]

Available commands are:
    agent          Runs a Consul agent
    catalog        Interact with the catalog
    connect        Interact with Consul Connect
    event          Fire a new event
    exec           Executes a command on Consul nodes
    force-leave    Forces a member of the cluster to enter the "left" state
    info           Provides debugging information for operators.
    intention      Interact with Connect service intentions
    join           Tell Consul agent to join cluster
    keygen         Generates a new encryption key
    keyring        Manages gossip layer encryption keys
    kv             Interact with the key-value store
    leave          Gracefully leaves the Consul cluster and shuts down
    lock           Execute a command holding a lock
    maint          Controls node or service maintenance mode
    members        Lists the members of a Consul cluster
    monitor        Stream logs from a Consul agent
    operator       Provides cluster-level tools for Consul operators
    reload         Triggers the agent to reload configuration files
    rtt            Estimates network round trip time between nodes
    snapshot       Saves, restores and inspects snapshots of Consul server state
    validate       Validate config files/directories
    version        Prints the Consul version
    watch          Watch for changes in Consul
```

就证明成功了

Consul 的角色

client: 客户端, 无状态, 将 HTTP 和 DNS 接口请求转发给局域网内的服务端集群. **server**: 服务端, 保存配置信息, 高可用集群, 在局域网内与本地客户端通讯, 通过广域网与其他数据中心通讯. 每个数据中心的 server 数量推荐为 3 个或是 5 个.

运行 Consul代理

Consul是典型的 C/S架构，可以运行服务模式或客户模式。每一个数据中心必须有至少一个服务节点， 3到5个服务节点最好。非常不建议只运行一个服务节点，因为在节点失效的情况下数据有极大的丢失风险。

运行Agent

完成Consul的安装后,必须运行agent. agent可以运行为server或client模式.每个数据中心至少必须拥有一台server . 建议在一个集群中有3或者5个server.部署单一的server,在出现失败时会不可避免的造成数据丢失.

其他的agent运行为client模式.一个client是一个非常轻量级的进程.用于注册服务,运行健康检查和转发对server的查询.agent必须在集群中的每个主机上运行.

启动 Consul Server

```
#S1:
$ consul agent -server -bootstrap-expect 2 -data-dir /tmp/consul -node=S2 -bind=192.168.110.123 -ui  -config-dir /etc/consul.d -rejoin -join 192.168.110.123 -client 0.0.0.0
#运行cosnul agent以server模式
-server ：  定义agent运行在server模式
-bootstrap-expect ：在一个datacenter中期望提供的server节点数目，当该值提供的时候，consul一直等到达到指定sever数目的时候才会引导整个集群，该标记不能和bootstrap共用
-data-dir：提供一个目录用来存放agent的状态，所有的agent允许都需要该目录，该目录必须是稳定的，系统重启后都继续存在
-node：节点在集群中的名称，在一个集群中必须是唯一的，默认是该节点的主机名
-bind：该地址用来在集群内部的通讯，集群内的所有节点到地址都必须是可达的，默认是0.0.0.0
-ui：  启动web界面
-config-dir：：配置文件目录，里面所有以.json结尾的文件都会被加载
-rejoin：使consul忽略先前的离开，在再次启动后仍旧尝试加入集群中。
-client：consul服务侦听地址，这个地址提供HTTP、DNS、RPC等服务，默认是127.0.0.1所以不对外提供服务，如果你要对外提供服务改成0.0.0.0

```

```
#n2:
$ consul agent -server -bootstrap-expect 2 -data-dir /tmp/consul -node=n1 -bind=192.168.110.7 -ui  -rejoin -join 192.168.110.123

-server ：  定义agent运行在server模式
-bootstrap-expect ：在一个datacenter中期望提供的server节点数目，当该值提供的时候，consul一直等到达到指定sever数目的时候才会引导整个集群，该标记不能和bootstrap共用
-bind：该地址用来在集群内部的通讯，集群内的所有节点到地址都必须是可达的，默认是0.0.0.0
-node：节点在集群中的名称，在一个集群中必须是唯一的，默认是该节点的主机名
-ui：  启动web界面
-rejoin：使consul忽略先前的离开，在再次启动后仍旧尝试加入集群中。
-config-dir：：配置文件目录，里面所有以.json结尾的文件都会被加载
-client：consul服务侦听地址，这个地址提供HTTP、DNS、RPC等服务，默认是127.0.0.1所以不对外提供服务，如果你要对外提供服务改成0.0.0.0
-join 192.168.110.121 ：  启动时加入这个集群
```

启动 Consul Client

```
#n3
$ consul agent  -data-dir /tmp/consul -node=n3 -bind=192.168.110.124 -config-dir /etc/consul.d -rejoin -join 192.168.110.123

运行cosnul agent以client模式，-join 加入到已有的集群中去。
```

查看集群成员

新开一个终端窗口运行consul members, 你可以看到Consul集群的成员.

```
$ consul members

Node  Address               Status  Type    Build  Protocol  DC   Segment
n1    192.168.110.7:8301    alive   server  1.1.0  2         dc1  <all>
n2    192.168.110.121:8301  alive   server  1.1.0  2         dc1  <all>
n3    192.168.110.122:8301  alive   client  1.1.0  2         dc1  <default>
```

停止Agent

你可以使用Ctrl-C 优雅的关闭Agent. 中断Agent之后你可以看到他离开了集群并关闭.

在退出中,Consul提醒其他集群成员,这个节点离开了.如果你强行杀掉进程.集群的其他成员应该能检测到这个节点失效了.当一个成员离开,他的服务和检测也会从目录中移除.当一个成员失效了,他的健康状况被简单的标记为危险,但是不会从目录中移除.Consul会自动尝试对失效的节点进行重连.允许他从某些网络条件下恢复过来.离开的节点则不会再继续联系.

此外,如果一个agent作为一个服务器,一个优雅的离开是很重要的,可以避免引起潜在的可用性故障影响达成一致性协议. consul优雅的退出

```
$ consul leave
```

注册服务

搭建好conusl集群后，用户或者程序就能到consul中去查询或者注册服务。可以通过提供服务定义文件或者调用HTTP API来注册一个服务.

首先,为Consul配置创建一个目录.Consul会载入配置文件夹里的所有配置文件.在Unix系统中通常类似/etc/consul.d (.d 后缀意思是这个路径包含了一组配置文件).

```
$ mkdir /etc/consul.d
```

然后,我们将编写服务定义配置文件.假设我们有一个名叫web的服务运行在 80端口.另外,我们将给他设置一个标签.这样我们可以使用他作为额外的查询方式:

```
{
  "service": {                                    #服务
     "name": "web",                               #名称
```

```
    "tags": ["master"],                                 #标记
    "address": "127.0.0.1",                             #ip
    "port": 10000,                                      #端口
    "checks": [
      {
        "http": "http://localhost:10000/health",
        "interval": "10s"                               #检查时间
      }
    ]
  }
}
```

测试程序

```
package main
import (
    "fmt"
    "net/http"
)
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Println("hello Web3! This is n3或者n2")
    fmt.Fprintf(w, "Hello Web3! This is n3或者n2")
}
func healthHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Println("health check! n3或者n2")
}
func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/health", healthHandler)
    http.ListenAndServe(":10000", nil)
}
```

查询服务

一旦agent启动并且服务同步了.我们可以通过DNS或者HTTP的API来查询服务.

DNS API

让我们首先使用DNS API来查询.在DNS API中,服务的DNS名字是 NAME.service.consul. 虽然是可配置的,但默认的所有DNS名字会都在consul命名空间下.这个子域告诉Consul,我们在查询服务,NAME则是服务的名称.

对于我们上面注册的Web服务.它的域名是 web.service.consul :

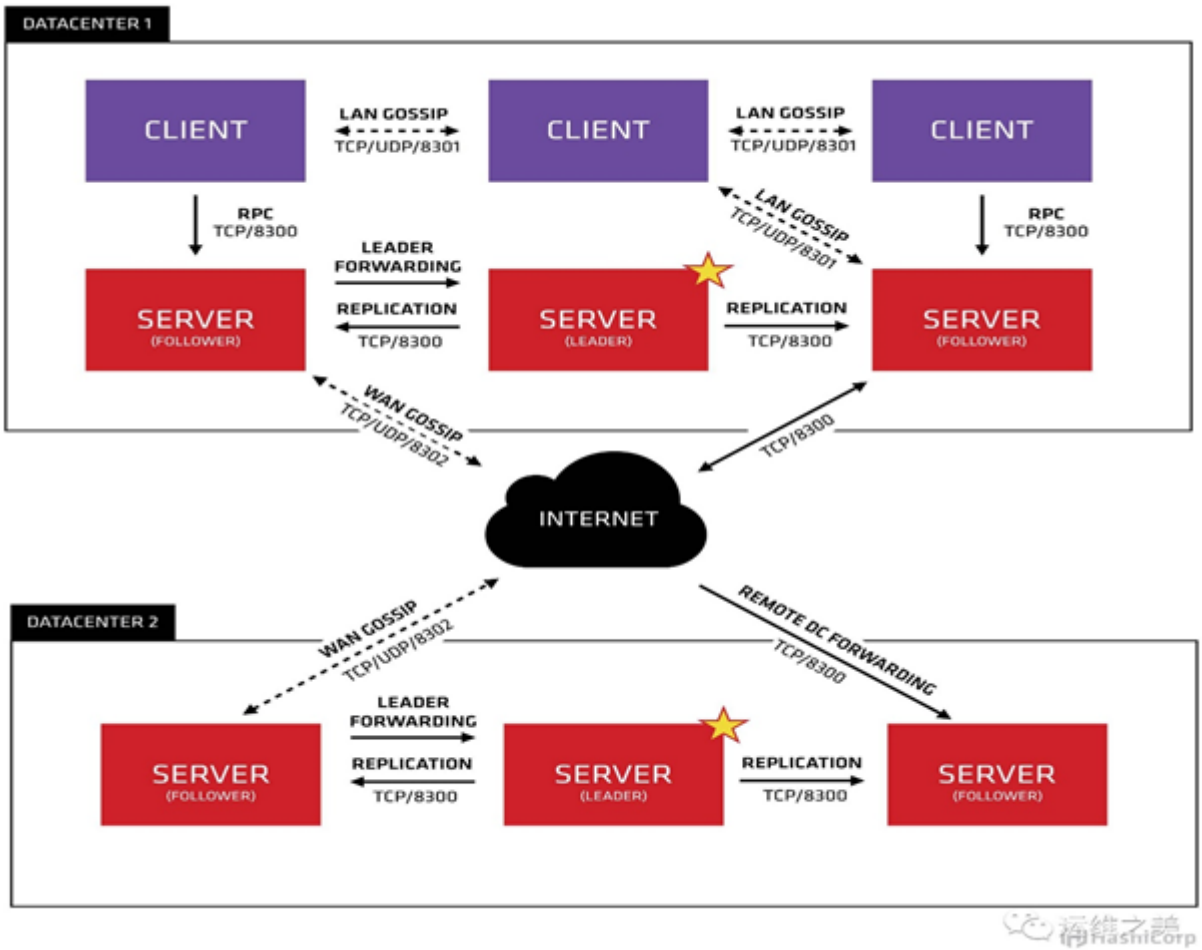
```
$ dig @127.0.0.1 -p 8600 web.service.consul
```

有也可用使用 DNS API 来接收包含 地址和端口的 SRV记录:

```
$ dig @127.0.0.1 -p 8600 web.service.consul SRV
```

SRV记录告诉我们 web 这个服务运行于节点dhcp-10-201-102-198 的80端口. DNS额外返回了节点的A记录.

Consul架构



我们只看数据中心1，可以看出consul的集群是由N个SERVER，加上M个CLIENT组成的。而不管是SERVER还是CLIENT，都是consul的一个节点，所有的服务都可以注册到这些节点上，正是通过这些节点实现服务注册信息的共享。除了这两个，还有一些小细节，一一简单介绍。 **CLIENT** CLIENT表示consul的client模式，就是客户端模式。是consul节点的一种模式，这种模式下，所有注册到当前节点的服务会被转发到SERVER【通过HTTP和DNS接口请求server】，本身是**不持久化**这些信息。 **SERVER** SERVER表示consul的server模式，表明这个consul是个server，这种模式下，功能和CLIENT都一样，唯一不同的是，它会把所有的信息持久化的本地，这样遇到故障，信息是可以被保留的 **SERVER-LEADER** 中间那个SERVER下面有LEADER的字眼，表明这个SERVER是它们的老大，它和其它SERVER不一样的一点是，它需要负责同步注册的信息给其它的SERVER，同时也要负责各个节点的健康监测。

Consul的client mode把请求转向server，那么client的作用是什么?

consul可以用来实现分布式系统的服务发现与配置。client把服务请求传递给server，server负责提供服务以及和其他数据中心交互。题主的问题是，既然server端提供了所有服务，那为何还需要多此一举地用client端来接收一次服务请求。我想，采用这种架构有以下几种理由： 首先server端的网络连接资源有限。对于一个分布式系统，一般情况下访问量是很大的。如果用户能不通过client直接地访问数据中心，那么数据中心必然要为每个用户提供一个单独的连接资源(线程，端口号等等)，那么server端的负担会非常大。所以很有必要用大量的client端来分散用户的连接请求，在client端先统一整合用户的服务请求，然后一次性地通过一个单一的链接发送大量的请求给server端，能够大量减少server端的网络负担。 其次，在client端可以对用户的请求进行一些处理来提高服务的效率，比如将相同的请求合并成同一个查询，再比如将之前的查询通过cookie的形式缓存下来。但是这些功能都需要消耗不少的计算和存储资源。如果在server端提供这些功能，必然加重server端的负担，使得server端更加不稳定。而通过client端来进行这些服务就没有这些问题了，因为client端不提供实际服务，有很充足的计算资源来进行这些处理这些工作。 最后还有一点，consul规定只要接入一个client就能将自己注册到一个服务网络当中。这种架构使得系统的可扩展性非常的强，网络的拓扑变化可以特别的灵活。这也是依赖于client—server结构的。如果系统中只有几个数据中心存在，那网络的扩张也无从谈起了。

Consul资料：http://www.liangxiansen.cn/2017/04/06/consul https://blog.csdn.net/yuanyuanispeak/article/details/54880743

五、Micro

Micro的介绍

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

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

micro的安装

下载micro

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

```
$ unzip hashicorp_consul.zip -d github.com/hashicorp/consul
# miekg_dns.zip 包解压在github.com/miekg/dns
$ unzip miekg_dns.zip -d github.com/miekg/dns
$ go get github.com/micro/micro
```

编译安装micro

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

插件安装

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

micro基本演示

创建微服务命令说明

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

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

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

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

创建2个服务

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

.
#主函数
├── main.go
#插件
├── plugin.go
#被调用函数
├── handler
│   └── example.go
#订阅服务
├── subscriber
│   └── example.go
#proto协议
├── proto/example
│   └── example.proto
#docker生成文件
├── Dockerfile
├── Makefile
└── README.md


download protobuf for micro:

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

compile the proto file example.proto:

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

#使用创建srv时给的protobuf命令保留用来将proto文件进行编译

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

```
├── Dockerfile
├── Makefile
└── README.md

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

启动consul进行监管

```
consul agent -dev
```

对srv服务进行的操作

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

对web服务进行的操作

main文件

```
package main

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

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

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

```
25
26      //处理请求   / 的路由    //当前这个web微服务的 html文件进行映射
27      service.Handle("/", http.FileServer(http.Dir("html")))
28
29      //处理请求 /example/call   的路由    这个相应函数 在当前项目下的handler
30      service.HandleFunc("/example/call", handler.ExampleCall)
31
32      //运行服务
33          if err := service.Run(); err != nil {
34                  log.Fatal(err)
35          }
36  }
37
```

将准备好的html文件替换掉原有的文件

handler文件

```
1   package handler
2
3   import (
4       "context"
5       "encoding/json"
6       "net/http"
7       "time"
8
9       "github.com/micro/go-micro/client"
10      //将srv中的proto的文件导入进来进行通信的使用
11      example "micro/rpc/srv/proto/example"
12  )
13  //相应请求的业务函数
14  func ExampleCall(w http.ResponseWriter, r *http.Request) {
15      // 将传入的请求解码为json
16      var request map[string]interface{}
17      if err := json.NewDecoder(r.Body).Decode(&request); err != nil {
18          http.Error(w, err.Error(), 500)
19          return
20      }
21
22      // 调用服务
23      //替换掉原有的服务名
24      //通过服务名和
25      exampleClient := example.NewExampleService("go.micro.srv.srv",
    client.DefaultClient)
26      rsp, err := exampleClient.Call(context.TODO(), &example.Request{
27          Name: request["name"].(string),
28      })
29      if err != nil {
30          http.Error(w, err.Error(), 500)
31          return
32      }
```

```
33
34      // we want to augment the response
35      response := map[string]interface{}{
36          "msg": rsp.Msg,
37          "ref": time.Now().UnixNano(),
38      }
39
40      // encode and write the response as json
41      if err := json.NewEncoder(w).Encode(response); err != nil {
42          http.Error(w, err.Error(), 500)
43          return
44      }
45  }
46
```

升级成为grpc的版本

重新生成proto文件

srv的main.go

```
1   package main
2
3   import (
4       "github.com/micro/go-log"
5       "github.com/micro/go-micro"
6       "micro/grpc/srv/handler"
7       "micro/grpc/srv/subscriber"
8       example "micro/grpc/srv/proto/example"
9       "github.com/micro/go-grpc"
10  )
11
12  func main() {
13      // New Service
14
15      service := grpc.NewService(
16          micro.Name("go.micro.srv.srv"),
17          micro.Version("latest"),
18      )
19
20      // Initialise service
21      service.Init()
22
23      // Register Handler
24      example.RegisterExampleHandler(service.Server(), new(handler.Example))
25
26      // Register Struct as Subscriber
27      micro.RegisterSubscriber("go.micro.srv.srv", service.Server(),
    new(subscriber.Example))
```

```
28
29      // Register Function as Subscriber
30      micro.RegisterSubscriber("go.micro.srv.srv", service.Server(),
    subscriber.Handler)
31
32      // Run service
33      if err := service.Run(); err != nil {
34          log.Fatal(err)
35      }
36  }
37
```

srv的example.go

```
1   package handler
2
3   import (
4       "context"
5
6       "github.com/micro/go-log"
7
8       example "micro/grpc/srv/proto/example"
9   )
10
11  type Example struct{}
12
13  // Call is a single request handler called via client.Call or the generated client
    code
14  func (e *Example) Call(ctx context.Context, req *example.Request, rsp
    *example.Response) error {
15      log.Log("Received Example.Call request")
16      rsp.Msg = "Hello " + req.Name
17      return nil
18  }
19
20  // Stream is a server side stream handler called via client.Stream or the generated
    client code
21  func (e *Example) Stream(ctx context.Context, req *example.StreamingRequest, stream
    example.Example_StreamStream) error {
22      log.Logf("Received Example.Stream request with count: %d", req.Count)
23
24      for i := 0; i < int(req.Count); i++ {
25          log.Logf("Responding: %d", i)
26          if err := stream.Send(&example.StreamingResponse{
27              Count: int64(i),
28          }); err != nil {
29              return err
30          }
31      }
32
33      return nil
34  }
35
```

```
// PingPong is a bidirectional stream handler called via client.Stream or the generated client code
func (e *Example) PingPong(ctx context.Context, stream example.Example_PingPongStream) error {
    for {
        req, err := stream.Recv()
        if err != nil {
            return err
        }
        log.Logf("Got ping %v", req.Stroke)
        if err := stream.Send(&example.Pong{Stroke: req.Stroke}); err != nil {
            return err
        }
    }
}

```

修改web的main.go

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

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

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

```

修改web的handler.go

```

```

```
package handler

import (
    "context"
    "encoding/json"
    "net/http"
    "time"
    example "micro/grpc/srv/proto/example"
    "github.com/micro/go-grpc"
)

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

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

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

关于插件化

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