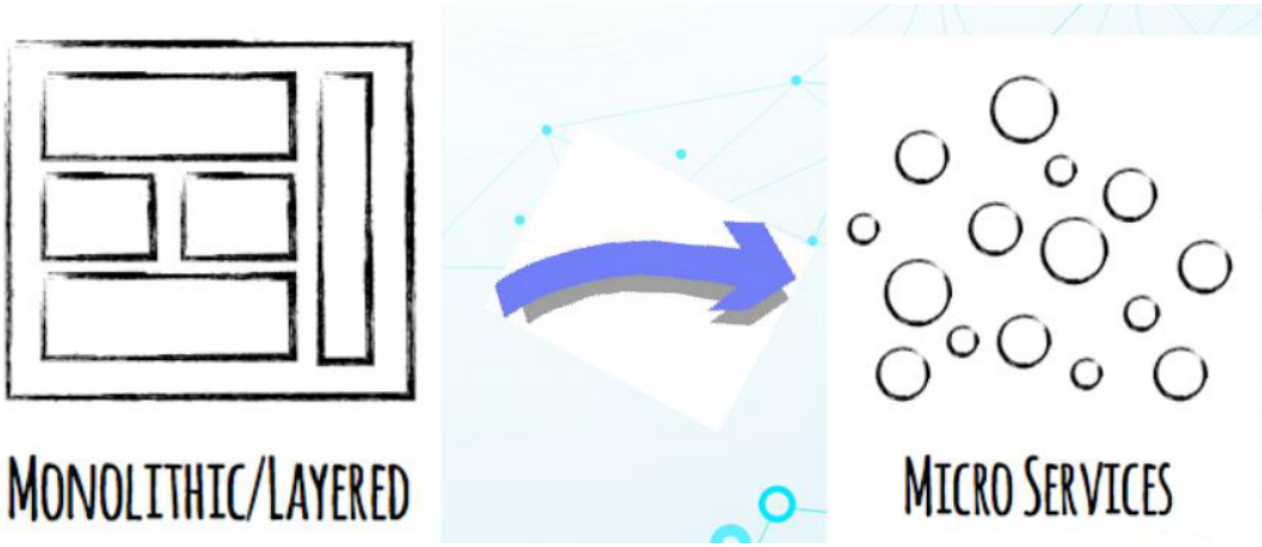
微服务



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。

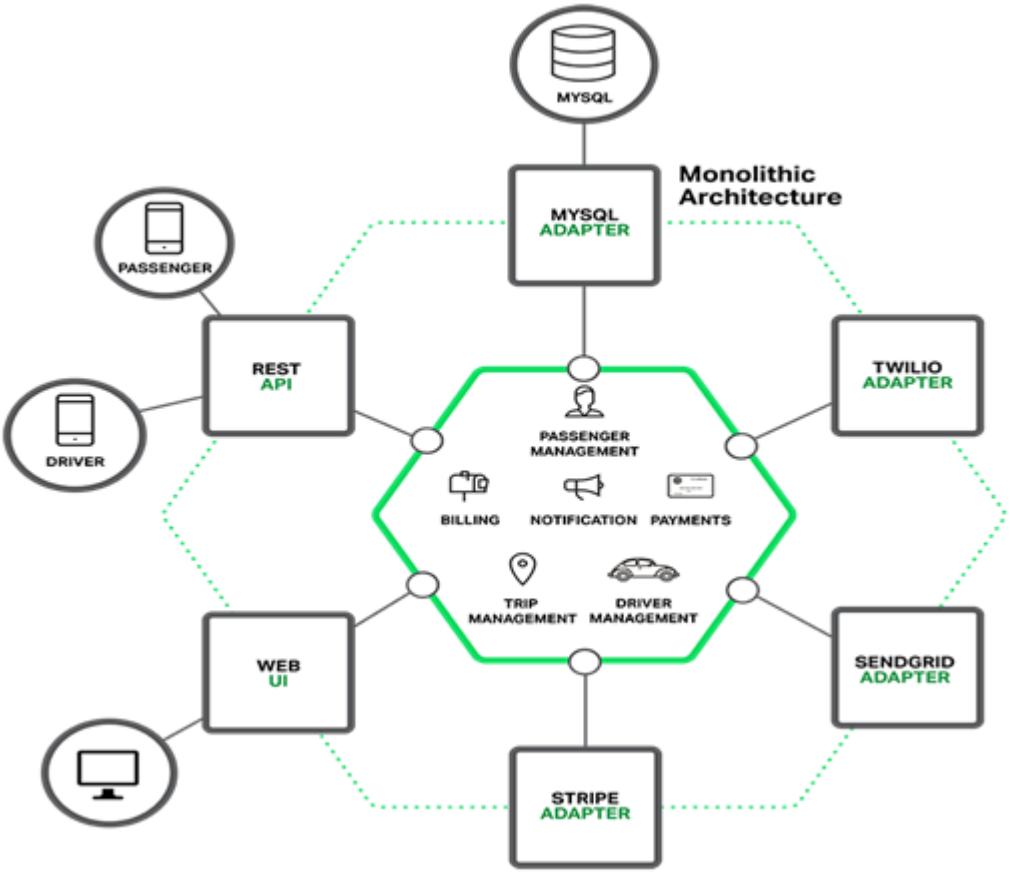
微服务架构定义的精髓，可以用一句话来描述，那就是“分而治之，合而用之”。

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。

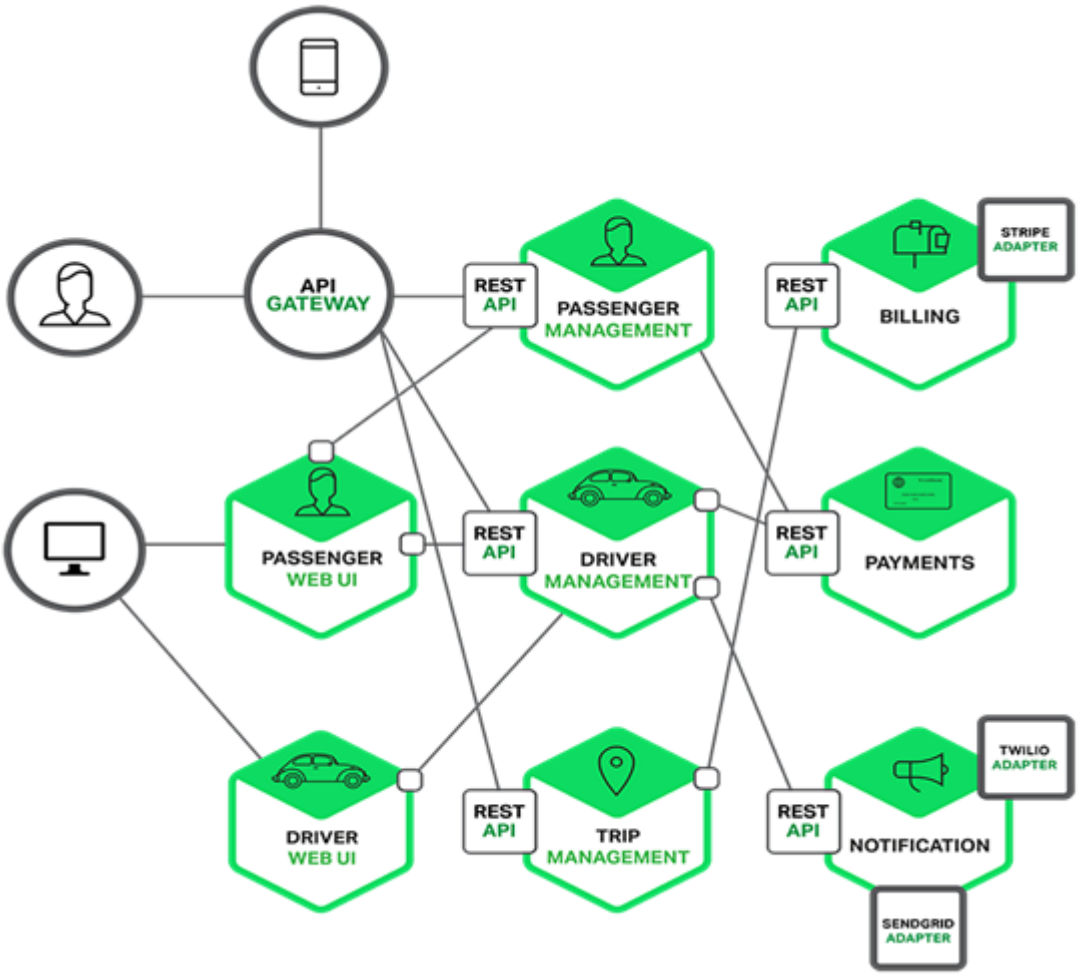
技术选型成本高

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

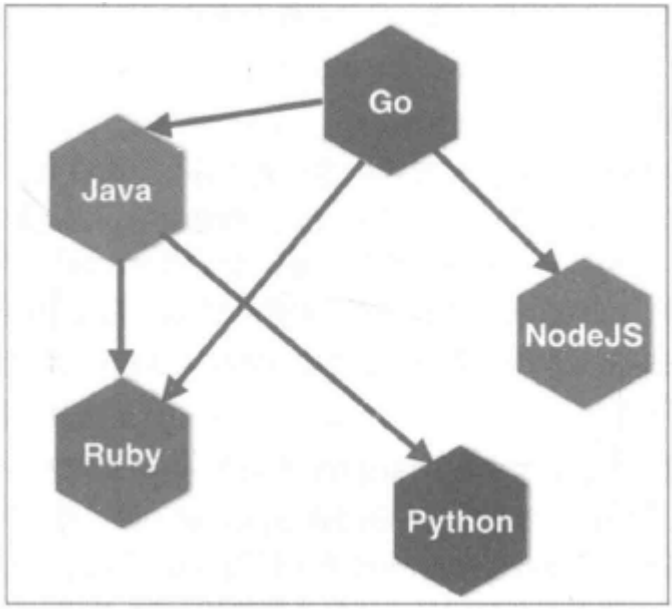


单一职责

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?

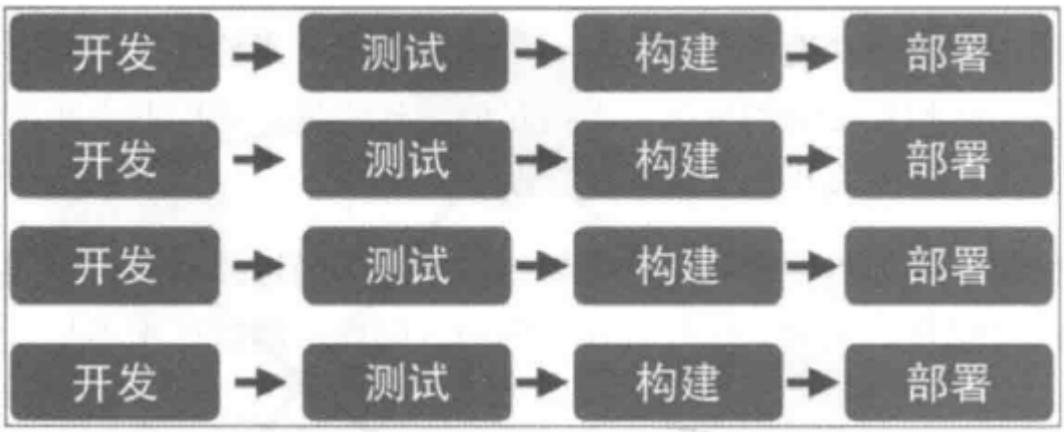
答：REST 指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。

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。

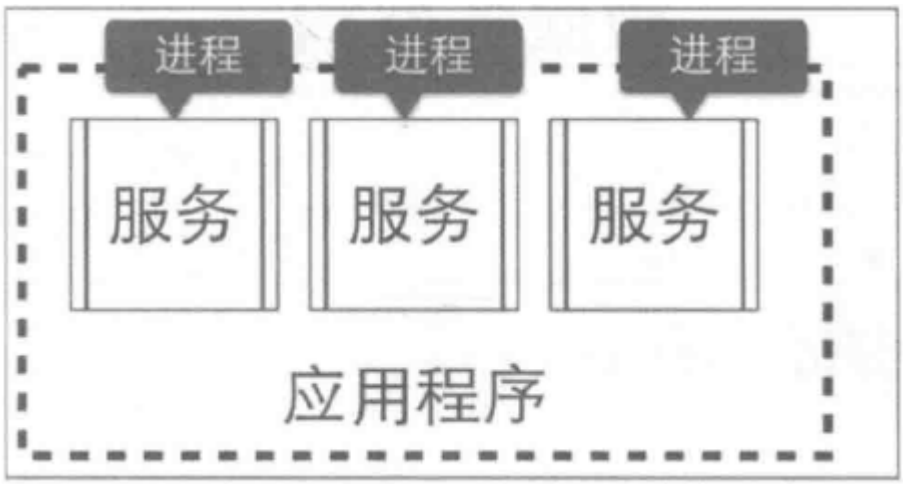


进程隔离

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

```

#下载 protoBuf:
$ git clone https://github.com/protocolbuffers/protobuf.git
```

```
#或者直接将压缩包拖入后解压
unzip protobuf.zip


#安装依赖库
$ sudo apt-get install autoconf  automake  libtool  curl  make  g++  unzip libffi-dev -y
#安装
$ cd protobuf/
$ ./autogen.sh
$ ./configure
$ make
$ sudo make install
$ sudo ldconfig    # 刷新共享库 很重要的一步啊
#安装的时候会比较卡
#成功后需要使用命令测试
$ protoc -h

```

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。

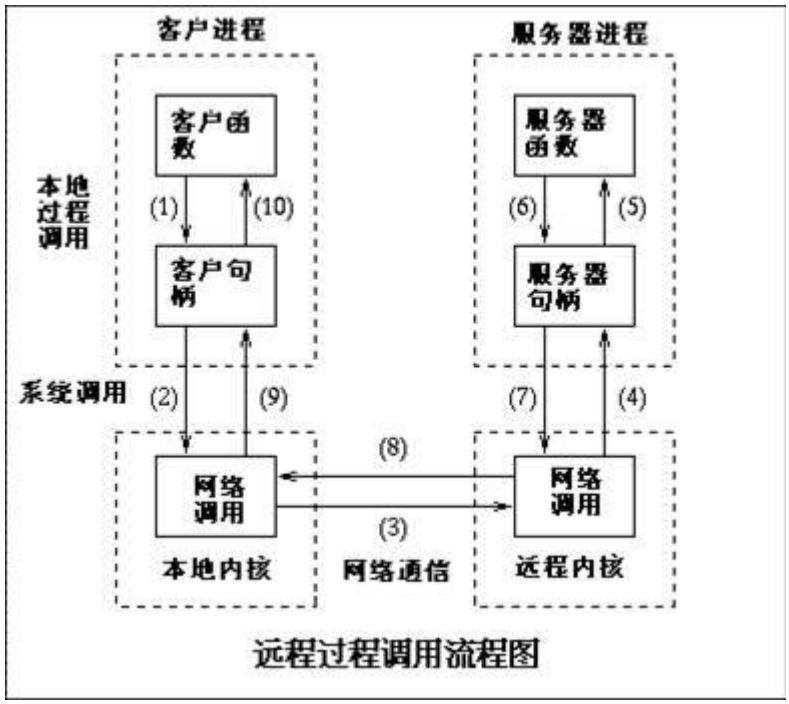
gRPC基于 HTTP/2标准设计，带来诸如双向流、流控、头部压缩、单 TCP连接上的多复用请求等特。这些特性使得其在移动设备上表现更好，更省电和节省空间占用。

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

```

```go
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{
        beego.Info("打call失败")
    }
    beego.Info("返回结果",val)

}

```

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）。

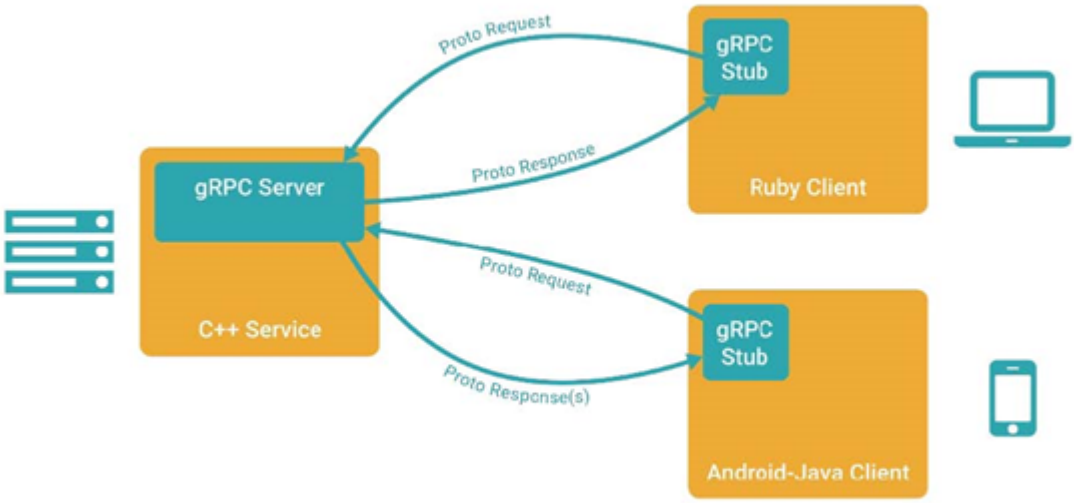
在服务端实现这个接口，并运行一个 gRPC服务器来处理客户端调用。

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"
)

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。

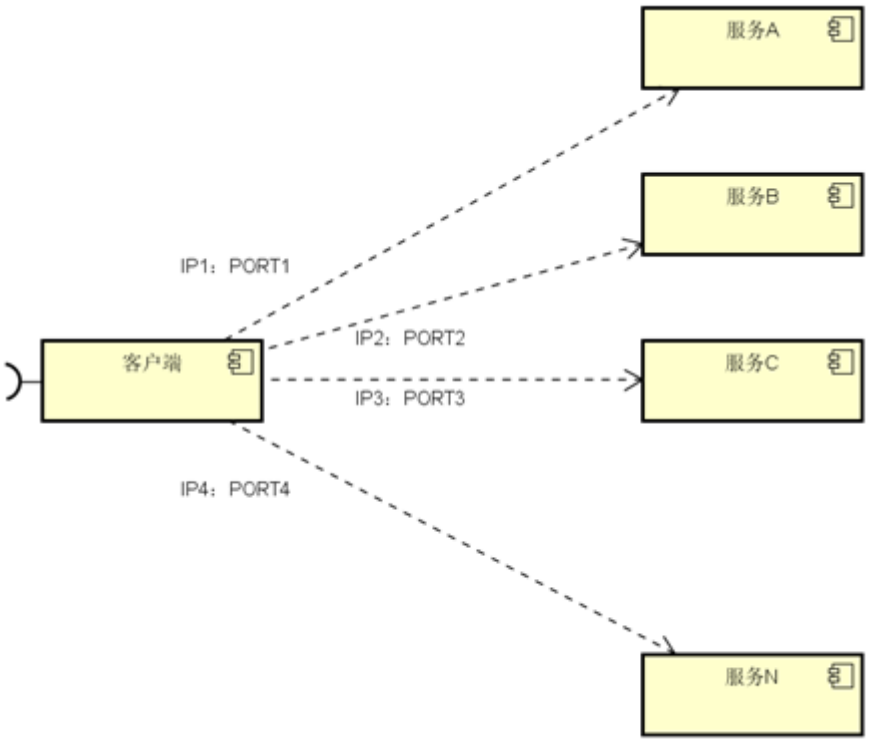
multi-datacenter：无需复杂的配置，即可支持任意数量的区域。

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：

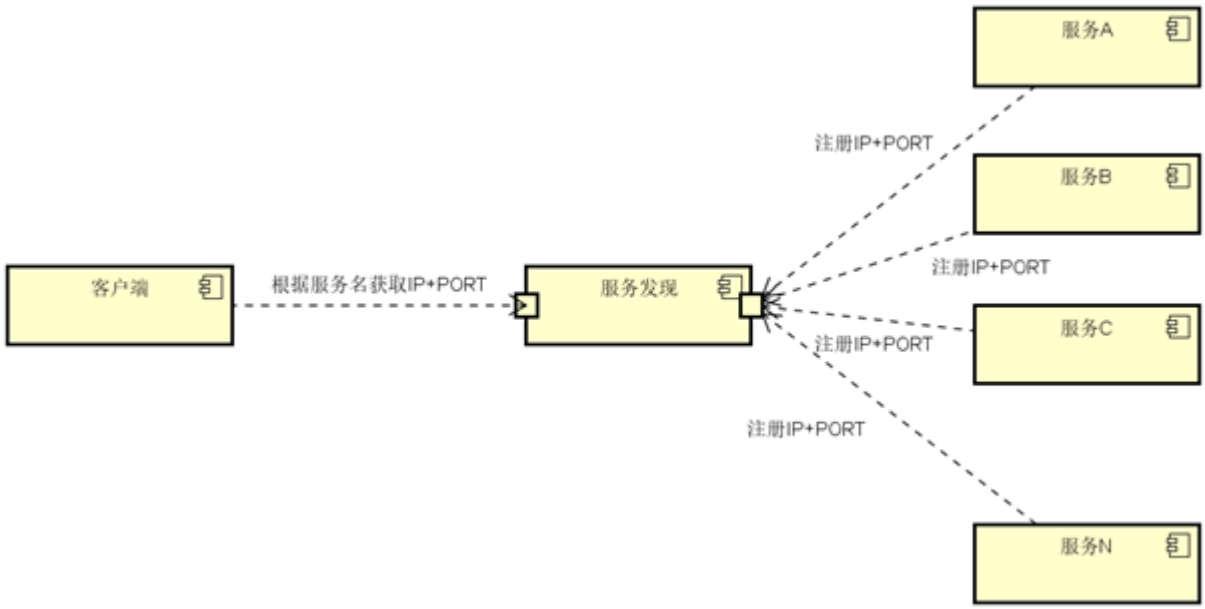
需要配置N个服务的网络位置，加大配置的复杂性

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！

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

```
#node1:
$ consul agent -server -bootstrap-expect 2 -data-dir /tmp/consul -node=n1 -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

```

```
#node2:
$ consul agent -server -bootstrap-expect 2 -data-dir /tmp/consul -node=n2 -bind=192.168.110.148 -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

```
#node3:
$ 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
#节点  网络地址               状态     类型     版本    协议       数据中心   分管部分
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 10000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.

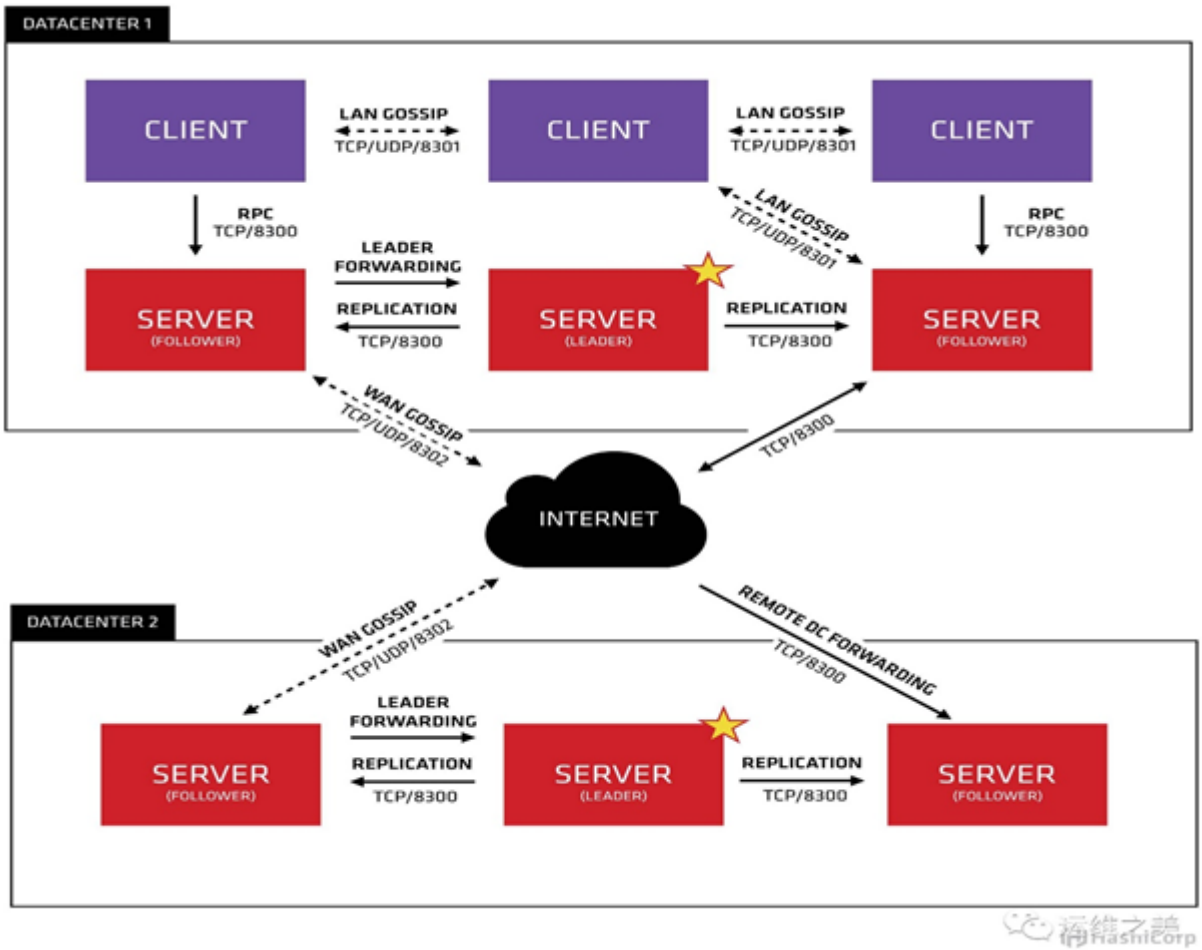
对于我们上面注册的Web服务.它的域名是 web.service.consul :

```
$ 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
```

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)
        }

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

```

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

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

```

将准备好的html文件替换掉原有的文件

handler文件

```
package handler

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

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

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

    // we want to augment the response
    response := map[string]interface{}{
        "msg": rsp.Msg,
```

```
        "ref": time.Now().UnixNano(),
    }

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

```

升级成为grpc的版本

重新生成proto文件

srv的main.go

```
package main

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

func main() {
    // 创建新服务

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

    // 初始化服务
    service.Init()

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

    // Register Struct as Subscriber
    //注册结构体来自于Subscriber
```

```
    micro.RegisterSubscriber("go.micro.srv.srv", service.Server(),
new(subscriber.Example))

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

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

```

srv的example.go

```
package handler

import (
    "context"

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

type Example struct{}

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

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

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

```
33
34      return nil
35  }
36
37  // PingPong is a bidirectional stream handler called via client.Stream or the
    generated client code
38  //心跳检测机制
39  func (e *Example) PingPong(ctx context.Context, stream
    example.Example_PingPongStream) error {
40      for {
41          req, err := stream.Recv()
42          if err != nil {
43              return err
44          }
45          log.Logf("Got ping %v", req.Stroke)
46          if err := stream.Send(&example.Pong{Stroke: req.Stroke}); err != nil {
47              return err
48          }
49      }
50  }
51
```

修改web的main.go

```
1   package main
2   import (
3           "github.com/micro/go-log"
4       "net/http"
5
6           "github.com/micro/go-web"
7           "micro/grpc/web/handler"
8   )
9   func main() {
10      // create new web service
11          service := web.NewService(
12                  web.Name("go.micro.web.web"),
13                  web.Version("latest"),
14                  web.Address(":8080"),
15          )
16      // initialise service
17          if err := service.Init(); err != nil {
18                  log.Fatal(err)
19          }
20
21      // register html handler
22      service.Handle("/", http.FileServer(http.Dir("html")))
23      // register call handler
24      service.HandleFunc("/example/call", handler.ExampleCall)
25      // run service
26          if err := service.Run(); err != nil {
27                  log.Fatal(err)
28          }
29  }
```

```
30
```

修改web的handler.go

```
package handler

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

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

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

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

关于插件化

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

六、租房网

项目启动

拆分原则

1、单一职责

2、服务粒度适中

3、考虑团队结构

4、以业务模型切入

5、演进式拆分

6、避免环形依赖和双向依赖

服务发现的启动

项目开始之前前我们先要启动我们单机版的consul

```
$ consul agent -dev
```

数据库的准备

在mysql中创建一个库

```
$ mysql -uroot -p
#输入root密码
#go1micro是库名
Mysql>create database if not exists go1micro default charset utf8 collate
utf8_general_ci; //创建1个库

Mysql> use go1micro
```

```
Mysql> show tables;
+------------------+
| Tables_in_go1micro|
+------------------+
| area             |
| facility         |
| facility_houses  |多对多的关联表
| house            |
| house_image      |
| order_house      |
| user             |
+------------------+
7 rows in set (0.00 sec)
mysql> desc facility_houses;
+-------------+------------+------+-----+---------+----------------+
| Field       | Type       | Null | Key | Default | Extra          |
+-------------+------------+------+-----+---------+----------------+
| id          | bigint(20) | NO   | PRI | NULL    | auto_increment |
| facility_id | int(11)    | NO   |     | NULL    |                |
| house_id    | int(11)    | NO   |     | NULL    |                |
+-------------+------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)

```

01 web端

创建命令

```
$ micro new --type "web" sss/IhomeWeb
```

REST

RESTful，是目前最为流行的一种互联网软件架构。因为它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用

什么是REST

REST(REpresentational State Transfer)这个概念，首次出现是在 2000年Roy Thomas Fielding（他是HTTP规范的主要编写者之一）的博士论文中，它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的。

要理解什么是REST，我们需要理解下面几个概念:

1 、**资源**（Resources）

REST是"表现层状态转化"，其实它省略了主语。"表现层"其实指的是"资源"的"表现层"。 那么什么是资源呢？就是我们平常上网访问的一张图片、一个文档、一个视频等。这些资源我们通过URI来定位，也就是一个URI表示一个资源。

2、**表现层**（Representation） 资源是做一个具体的实体信息，他可以有多种的展现方式。而把实体展现出来就是表现层，例如一个txt文本信息，他可以输出成html、json、xml等格式，一个图片他可以jpg、png等方式展现，这个就是表现层的意思。 URI确定一个资源，但是如何确定它的具体表现形式呢？应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对"表现层"的描述。

3、 **状态转化**（State Transfer） 访问一个网站，就代表了客户端（浏览器）和服务器的一个互动过程。在这个过程中，肯定涉及到数据和状态的变化。而HTTP协议是无状态的，那么这些状态肯定保存在服务器端，所以如果客户端想要通知服务器端改变数据和状态的变化，肯定要通过某种方式来通知它。

客户端能通知服务器端的手段，只能是HTTP协议。

具体来说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。

它们分别对应四种基本操作：

查：GET用来获取资源

增：POST用来新建资源（也可以用于更新资源）

改：PUT用来更新资源

删：DELETE用来删除资源。

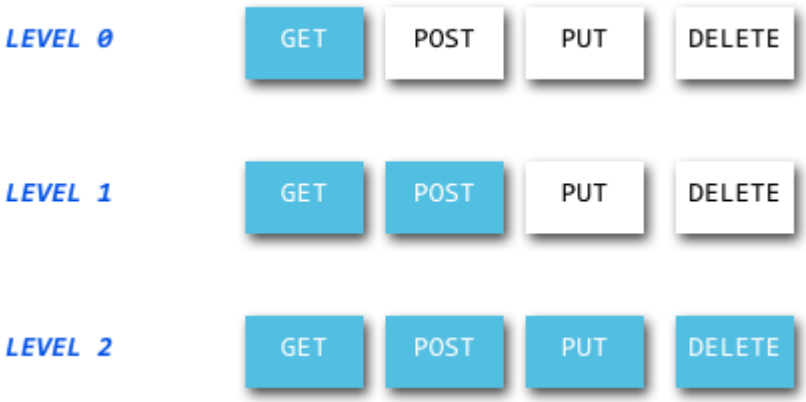
综合上面的解释，我们总结一下什么是RESTful架构： （1）每一个URI代表一种资源；

（2）客户端和服务器之间，传递这种资源的某种表现层；

（3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

RESTful的实现

Go没有为REST提供直接支持，但是因为RESTful是基于HTTP协议实现的，所以我们可以利用net/http包来自己实现，当然需要针对REST做一些改造，REST是根据不同的method来处理相应的资源，目前已经存在的很多自称是REST的应用，其实并没有真正的实现REST，我暂且把这些应用根据实现的method分成几个级别



我们目前实现REST的三个level，我们在应用开发的时候也不一定全部按照RESTful的规则全部实现他的方式，因为有些时候完全按照RESTful的方式未必是可行的，RESTful服务充分利用每一个HTTP方法，包括DELETE和PUT。可有时，HTTP客户端只能发出GET和POST请求：

1、 HTML标准只能通过链接和表单支持GET和POST。在没有Ajax支持的网页浏览器中不能发出PUT或DELETE命令有些防火墙会挡住HTTP PUT和DELETE请求要绕过这个限制，客户端需要把实际的PUT和DELETE请求通过 POST 请求穿透过来。RESTful 服务则要负责在收到的 POST 请求中找到原始的 HTTP 方法并还原。

2、我们现在可以通过POST里面增加隐藏字段_method这种方式可以来模拟PUT、DELETE等方式，但是服务器端需要做转换。我现在的项目里面就按照这种方式来做的REST接口。当然Go语言里面完全按照RESTful来实现是很容易的

```
package main

import (
    "fmt"
    "github.com/julienschmidt/httprouter"
    "net/http"
)

func getuser(w http.ResponseWriter, r *http.Request,ps httprouter.Params) {
    uid:=ps.ByName("uid")
    fmt.Fprintf(w, "获取用户名 %s", uid)
    mo:= r.Method
    fmt.Fprintf(w, "获取请求方式 %s", mo)

}

func main() {

    router := httprouter.New()
    router.GET("/getuser/:uid", getuser)

    http.Handle("/", router)
    http.ListenAndServe(":8088", nil)
}

```

上面的代码演示了如何编写一个REST的应用，我们通过get请求来访问对应的函数获取到了我传入的数据，这是离我们使用的是1个三方库

三方库安装

```
$ go get github.com/julienschmidt/httprouter
```

总结

REST是一种架构风格，汲取了WWW的成功经验：无状态，以资源为中心，充分利用HTTP协议和URI协议，提供统一的接口定义，使得它作为一种设计Web服务的方法而变得流行。在某种意义上，通过强调URI和HTTP等早期Internet标准，REST是对大型应用程序服务器时代之前的Web方式的回归。目前Go对于REST的支持还是很简单的，通过实现自定义的路由规则，我们就可以通过不同的请求方法访问不同的函数。这样就实现了REST的架构。

web服务的调整

修改main.go

```
package main

import (
    "github.com/micro/go-log"
    "net/http"
    "github.com/micro/go-web"
    "sss/ihomeweb/handler"
    "github.com/julienschmidt/httprouter"
)

func main() {
    //创建web服务
        service := web.NewService(
                web.Name("go.micro.web.ihomeweb"),
                web.Version("latest"),
                web.Address(":8999"),
        )
    // 初始化服务
        if err := service.Init(); err != nil {
                log.Fatal(err)
        }
    rou := httprouter.New()
    //映射静态页面
    rou.NotFound = http.FileServer(http.Dir("html"))

    //后续陆续添加服务所以这个文件的这个地方会一直添加内容


    // 注册服务
```

```
    service.Handle("/", rou)

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

```

创建工具函数文件夹

```
#创建工具函数文件夹
$ mkdir utils
# 进入文件夹创建文件
$ cd utils
# 配置文件读取函数文件
$ vim config.go
# 错误码文件
$ vim error.go
# 字符串拼接文件
$ vim misc.go

```

配置文件读取文件config.go

```
package utils

import (
    "github.com/astaxie/beego"
    //使用了beego框架的配置文件读取模块
    "github.com/astaxie/beego/config"
)

var (
    G_server_name  string //项目名称
    G_server_addr  string //服务器ip地址
    G_server_port  string //服务器端口
    G_redis_addr   string //redis ip地址
    G_redis_port   string //redis port端口
    G_redis_dbnum  string //redis db 编号
    G_mysql_addr   string //mysql ip 地址
    G_mysql_port   string //mysql 端口
    G_mysql_dbname string //mysql db name
    G_fastdfs_port   string //fastdfs 端口
    G_fastdfs_addr string //fastdfs ip
)


func InitConfig() {
    //从配置文件读取配置信息
```

```
    appconf, err := config.NewConfig("ini", "./conf/app.conf")
    if err != nil {
        beego.Debug(err)
        return
    }
    G_server_name = appconf.String("appname")
    G_server_addr = appconf.String("httpaddr")
    G_server_port = appconf.String("httpport")
    G_redis_addr = appconf.String("redisaddr")
    G_redis_port = appconf.String("redisport")
    G_redis_dbnum = appconf.String("redisdbnum")
    G_mysql_addr = appconf.String("mysqladdr")
    G_mysql_port = appconf.String("mysqlport")
    G_mysql_dbname = appconf.String("mysqldbname")
    G_fastdfs_port  = appconf.String("fastdfsport")
    G_fastdfs_addr = appconf.String("fastdfsaddr")
    return
}

func init() {
    InitConfig()
}

```

错误码文件

```
package utils

const (
    RECODE_OK          = "0"
    RECODE_DBERR       = "4001"
    RECODE_NODATA      = "4002"
    RECODE_DATAEXIST   = "4003"
    RECODE_DATAERR     = "4004"
    RECODE_SESSIONERR  = "4101"
    RECODE_LOGINERR    = "4102"
    RECODE_PARAMERR    = "4103"
    RECODE_USERERR     = "4104"
    RECODE_ROLEERR     = "4105"
    RECODE_PWDERR      = "4106"
    RECODE_SMSERR      = "4017"
    RECODE_REQERR      = "4201"
    RECODE_IPERR       = "4202"
    RECODE_THIRDERR    = "4301"
    RECODE_IOERR       = "4302"
    RECODE_SERVERERR   = "4500"
    RECODE_UNKNOWERR   = "4501"
)

var recodeText = map[string]string{
    RECODE_OK:          "成功",
    RECODE_DBERR:       "数据库查询错误",
    RECODE_NODATA:      "无数据",
```

```
    RECODE_DATAEXIST:  "数据已存在",
    RECODE_DATAERR:    "数据错误",
    RECODE_SESSIONERR: "用户未登录",
    RECODE_LOGINERR:   "用户登录失败",
    RECODE_PARAMERR:   "参数错误",
    RECODE_USERERR:    "用户不存在或未激活",
    RECODE_ROLEERR:    "用户身份错误",
    RECODE_PWDERR:     "密码错误",
    RECODE_REQERR:     "非法请求或请求次数受限",
    RECODE_IPERR:      "IP受限",
    RECODE_THIRDERR:   "第三方系统错误",
    RECODE_IOERR:      "文件读写错误",
    RECODE_SERVERERR:  "内部错误",
    RECODE_UNKNOWERR:  "未知错误",
    RECODE_SMSERR:     "短信失败",
}

func RecodeText(code string) string {
    str, ok := recodeText[code]
    if ok {
        return str
    }
    return recodeText[RECODE_UNKNOWERR]
}
```

字符串拼接文件

```
package utils

/* 将url加上 http://IP:PROT/  前缀 */
//http:// + 127.0.0.1 + : + 8080 + 请求
func AddDomain2Url(url string) (domain_url string) {
    domain_url = "http://" + G_fastdfs_addr + ":" + G_fastdfs_port + "/" + url

    return domain_url
}
```

创建数据库文件

```
$ mkdir models
#创建数据库文件
$ vim models.go
```

models.go文件基础内容

```
import (
   "github.com/astaxie/beego/orm"
   _ "github.com/go-sql-driver/mysql"
)
func init()  {
}
```

models.go文件包的导入

```
import (
    //beego的orm模块
    "github.com/astaxie/beego/orm"
    //go操作数据库的模块
    _ "github.com/go-sql-driver/mysql"
    //工具模块
    "sss/ihomeweb/utils"
    //时间包
    "time"
    //beego
    "github.com/astaxie/beego"
)
```

models.go的表单与辅助工具

```
/* 用户 table_name = user */
type User struct {
Id            int           `json:"user_id"`                      //用户编号
Name          string        `orm:"size(32)"  json:"name"`         //用户昵称
Password_hash string        `orm:"size(128)" json:"password"`     //用户密码加密的
Mobile        string        `orm:"size(11);unique"  json:"mobile"` //手机号
Real_name     string        `orm:"size(32)" json:"real_name"`     //真实姓名   实名认证
Id_card       string        `orm:"size(20)" json:"id_card"`       //身份证号   实名认证
Avatar_url    string        `orm:"size(256)" json:"avatar_url"`   //用户头像路径 通过fastdfs进行图片存储
Houses        []*House      `orm:"reverse(many)" json:"houses"`   //用户发布的房屋信息  一个人多套房
Orders        []*OrderHouse `orm:"reverse(many)" json:"orders"`   //用户下的订单 一个人多次订单
}

/* 房屋信息 table_name = house */
type House struct {
Id              int           `json:"house_id"`
       //房屋编号
User            *User         `orm:"rel(fk)" json:"user_id"`
      //房屋主人的用户编号  与用户进行关联
Area            *Area         `orm:"rel(fk)" json:"area_id"`
      //归属地的区域编号   和地区表进行关联
Title           string        `orm:"size(64)" json:"title"`
       //房屋标题
```

```
Price           int           `orm:"default(0)" json:"price"`
       //单价,单位:分    每次的价格要乘以100
Address         string        `orm:"size(512)" orm:"default("")" json:"address"`
      //地址
Room_count      int           `orm:"default(1)" json:"room_count"`
      //房间数目
Acreage         int           `orm:"default(0)" json:"acreage"`
       //房屋总面积
Unit            string        `orm:"size(32)" orm:"default("")" json:"unit"`
      //房屋单元,如 几室几厅
Capacity        int           `orm:"default(1)" json:"capacity"`
      //房屋容纳的总人数
Beds            string        `orm:"size(64)" orm:"default("")" json:"beds"`
      //房屋床铺的配置
Deposit         int           `orm:"default(0)" json:"deposit"`
       //押金
Min_days        int           `orm:"default(1)" json:"min_days"`
      //最少入住的天数
Max_days        int           `orm:"default(0)" json:"max_days"`
      //最多入住的天数 0表示不限制
Order_count     int           `orm:"default(0)" json:"order_count"`
       //预定完成的该房屋的订单数
Index_image_url string        `orm:"size(256)" orm:"default("")"
json:"index_image_url"` //房屋主图片路径
Facilities      []*Facility    `orm:"reverse(many)" json:"facilities"`
       //房屋设施    与设施表进行关联
Images          []*HouseImage `orm:"reverse(many)" json:"img_urls"`
       //房屋的图片    除主要图片之外的其他图片地址
Orders          []*OrderHouse `orm:"reverse(many)" json:"orders"`
       //房屋的订单     与房屋表进行管理
Ctime           time.Time     `orm:"auto_now_add;type(datetime)" json:"ctime"`
}

//首页最高展示的房屋数量
var HOME_PAGE_MAX_HOUSES int = 5

//房屋列表页面每页显示条目数
var HOUSE_LIST_PAGE_CAPACITY int = 2
//处理房子信息
func (this *House) To_house_info() interface{} {
    house_info := map[string]interface{}{
    "house_id":    this.Id,
    "title":       this.Title,
    "price":       this.Price,
    "area_name":   this.Area.Name,
    "img_url":     utils.AddDomain2Url(this.Index_image_url),
    "room_count":  this.Room_count,
    "order_count": this.Order_count,
    "address":     this.Address,
    "user_avatar": utils.AddDomain2Url(this.User.Avatar_url),
    "ctime":       this.Ctime.Format("2006-01-02 15:04:05"),
    }

```

```
    return house_info
}
//处理1个房子的全部信息
func (this *House) To_one_house_desc() interface{} {
    house_desc := map[string]interface{}{
    "hid":          this.Id,
    "user_id":      this.User.Id,
    "user_name":    this.User.Name,
    "user_avatar": utils.AddDomain2Url(this.User.Avatar_url),
    "title":        this.Title,
    "price":        this.Price,
    "address":      this.Address,
    "room_count":   this.Room_count,
    "acreage":      this.Acreage,
    "unit":         this.Unit,
    "capacity":     this.Capacity,
    "beds":         this.Beds,
    "deposit":      this.Deposit,
    "min_days":     this.Min_days,
    "max_days":     this.Max_days,
    }

    //房屋图片
    img_urls := []string{}
    for _, img_url := range this.Images {
    img_urls = append(img_urls, utils.AddDomain2Url(img_url.Url))
    }
    house_desc["img_urls"] = img_urls

    //房屋设施
    facilities := []int{}
    for _, facility := range this.Facilities {
    facilities = append(facilities, facility.Id)
    }
    house_desc["facilities"] = facilities

    //评论信息

    comments := []interface{}{}
    orders := []OrderHouse{}
    o := orm.NewOrm()
    order_num, err := o.QueryTable("order_house").Filter("house__id",
this.Id).Filter("status", ORDER_STATUS_COMPLETE).OrderBy("-
ctime").Limit(10).All(&orders)
    if err != nil {
        beego.Error("select orders comments error, err =", err, "house id = ",
this.Id)
    }
    for i := 0; i < int(order_num); i++ {
    o.LoadRelated(&orders[i], "User")
    var username string
    if orders[i].User.Name == "" {
    username = "匿名用户"
```

```
    } else {
    username = orders[i].User.Name
    }

    comment := map[string]string{
    "comment":   orders[i].Comment,
    "user_name": username,
    "ctime":     orders[i].Ctime.Format("2006-01-02 15:04:05"),
    }
    comments = append(comments, comment)
    }
    house_desc["comments"] = comments

    return house_desc
}

/* 区域信息 table_name = area */  //区域信息是需要我们手动添加到数据库中的
type Area struct {
Id     int      `json:"aid"`                          //区域编号      1    2
Name   string   `orm:"size(32)" json:"aname"`        //区域名字      昌平 海淀
Houses []*House `orm:"reverse(many)" json:"houses"` //区域所有的房屋    与房屋表进行关联
}

/* 设施信息 table_name = "facility"*/     //设施信息 需要我们提前手动添加的
type Facility struct {
Id     int      `json:"fid"`     //设施编号
Name   string   `orm:"size(32)"` //设施名字
Houses []*House `orm:"rel(m2m)"` //都有哪些房屋有此设施   与房屋表进行关联的
}

/* 房屋图片 table_name = "house_image"*/
type HouseImage struct {
Id    int    `json:"house_image_id"`         //图片id
Url   string `orm:"size(256)" json:"url"`    //图片url     存放我们房屋的图片
House *House `orm:"rel(fk)" json:"house_id"` //图片所属房屋编号
}

const (
ORDER_STATUS_WAIT_ACCEPT  = "WAIT_ACCEPT"  //待接单
ORDER_STATUS_WAIT_PAYMENT = "WAIT_PAYMENT" //待支付
ORDER_STATUS_PAID         = "PAID"         //已支付
ORDER_STATUS_WAIT_COMMENT = "WAIT_COMMENT" //待评价
ORDER_STATUS_COMPLETE     = "COMPLETE"     //已完成
ORDER_STATUS_CANCELED     = "CONCELED"     //已取消
ORDER_STATUS_REJECTED     = "REJECTED"     //已拒单
)

/* 订单 table_name = order */
type OrderHouse struct {
Id          int       `json:"order_id"`               //订单编号
User        *User     `orm:"rel(fk)" json:"user_id"`  //下单的用户编号    //与用户表进行关联
```

```
House       *House    `orm:"rel(fk)" json:"house_id"` //预定的房间编号    //与房屋信息进行关联
Begin_date  time.Time `orm:"type(datetime)"`          //预定的起始时间
End_date    time.Time `orm:"type(datetime)"`          //预定的结束时间
Days        int       //预定总天数
House_price int       //房屋的单价
Amount      int       //订单总金额
Status      string    `orm:"default(WAIT_ACCEPT)"`                 //订单状态
Comment     string    `orm:"size(512)"`                            //订单评论
Ctime       time.Time `orm:"auto_now;type(datetime)" json:"ctime"` //每次更新此表，都会更新这个字段
Credit      bool                                                     //表示个人征信情况 true表示良好
}
//处理订单信息
func (this *OrderHouse) To_order_info() interface{} {
    order_info := map[string]interface{}{
    "order_id":   this.Id,
    "title":      this.House.Title,
    "img_url":    utils.AddDomain2Url(this.House.Index_image_url),
    "start_date": this.Begin_date.Format("2006-01-02 15:04:05"),
    "end_date":   this.End_date.Format("2006-01-02 15:04:05"),
    "ctime":      this.Ctime.Format("2006-01-02 15:04:05"),
    "days":       this.Days,
    "amount":     this.Amount,
    "status":     this.Status,
    "comment":    this.Comment,
    "credit":     this.Credit,
    }

    return order_info
}

```

models.go的函数

```
func init() {
    //注册mysql的驱动
    orm.RegisterDriver("mysql", orm.DRMySQL)

    // 设置默认数据库
    orm.RegisterDataBase("default", "mysql", "root:1@tcp("+utils.G_mysql_addr+":"+utils.G_mysql_port+")/micropgc?charset=utf8", 30)

    //注册model
    orm.RegisterModel(new(User), new(House), new(Area), new(Facility), new(HouseImage), new(OrderHouse))

    // create table
    //第一个是别名
    // 第二个是是否强制替换模块    如果表变更就将false 换成true 之后再换回来表就便更好来了
    //第三个参数是如果没有则同步或创建
```

```
    orm.RunSyncdb("default", false, true)

}

```

运行服务并且创建表单

```
#创建conf文件夹用来存放配置文件
$ mkdir conf
#创建data.sql文件
$ vim data.sql
```

datat.sql文件内容

```
INSERT INTO `area`(`name`) VALUES ('东城区'),('西城区'),('朝阳区'),('海淀区'),('昌平区'),('丰台区'),('房山区'),('通州区'),('顺义区'),('大兴区'),('怀柔区'),('平谷区'),('密云区'),('延庆区'),('石景山区');
INSERT INTO `facility`(`name`) VALUES('无线网络'),('热水淋浴'),('空调'),('暖气'),('允许吸烟'),('饮水设备'),('牙具'),('香皂'),('拖鞋'),('手纸'),('毛巾'),('沐浴露、洗发露'),('冰箱'),('洗衣机'),('电梯'),('允许做饭'),('允许带宠物'),('允许聚会'),('门禁系统'),('停车位'),('有线网络'),('电视'),('浴缸'),('吃鸡'),('打台球');
```

登入mysql进行数据导入

```
#登录mysql
$ mysql -uroot -p
#输入root密码
Mysql> use go1micro
#数据的导入
mysql> source ./conf/data.sql
#数据检查
mysql> select * from area;
mysql> select * from facility;

```

创建1个app.conf文件

app.conf文件内容

```
#应用名称
appname = ihome
#地址
httpaddr = 127.0.0.1
#端口
httpport = 8080
#数据库地址
mysqladdr = 192.168.110.20
#数据库端口
mysqlport = 3306

```

导入前端页面

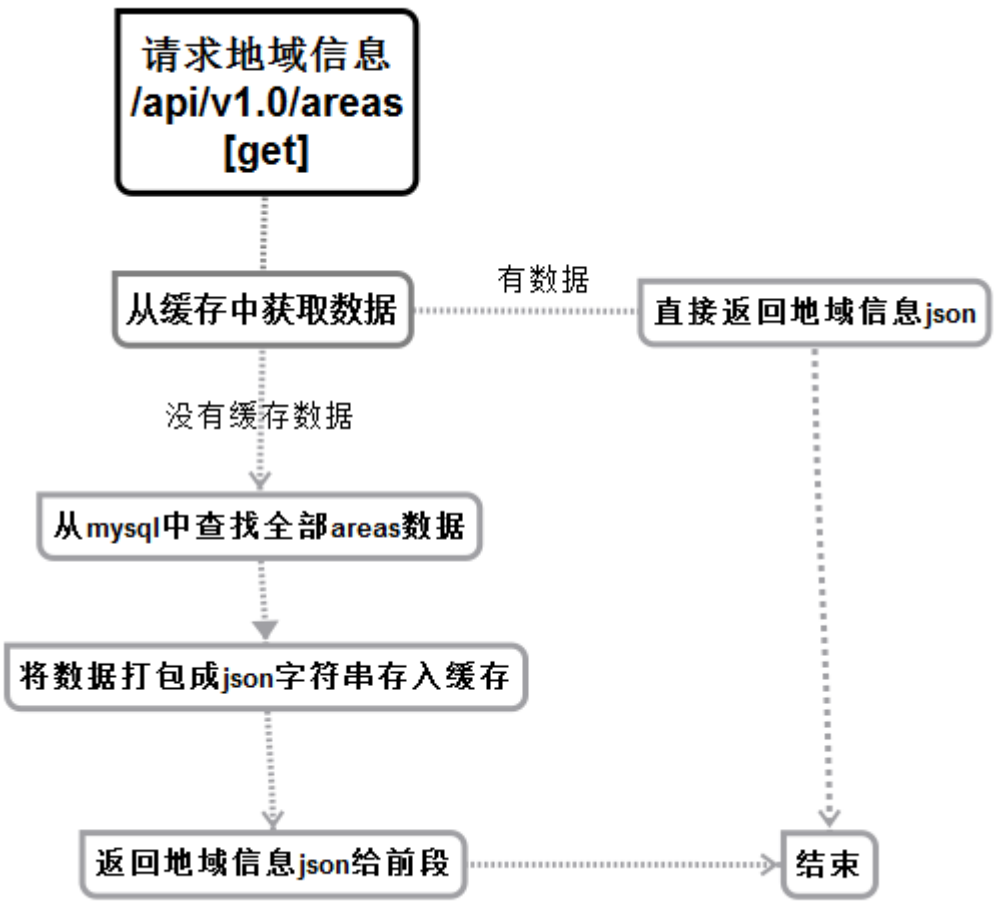
```
将html压缩到拷贝到项目将内容替换掉html文件夹中的内容
$unzip html

```

02 获取地区信息

创建命令

```
$ micro new --type "srv" sss/GetArea
```

流程与接口



```
#Request:
method: GET
url:api/v1.0/areas
#data:
no input data
#Response
#返回成功:
{
    "errno": 0,
    "errmsg":"OK",
    "data": [
    {"aid": 1, "aname": "东城区"},
    {"aid": 2, "aname": "西城区"},
    {"aid": 3, "aname": "通州区"},
    {"aid": 4, "aname": "顺义区"}]
    //...
}
#注册失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}

```

redis的使用

安装redis

```
#下载
$ wget http://download.redis.io/releases/redis-4.0.9.tar.gz
#解压
$ tar xzf redis-4.0.9.tar.gz
#进入
$ cd redis-4.0.9
#编译
$ make
#安装
$ sudo make install
#验证
$ redis-cli
Could not connect to Redis at 127.0.0.1:6379: Connection refused
Could not connect to Redis at 127.0.0.1:6379: Connection refused
not connected>


```

redis启动

```
#将redis安装包的redis.conf文件复制到我们项目中ihome服务的conf文件当中
#修改redis.conf文件
#69行
69 bind 127.0.0.1 当前主机ip
#136行修改为yes表示守护进程启动
 136 daemonize yes

#在ihome服务中创建一个启动文件
$ vim server.sh
#文件内容
redis-server ./conf/redis.conf
#给文件赋予启动权限
$ chmod 777  server.sh
```

安装go语言redis aip驱动

```
$ go get -v -u github.com/gomodule/redigo/redis
$ go get -v -u github.com/garyburd/redigo`
```

安装beego的cache缓存模块

```
$ go   get github.com/astaxie/beego/cache
```

proto文件

```
service Example {
    rpc GetArea(Request) returns (Response) {}
}

message Request {
}

message Response {
    //返回错误码
    string Errno = 1;
    //返回错误信息
    string Errmsg = 2;
    //返回数据类型
    message Address{
        int32 aid = 1;
        string aname = 2;
    }
    //用自定义类型返回的数组
    repeated Address Data = 3;
}
```

web服务main.go添加3条路由

```
//获取地区信息
rou.GET("/api/v1.0/areas",handler.GetArea)
//下面两个目前并不实现服务
//获取session
rou.GET("/api/v1.0/session",handler.GetSession)
//获取index
rou.GET("/api/v1.0/house/index",handler.GetIndex)
```

web服务handler.go添加三个客户端函数

```
//获取地区
func GetArea(w http.ResponseWriter, r *http.Request,_ httprouter.Params) {
    beego.Info("获取地区请求客户端 url: api/v1.0/areas")

    //创建新的grpc返回句柄
    server :=grpc.NewService()
    //服务出初始化
    server.Init()


    //创建获取地区的服务并且返回句柄
    exampleClient := GETAREA.NewExampleService("go.micro.srv.GetArea",
server.Client())

    //调用函数并且获得返回数据
    rsp, err := exampleClient.GetArea(context.TODO(), &GETAREA.Request{})
    if err != nil {
        http.Error(w, err.Error(), 502)
        return
    }
    //创建返回类型的切片
    area_list := []models.Area{}
    //循环读取服务返回的数据
    for _,value := range rsp.Data{
        tmp :=models.Area{Id:int(value.Aid),Name:value.Aname,Houses:nil}
        area_list = append(area_list,tmp)
    }
    //创建返回数据map
    response := map[string]interface{}{
        "errno": rsp.Errno,
        "errmsg": rsp.Errmsg,
        "data" : area_list,
    }
    //注意的点
    w.Header().Set("Content-Type", "application/json")

    // 将返回数据map发送给前端
    if err := json.NewEncoder(w).Encode(response); err != nil {
        http.Error(w, err.Error(), 503)
```

```
        return
    }
}


//获取session
func GetSession(w http.ResponseWriter, r *http.Request,_ httprouter.Params) {
    beego.Info("获取Session url: api/v1.0/session")

    //创建返回数据map
    response := map[string]interface{}{
        "errno": utils.RECODE_SESSIONERR,
        "errmsg": utils.RecodeText(utils.RECODE_SESSIONERR),

    }
    w.Header().Set("Content-Type", "application/json")

    // 将返回数据map发送给前端
    if err := json.NewEncoder(w).Encode(response); err != nil {
        http.Error(w, err.Error(), 503)
        return
    }
}

//获取首页轮播
func GetIndex(w http.ResponseWriter, r *http.Request,_ httprouter.Params) {
    beego.Info("获取首页轮播 url: api/v1.0/houses/index")

    //创建返回数据map
    response := map[string]interface{}{
        "errno": utils.RECODE_OK,
        "errmsg": utils.RecodeText(utils.RECODE_OK),

    }
    w.Header().Set("Content-Type", "application/json")

    // 将返回数据map发送给前端
    if err := json.NewEncoder(w).Encode(response); err != nil {
        http.Error(w, err.Error(), 503)
        return
    }
}
```

GetArea服务handler.go

```
package handler
import (
    "context"

    "github.com/micro/go-log"

    example "go-1/GetArea/proto/example"
    "github.com/astaxie/beego"
```

```
    "go-1/homeweb/utils"
    "encoding/json"
    "github.com/astaxie/beego/cache"
    _ "github.com/astaxie/beego/cache/redis"
    _ "github.com/gomodule/redigo/redis"
    "github.com/astaxie/beego/orm"
    "go-1/homeweb/models"
    "time"
)


func (e *Example) GetArea(ctx context.Context, req *example.Request, rsp
*example.Response) error {
    beego.Info(" GetArea    api/v1.0/areas !!!")

    //初始化返回值
    rsp.Errno = utils.RECODE_OK
    rsp.Errmsg = utils.RecodeText(rsp.Errno)

    //连接redis创建句柄
    redis_config_map := map[string]string{
        "key":utils.G_server_name,

        "conn":utils.G_redis_addr+":"+utils.G_redis_port,
        "dbNum":utils.G_redis_dbnum,
    }
    //确定连接信息
    beego.Info(redis_config_map)
    //将map转化为json
    redis_config ,_:=json.Marshal(redis_config_map)
    //连接redis
    bm, err := cache.NewCache("redis", string(redis_config) )
    if err != nil {
        beego.Info("缓存创建失败",err)
        rsp.Errno  =  utils.RECODE_DBERR
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
        return  nil
    }
    /*1获取缓存数据*/
    areas_info_value:=bm.Get("areas_info")
    //如果不为空则说明成功
    if areas_info_value  !=nil{
        beego.Info("获取到缓存发送给前端")

        //用来存放解码的json
        ares_info := []map[string]interface{}{}
        //解码
        err = json.Unmarshal(areas_info_value.([]byte),&ares_info)

        //进行循环赋值
        for key, value := range ares_info {

            beego.Info(key,value)
```

```
            //创建对于数据类型并进行赋值
            area := example.Response_Address{Aid :int32(value["aid"].(float64)),
Aname :value["aname"].(string)}

            //递增到切片
            rsp.Data = append(rsp.Data,&area)
        }

        return nil
    }
    beego.Info("没有拿到缓存")

    /*2如果没有缓存我们就从mysql 里进行查询*/

    //orm的操作创建orm句柄
    o:=orm.NewOrm()

    //接受地区信息的切片
    var areas []models.Area
    //创建查询条件
    qs:= o.QueryTable("area")
    //查询全部地区
    num ,err :=qs.All(&areas)
    if err != nil {
        rsp.Errno  =  utils.RECODE_DBERR
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
        return nil
    }
    if num == 0 {
        rsp.Errno  =  utils.RECODE_NODATA
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
        return nil
    }

    beego.Info("写入缓存")

    /*3获取数据写入缓存*/

    //将查询到的数据编码成json格式
    ares_info_str,_:=json.Marshal(areas)

    //Put(key string, val interface{}, timeout time.Duration) error
    //存入缓存中
    err =bm.Put("areas_info",ares_info_str,time.Second*3600)
    if err != nil {
        beego.Info("数据库中查出数据信息存入缓存中失误",err)
        rsp.Errno  =  utils.RECODE_NODATA
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
        return nil
    }

    //返回地区信息
    for key, value := range areas {
```

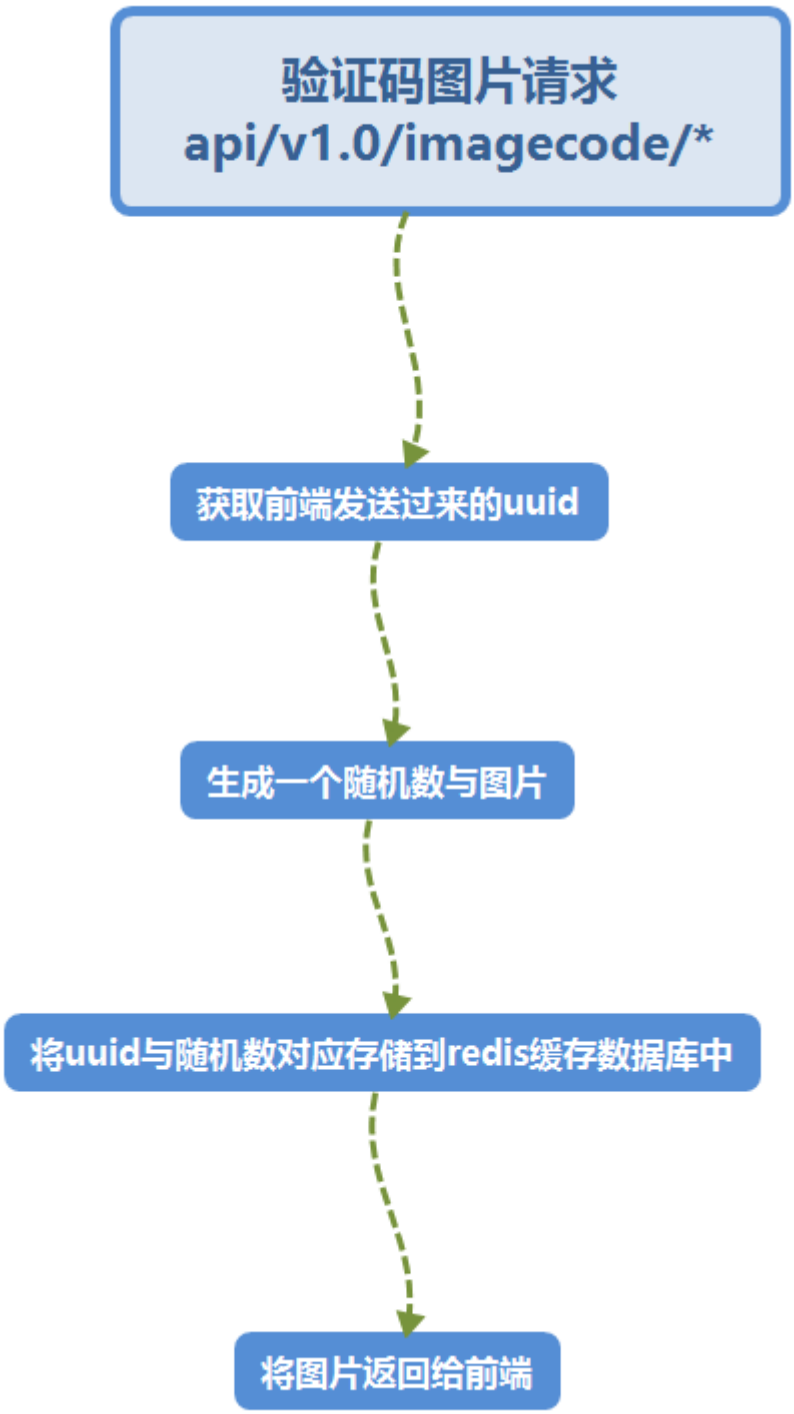
```
        beego.Info(key,value)

        area := example.Response_Address{Aid :int32(value.Id), Aname
:string(value.Name)}
        //var area  example.ResponseArea_Address
        //area.Aid = value["aid"].(int32)
        //area.Aname = value["aname"].(string)
        rsp.Data = append(rsp.Data,&area)
    }

    return nil
}

```

03 获取验证码图片

创建命令

```
$ micro new --type "srv" sss/GetImageCd
```

流程与接口




```
#Request:
method: GET
url:api/v1.0/imagecode/:uuid
#data:
no input data
#Response
#返回成功:
{
    二进制图片
}
#注册失败:
```

```
{
    "errno": "4001",   //状态码
    "errmsg":"状态错误信息"
}

```

使用验证码库

```
$ go get -u -v github.com/afocus/captcha
```

接口api

```
    //创建1个句柄
    cap := captcha.New()
    //通过句柄调用 字体文件
    if err := cap.SetFont("comic.ttf"); err != nil {
        panic(err.Error())
    }

    //设置图片的大小
    cap.SetSize(91, 41)
    // 设置干扰强度
    cap.SetDisturbance(captcha.MEDIUM)
    // 设置前景色 可以多个 随机替换文字颜色 默认黑色
    //SetFrontColor(colors ...color.Color)  这两个颜色设置的函数属于不定参函数
    cap.SetFrontColor(color.RGBA{255, 255, 255, 255})
    // 设置背景色 可以多个 随机替换背景色 默认白色
    cap.SetBkgColor(color.RGBA{255, 0, 0, 255}, color.RGBA{0, 0, 255, 255},
color.RGBA{0, 153, 0, 255})
    //生成图片 返回图片和 字符串(图片内容的文本形式)
    img, str := cap.Create(4, captcha.NUM)
```

proto文件

```
service Example {
    rpc GetImageCd(Request) returns (Response) {}
}

message Request {
    string uuid = 1;
}

message Response {
    //错误码
    string Errno =1 ;
    //错误消息
    string Errmsg = 2;
    //图片结构信息
    bytes Pix = 3;
```

```
16
17      //图片跨步
18      int64 Stride =4 ;
19
20      message  Point{
21      int64 X =1;
22      int64 Y =2;
23      }
24
25      // Min, Max Point
26      Point Min =5;
27      Point Max =6;
28  }
29
```

web服务main.go添加1条路由

```
1  //获取图片验证码
2  rou.GET("/api/v1.0/imagecode/:uuid",handler.GetImageCd)
```

web服务handler.go添加代码

```
1  func GetImageCd(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
2      beego.Info("获取图片验证码 url: api/v1.0/imagecode/:uuid")
3      //创建服务
4      server :=grpc.NewService()
5      //服务初始化
6      server.Init()
7
8      //连接服务
9      exampleClient := GETIMAGECD.NewExampleService("go.micro.srv.GetImageCd",
   server.Client())
10
11     //获取前端发送过来的唯一uuid
12     beego.Info(ps.ByName("uuid"))
13     //通过句柄调用我们proto协议中准备好的函数
14     //第一个参数为默认,第二个参数 proto协议中准备好的请求包
15     rsp, err := exampleClient.GetImageCd(context.TODO(),&GETIMAGECD.Request{
16         Uuid:ps.ByName("uuid"),
17     })
18     //判断函数调用是否成功
19     if err != nil {
20         beego.Info(err)
21         http.Error(w, err.Error(), 502)
22         return
23     }
24
25     //处理前端发送过来的图片信息
26     var img image.RGBA
27
28     img.Stride = int(rsp.Stride)
29
```

```
    img.Rect.Min.X = int(rsp.Min.X)
    img.Rect.Min.Y = int(rsp.Min.Y)
    img.Rect.Max.X = int(rsp.Max.X)
    img.Rect.Max.Y = int(rsp.Max.Y)

    img.Pix =[]uint8(rsp.Pix)

    var image captcha.Image

    image.RGBA = &img

    //将图片发送给前端
    png.Encode(w, image)
}
```

GetImageCd服务业务代码

```
package handler

import (
    "context"

    "github.com/micro/go-log"

    example "go-1/GetImageCd/proto/example"
    "go-1/homeweb/utils"
    "github.com/astaxie/beego"
    "encoding/json"
    "github.com/astaxie/beego/cache"
    _ "github.com/astaxie/beego/cache/redis"
    _ "github.com/gomodule/redigo/redis"
    "time"
    "github.com/afocus/captcha"
    "image/color"
)

type Example struct{}

// Call is a single request handler called via client.Call or the generated client code
func (e *Example) GetImageCd(ctx context.Context, req *example.Request, rsp *example.Response) error {

    beego.Info("---------------- GET  /api/v1.0/imagecode/:uuid GetImage() ----------")


    //创建1个句柄
    cap := captcha.New()
    //通过句柄调用 字体文件
    if err := cap.SetFont("comic.ttf"); err != nil {
        beego.Info("没有字体文件")
        panic(err.Error())
```

```
    }
    //设置图片的大小
    cap.SetSize(91, 41)
    // 设置干扰强度
    cap.SetDisturbance(captcha.MEDIUM)
    // 设置前景色 可以多个 随机替换文字颜色 默认黑色
    //SetFrontColor(colors ...color.Color)  这两个颜色设置的函数属于不定参函数
    cap.SetFrontColor(color.RGBA{255, 255, 255, 255})
    // 设置背景色 可以多个 随机替换背景色 默认白色
    cap.SetBkgColor(color.RGBA{255, 0, 0, 255}, color.RGBA{0, 0, 255, 255}, color.RGBA{0, 153, 0, 255})
    //生成图片 返回图片和 字符串(图片内容的文本形式)
    img, str := cap.Create(4, captcha.NUM)
    beego.Info(str)


    b := *img //解引用
    c := *(b.RGBA) //解引用
    //成功返回
    rsp.Errno = utils.RECODE_OK
    rsp.Errmsg = utils.RecodeText(rsp.Errno)

    //图片信息
    rsp.Pix = []byte(c.Pix)
    rsp.Stride =  int64(c.Stride)
    rsp.Max  =  &example.Response_Point{X:int64(c.Rect.Max.X),Y:int64(c.Rect.Max.Y)}
    rsp.Min  =  &example.Response_Point{X:int64(c.Rect.Min.X),Y:int64(c.Rect.Min.Y)}



    /*将uuid与 随机数验证码对应的存储在redis缓存中*/
    //初始化缓存全局变量的对象

    redis_config_map := map[string]string{
        "key":"ihome",

        "conn":utils.G_redis_addr+":"+utils.G_redis_port,
        "dbNum":utils.G_redis_dbnum,
    }
    beego.Info(redis_config_map)
    redis_config ,_:=json.Marshal(redis_config_map)



    //连接redis数据库 创建句柄
    bm, err := cache.NewCache("redis", string(redis_config) )
    if err !=nil{
        beego.Info("GetImage()   cache.NewCache err ",err)
        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)
    }
```

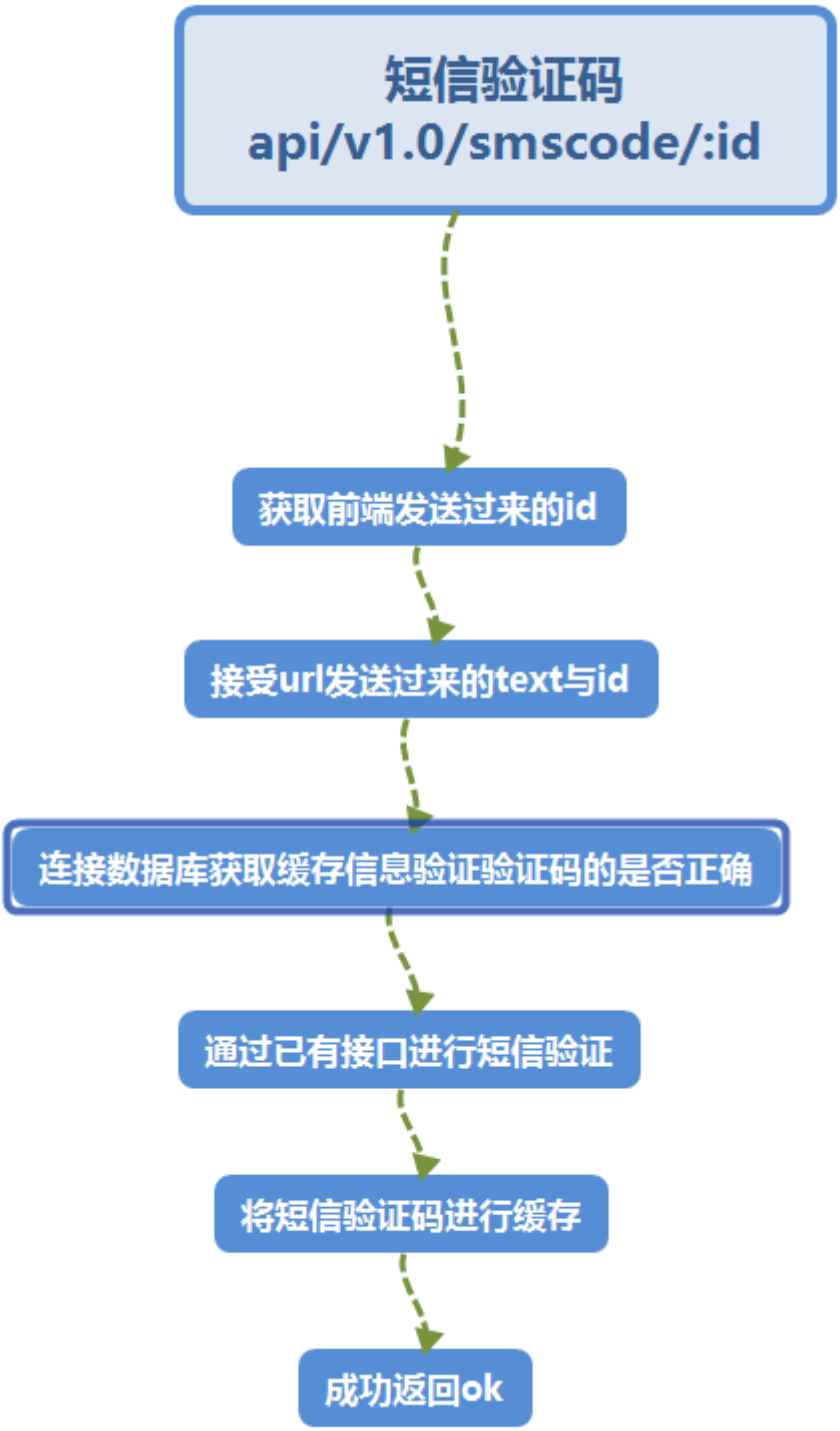
```
84      //验证码进行1个小时缓存
85      bm.Put(req.Uuid ,str, 300 *time.Second )
86
87
88
89      return nil
90  }
```

04 获取短信验证码

创建命令

```
1  $ micro new --type "srv" sss/GetSmscd
```

流程与接口




```
#Request:
method: GET
#111表示的是手机号
url:api/v1.0/smscode/:mobile
url:api/v1.0/smscode/111?  text=248484&id=9cd8faa9-5653-4f7c-b653-0a58a8a98c81
data:no input data
#Response
#返回成功:
{
    "errno": "0",   //状态码
```

```
    "errmsg":"ok"
}
#注册失败:
{
    "errno": "4001",   //状态码
    "errmsg":"状态错误信息"
}

```

使用短信验证的api

```
//网站: https://www.mysubmail.com


//短信map
    messageconfig := make(map[string]string)
    //id
    messageconfig["appid"] = "29672"
    //key
    messageconfig["appkey"] = "89d90165cbea8cae80137d7584179bdb"
    //编码格式
    messageconfig["signtype"] = "md5"


    //短信操作对象
    messagexsend := submail.CreateMessageXSend()
    //短信发送到那个手机号
    submail.MessageXSendAddTo(messagexsend, “手机号”)
    //短信发送的模板
    submail.MessageXSendSetProject(messagexsend, "NQ1J94")
    //发送的验证码
    submail.MessageXSendAddVar(messagexsend, "code", "1111")
    //发送
    fmt.Println("MessageXSend ",
submail.MessageXSendRun(submail.MessageXSendBuildRequest(messagexsend),
messageconfig))
```

proto文件

```
service Example {
    rpc GetSmscd(Request) returns (Response) {}
}

//客户端发送给服务器
message Request {
    //手机号
```

```
    string Mobile = 1;
    //uuid
    string Id = 2;
    //验证码
    string Text = 3;
}
//服务器回发给客户端
message Response {
    string Errno = 1;
    string Errmsg = 2;
}
```

web服务main.go添加1条路由

```
//获取短信验证码
rou.GET("/api/v1.0/smscode/:mobile",handler.Getsmscd)
```

web服务中的handler.go添加内容

```
func Getsmscd(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
    beego.Info(" 获取短信验证   api/v1.0/smscode/:id ")

    //创建服务
    service := grpc.NewService()
    service.Init()

    //获取 前端发送过来的手机号
    mobile := ps.ByName("mobile")
    beego.Info(mobile)
    //后端进行正则匹配
    //创建正则句柄
    myreg := regexp.MustCompile(`0?(13|14|15|17|18|19)[0-9]{9}`)
    //进行正则匹配
    bo := myreg.MatchString(mobile)

    //如果手机号错误则
    if bo == false {
        // we want to augment the response
        resp := map[string]interface{}{
            "errno":  utils.RECODE_NODATA,
            "errmsg": "手机号错误",
        }
        //设置返回数据格式
        w.Header().Set("Content-Type", "application/json")

        //将错误发送给前端
        if err := json.NewEncoder(w).Encode(resp); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        beego.Info("手机号错误返回")
```

```
        return
    }


    //获取url携带的验证码 和key (uuid)
    beego.Info(r.URL.Query())
    //获取url携带的参数
    text := r.URL.Query()["text"][0] //text=248484

    id := r.URL.Query()["id"][0]     //id=9cd8faa9-5653-4f7c-b653-0a58a8a98c81


    //调用服务
    exampleClient := GETSMSCD.NewExampleService("go.micro.srv.GetSmscd",
service.Client())
    rsp, err := exampleClient.GetSmscd(context.TODO(), &GETSMSCD.Request{
        Mobile: mobile,
        Id:     id,
        Text:   text,
    })

    if err != nil {
        http.Error(w, err.Error(), 502)
        beego.Info(err)
        //beego.Debug(err)
        return
    }
    //创建返回map
    resp := map[string]interface{}{
        "errno":  rsp.Errno,
        "errmsg": rsp.Errmsg,
    }
    //设置返回格式
    w.Header().Set("Content-Type", "application/json")

    //将数据回发给前端
    if err := json.NewEncoder(w).Encode(resp); err != nil {
        http.Error(w, err.Error(), 503)
        beego.Info(err)
        return
    }
}
```

服务下的handler

```
package handler

import (
    "context"

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

```
7
8       example "go-1/GetSmscd/proto/example"
9       "strconv"
10      "fmt"
11      "time"
12      "github.com/astaxie/beego"
13      "go-1/homeweb/utils"
14      "encoding/json"
15      "github.com/astaxie/beego/cache"
16      _ "github.com/astaxie/beego/cache/redis"
17      _ "github.com/gomodule/redigo/redis"
18      "github.com/garyburd/redigo/redis"
19      "reflect"
20      "math/rand"
21      "github.com/SubmailDem/submail"
22      "github.com/astaxie/beego/orm"
23      "go-1/homeweb/models"
24  )
25
26  //获取短信验证码
27  func (e *Example) GetSmscd(ctx context.Context, req *example.Request, rsp
    *example.Response) error {
28      beego.Info(" GET smscd  api/v1.0/smscode/:id ")
29      //初始化返回正确的返回值
30      rsp.Errno = utils.RECODE_OK
31      rsp.Errmsg = utils.RecodeText(rsp.Errno)
32      /*验证uuid的缓存*/
33
34
35      //验证手机号
36      o:=orm.NewOrm()
37      user := models.User{Mobile:req.Mobile}
38      err := o.Read(&user)
39      if err==nil{
40          beego.Info("用户已经存在")
41          rsp.Errno  =  utils.RECODE_DBERR
42          rsp.Errmsg  = utils.RecodeText(rsp.Errno)
43          return  nil
44      }
45      beego.Info(err)
46
47      //连接redis数据库
48      redis_config_map := map[string]string{
49          "key":utils.G_server_name,
50          "conn":utils.G_redis_addr+":"+utils.G_redis_port,
51          "dbNum":utils.G_redis_dbnum,
52      }
53      beego.Info(redis_config_map)
54      redis_config ,_:=json.Marshal(redis_config_map)
55      beego.Info( string(redis_config) )
56
57
58      //连接redis数据库 创建句柄
```

```
    bm, err := cache.NewCache("redis", string(redis_config) )

//bm,err:=cache.NewCache("redis",`{"key":"ihome","conn":"127.0.0.1:6379","dbNum":"0"} `)//创建1个缓存句柄
    if err != nil {
        beego.Info("缓存创建失败",err)
        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)
        return nil
    }

    beego.Info(req.Id,reflect.TypeOf(req.Id))
    //查询相关数据

    value:=bm.Get(req.Id)
    if value  ==nil{
        beego.Info("获取到缓存数据查询失败",value)

        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)

        return nil
    }
    beego.Info(value,reflect.TypeOf(value))
    value_str ,_ :=redis.String(value,nil)


    beego.Info(value_str,reflect.TypeOf(value_str))
    //数据对比
    if req.Text != value_str{
        beego.Info("图片验证码 错误 ")
        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)
        return nil
    }







    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    size:=r.Intn(9999)+1001
    beego.Info(size)

    //短信map
    messageconfig := make(map[string]string)
    //id
    messageconfig["appid"] = "29672"
    //key
    messageconfig["appkey"] = "89d90165cbea8cae80137d7584179bdb"
    //编码格式
    messageconfig["signtype"] = "md5"

```

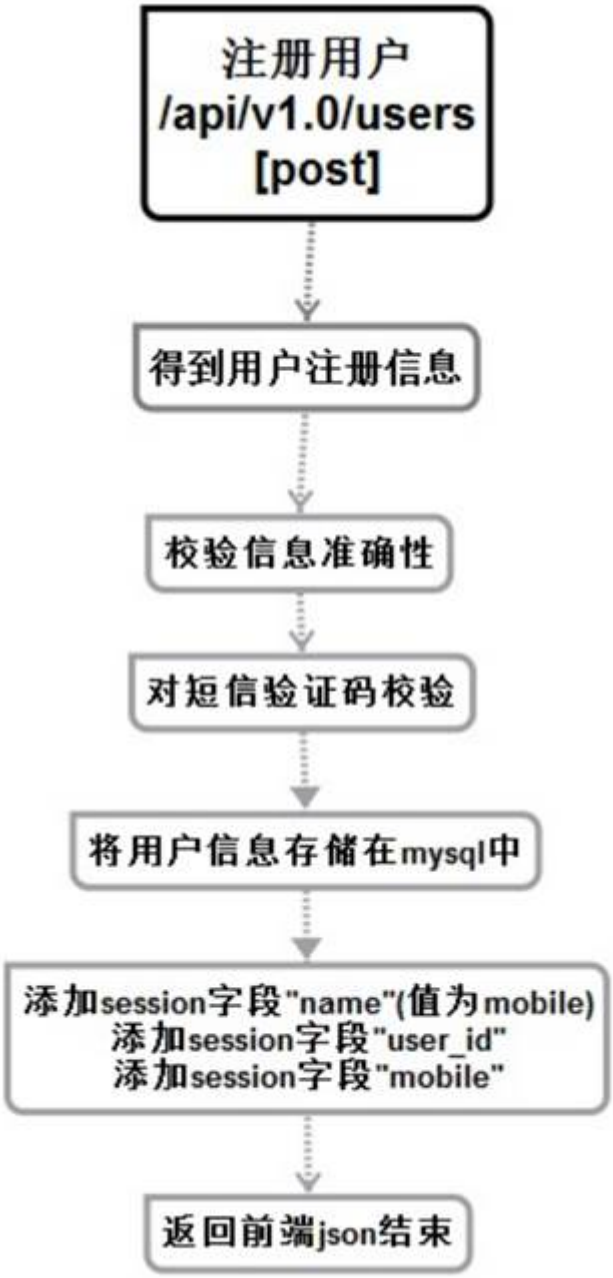
```
110
111     //短信操作对象
112     messagexsend := submail.CreateMessageXSend()
113     //短信发送到那个手机号
114     submail.MessageXSendAddTo(messagexsend, req.Mobile  )
115     //短信发送的模板
116     submail.MessageXSendSetProject(messagexsend, "NQ1J94")
117     //发送的验证码
118     submail.MessageXSendAddVar(messagexsend, "code", strconv.Itoa(size))
119     //发送
120     fmt.Println("MessageXSend ",
    submail.MessageXSendRun(submail.MessageXSendBuildRequest(messagexsend),
    messageconfig))
121
122
123     /*通过手机号将验证短信进行缓存*/
124
125     err = bm.Put(req.Mobile,size,time.Second*300)
126     if err !=nil{
127         beego.Info("缓存出现问题")
128         rsp.Errno = utils.RECODE_DBERR
129         rsp.Errmsg = utils.RecodeText(rsp.Errno)
130         return nil
131     }
132
133     return  nil
134 }
```

05 注册请求

创建命令

```
1 $ micro new --type "srv" sss/PostRet
```

流程和接口




```
#Request:
method: POST
url:api/v1.0/users
#data:
{
    mobile: "123", //手机号
    password: "123", //密码
    sms_code: "123"   //短信验证码
}
#Response
#返回成功:
{
    "errno": "0",   //状态码
    "errmsg":"ok"
}
#注册失败:
```

```
{
    "errno": "4001",   //状态码
    "errmsg":"状态错误信息"
}
```

proto文件

```
service Example {
    rpc PostRet(Request) returns (Response) {}
}


message Request {
    //手机号
    string Mobile = 1;
    //密码
    string Password = 2;
    //短信验证码
    string Sms_code = 3;
}

message Response {
    string Errno = 1;
    string Errmsg = 2;
    string SessionID = 3;
}
```

web服务main.go添加1条路由

```
//注册
rou.POST("/api/v1.0/users",handler.Postreg)
```

web服务的handler.go中

```
//注册请求
func Postreg(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
    beego.Info(" 注册请求   /api/v1.0/users ")
    /*获取前端发送过来的json数据*/
    var request map[string]interface{}
    if err := json.NewDecoder(r.Body).Decode(&request); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }
    for key, value := range request {
        beego.Info(key,value,reflect.TypeOf(value))
    }

    //由于前端每作所以后端进行下操作
    if request["mobile"] == "" || request["password"] =="" || request["sms_code"]
=="" {
```

```
        resp := map[string]interface{}{
            "errno": utils.RECODE_NODATA,
            "errmsg": "信息有误请从新输入",
        }

        //如果不存在直接给前端返回
        w.Header().Set("Content-Type", "application/json")

        if err := json.NewEncoder(w).Encode(resp); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        beego.Info("有数据为空")
        return
    }


    //创建服务
    service := grpc.NewService()
    service.Init()

    // 连接服务将数据发送给注册服务进行注册
    exampleClient := POSTRET.NewExampleService("go.micro.srv.PostRet",
service.Client())
    rsp, err := exampleClient.PostRet(context.TODO(), &POSTRET.Request{
        Mobile:request["mobile"].(string),
        Password:request["password"].(string),
        SmsCode:request["sms_code"].(string),
    })
    if err != nil {
        http.Error(w, err.Error(), 502)

        beego.Info(err)
        //beego.Debug(err)
        return
    }




    resp := map[string]interface{}{
        "errno": rsp.Errno,
        "errmsg": rsp.Errmsg,
    }

    //读取cookie
    cookie,err :=r.Cookie("userlogin")


    //如果读取失败或者cookie的value中不存在则创建cookie
    if err !=nil || ""==cookie.Value{
```

```
        cookie  := http.Cookie{Name: "userlogin", Value: rsp.SessionID, Path: "/", MaxAge: 600}
        http.SetCookie(w, &cookie)
    }

    //设置回发数据格式
    w.Header().Set("Content-Type", "application/json")

    //将数据回发给前端
    if err := json.NewEncoder(w).Encode(resp); err != nil {
        http.Error(w, err.Error(), 503)
        beego.Info(err)
        return
    }


    return
}
```

服务端

```
func GetMd5String(s string) string {
    h := md5.New()
    h.Write([]byte(s))
    return hex.EncodeToString(h.Sum(nil))
}
// Call is a single request handler called via client.Call or the generated client code
func (e *Example) PostRet(ctx context.Context, req *example.Request, rsp *example.Response) error {
    beego.Info(" POST userreg    /api/v1.0/users !!!")
    //初始化错误码
    rsp.Errno = utils.RECODE_OK
    rsp.Errmsg = utils.RecodeText(rsp.Errno)


    //构建连接缓存的数据
    redis_config_map := map[string]string{
        "key":utils.G_server_name,
        "conn":utils.G_redis_addr+":"+utils.G_redis_port,
        "dbNum":utils.G_redis_dbnum,
    }
    redis_config ,_:=json.Marshal(redis_config_map)


    //连接redis数据库 创建句柄
    bm, err := cache.NewCache("redis", string(redis_config) )
    if err != nil {
        beego.Info("缓存创建失败",err)
        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)
```

```
        return nil
    }

    //查询相关数据
    value:=bm.Get(req.Mobile)
    if value  ==nil{
        beego.Info("获取到缓存数据查询失败",value)
        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)

        return nil
    }
    beego.Info(value,reflect.TypeOf(value))
    //进行解码
    var info interface{}
    json.Unmarshal(value.([]byte),&info)
    beego.Info(info,reflect.TypeOf(info))


    //类型转换
    s := int(info.(float64))
    beego.Info(s,reflect.TypeOf(s))
    s1,err :=strconv.Atoi( req.SmsCode)

    if s1 != s{
        beego.Info("短信验证码错误")
        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)
        return nil
    }

    user := models.User{}
    user.Name = req.Mobile   //就用手机号登陆
    //密码正常情况下 md5 sha256 sm9  存入数据库的是你加密后的编码不是明文存入
    //user.Password_hash = GetMd5String(req.Password)
    user.Password_hash = req.Password
    user.Mobile = req.Mobile
    //创建数据库剧本
    o:=orm.NewOrm()
    //插入数据库
    id,err:=o.Insert(&user)
    if err != nil {
        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)
        return nil
    }
    beego.Info("id",id)

    //生成sessionID 保证唯一性
    h := GetMd5String(req.Mobile+req.Password)
    //返回给客户端session
    rsp.SessionID = h
```

```
82
83      //拼接key sessionid + name
84      bm.Put(h+"name",string(user.Mobile),time.Second*3600)
85      //拼接key sessionid + user_id
86      bm.Put(h+"user_id", string(user.Id) ,time.Second*3600)
87      //拼接key sessionid + mobile
88      bm.Put(h+"mobile",string(user.Mobile) ,time.Second*3600)
89
90
91      return nil
92  }
93
```

cookie与session（了解）

Web开发中一个很重要的议题就是如何做好用户的整个浏览过程的控制，因为HTTP协议是无状态的，所以用户的每一次请求都是无状态的，我们不知道在整个web操作过程中那些连接与该用户有关系，我们应该如何解决这个问题呢?

Web的经典解决方案就是COOKIE和SESSION，cookie机制是一种客户端机制，把用户的数据保存在客户端，而session是一种服务器机制，服务器会使用一种类似于散列表的结构来保存信息，每一个网站客户都会被分配一个唯一的标识符，即sessionID ，它的存放形式无非是两种：要么经过url传递，要么保存在客户端的cookies里。当然最安全的方式还是将session保存在数据库中，但是这样效率会下降很多。

Session和cookie是浏览器中常见的两个概念，也是比较难以辨析的两个概念，但是它们在浏览需要认证的服务器页面以及页面统计中还是相当关键的。我们先通过日常的应用来了解下session和cookie怎么来的?

首先考虑如何抓取一个访问受限的网页这个问题。例如新浪微博好友的主页，这个人微博页面等

显然，通过浏览器，我们可以手动的输入用户名和密码来访问页面，而所谓的“抓取”，其实就是使用程序来模拟完成同样的工作，因此我们需要了解“登陆”过程到底发生了什么。

当用户来到微博登陆页面，输入用户名密码之后点击登陆后，浏览器会将认证信息POST发送给远端的服务器，服务器执行验证逻辑，然后验证通过。

则浏览器会跳转到登录用户的微博首页

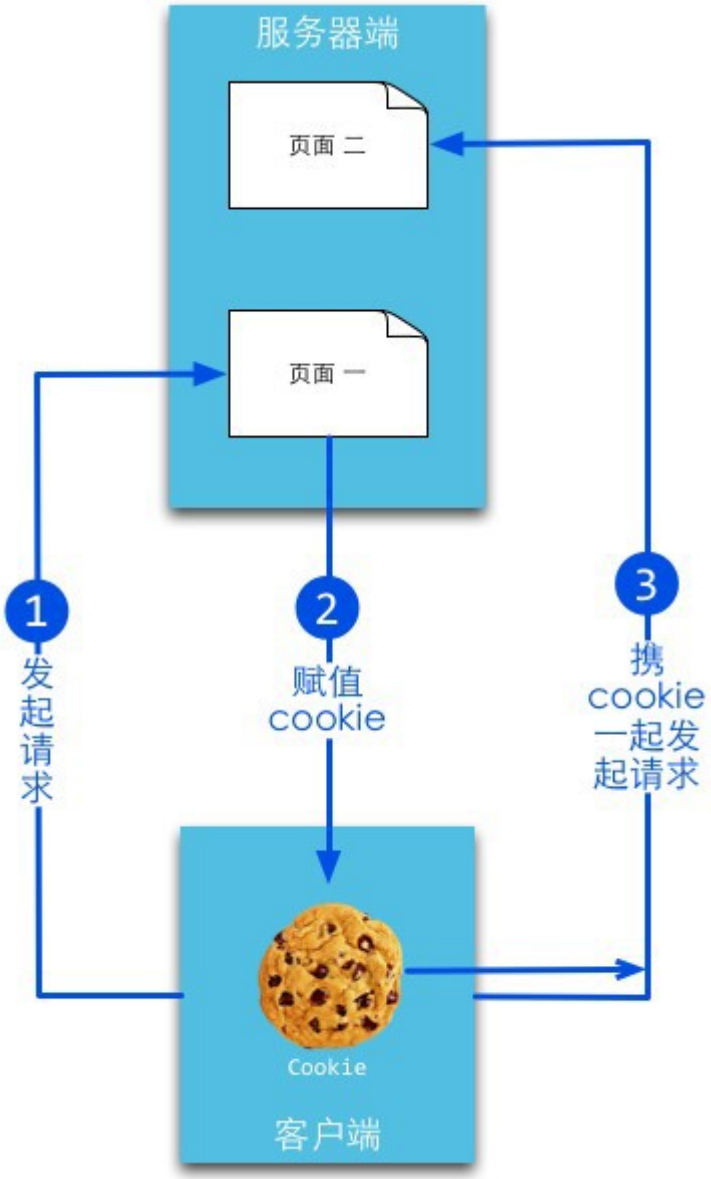
在登陆成功后，浏览器如何验证我们对其他受限制页面的访问？因为HTTP协议是无状态的，所以很显然服务器不可能知道我们已经在上一次的HTTP请求中通过了登陆验证。

当然最简单的解决方案就是所有的请求里面都带上用户名和密码，这样虽然可行，但是大大加重了服务器的负担（对于每个request都需要到数据库中进行验证），也大大降低了用户的体验（每个页面都需要重新输入用户名和密码，每个页面都需要带有登录表单）。

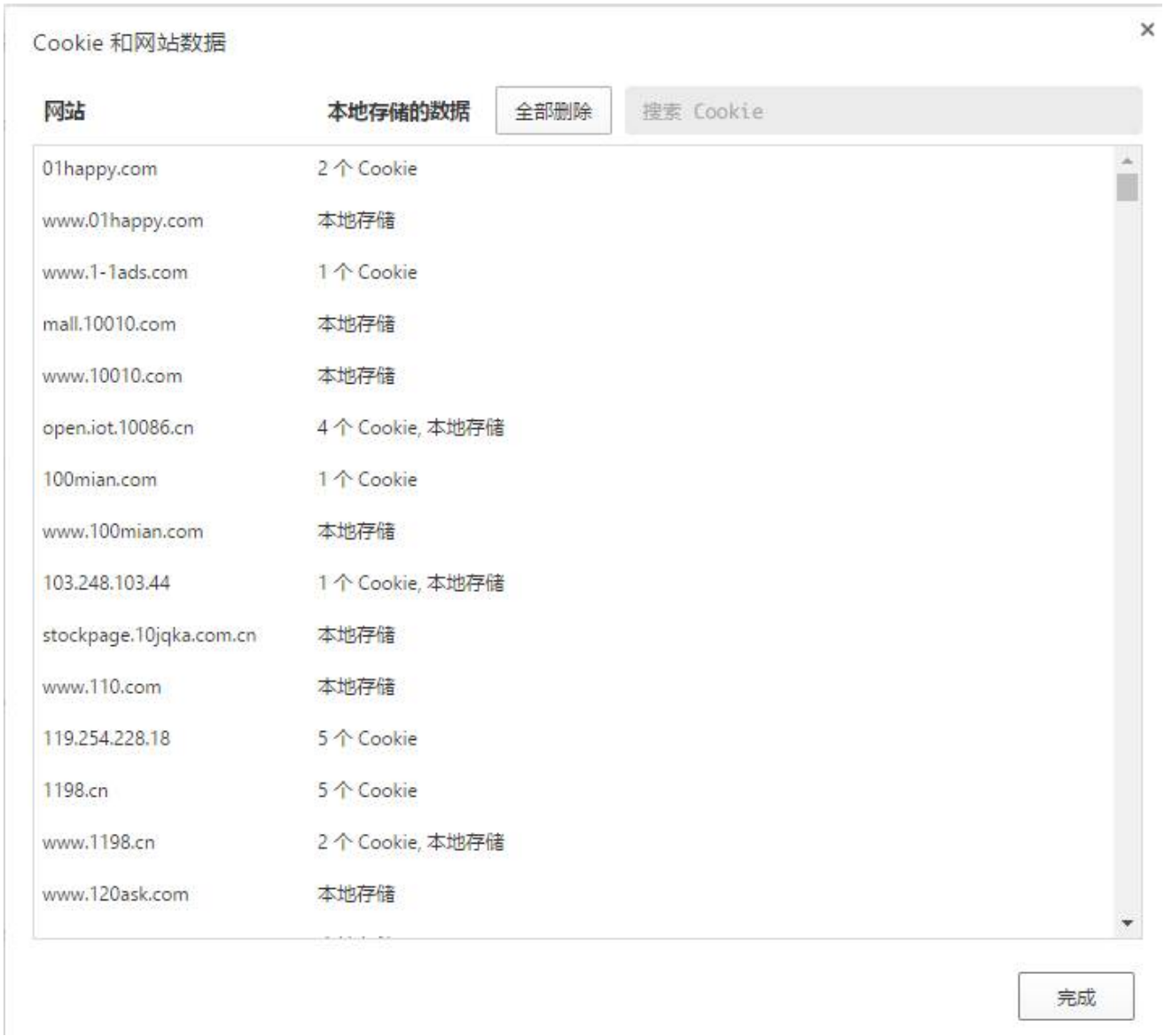
既然直接在请求中带上用户名和密码不行，那么只有在服务器或者客户端保存一些类似的代表身份的信息，所有就有了COOKIE和SESSION。

cookie

Cookie，简而言之就是在本地计算机保存一些用户操作的历史信息（当然也包括登陆信息）并在用户再次访问该站点时浏览器通过HTTP协议将本地cookie内容发送给服务器，从而完成验证，或继续上一步操作。。



Cookie是由浏览器维持的，存储在客户端的一小段文本信息，伴随着用户请求和页面在Web服务器和浏览器之间传递。用户每次访问站点时，Web应用程序都可以读取cookie包含的信息。浏览器设置里面有cookie隐私数据选项，打开它，可以看到很多已访问网站的cookies。



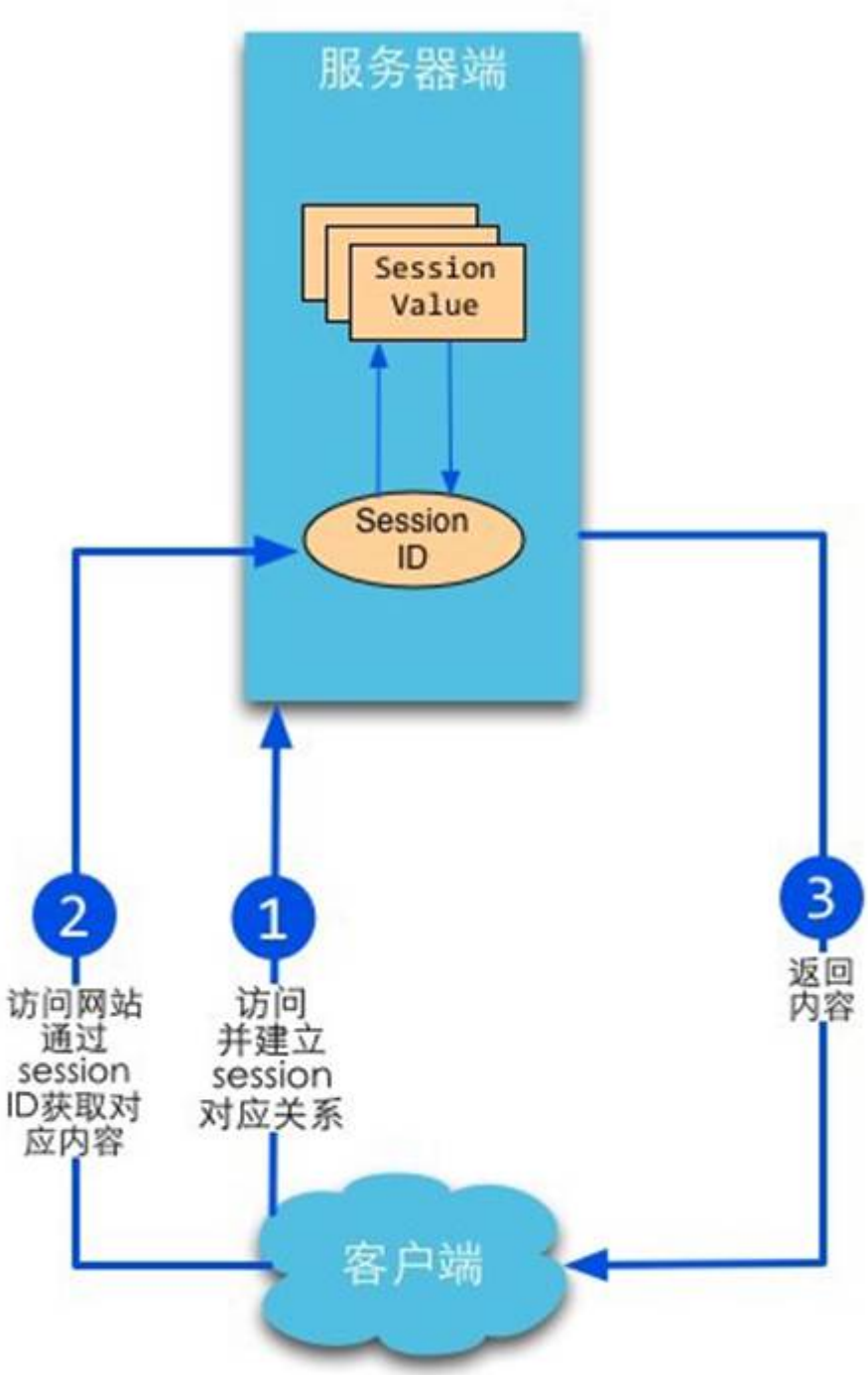
cookie是有时间限制的，根据生命期不同分成两种：会话cookie和持久cookie；

如果不设置过期时间，则表示这个cookie生命周期为从创建到浏览器关闭止，只要关闭浏览器窗口，cookie就消失了。这种生命期为浏览会话期的cookie被称为会话cookie。会话cookie一般不保存在硬盘上而是保存在内存里。

如果设置了过期时间(setMaxAge(60*60*24))，浏览器就会把cookie保存到硬盘上，关闭后再次打开浏览器，这些cookie依然有效直到超过设定的过期时间。存储在硬盘上的cookie可以在不同的浏览器进程间共享，比如两个IE窗口。而对于保存在内存的cookie，不同的浏览器有不同的处理方式。

session

Session，简而言之就是在服务器上保存用户操作的历史信息。服务器使用sessionid来标识session，sessionid由服务器负责产生，保证随机性和唯一性，相当于一个随机秘钥，避免在握手和传输中暴露用户真正密码。单在该方式下，仍然需要将发送请求给客户端与session进行对应，所以可以借助cookie机制来获取客户端的标识（即sessionid），也可通过GET方式将ID提交给服务器。



Session，中文通常被翻译为会话，本来的含义是指有使用中的一系列动作/信息，比如打电话，从拿起电话拨号到挂断电话这中间的一系列工程称之为一个session。然而session一次与网络协议向管理时，他往往隐含了“面向连接”或者“保持状态”这两个含义。

Session在web开发环境下的语义又有了新的扩展，它的含义是指一类用来在客户端与服务器之间保持状态的解决方案。有时候session也是用来指这种解决方案的存储结构。

Session机制是一种服务器端的机制，服务器只用一种类似于散列报的结构（也可能是散列表）来保存信息。

当程序需要为某个客户端请求创建一个session的时候，服务器首先要先检查这个客户端的请求里是否包含一个session表示——称之为session id，如果已经包含一个session id则说明以前已经为此用户创建过session，服务器就按照session id把这个session检索出来使用（如果检索不到，可能会创建一个新的。这种情况可能会出现在服务端已经删除了该用户对应的session对象，单用户认为地在请求的url后面加上一个SESSION的参数。）如果客户端

请求不包含session id，则为此服务创建一个session并同时生成一个与此session向管理的session id，在这个session id将本次响应包中返回给客户端。

Session机制本身并不复杂，然而其实现和配置上的灵活却使得具体情况复杂多变。这也要求我们不能仅仅某一次的经验或者某一个浏览器、服务器的经验当做普遍使用。

Session和cookie的目的相同，都是为了克服HTTP协议的无状态缺陷，但完成方法不同。Session通过cookie，在客户端保存session id，而将用户的其他会话消息保存在服务端的session对象中，与此相对的，cookie需要将所有信息都保存在客户端。因此cookie纯在这一定的安全隐患。例如本地cookie中保存的用户名密码被破译，或者cooie被其他网站收集（例如：

1.aapA主动设置域B cookie，让域B cookie获取；

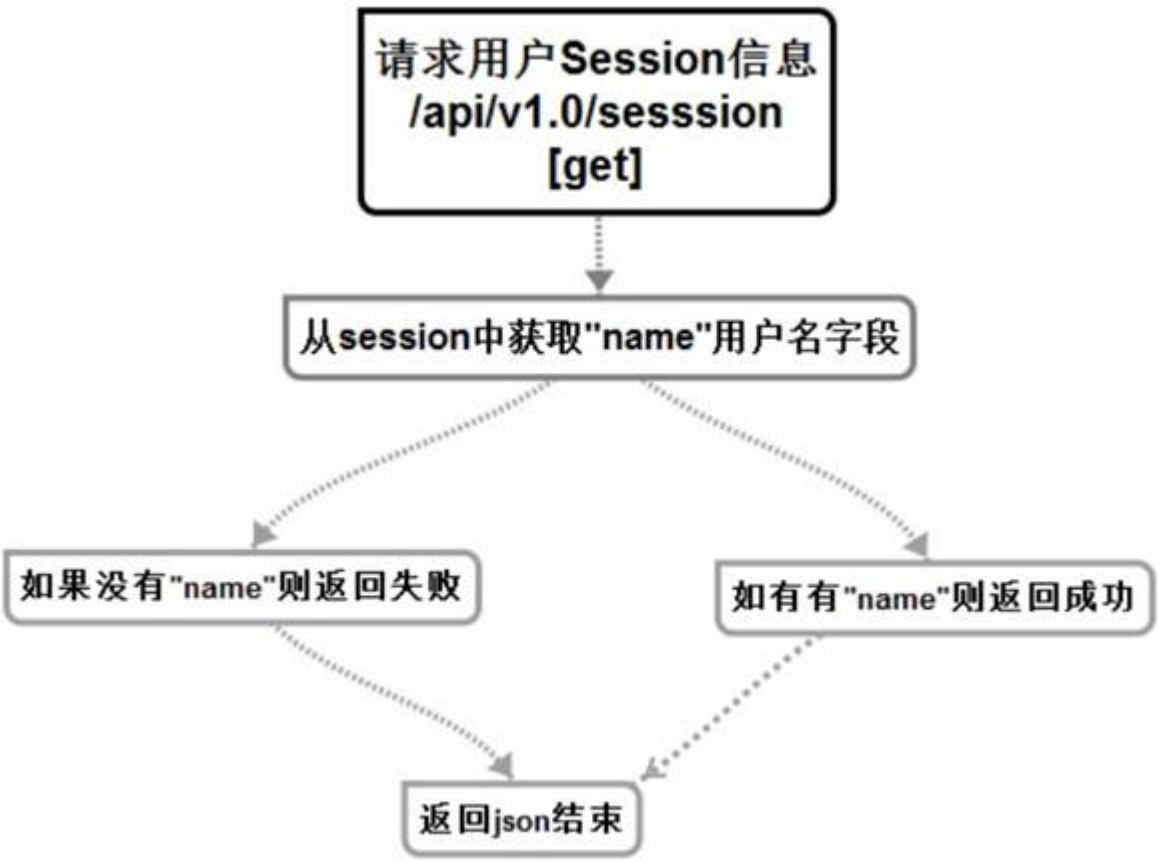
2.XSS（跨站脚本*攻击*），在appA中通过javascript获取document.cookie,并传递给自己的appB）

06 获取session信息

创建命令

```
$ micro new --type "srv" sss/GetSession
```

流程与接口




```
#Request:
method: GET
url:api/v1.0/session
data:no input data
#Response
#返回成功:
{
    "errno": "0",
    "errmsg":"OK",
    "data": {"name" : "13313331333"}

}
#注册失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

proto协议

```
service Example {
    rpc GetSession(Request) returns (Response) {}
}

message Request {
    string Sessionid = 1;
}

message Response {
    string Errno = 1;
    string Errmsg = 2;
    string Data = 3;
}
```

修改web服务中的GetSession

```
//获取session
func GetSession(w http.ResponseWriter, r *http.Request,_ httprouter.Params) {
    beego.Info("获取Session url: api/v1.0/session")

    //创建服务
    service := grpc.NewService()
    service.Init()

    //创建句柄
    exampleClient := GETSESSION.NewExampleService("go.micro.srv.GetSession",
service.Client())

    //获取cookie
    userlogin,err:=r.Cookie("userlogin")

    //如果不存在就返回
    if err != nil{

        //创建返回数据map
        response := map[string]interface{}{
            "errno": utils.RECODE_SESSIONERR,
            "errmsg": utils.RecodeText(utils.RECODE_SESSIONERR),

        }

        w.Header().Set("Content-Type", "application/json")
        // encode and write the response as json
        if err := json.NewEncoder(w).Encode(response); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        return
    }

    //存在就发送数据给服务
```

```
    rsp, err := exampleClient.GetSession(context.TODO(),&GETSESSION.Request{
        Sessionid:userlogin.Value,
    })

    if err != nil {
        http.Error(w, err.Error(), 502)

        beego.Info(err)
        //beego.Debug(err)
        return
    }


    // we want to augment the response
    //将获取到的用户名返回给前端
    data := make(map[string]string)
    data["name"]=rsp.Data
    response := map[string]interface{}{
        "errno": rsp.Errno,
        "errmsg": rsp.Errmsg,
        "data" : data,
    }

    w.Header().Set("Content-Type", "application/json")

    // 将返回数据map发送给前端
    if err := json.NewEncoder(w).Encode(response); err != nil {
        http.Error(w, err.Error(), 503)
        return
    }
}

```

服务端

```
func (e *Example) GetSession(ctx context.Context, req *example.Request, rsp
*example.Response) error {
    beego.Info(" GET session    /api/v1.0/session !!!")
    //创建返回空间
    //初始化的是否返回不存在
    rsp.Errno = utils.RECODE_SESSIONERR
    rsp.Errmsg = utils.RecodeText(rsp.Errno)

    ////获取前端的cookie
    beego.Info(req.Sessionid,reflect.TypeOf(req.Sessionid))
    //构建连接缓存的数据
    redis_config_map := map[string]string{
        "key":utils.G_server_name,
        "conn":utils.G_redis_addr+":"+utils.G_redis_port,
        "dbNum":utils.G_redis_dbnum,
    }
    beego.Info(redis_config_map)
    redis_config ,_:=json.Marshal(redis_config_map)
```

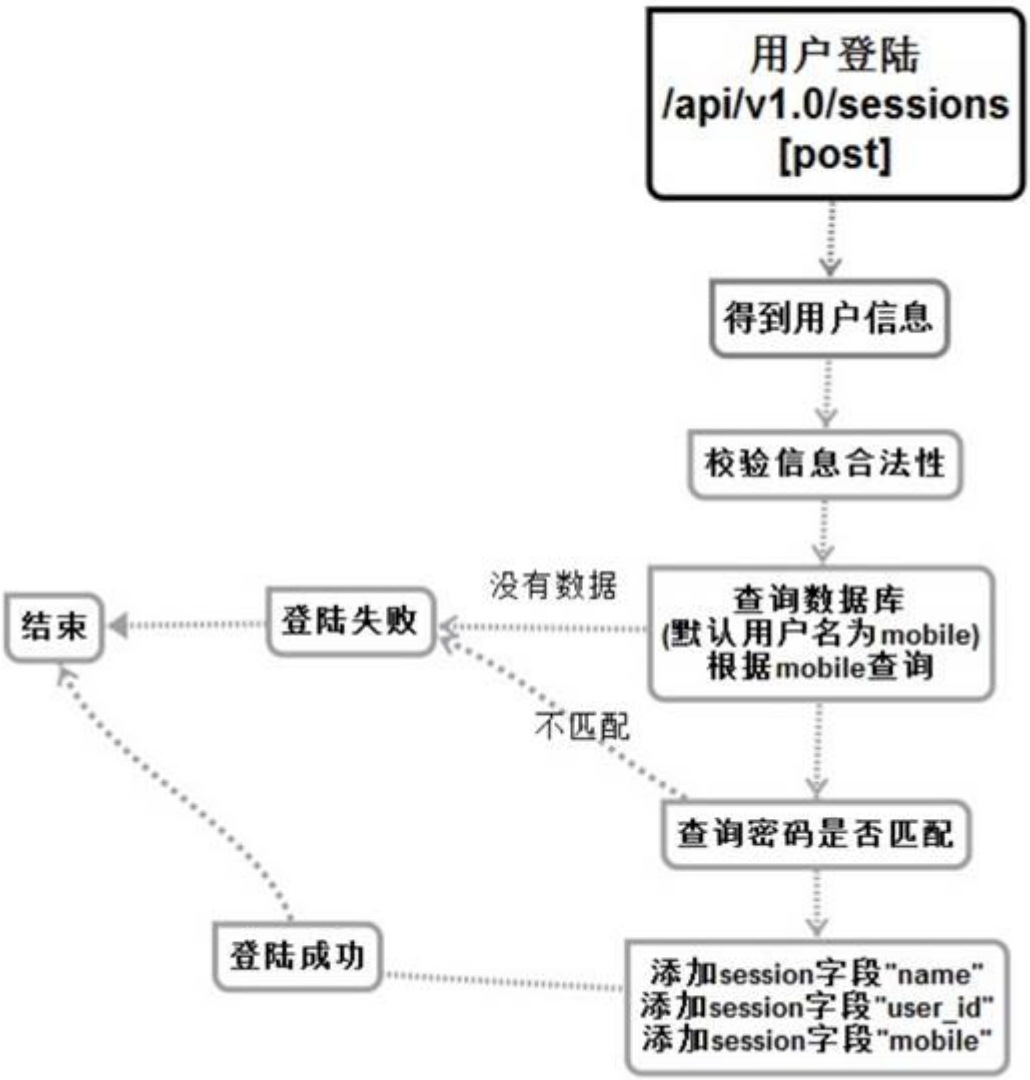
```
    beego.Info( string(redis_config) )

    //连接redis数据库 创建句柄
    bm, err := cache.NewCache("redis", string(redis_config) )
    if err != nil {
        beego.Info("缓存创建失败",err)
        rsp.Errno  =  utils.RECODE_DBERR
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
        return  nil
    }

    //拼接key
    sessionidname :=  req.Sessionid + "name"
    //从缓存中获取session 那么使用唯一识别码  通过key查询用户名
    areas_info_value:=bm.Get(sessionidname)
    //查看返回数据类型
    beego.Info(reflect.TypeOf(areas_info_value) ,areas_info_value )

    //通过redis方法进行转换
    name ,err :=redis.String(areas_info_value,nil)
    if err!=nil{
        rsp.Errno = utils.RECODE_DATAERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)
    }
    //查看返回数据类型
    beego.Info(name,reflect.TypeOf(name))


    //获取到了session
    rsp.Errno = utils.RECODE_OK
    rsp.Errmsg = utils.RecodeText(rsp.Errno)
    rsp.Data = name

    return nil

}
```

07 登录请求

创建命令

```
$ micro new --type "srv" sss/PostLogin
```

流程与接口




```
#Request:
method: POST
url:api/v1.0/sessions
#data:
{
    mobile: "133", //手机号
    password: "itcast"//密码
}
#Response
#返回成功:
{
    "errno": "0",
    "errmsg":"OK",
}
#返回失败:
{
```

```
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

proto

```
service Example {
    rpc PostLogin(Request) returns (Response) {}

}

message Request {
    string Mobile =1 ;
    string Password =2 ;
}

message Response {
    string Errno = 1 ;
    string Errmsg =2 ;
    string SessionID = 3 ;
}
```

web中添加路由

```
//登陆
rou.POST("/api/v1.0/sessions",handler.PostLogin)
```

web的handler中添加

```
//登陆
func PostLogin(w http.ResponseWriter, r *http.Request, _ httprouter.Params) {
    beego.Info("登陆 api/v1.0/sessions")
    //获取前端post请求发送的内容
    var request map[string]interface{}
    if err := json.NewDecoder(r.Body).Decode(&request); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }

    for key, value := range request {
        beego.Info(key,value,reflect.TypeOf(value))
    }
    //判断账号密码是否为空
    if request["mobile"] == "" || request["password"] =="" {
        resp := map[string]interface{}{
            "errno": utils.RECODE_NODATA,
            "errmsg": "信息有误请从新输入",
        }
        w.Header().Set("Content-Type", "application/json")

        // encode and write the response as json
```

```
        if err := json.NewEncoder(w).Encode(resp); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        beego.Info("有数据为空")
        return
    }

    //创建连接

    service := grpc.NewService()
    service.Init()
    exampleClient
:=POSTLOGIN.NewExampleService("go.micro.srv.PostLogin",service.Client())

    rsp, err := exampleClient.PostLogin(context.TODO(),&POSTLOGIN.Request{
        Password:request["password"].(string),
        Mobile:request["mobile"].(string),
    })


    if err != nil {
        http.Error(w, err.Error(), 502)

        beego.Info(err)
        //beego.Debug(err)
        return
    }

    cookie,err :=r.Cookie("userlogin")
    if err !=nil || ""==cookie.Value{
        cookie  := http.Cookie{Name: "userlogin", Value: rsp.SessionID, Path: "/",
MaxAge: 600}
        http.SetCookie(w, &cookie)
    }
    beego.Info(rsp.SessionID)
    resp := map[string]interface{}{
        "errno": rsp.Errno,
        "errmsg": rsp.Errmsg,

    }
    w.Header().Set("Content-Type", "application/json")

    // encode and write the response as json
    if err := json.NewEncoder(w).Encode(resp); err != nil {
        http.Error(w, err.Error(), 503)
        beego.Info(err)
        return
    }
}
```

服务

```go
func (e *Example) PostLogin(ctx context.Context, req *example.Request, rsp *example.Response) error {
    beego.Info("登陆 api/v1.0/sessions")

    //返回给前端的map结构体
    rsp.Errno =  utils.RECODE_OK
    rsp.Errmsg = utils.RecodeText(rsp.Errno)


    //查询数据库
    var user models.User
    o:=orm.NewOrm()

    //select * from user
    //创建查询句柄
    qs:=o.QueryTable("user")
    //qs.Filter("profile__age", 18)
    //查询符合的数据
    err:=qs.Filter("mobile",req.Mobile).One(&user)
    if err != nil {

        rsp.Errno =  utils.RECODE_NODATA
        rsp.Errmsg = utils.RecodeText(rsp.Errno)

        return nil
    }

    //判断密码是否正确
    if req.Password != user.Password_hash{
        rsp.Errno =  utils.RECODE_PWDERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)

        return nil
    }

    //编写redis缓存数据库信息
    redis_config_map := map[string]string{
        "key":utils.G_server_name,
        //"conn":"127.0.0.1:6379",
        "conn":utils.G_redis_addr+":"+utils.G_redis_port,
        "dbNum":utils.G_redis_dbnum,
    }
    beego.Info(redis_config_map)
    redis_config ,_:=json.Marshal(redis_config_map)


    //连接redis数据库 创建句柄
    bm, err := cache.NewCache("redis", string(redis_config) )

    if err != nil {
        beego.Info("缓存创建失败",err)
        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)
```

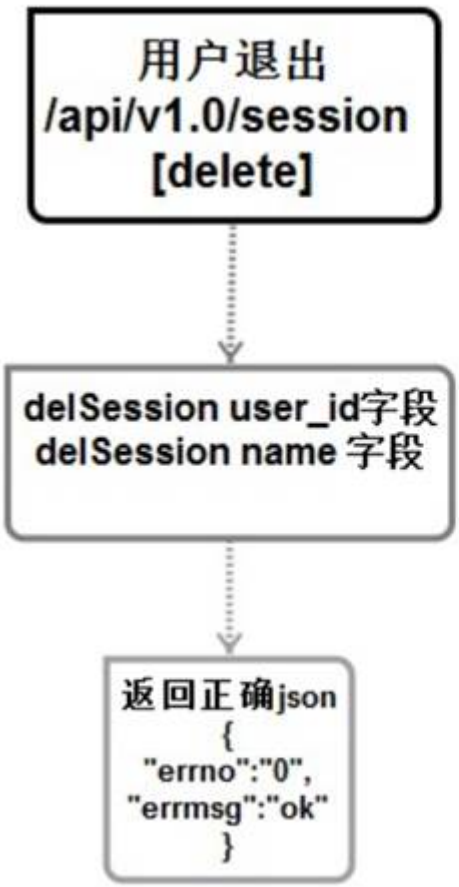
```
53
54          return nil
55      }
56
57      //生成sessionID
58      h := GetMd5String(req.Mobile+req.Password)
59      rsp.SessionID = h
60
61      beego.Info(h)
62
63      //拼接key sessionid + name
64      bm.Put(h+"name",string(user.Name),time.Second*3600)
65      //拼接key sessionid + user_id
66      bm.Put(h+"user_id", string(user.Id) ,time.Second*3600)
67      //拼接key sessionid + mobile
68      bm.Put(h+"mobile",string(user.Mobile) ,time.Second*3600)
69
70      //成功返回数据
71      return nil
72
73  }
```

08 退出请求

创建命令

```
1  $ micro new --type "srv" sss/DeleteSession
```

流程与接口




```
#Request:
method: DELETE
url:api/v1.0/session
#data:
no input data
#Response
#返回成功:
{
    "errno": "0",
    "errmsg":"OK",
}
#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

proto

```
service Example {
    rpc DeleteSession(Request) returns (Response) {}
}


message Request {
    string Sessionid = 1;
}

message Response {
    string Errno = 1;
    string Errmsg = 2;
}
```

在web中添加1个路由

```
//退出登陆
rou.DELETE("/api/v1.0/session", handler.DeleteSession)
```

web下的handler

```
func DeleteSession(w http.ResponseWriter, r *http.Request,_ httprouter.Params) {
    beego.Info("---------------- DELETE  /api/v1.0/session Deletesession() ----------")
    //创建返回空间

    server :=grpc.NewService()
    server.Init()
    exampleClient := DELETESESSION.NewExampleService("go.micro.srv.DeleteSession", server.Client())

    //获取session
    userlogin,err:=r.Cookie("userlogin")
    //如果没有数据说明没有的登陆直接返回错误
    if err != nil{
        resp := map[string]interface{}{
            "errno": utils.RECODE_SESSIONERR,
            "errmsg": utils.RecodeText(utils.RECODE_SESSIONERR),
        }

        w.Header().Set("Content-Type", "application/json")
        // encode and write the response as json
        if err := json.NewEncoder(w).Encode(resp); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        return
    }

    rsp, err := exampleClient.DeleteSession(context.TODO(),&DELETESESSION.Request{
        Sessionid:userlogin.Value,
```

```
    })

    if err != nil {
        http.Error(w, err.Error(), 502)

        beego.Info(err)
        //beego.Debug(err)
        return
    }
    //再次读取数据
    cookie,err :=r.Cookie("userlogin")

    //数据不为空则将数据设置副的
    if err !=nil || ""==cookie.Value{
        return
    }else {
        cookie := http.Cookie{Name: "userlogin", Path: "/", MaxAge: -1}
        http.SetCookie(w, &cookie)
    }


    //返回数据
    resp := map[string]interface{}{
        "errno": rsp.Errno,
        "errmsg": rsp.Errmsg,
    }
    //设置格式
    w.Header().Set("Content-Type", "application/json")

    // encode and write the response as json
    if err := json.NewEncoder(w).Encode(resp); err != nil {
        http.Error(w, err.Error(), 503)
        beego.Info(err)
        return
    }


    return
}
```

服务端

```
func (e *Example) DeleteSession(ctx context.Context, req *example.Request, rsp
*example.Response) error {
    beego.Info(" DELETE session    /api/v1.0/session !!!")

    //创建返回空间
    //初始化的是否返回不存在
    rsp.Errno = utils.RECODE_OK
    rsp.Errmsg = utils.RecodeText(rsp.Errno)

```

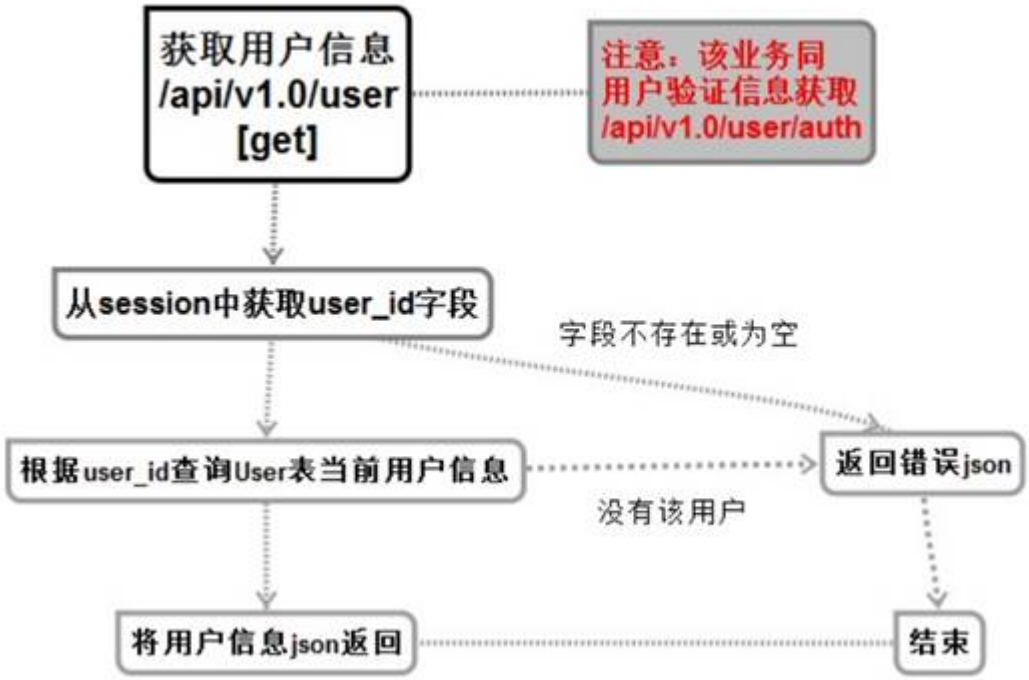
```
    ////获取前端的cookie
    beego.Info(req.Sessionid,reflect.TypeOf(req.Sessionid))


    //构建连接缓存的数据
    redis_config_map := map[string]string{
        "key":utils.G_server_name,
        "conn":utils.G_redis_addr+":"+utils.G_redis_port,
        "dbNum":utils.G_redis_dbnum,
    }
    beego.Info(redis_config_map)
    redis_config ,_:=json.Marshal(redis_config_map)

    //连接redis数据库 创建句柄
    bm, err := cache.NewCache("redis", string(redis_config) )
    if err != nil {
        beego.Info("缓存创建失败",err)
        rsp.Errno  =  utils.RECODE_DBERR
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
        return  nil
    }
    sessionidname :=  req.Sessionid + "name"
    sessioniduserid :=  req.Sessionid + "user_id"
    sessionidmobile :=  req.Sessionid + "mobile"

    //从缓存中获取session 那么使用唯一识别码
    bm.Delete(sessionidname)
    bm.Delete(sessioniduserid)
    bm.Delete(sessionidmobile)

    return nil
}
```

09 获取用户信息

创建命令

```
$ micro new  --type "srv" sss/GetUserInfo
```

流程与接口




```
#Request:
method: GET
url:api/v1.0/user
#data:
no input data
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
    "user_id": 1,
    "name": "Panda",
    "mobile": "110",
    "real_name": "熊猫",
    "id_card": "210112244556677",
    "avatar_url":
"http://101.200.170.171:9998/group1/M00/00/00/Zciqq1n7It2ANn1dAADexS5wJKs808.png"
  }
}
#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

proto

```
service Example {
    rpc GetUserInfo(Request) returns (Response) {}
}
message Request {
    string Sessionid = 1 ;
}
message Response {
    //错误码
    string Errno =1 ;
    //错误信息
    string Errmsg = 2;
    //用户id
    int64  User_id = 3 ;
    //用户名
    string Name =4;
    //手机号
    string Mobile =5 ;
    //真实姓名
    string Real_name =6 ;
    //身份证号
    string Id_card =7;
    //头衔地址
    string Avatar_url =8 ;
}
```

web中添加路由

```
//请求用户基本信息 GET /api/v1.0/user
rou.GET("/api/v1.0/user", handler.GetUserInfo)
```

web下handler添加函数

```
func GetUserInfo(w http.ResponseWriter, r *http.Request,_ httprouter.Params) {
    beego.Info("GetUserInfo  获取用户信息   /api/v1.0/user")
    //初始化服务
    service := grpc.NewService()
    service.Init()

    //创建句柄
    exampleClient := GETUSERINFO.NewExampleService("go.micro.srv.GetUserInfo",
service.Client())

    //获取用户的登陆信息
    userlogin,err:=r.Cookie("userlogin")

    //判断是否成功不成功就直接返回
    if err != nil{
        resp := map[string]interface{}{
            "errno": utils.RECODE_SESSIONERR,
            "errmsg": utils.RecodeText(utils.RECODE_SESSIONERR),
        }

```

```
        w.Header().Set("Content-Type", "application/json")
        // encode and write the response as json
        if err := json.NewEncoder(w).Encode(resp); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        return
    }

    //成功就将信息发送给前端
    rsp, err := exampleClient.GetUserInfo(context.TODO(),&GETUSERINFO.Request{
        Sessionid:userlogin.Value,
    })

    if err != nil {
        http.Error(w, err.Error(), 502)

        beego.Info(err)
        //beego.Debug(err)
        return
    }
    //


    // 准备1个数据的map
    data := make(map[string]interface{})
    //将信息发送给前端
    data["user_id"] = int(rsp.UserId)
    data["name"] = rsp.Name
    data["mobile"] = rsp.Mobile
    data["real_name"] = rsp.RealName
    data["id_card"] = rsp.IdCard
    data["avatar_url"] = utils.AddDomain2Url(rsp.AvatarUrl)

    resp := map[string]interface{}{
        "errno": rsp.Errno,
        "errmsg": rsp.Errmsg,
        "data" : data,
    }
    //设置格式
    w.Header().Set("Content-Type", "application/json")

    // encode and write the response as json
    if err := json.NewEncoder(w).Encode(resp); err != nil {
        http.Error(w, err.Error(), 503)
        beego.Info(err)
        return
    }

    return
}

```

服务端

```
func (e *Example) GetUserInfo(ctx context.Context, req *example.Request, rsp *example.Response) error {

    beego.Info("---------------- GET  /api/v1.0/user Getuserinfo() ----------------")
    //打印sessionid
    beego.Info(req.Sessionid,reflect.TypeOf(req.Sessionid))
    //错误码
    rsp.Errno  =  utils.RECODE_OK
    rsp.Errmsg  = utils.RecodeText(rsp.Errno)


    //构建连接缓存的数据
    redis_config_map := map[string]string{
        "key":utils.G_server_name,
        //"conn":"127.0.0.1:6379",
        "conn":utils.G_redis_addr+":"+utils.G_redis_port,
        "dbNum":utils.G_redis_dbnum,
    }
    beego.Info(redis_config_map)
    redis_config ,_:=json.Marshal(redis_config_map)

    //连接redis数据库 创建句柄
    bm, err := cache.NewCache("redis", string(redis_config) )
    if err != nil {
        beego.Info("缓存创建失败",err)
        rsp.Errno  =  utils.RECODE_DBERR
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
        return  nil
    }

    //拼接用户信息缓存字段
    sessioniduserid :=  req.Sessionid + "user_id"

    //获取到当前登陆用户的user_id
    value_id :=bm.Get(sessioniduserid)
    //打印
    beego.Info(value_id,reflect.TypeOf(value_id))

    //数据格式转换
    id :=  int(value_id.([]uint8)[0])
    beego.Info(id ,reflect.TypeOf(id))
    //创建user表
    user := models.User{Id:id}
    //创建数据库orm句柄
    o := orm.NewOrm()
    //查询表
    err =o.Read(&user)
    if err !=nil{
        rsp.Errno  =  utils.RECODE_DBERR
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
```

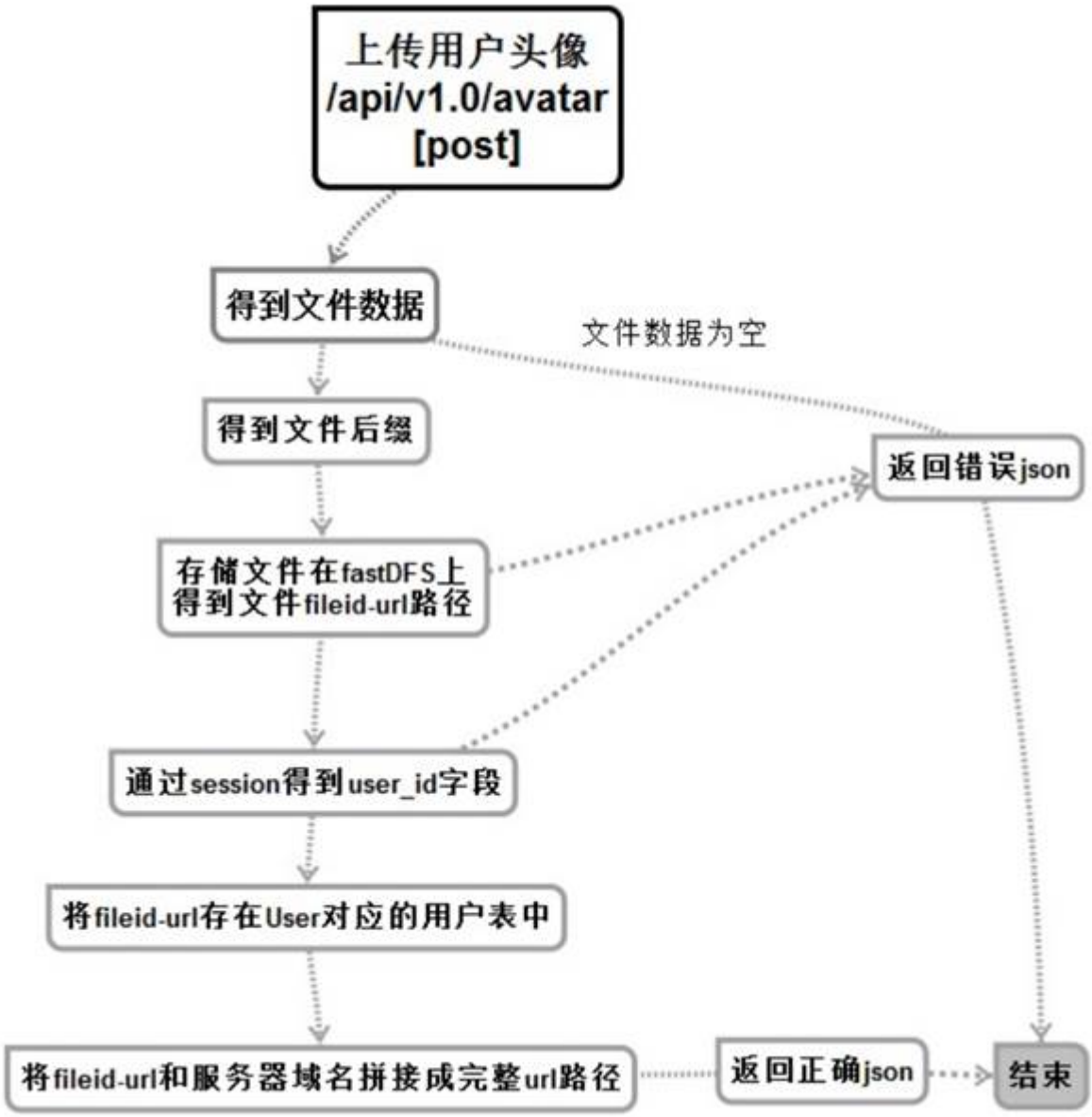
```
        return  nil
    }
    //将查询到的数据依次赋值
    rsp.UserId= int64(user.Id)
    rsp.Name= user.Name
    rsp.Mobile = user.Mobile
    rsp.RealName = user.Real_name
    rsp.IdCard = user.Id_card
    rsp.AvatarUrl = user.Avatar_url


    return nil
}
```

10 上传用户头像

创建命令

```
$ micro new  --type "srv" sss/PostAvatar
```

流程与接口




```
#Request:
method: POST
url:api/v1.0/user/avatar
#data:
图片的二进制数据
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
    "avatar_url": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1n6_L-AOB04AADexS5wJKs662.png" //图片地址需要进行拼接
  }
```

```
14
15  }
16  #返回失败:
17  {
18      "errno": "400x",    //状态码
19      "errmsg":"状态错误信息"
20  }
```

fastdfs操纵函数

```
1   package models
2
3   import (
4       "github.com/weilaihui/fdfs_client"
5       "github.com/astaxie/beego"
6   )
7
8   //通过文件名的方式进行上传
9   func UploadByFilename( filename string)(GroupName,RemoteFileId string ,err error )
    {
10      //通过配置文件创建fdfs操作句柄
11      fdfsClient, thiserr :=fdfs_client.NewFdfsClient("/home/itcast/go/src/go-
    1/homeweb/conf/client.conf")
12      if thiserr  !=nil{
13          //说一下那里出问题了
14          beego.Info("UploadByFilename( ) fdfs_client.NewFdfsClient  err",err)
15          GroupName = ""
16          RemoteFileId = ""
17          err = thiserr
18          return
19      }
20
21      //unc (this *FdfsClient) UploadByFilename(filename string)
    (*UploadFileResponse, error)
22      //通过句柄上传文件（被上传的文件）
23
24      uploadResponse, thiserr := fdfsClient.UploadByFilename(filename)
25      if thiserr !=nil{
26          beego.Info("UploadByFilename( ) fdfsClient.UploadByFilename(filename)
    err",err)
27          GroupName = ""
28          RemoteFileId = ""
29          err = thiserr
30          return
31      }
32
33      beego.Info(uploadResponse.GroupName)
34      beego.Info(uploadResponse.RemoteFileId)
35      //回传
36      return  uploadResponse.GroupName , uploadResponse.RemoteFileId ,nil
37
38  }
39
```

```
40
41
42
43 //功能函数 操作fdfs上传二进制文件
44 func UploadByBuffer(filebuffer []byte, fileExtName string)(GroupName,RemoteFileId
   string ,err error ){
45
46     //通过配置文件创建fdfs操作句柄
47     fdfsClient, thiserr :=fdfs_client.NewFdfsClient("/home/itcast/go/src/go-
   1/homeweb/conf/client.conf")
48     if thiserr  !=nil{
49         beego.Info("UploadByBuffer( ) fdfs_client.NewFdfsClient  err",err)
50         GroupName = ""
51         RemoteFileId = ""
52         err = thiserr
53         return
54     }
55
56     //通过句柄上传二进制的文件
57     uploadResponse, thiserr :=fdfsClient.UploadByBuffer(filebuffer,fileExtName)
58     if thiserr  !=nil{
59         beego.Info("UploadByBuffer( ) fdfs_client.UploadByBuffer  err",err)
60         GroupName = ""
61         RemoteFileId = ""
62         err = thiserr
63         return
64     }
65     beego.Info(uploadResponse.GroupName)
66     beego.Info(uploadResponse.RemoteFileId)
67     //回传入
68     return uploadResponse.GroupName,uploadResponse.RemoteFileId,nil
69
70 }
71
```

proto

```
1  service Example {
2      rpc PostAvatar(Request) returns (Response) {}
3  }
4
5  message Message {
6      string say = 1;
7  }
8
9  message Request {
10     //二进制图片
11     bytes Avatar =1;
12     //sessionid
13     string Sessionid=2;
14     //文件大小
15     int64 filesize =3;
16     //文件名字
```

```
    string filename =4 ;
}

message Response {
    //错误码
    string Errno = 1;
    //错误信息
    string Errmsg = 2;
    //回传的url
    string Avatar_url = 3;
}
```

web中添加路由

```
//上传头像 POST
rou.POST("/api/v1.0/user/avatar",handler.PostAvatar)
```

web下handle

```
//上传用户头像 PostAvatar
func PostAvatar(w http.ResponseWriter, r *http.Request,_ httprouter.Params) {
    beego.Info("上传用户头像 PostAvatar /api/v1.0/user/avatar")

    //创建服务
    service := grpc.NewService()
    service.Init()

    //创建句柄
    exampleClient := POSTAVATAR.NewExampleService("go.micro.srv.PostAvatar",
service.Client())

    //查看登陆信息
    userlogin,err:=r.Cookie("userlogin")

    //如果没有登陆就返回错误
    if err != nil{
        resp := map[string]interface{}{
            "errno": utils.RECODE_SESSIONERR,
            "errmsg": utils.RecodeText(utils.RECODE_SESSIONERR),
        }

        w.Header().Set("Content-Type", "application/json")
        // encode and write the response as json
        if err := json.NewEncoder(w).Encode(resp); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        return
    }

    //接收前端发送过来的文集
```

```
    file,hander,err:=r.FormFile("avatar")

    //判断是否接受成功
    if err != nil{
        beego.Info("Postupavatar   c.GetFile(avatar) err" ,err)

        resp := map[string]interface{}{
            "errno": utils.RECODE_IOERR,
            "errmsg": utils.RecodeText(utils.RECODE_IOERR),
        }
        w.Header().Set("Content-Type", "application/json")
        // encode and write the response as json
        if err := json.NewEncoder(w).Encode(resp); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        return
    }
    //打印基本信息
    beego.Info(file ,hander)
    beego.Info("文件大小",hander.Size)
    beego.Info("文件名",hander.Filename)

    //二进制的空间用来存储文件
    filebuffer:= make([]byte,hander.Size)

    //将文件读取到filebuffer里
    _,err = file.Read(filebuffer)
    if err !=nil{
        beego.Info("Postupavatar   file.Read(filebuffer) err" ,err)
        resp := map[string]interface{}{
            "errno": utils.RECODE_IOERR,
            "errmsg": utils.RecodeText(utils.RECODE_IOERR),
        }
        w.Header().Set("Content-Type", "application/json")
        // encode and write the response as json
        if err := json.NewEncoder(w).Encode(resp); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        return
    }
    //调用函数传入数据
    rsp, err := exampleClient.PostAvatar(context.TODO(),&POSTAVATAR.Request{
        Sessionid:userlogin.Value,
        Filename:hander.Filename,
        Filesize:hander.Size,
        Avatar:filebuffer,
    })
    if err != nil {
        http.Error(w, err.Error(), 502)
```

```
86
87          beego.Info(err)
88          //beego.Debug(err)
89          return
90      }
91      //
92
93      //准备回传数据空间
94      data := make(map[string]interface{})
95      //url拼接然回回传数据
96      data["avatar_url"]=utils.AddDomain2Url(rsp.AvatarUrl)
97
98
99
100     resp := map[string]interface{}{
101         "errno": rsp.Errno,
102         "errmsg": rsp.Errmsg,
103         "data":data,
104     }
105
106     w.Header().Set("Content-Type", "application/json")
107
108     // encode and write the response as json
109     if err := json.NewEncoder(w).Encode(resp); err != nil {
110         http.Error(w, err.Error(), 503)
111         beego.Info(err)
112         return
113     }
114
115
116     return
117 }
```

服务端

```
1   // Call is a single request handler called via client.Call or the generated client
    code
2   func (e *Example) PostAvatar(ctx context.Context, req *example.Request, rsp
    *example.Response) error {
3       beego.Info("上传用户头像 PostAvatar /api/v1.0/user/avatar")
4       //初始化返回正确的返回值
5       rsp.Errno = utils.RECODE_OK
6       rsp.Errmsg = utils.RecodeText(rsp.Errno)
7
8       //检查下数据是否正常
9       beego.Info(len(req.Avatar),req.Filesize)
10
11
12      /*获取文件的后缀名*/      //dsnlkjfajadskfksda.sadsdasd.sdasd.jpg
13      beego.Info("后缀名",path.Ext(req.Filename))
14
15      /*存储文件到fastdfs当中并且获取 url*/
16      //.jpg
```

```
    fileext :=path.Ext(req.Filename)
    //group1 group1/M00/00/00/wKgLg1t08pmANXH1AAaInSze-cQ589.jpg
    //上传数据
    Group,FileId ,err :=  models.UploadByBuffer(req.Avatar,fileext[1:])
    if err != nil {
        beego.Info("Postupavatar  models.UploadByBuffer err" ,err)
        rsp.Errno = utils.RECODE_IOERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)
        return nil
    }
    beego.Info(Group)

    /*通过session 获取我们当前现在用户的uesr_id*/
    redis_config_map := map[string]string{
        "key":utils.G_server_name,
        //"conn":"127.0.0.1:6379",
        "conn":utils.G_redis_addr+":"+utils.G_redis_port,
        "dbNum":utils.G_redis_dbnum,
    }
    beego.Info(redis_config_map)
    redis_config ,_:=json.Marshal(redis_config_map)
    beego.Info( string(redis_config) )
    //连接redis数据库 创建句柄
    bm, err := cache.NewCache("redis", string(redis_config) )
    if err != nil {
        beego.Info("缓存创建失败",err)
        rsp.Errno  =  utils.RECODE_DBERR
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
        return  nil
    }
    //拼接key
    sessioniduserid :=  req.Sessionid + "user_id"

    //获得当前用户的userid
    value_id :=bm.Get(sessioniduserid)
    beego.Info(value_id,reflect.TypeOf(value_id))


    id :=  int(value_id.([]uint8)[0])
    beego.Info(id ,reflect.TypeOf(id))

    //创建表对象
    user := models.User{Id:id,Avatar_url:FileId}
    /*将当前fastdfs-url 存储到我们当前用户的表中*/
    o:=orm.NewOrm()
    //将图片的地址存入表中
    _ ,err =o.Update(&user ,"avatar_url")
    if err !=nil {

        rsp.Errno = utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)

    }
```

```
70
71      //回传图片地址
72      rsp.AvatarUrl=FileId
73
74      return nil
75  }
```

使用fastdfs+nginx

fastdfs安装

```
#安装libevent
#解压
$ tar zxvf libevent-2.1.8-stable.tar.gz
$ cd libevent-2.1.8-stable/
#检查
$ ./configure
#编译
$ make
#安装
$ sudo make install
#检查安装是否成功
$ ls -al /usr/local/lib/ | grep libevent
#防止在系统默认路径下 找不到库文件
$ sudo ln -s /usr/local/lib/libevent-2.1.so.6 /usr/lib/libevent-2.1.so.6

#安装libfastcommon
$ unzip libfastcommon-1.36.zip
$ cd libfastcommon-master/
$ ./make.sh
$ sudo ./make.sh install

#安装fastdfs
#解压
$ tar -zxvf fastdfs-5.10.tar.gz
$ cd fastdfs-5.10/
#编译
$ ./make.sh
#安装
$ sudo ./make.sh install
#验证
$ ls -al /usr/bin/fdfs*
$ fdfs_test
```

配置 fastdfs

```
#默认的配置文件地址在/etc/fdfs下
#因为我们的项目需要
#所以我们需要将配置文件拷贝到我们的conf文件夹下进配置
```

```
$ cd /etc/fdfs
$ cp  client.conf.sample  storage.conf.sample  tracker.conf.sample
$GOPATH/src/sss/Ihomeweb/conf
#将配置文件的名字进行下修改
#客户端
$ mv client.conf.sample  client.conf
#存储文件
$ mv storage.conf.sample  storage.conf
#跟踪器
$ mv tracker.conf.sample  tracker.conf
#准备文件夹在项目下创建文件夹
$ cd /home/itcast/go/src/sss
$ mkdir fastdfs
#在fastdfs下创建四个文件夹
$ cd fastdfs
$ mkdir tracker client storage storage_data

##配置tracker.conf
#6行的ip地址设置
# bind an address of this host
# empty for bind all addresses of this host
bind_addr=192.168.110.123 (需要配置的ip)

#21行的的log日志设置
# the base path to store data and log files
base_path=/home/itcast/go/src/sss/fastdfs/tracker (log目录)

##配置storage.conf
#13行配置ip
# bind an address of this host
# empty for bind all addresses of this host
bind_addr=192.168.110.123
#40行的log日志设置
# the base path to store data and log files
base_path=/home/itcast/go/src/sss/fastdfs/storage (log目录)

#107行的文件存放设置
# store_path#, based 0, if store_path0 not exists, it's value is base_path
# the paths must be exist
store_path0=/home/itcast/go/src/sss/fastdfs/storage_data (文件存放路径)
#store_path1=/home/yuqing/fastdfs2

#116行tracker的ip地址
# tracker_server can ocur more than once, and tracker_server format is
#  "host:port", host can be hostname or ip address
tracker_server=192.168.110.7:22122

##配置client.conf日志
#9行的log日志地址
# the base path to store log files
base_path=/home/itcast/go/src/sss/fastdfs/client (log目录)
#12行追踪器的ip
# tracker_server can ocur more than once, and tracker_server format is
```

```
#  "host:port", host can be hostname or ip address
tracker_server=192.168.110.123:22122

更新项目目录下的的setup_server.sh文件
#启动redis服务
redis-server ./conf/redis.conf

#启动trackerd

fdfs_trackerd  /home/itcast/go/src/sss/homeweb/conf/tracker.conf restart
#启动storaged

fdfs_storaged  /home/itcast/go/src/sss/homeweb/conf/storage.conf restart

我们先测试下
Fdfs_upload_file ./conf/client.conf cj.jpg

```

安装步骤nginx操作fast步骤

关于nginx操作fastdfs的插件

```
#将事先准备好的fastdfs的插件拷贝到ubuntu中
#解压nginx关于fastdfs的插件
$ tar -zxvf fastdfs-nginx-module_v1.16.tar.gz
#查看当前目录下存在两个解压好的文件
$ ls
fastdfs-nginx-module    nginx-1.10.1
#进入fastdfs-nginx-module
$ cd fastdfs-nginx-module
#查看当前文件夹下文件
$ tree
.
├── HISTORY
├── INSTALL
└── src
    ├── common.c
    ├── common.h
    ├── config
    ├── mod_fastdfs.conf
    └── ngx_http_fastdfs_module.c
#查看INSTALL文件
$ vim INSTALL
```

关于fastdfs插件的安装介绍

```
版权所有 2010开心鱼 / 余庆
这个软件只能在GNU通用条款下被复制
```

```
公共许可证V3，请访问FastDFS主页以获得更多详细信息。
英语:http://english.csource.org/
中文:http://www.csource.org/

#步骤1.首先安装FastDFS存储服务器和客户端库，
FastDFS版本应该是>= 2.09。下载地址:
https://code.google.com/p/fastdfs/downloads/list

#步骤2.安装nginx服务器
FastDFS nginx模块测试通过nginx 0.8.53，
我的nginx安装在/usr/local/nginx

#步骤3.下载FastDFS nginx模块源代码包并解压，如:
tar xzf fastdfs_nginx_module_v1.16.tar.gz

#步骤4.输入nginx源目录，编译并安装模块，如:
cd nginx-1.5.12
./configure --add-module=/home/yuqing/fastdfs-nginx-module/src
make; make install

注意:在编译之前，你可以更改FDFS_OUTPUT_CHUNK_SIZE和
FDFS_MOD_CONF_FILENAME的宏定义在配置文件中:

CFLAGS="$CFLAGS -D_FILE_OFFSET_BITS=64 -DFDFS_OUTPUT_CHUNK_SIZE='256*1024' -
DFDFS_MOD_CONF_FILENAME='\"/etc/fdfs/mod_fastdfs.conf\"'"



#步骤5.配置nginx配置文件，例如nginx.conf，添加以下行:

        location /M00 {
            root /home/yuqing/fastdfs/data;
            ngx_fastdfs_module;
        }

#步骤6.将一个符号链接${fastdfs_base_path}/data/M00到${fastdfs_base_path}/data，

命令行，例如:
ln -s /home/yuqing/fast /data/ home/yuqing/fast /data/ m00

#步骤7.更改配置文件/etc/fdfs/mod_fastdfs.conf，更多细节请看

#步骤8.重新启动nginx服务器，例如:
/usr/local/nginx/sbin/nginx -s stop; /usr/local/nginx/sbin/nginx
```

修改fastdfs插件安装的配置文件

```
#进入到插件的src目录下
$ cd/home/itcast/ffdfs/fastdfs-nginx-module/src
#修改config配置文件
$ vi config
CORE_INCS="$CORE_INCS /usr/local/include/fastdfs /usr/local/include/fastcommon/"
#修改为:
CORE_INCS="$CORE_INCS /usr/include/fastdfs /usr/include/fastcommon/"
```

nginx的安装依赖

```
#讲准备好的依赖文件拷贝到ubuntu中
#依次解压依赖
$ tar -zxvf openssl-1.0.1t.tar.gz
$ tar -zxvf zlib-1.2.11.tar.gz
$ tar -jxvf pcre-8.40.tar.bz2


#进入openssl-1.0.1t
$ cd openssl-1.0.1t
#检查
$ sudo ./config
#编译
$ sudo make
#安装
$ sudo make install

#进入zlib-1.2.11
$ cd zlib-1.2.11
#检查
$ sudo ./configure
#编译
$ sudo make
#安装
$ sudo make install

#进入pcre-8.40
$ cd pcre-8.40
#检查
$ sudo ./configure
#编译
$ sudo make
#安装
$ sudo make install
```

nginx正规联合编译方法

```
#将事先准备好的nginx安装包拷贝到ubuntu中
#解压nginx压缩包
```

```
$ tar -zxvf nginx-1.10.1.tar.gz
#进入nginx
$ cd /home/itcast/ffdfs/nginx-1.10.1
#进行nginx和fastdfs插件的联合编译
#检查
$ ./configure --with-openssl=openssl源码目录 --with-pcre=pcre的源码目录 --with-zlib=zlib  --add-module=插件src目录

$ ./configure --with-openssl=/home/itcast/openssl-1.0.1t --with-pcre=/home/itcast/pcre-8.40 --with-zlib=/home/itcast/zlib-1.2.11 --with-http_ssl_module  --add-module=/home/itcast/fastdfs-nginx-module_v1.16/fastdfs-nginx-module/src
#编译
$ make
#安装
$ make install
#创建软连接方便调用
$ ln -s /usr/local/nginx/sbin/nginx /usr/bin/nginx
```

注意1：找不到sha1的库

```
#在nginx 下执行 ./configure后最近结果显示会出现如下问题
sha1 library is not found
#找不到sha1的库
#configure的时候增加参数 --with-http_ssl_module
```

注意2：编译异常

```
#在 make时出现如下问题
src/core/ngx_murmurhash.c: In function ‘ngx_murmur_hash2’:
src/core/ngx_murmurhash.c:37:11: error: this statement may fall through [-Werror=implicit-fallthrough=]
         h ^= data[2] << 16;
         ~~^~~~~~~~~~~~~~~~
src/core/ngx_murmurhash.c:38:5: note: here
     case 2:
     ^~~~
src/core/ngx_murmurhash.c:39:11: error: this statement may fall through [-Werror=implicit-fallthrough=]
         h ^= data[1] << 8;
         ~~^~~~~~~~~~~~~~~~
src/core/ngx_murmurhash.c:40:5: note: here
     case 1:
     ^~~~
cc1: all warnings being treated as errors
objs/Makefile:485: recipe for target 'objs/src/core/ngx_murmurhash.o' failed
make[1]: *** [objs/src/core/ngx_murmurhash.o] Error 1
make[1]: 离开目录“/home/itcast/ffdfs/nginx-1.10.1”
Makefile:8: recipe for target 'build' failed
make: *** [build] Error 2

```

```
#主要原因是 Makefile 里面  gcc 的参数多了一个"-Werror"
#进入到/home/itcast/ffdfs/nginx-1.10.1/objs中修改Makefile
vim Makefile
#删除"-Werror"得到如下内容
CFLAGS =  -pipe  -O -W -Wall -Wpointer-arith -Wno-unused  -g  -
D_FILE_OFFSET_BITS=64 -DFDFS_OUTPUT_CHUNK_SIZE='256*1024' -
DFDFS_MOD_CONF_FILENAME='"/etc/fdfs/mod_fastdfs.conf"'

```

注意3：插件配置文件不存在

```
#联合编译安装好nginx后启动nginx如下效果发现只有1个进程
$ nginx
$ ps aux|grep nginx
ps aux |grep nginx
root  105455 cd
#查看 /usr/local/nginx/logs 目录下的日志
$ cat /usr/local/nginx/logs/error.log
#发现这样一条错误
ERROR - file: shared_func.c, line: 968, file /etc/fdfs/mod_fastdfs.conf not exist
错误 - 文件 :shared_func.c 968行 访问 /etc/fdfs/mod_fastdfs.conf 不存在

#确认下文件是否存在
$ cd /etc/fdfs/
$ ls
client.conf.sample  storage.conf.sample  storage_ids.conf.sample
tracker.conf.sample
#确实不存在

#解决
#需要从fastdfs源码目录中复制过来
$ cd /home/itcast/ffdfs/fastdfs-nginx-module/src
#将mod_fastdfs.conf 文件从当前目录下拷贝到/etc/fdfs/目录下
$ sudo cp /home/itcast/ffdfs/fastdfs-nginx-module/src/mod_fastdfs.conf /etc/fdfs/
```

注意4：配置文件修改

```
#修改fastdfs插件的配置文件进行修改，参数当前存储节点的storage.conf进行修改
#进入到/etc/fdfs/目录
$ cd /etc/fdfs/
#编辑 mod_fastdfs.conf 文件
$ sudo vim mod_fastdfs.conf

# 存储log日志的目录
 9 # the base path to store log files
 10 base_path=/home/itcast/go/src/sss/fastdfs/storage
# 追踪器的地址信息
 37 # FastDFS tracker_server can ocur more than once, and tracker_server format is
 38 #  "host:port", host can be hostname or ip address
```

```
 39 # valid only when load_fdfs_parameters_from_tracker is true
 40 tracker_server=192.168.110.20:22122
# 当前存储节点监听的端口
 42 # the port of the local storage server
 43 # the default value is 23000
 44 storage_server_port=23000
 # 当前存储节点所属的组
 46 # the group name of the local storage server
 47 group_name=group1
# 客户端访问的url中是不是出现组名
 49 # if the url / uri including the group name
 50 # set to false when uri like /M00/00/00/xxx
 51 # set to true when uri like ${group_name}/M00/00/00/xxx, such as group1/M00/xxx
 52 # default value is false
 53 url_have_group_name = true
 # 当前存储节点存储路径的个数 参照 storage.conf
 55 # path(disk or mount point) count, default value is 1
 56 # must same as storage.conf
 57 store_path_count=1
# 详细的存储路径
 59 # store_path#, based 0, if store_path0 not exists, it's value is base_path
 60 # the paths must be exist
 61 # must same as storage.conf
 62 store_path0=/home/itcast/go/src/sss/fastdfs/storage_data


#完成修改后再次启动nginx 依然是1个nginx进程
$ sudo nginx
$ ps aux |grep nginx
root  106838  0.0  0.0 31384   764 ?     Ss 22:09   0:00 nginx: master process nginx
itcast 106841 0.0  0.0 21536  1048 pts/15  S+ 22:09   0:00 grep --color=auto nginx
#查看日志
$ cat /usr/local/nginx/logs/error.log
#发现新的错误
ERROR - file: ini_file_reader.c, line: 631, include file "http.conf" not exists,
line: "#include http.conf"
#那么说明我们的配置文件就改好了
```

注意5：新的错误

```
#上面我们查看了新的日志发现了新的错误
ERROR - file: ini_file_reader.c, line: 631, include file "http.conf" not exists,
line: "#include http.conf"
#错误 -文件 ： ini_file_reader.c 的第631行 导入文件的 http.conf 不存在
    - 原因是从/etc/fdfs下加载该http.conf文件没找到
    - 从fastdfs的源码安装目录中找
    - fastdfs源码安装目录/conf/http.conf
    - cp fastdfs源码安装目录/conf/http.conf /etc/fdfs
$ sudo cp ./http.conf /etc/fdfs/
#进入etc下验证是否成功
$ cd /etc/fdfs/
$ ls
```

```
client.conf.sample  http.conf  mod_fastdfs.conf  storage.conf.sample
storage_ids.conf.sample  tracker.conf.sample
#停掉之前启动的nginx
$ sudo nginx -s stop
#查看是否停止
$ ps aux |grep nginx
itcast 107419 0.0 0.0 21536 1040 pts/15 S+ 09:36  0:00 grep --color=auto nginx
#启动新的nginx
$ sudo nginx
#查看还是1个线程
$ ps aux |grep nginx
root   107422 0.0 0.0 31384  760 ?      Ss  09:36  0:00 nginx: master process nginx
itcast 107425 0.0 0.0 21536 1048 pts/15 S+  09:36  0:00 grep --color=auto nginx

#发现还是有错误所以我们再次查看日志
$ cat /usr/local/nginx/logs/error.log

#又一次出现了新的错误
ERROR - file: shared_func.c, line: 968, file /etc/fdfs/mime.types not exist
#说明我们http.conf的错误已经解决了
```

注意6：找不到文件

```
#ERROR - file: shared_func.c, line: 968, file /etc/fdfs/mime.types not exist
#错误 - file ： shared_func.c 文件的 第968 行 不能找到 /etc/fdfs/mime.types
- 从nginx的源码安装目录中找
- nginx源码安装目录/conf/mime.types
- cp nginx源码安装目录/conf/mime.types /etc/fdfs
#找到nginx源码安装目录的文件
$ cd /home/itcast/ffdfs/nginx-1.10.1/conf
$ ls
fastcgi.conf  fastcgi_params  koi-utf  koi-win  mime.types  nginx.conf  scgi_params
uwsgi_params  win-utf
#发现文件存在后将其拷贝到/etc/fdfs
$ sudo cp ./mime.types  /etc/fdfs/

#再次验证是否成功
#停掉之前启动的nginx
$ sudo nginx -s stop
#查看是否停止
$ ps aux |grep nginx
itcast 107488 0.0 0.0 21536 1092 pts/15  S+  09:52  0:00 grep --color=auto nginx
#启动nginx
$ sudo nginx
itcast@itcast-virtual-machine:~/ffdfs/nginx-1.10.1/conf$ ps aux |grep nginx
root   107491 0.0  0.0 31384   764 ?     Ss  09:53  0:00 nginx: master process
nginx
nobody 107492 0.0  0.0 36212  4016 ?      S   09:53  0:00 nginx: worker process
itcast 107494 0.0  0.0 21536  1152 pts/15 S+  09:53  0:00 grep --color=auto nginx
#发现已经出现了worker的进程说明我们的nginx安装环境算是成功了
```

注意7：可能出现的问题

```
#可能出现的问题
line: 177, "Permission denied" can't be accessed
#没有权限访问
    - 修改nginx.conf，将worker进程的所有者改为root
    - user root;
```

修改nginx的访问配置文件

```
#我们在虚拟机中访问我们的资源图片
http://127.0.0.1/group1/M00/00/00/wKhuFFuRWriATb4DAADprYlRZN0624.jpg
#会出现如下效果
```


```
#此时我们需要再次访问我们的日志文件
$ cat /usr/local/nginx/logs/error.log
# open()
"/usr/local/nginx/html/group1/M00/00/00/wKhuFFuRWriATb4DAADprYlRZN0624.jpg" failed
(2: No such file or directory), client: 127.0.0.1, server: localhost, request: "GET
/group1/M00/00/00/wKhuFFuRWriATb4DAADprYlRZN0624.jpg HTTP/1.1", host: "127.0.0.1"
#打开图片文件失败（没有这样的文件或者是目录） 客户端是 127.0.0.1 服务器是localhost
#发送请求"GET /group1/M00/00/00/wKhuFFuRWriATb4DAADprYlRZN0624.jpg HTTP/1.1"
#主机127.0.0.1
#为什么是404?
#nginx在解析客户端请求的时查找的资源目录是不对的，所以找不到我们想要的文件
#所以我们需要给nginx制定争取的目录

#nginx 处理 http://127.0.0.1/group1/M00/00/00/wKhuFFuRWriATb4DAADprYlRZN0624.jpg
#去掉ip与文件名，得到 /group1/M00/00/00/

#需要给修改nginx的配置文件添加location处理指令
    location /group1/M00/00/00/
    {
        # 指定正确的资源路径
        root /root/fdfs/storage/data; // M00映射的实际路径
        ngx_fastdfs_module; # 添加nginx对fastdfs模块的调用
    }

```

```
22
23
24  #进入nginx的配置文件目录下
25  $ cd /usr/local/nginx/conf
26  #备份 nginx.conf
27  $ sudo cp nginx.conf nginx.conf.old
28  #编辑nginx.conf
29  $ sudo vim nginx.conf
30  #修改内容为
31      listen       8888;
32      server_name  localhost;
33      #location  /group1/M00/ {
34      location  ~/group([0-9])/M00 {
35      #正确的数据路径
36          root   /home/itcast/go/src/sss/fastdfs/storage_data/data;
37          ngx_fastdfs_module; # 添加nginx对fastdfs模块的调用
38      }
39
40
41  #修改好文件之后加载文件则会出现1个新的进程就是这个插件的进程
42  $ sudo nginx -s reload
43  [sudo] itcast 的密码:
44  ngx_http_fastdfs_set pid=107791
45  #修改app.conf文件添加配置信息
46
47  #成功之后我们再次访问图片
48  http://127.0.0.1:8888/group1/M00/00/00/wKhuFFuRWriATb4DAADprYlRZN0624.jpg
49  #会出现以下效果
```

go操作fastdfs

安装库

```
go get github.com/weilaihui/fdfs_client
```

在models中创建fastdfs_client.go

```
package models

import (
    "github.com/weilaihui/fdfs_client"
    "github.com/astaxie/beego"
)


```

```go
//通过文件名的方式进行上传
func UploadByFilename( filename string)(GroupName,RemoteFileId string ,err error )
{
    //通过配置文件创建fdfs操作句柄
    fdfsClient, thiserr
:=fdfs_client.NewFdfsClient("/home/itcast/go/srcsss/homeweb/conf/client.conf")
    if thiserr  !=nil{
        //说一下那里出问题了
        beego.Info("UploadByFilename( ) fdfs_client.NewFdfsClient  err",err)
        GroupName = ""
        RemoteFileId = ""
        err = thiserr
        return
    }

    //unc (this *FdfsClient) UploadByFilename(filename string)
(*UploadFileResponse, error)
    //通过句柄上传文件（被上传的文件）

    uploadResponse, thiserr := fdfsClient.UploadByFilename(filename)
    if thiserr !=nil{
        beego.Info("UploadByFilename( ) fdfsClient.UploadByFilename(filename)
err",err)
        GroupName = ""
        RemoteFileId = ""
        err = thiserr
        return
    }

    beego.Info(uploadResponse.GroupName)
    beego.Info(uploadResponse.RemoteFileId)
    //回传
    return  uploadResponse.GroupName , uploadResponse.RemoteFileId ,nil

}

//功能函数 操作fdfs上传二进制文件
func UploadByBuffer(filebuffer []byte, fileExtName string)(GroupName,RemoteFileId
string ,err error ){

    //通过配置文件创建fdfs操作句柄
    fdfsClient, thiserr
:=fdfs_client.NewFdfsClient("/home/itcast/go/src/sss/homeweb/conf/client.conf")
    if thiserr  !=nil{
        beego.Info("UploadByBuffer( ) fdfs_client.NewFdfsClient  err",err)
        GroupName = ""
        RemoteFileId = ""
        err = thiserr
        return
    }

    //通过句柄上传二进制的文件
    uploadResponse, thiserr :=fdfsClient.UploadByBuffer(filebuffer,fileExtName)
```

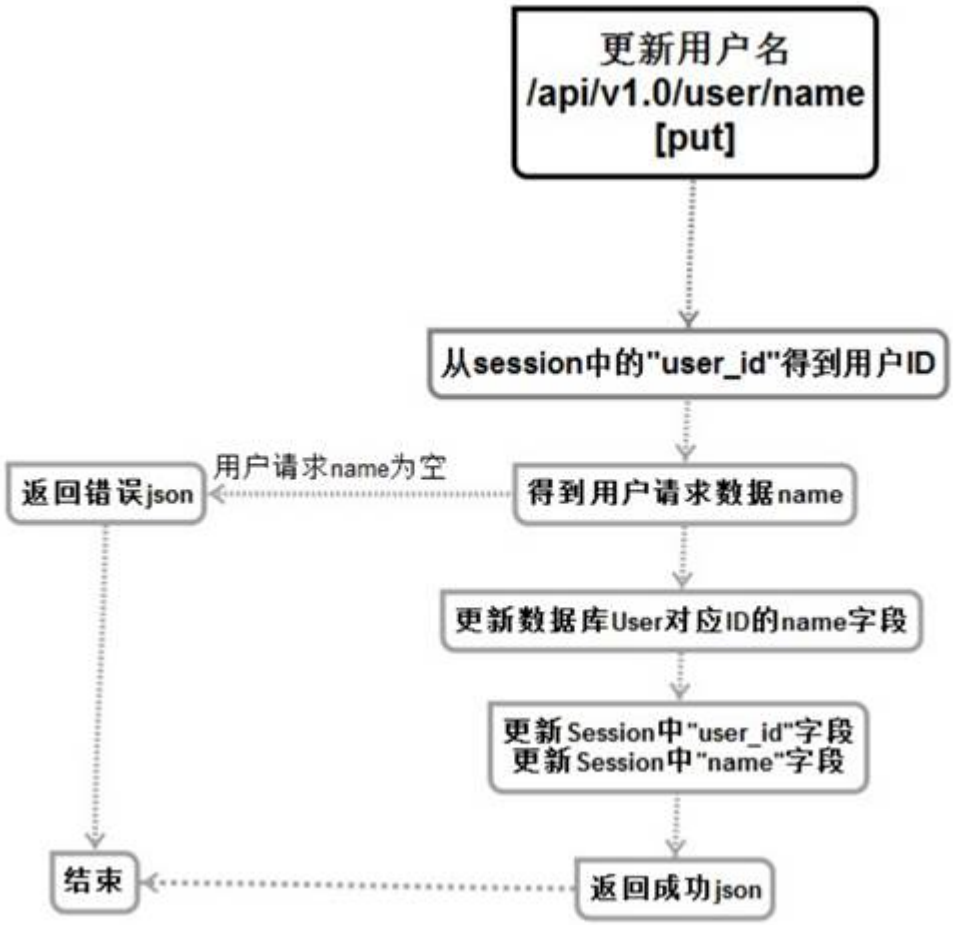
```
    if thiserr  !=nil{
        beego.Info("UploadByBuffer( ) fdfs_client.UploadByBuffer  err",err)
        GroupName = ""
        RemoteFileId = ""
        err = thiserr
        return
    }
    beego.Info(uploadResponse.GroupName)
    beego.Info(uploadResponse.RemoteFileId)
    //回传
    return uploadResponse.GroupName,uploadResponse.RemoteFileId,nil

}
```

11 更新用户名

创建命令

```
$ micro new --type "srv" sss/PutUserInfo
```

流程与接口




```
#Request:
method: PUT
url:api/v1.0/user/name
#data:
{
 "name":"panda"
}
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
    "name": "Panda"
  }
}

#返回失败:
{
    "errno": "400x",    //状态码
    "errmsg":"状态错误信息"
}
```

proto

```
service Example {
    rpc PutUserInfo(Request) returns (Response) {}
    rpc Stream(StreamingRequest) returns (stream StreamingResponse) {}
    rpc PingPong(stream Ping) returns (stream Pong) {}
}

message Message {
    string say = 1;
}

message Request {
    string Sessionid=1;
    string Username = 2 ;
}

message Response {
    string Errno = 1;
    string Errmsg = 2;
    string Username = 3;
}

```

web中添加路由

```
//请求更新用户名 PUT
rou.PUT("/api/v1.0/user/name",handler.PutUserInfo)
```

web中的handler添加函数

```
//更新用户名//PutUserInfo
func PutUserInfo(w http.ResponseWriter, r *http.Request,_ httprouter.Params) {
    beego.Info(" 更新用户名 Putuserinfo /api/v1.0/user/name")
    //创建服务
    service := grpc.NewService()
    service.Init()
    // 接收前端发送内容
    var request map[string]interface{}
    if err := json.NewDecoder(r.Body).Decode(&request); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }

    // 调用服务
    exampleClient := PUTUSERINFO.NewExampleService("go.micro.srv.PutUserInfo",
service.Client())

```

```
    //获取用户登陆信息
    userlogin,err:=r.Cookie("userlogin")
    if err != nil{
        resp := map[string]interface{}{
            "errno": utils.RECODE_SESSIONERR,
            "errmsg": utils.RecodeText(utils.RECODE_SESSIONERR),
        }

        w.Header().Set("Content-Type", "application/json")
        // encode and write the response as json
        if err := json.NewEncoder(w).Encode(resp); err != nil {
            http.Error(w, err.Error(), 503)
            beego.Info(err)
            return
        }
        return
    }


    rsp, err := exampleClient.PutUserInfo(context.TODO(), &PUTUSERINFO.Request{
        Sessionid:userlogin.Value,
        Username:request["name"].(string),
    })
    if err != nil {
        http.Error(w, err.Error(), 500)
        return
    }

    //接收回发数据
    data := make(map[string]interface{})
    data["name"]=rsp.Username


    response := map[string]interface{}{
        "errno": rsp.Errno,
        "errmsg": rsp.Errmsg,
        "data":data,
    }
    w.Header().Set("Content-Type", "application/json")

    // 返回前端
    if err := json.NewEncoder(w).Encode(response); err != nil {
        http.Error(w, err.Error(), 501)
        return
    }
}

```

服务端

```
func (e *Example) PutUserInfo(ctx context.Context, req *example.Request, rsp
*example.Response) error {

```

```
    //打印被调用的函数
    beego.Info("---------------- PUT  /api/v1.0/user/name PutUersinfo() ----------
-------")

    //创建返回空间
    rsp.Errno= utils.RECODE_OK
    rsp.Errmsg = utils.RecodeText(rsp.Errno)

    /*得到用户发送过来的name*/
    beego.Info(rsp.Username)

    /*从从sessionid获取当前的userid*/
    //连接redis
    redis_config_map := map[string]string{
        "key":utils.G_server_name,
        //"conn":"127.0.0.1:6379",
        "conn":utils.G_redis_addr+":"+utils.G_redis_port,
        "dbNum":utils.G_redis_dbnum,
    }
    beego.Info(redis_config_map)
    redis_config ,_:=json.Marshal(redis_config_map)
    beego.Info( string(redis_config) )

    //连接redis数据库 创建句柄
    bm, err := cache.NewCache("redis", string(redis_config) )
    if err != nil {
        beego.Info("缓存创建失败",err)
        rsp.Errno  =  utils.RECODE_DBERR
        rsp.Errmsg  = utils.RecodeText(rsp.Errno)
        return  nil
    }
    //拼接key
    sessioniduserid :=  req.Sessionid + "user_id"
    //获取userid
    value_id :=bm.Get(sessioniduserid)
    beego.Info(value_id,reflect.TypeOf(value_id))

    id :=  int(value_id.([]uint8)[0])
    beego.Info(id ,reflect.TypeOf(id))

    //创建表对象
    user:=models.User{Id:id,Name:req.Username}
    /*更新对应user_id的name字段的内容*/
    //创建数据库句柄
    o:= orm.NewOrm()
    //更新
    _ , err =o.Update(&user ,"name")
    if err !=nil{
        rsp.Errno= utils.RECODE_DBERR
        rsp.Errmsg = utils.RecodeText(rsp.Errno)

        return nil
    }
```

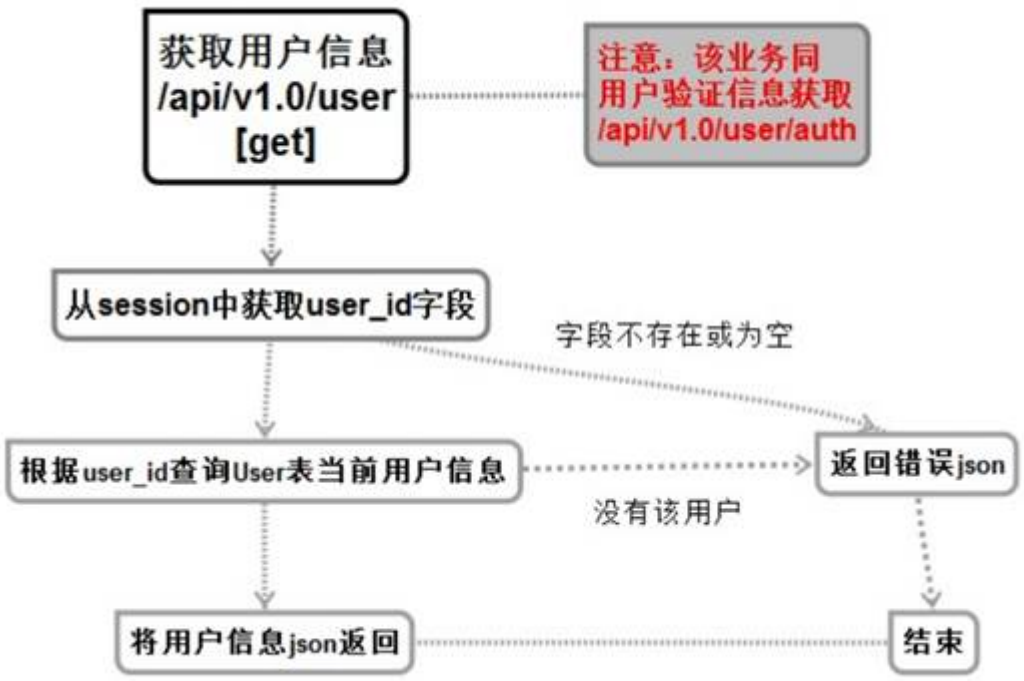
```
55
56      /*更新session user_id*/
57      sessionidname :=  req.Sessionid + "name"
58      bm.Put(sessioniduserid,string(user.Id),time.Second*600)
59      /*更新session name*/
60      bm.Put(sessionidname,string(user.Name),time.Second*600)
61
62      /*成功返回数据*/
63      rsp.Username = user.Name
64      return nil
65  }
66
```

12 检查用户实名认证

创建命令

```
1  $ micro new --type "srv" sss/GetUserAuth
```

流程与接口




```
1
```

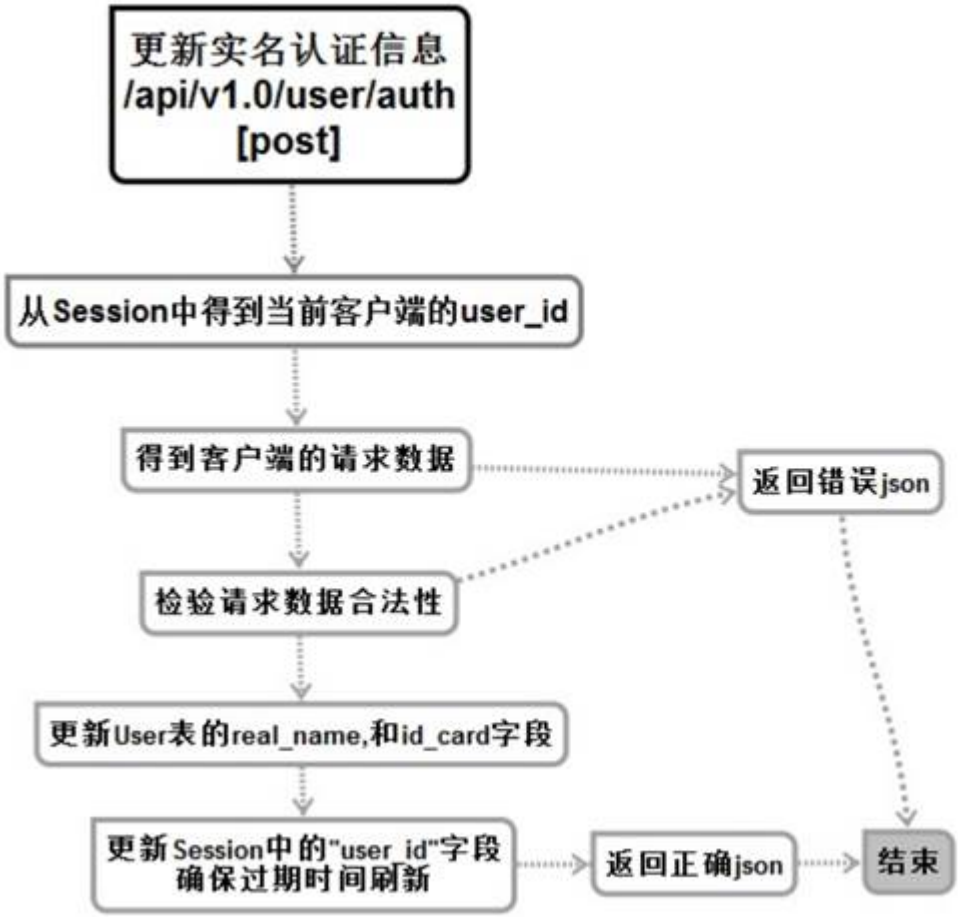
```
#Request:
method: GET
url:api/v1.0/user/auth
#data:
no input data
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
    "user_id": 1,
    "name": "Panda",
    "password": "123123",
    "mobile": "110",
    "real_name": "熊猫",
    "id_card": "210112244556677",
    "avatar_url":
"http://101.200.170.171:9998/group1/M00/00/00/Zciqq1n7It2ANn1dAADexS5wJKs808.png"
  }
}

#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

13 更新实名认证信息

创建命令

```
$ micro new -type "srv" sss/PostUserAuth
```

流程与接口



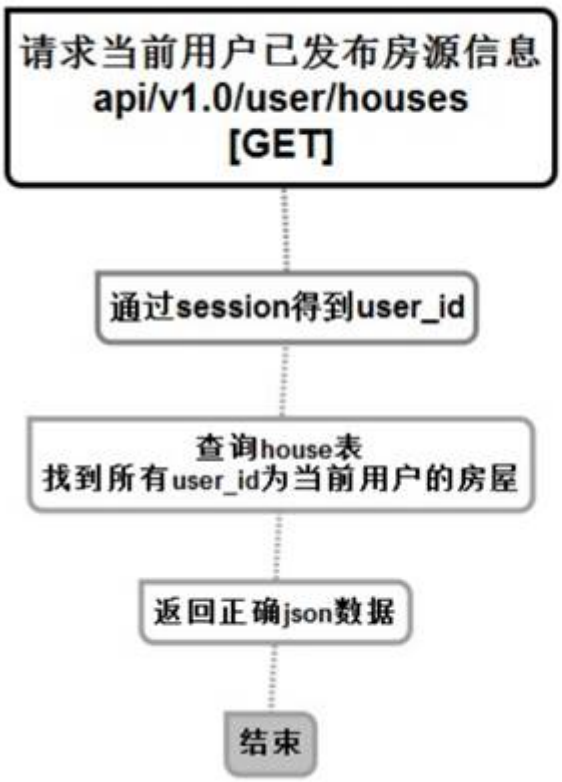

```
#Request:
method: POST
url:api/v1.0/user/auth
#data:
{
    real_name: "熊猫",
    id_card: "21011223344556677"
}
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功"
}
#返回失败:
{
    "errno": "400x",    //状态码
    "errmsg":"状态错误信息"
}
```

14 获取当前用户已发布房源信息

创建命令

```
$ micro new -type "srv" sss/GetUserHouses
```

流程与接口




```
#Request:
method: GET
url:api/v1.0/user/houses
#data:
no input data
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
    "houses": [
      {
        "address": "西三旗桥东",
        "area_name": "昌平区",
        "ctime": "2017-11-06 11:16:24",
        "house_id": 1,
```

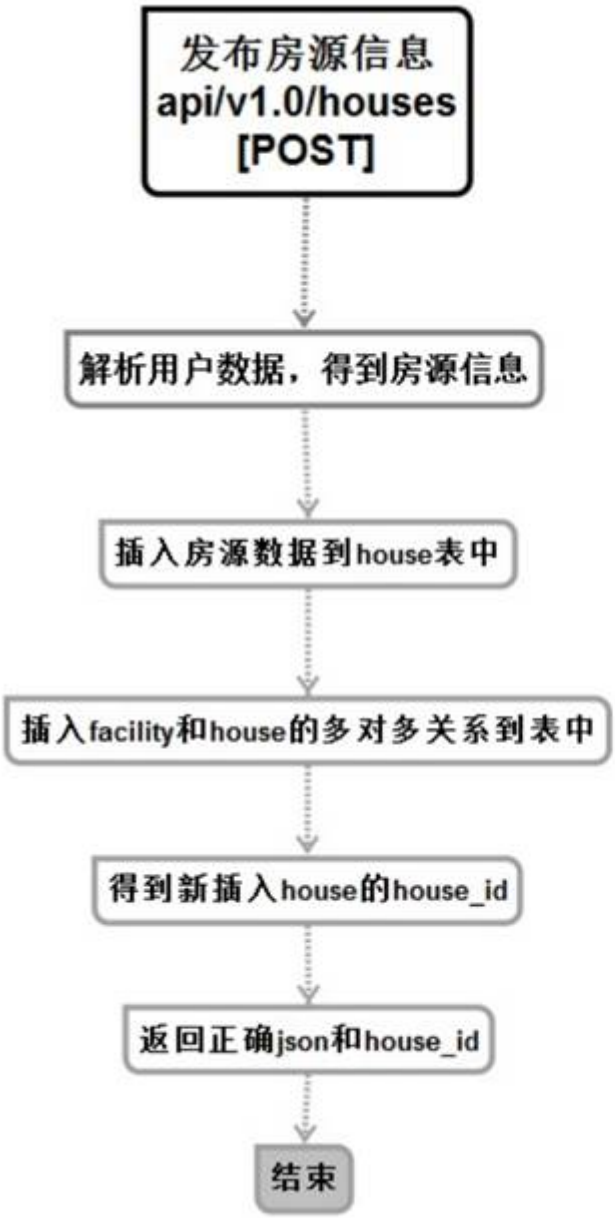
```
        "img_url": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBJY-AL3m8AAS8K2x8TDE052.jpg",
        "order_count": 0,
        "price": 100,
        "room_count": 2,
        "title": "上奥世纪中心",
        "user_avatar": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBLFeALIEjAADexS5wJKs340.png"
      },
      {
        "address": "北清路郑上路",
        "area_name": "顺义区",
        "ctime": "2017-11-06 11:38:54",
        "house_id": 2,
        "img_url": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBKtmAC8y8AAZcKg5PznU817.jpg",
        "order_count": 0,
        "price": 1000,
        "room_count": 1,
        "title": "修正大厦302教室",
        "user_avatar": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBLFeALIEjAADexS5wJKs340.png"
      }
    ]
  }
}

#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

15 发布房源信息

创建命令

```
$ micro new --type "srv" sss/PostHouses
```

流程与接口




```
#Request:
method: POST
url:api/v1.0/houses
#data:
{
"title":"上奥世纪中心",
"price":"666",
"area_id":"5",
"address":"西三旗桥东建材城1号",
"room_count":"2",
"acreage":"60",
"unit":"2室1厅",
"capacity":"3",
"beds":"双人床2张",
"deposit":"200",
"min_days":"3",
"max_days":"0",
```

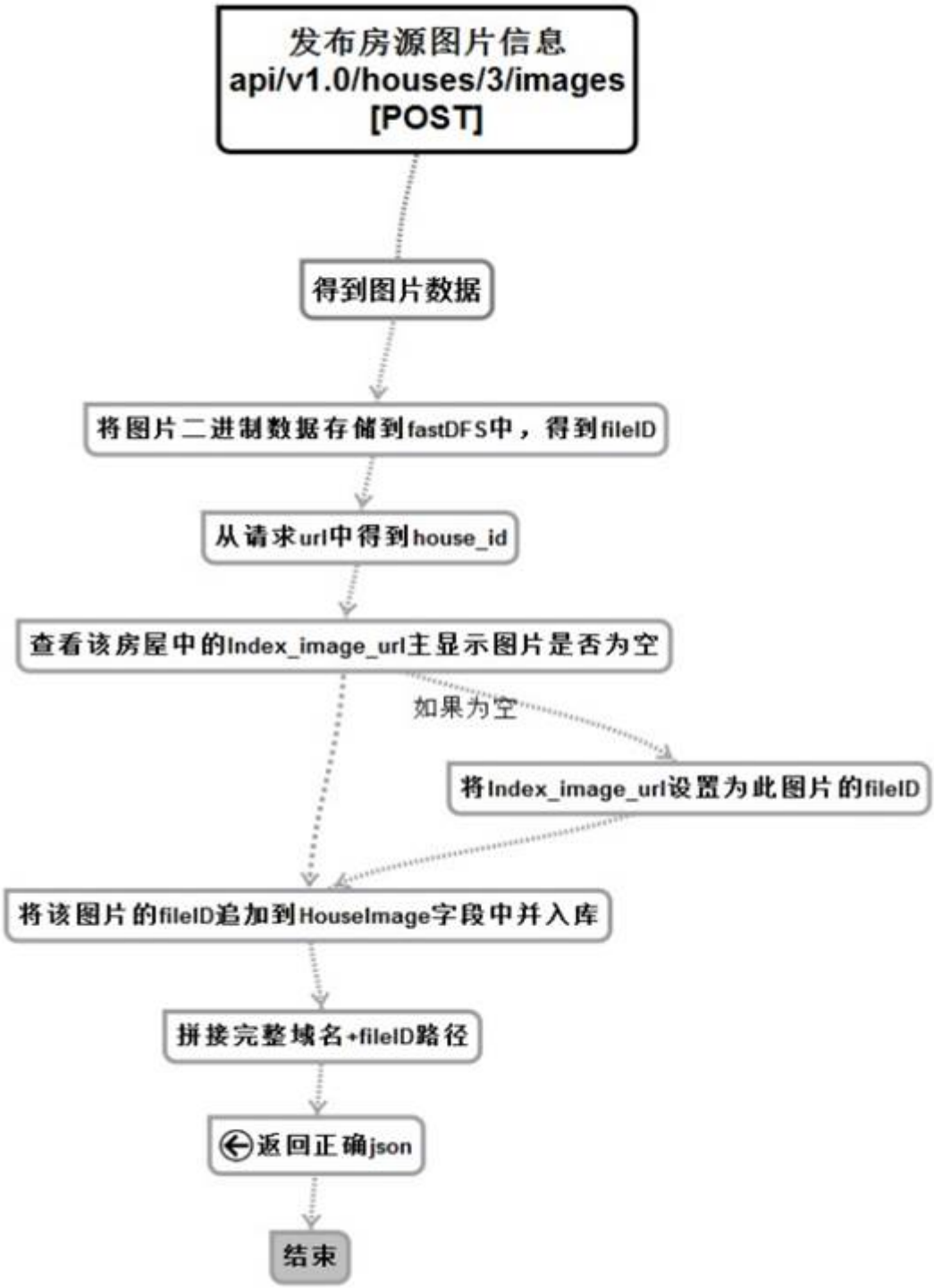
```
"facility":["1","2","3","7","12","14","16","17","18","21","22"]
}
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功"
  "data" :{
        "house_id": "1"
  }
}
#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

16 上传房屋图片

创建命令

```
$ micro new --type "srv" sss/PostHousesImage
```

流程与接口




```
#Request:
method: POST
#3表示房源id
url:api/v1.0/houses/3/images
url:api/v1.0/houses/:id/images

#data:
图片二进制数据
#Response
#返回成功:
```

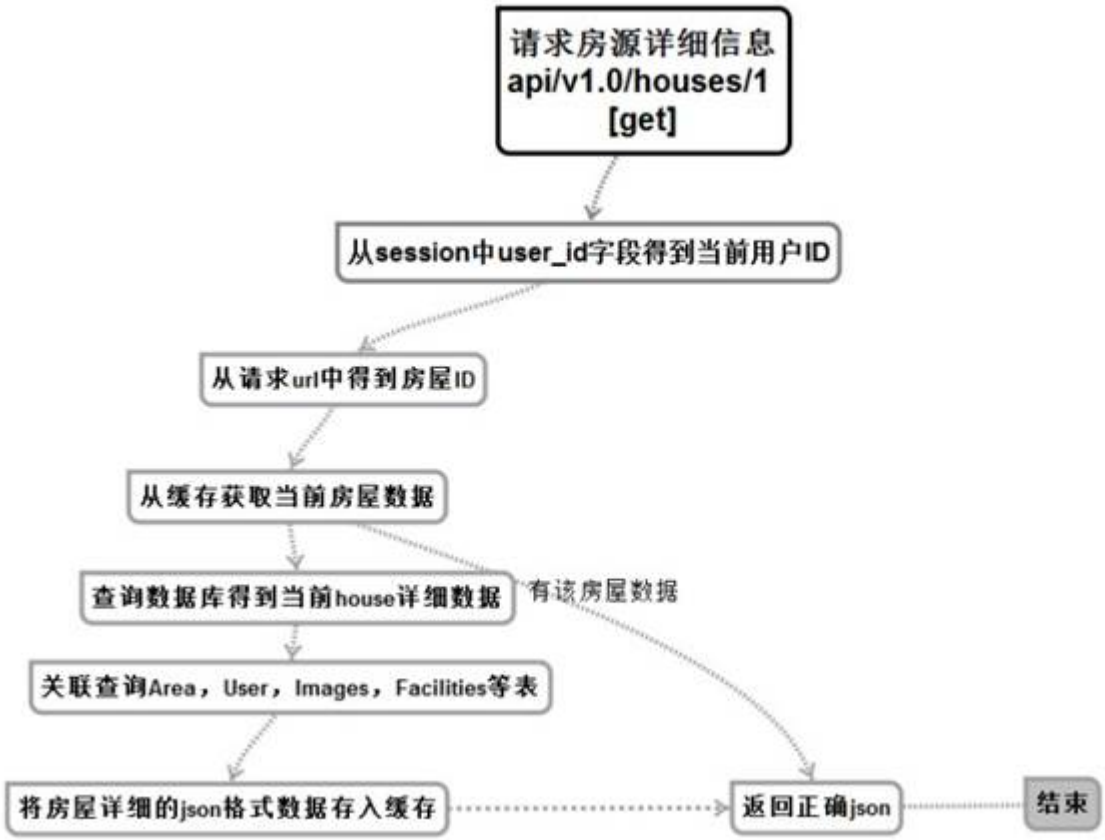
```
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
    "url": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBLmWAHlsrAAaInSze-cQ719.jpg"
  }
}
#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

17 获取房源详细信息

创建命令

```
$ micro new --type "srv" sss/GetHouseInfo
```

流程与接口




```
#Request:
method: GET
#1表示房源id
url:api/v1.0/houses/1
url:api/v1.0/houses/:id
#data:
no input data
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
    "house": {
      "acreage": 80,
      "address": "西三旗桥东",
      "beds": "2双人床",
      "capacity": 3,
      "comments": [
        {
          "comment": "评论的内容",
          "ctime": "2017-11-12 12:30:30",
          "user_name": "评论人的姓名"
        },
        {
          "comment": "评论的内容",
```

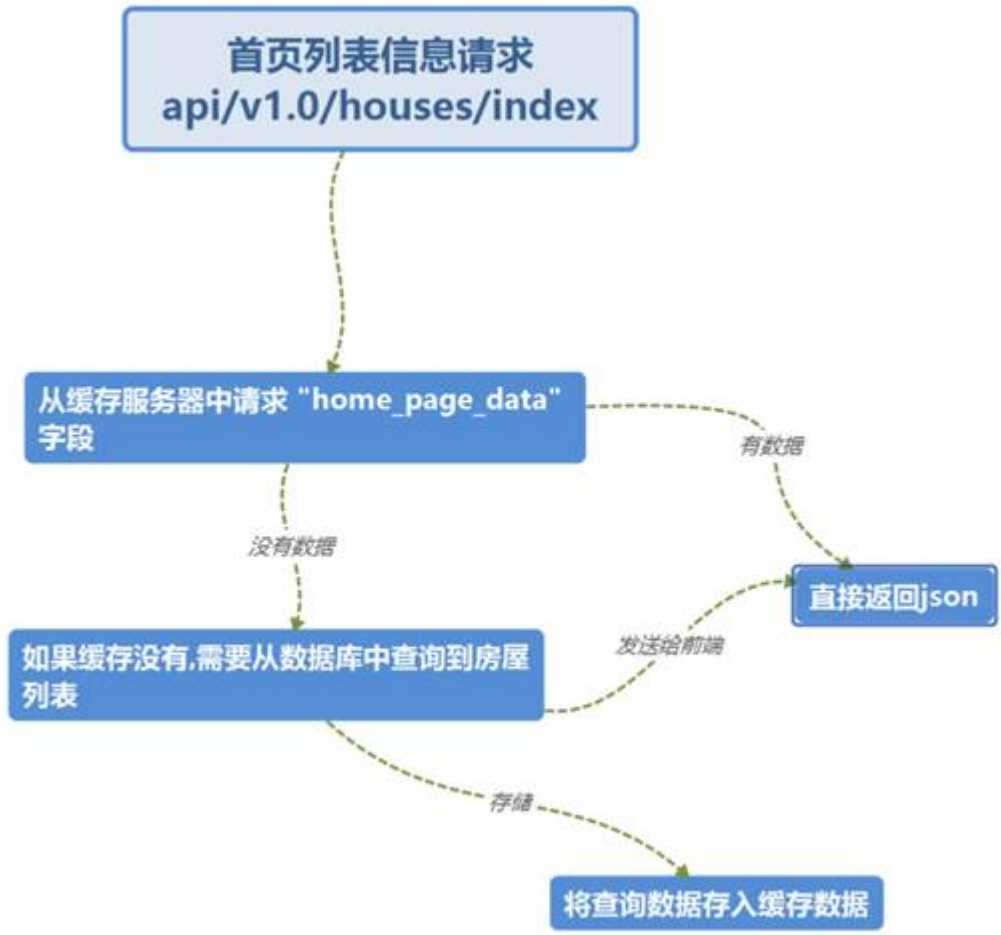
```
          "ctime": "2017-11-12 12:30:30",
          "user_name": "评论人的姓名"
        },
        {
          "comment": "评论的内容",
          "ctime": "2017-11-12 12:30:30",
          "user_name": "评论人的姓名"
        }
      ],
      "deposit": 200,
      "facilities": [9,11,13,16,19,20,21,23],
      "hid": 1,
      "img_urls": [
        "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBJY-AL3m8AAS8K2x8TDE052.jpg",
        "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBJZmAYqGWAAaInSze-cQ230.jpg"
      ],
      "max_days": 30,
      "min_days": 1,
      "price": 100,
      "room_count": 2,
      "title": "上奥世纪中心",
      "unit": "3室3厅",
      "user_avatar": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBLFeALIEjAADexS5wJKs340.png",
      "user_id": 1,
      "user_name": "Panda"
    },
    "user_id": 1
  }
}

#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

18 获取首页动画图片

创建命令

```
$ micro new --type "srv" sss/GetIndex
```

流程与接口




```
#Request:
method: GET
url:api/v1.0/houses/index
#data:
no input data
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
        "houses": [
      {
        "house_id":    this.Id,
        "title":       this.Title,
        "price":       this.Price,
        "area_name":   this.Area.Name,
        "img_url":     utils.AddDomain2Url(this.Index_image_url),
        "room_count":  this.Room_count,
        "order_count": this.Order_count,
        "address":     this.Address,
        "user_avatar": utils.AddDomain2Url(this.User.Avatar_url),
        "ctime":       this.Ctime.Format("2006-01-02 15:04:05"),
```

```
    },
      {
        "house_id":    this.Id,
        "title":       this.Title,
        "price":       this.Price,
        "area_name":   this.Area.Name,
        "img_url":     utils.AddDomain2Url(this.Index_image_url),
        "room_count":  this.Room_count,
        "order_count": this.Order_count,
        "address":     this.Address,
        "user_avatar": utils.AddDomain2Url(this.User.Avatar_url),
        "ctime":       this.Ctime.Format("2006-01-02 15:04:05"),
    }
    ],

  }
}


#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

19 搜索房源

创建命令

```
$ micro new --type "srv" sss/GetHouses
```

流程与接口


```
#Request:
method: GET
#adi表示地区编号
#sd表示起始日期
#ed表示结束日期
#sk表示查询方式
#p表示页码
url:api/v1.0/houses?aid=5&sd=2017-11-12&ed=2017-11-30&sk=new&p=1
#data:
no input data
#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
    "current_page": 1,
    "houses": [
      {
```

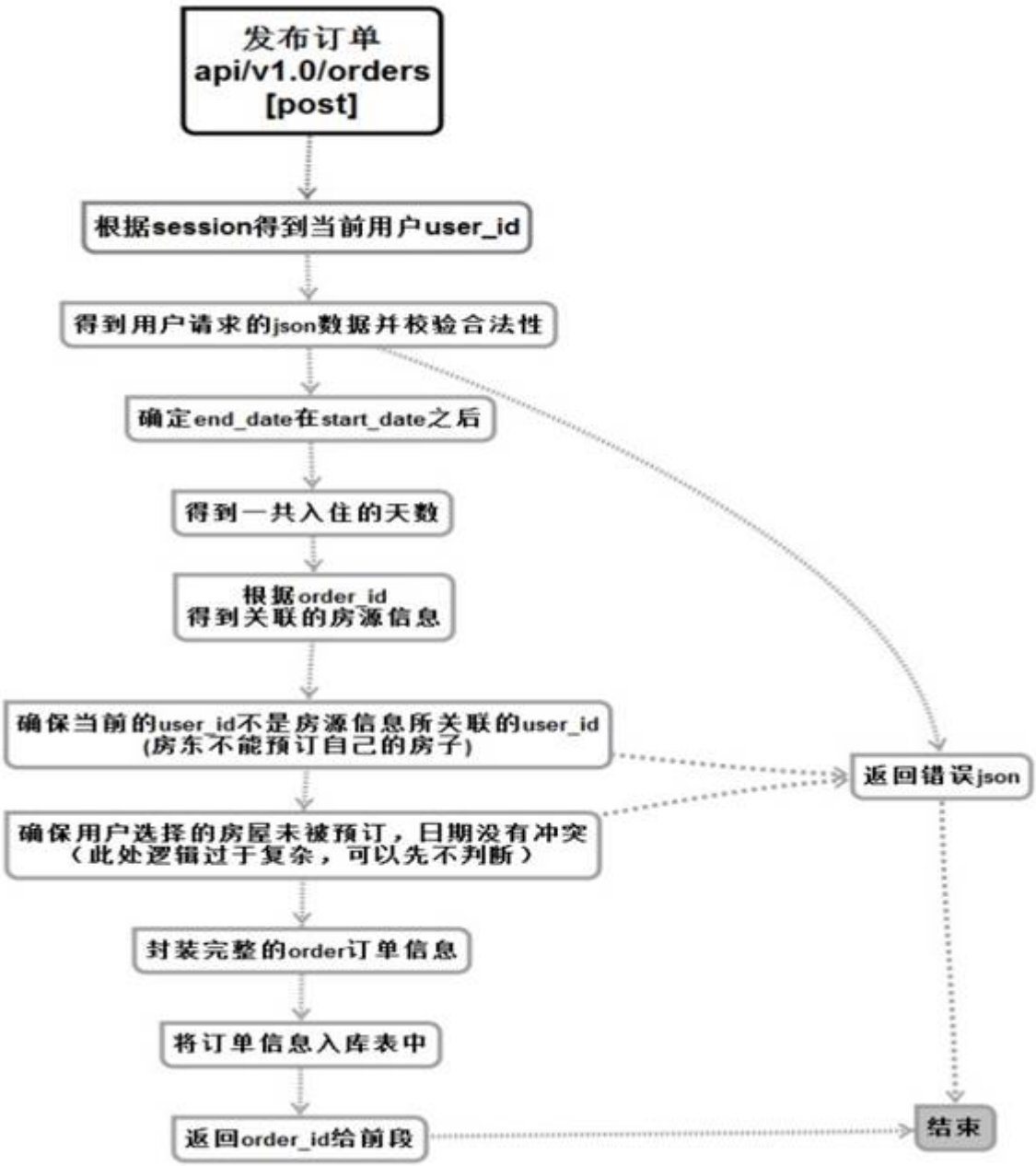
```
        "address": "西三旗桥东",
        "area_name": "昌平区",
        "ctime": "2017-11-06 11:16:24",
        "house_id": 1,
        "img_url": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBJY-AL3m8AAS8K2x8TDE052.jpg",
        "order_count": 0,
        "price": 100,
        "room_count": 2,
        "title": "上奥世纪中心13号楼",
        "user_avatar": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBLFeALIEjAADexS5wJKs340.png"
      },
      {
        "address": "西三旗桥东",
        "area_name": "昌平区",
        "ctime": "2017-11-06 11:16:24",
        "house_id": 1,
        "img_url": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBJY-AL3m8AAS8K2x8TDE052.jpg",
        "order_count": 0,
        "price": 100,
        "room_count": 2,
        "title": "上奥世纪中心18号楼",
        "user_avatar": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBLFeALIEjAADexS5wJKs340.png"
        }
    ],
    "total_page": 1
  }
}

#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

20 发布订单

创建命令

```
$ micro new --type "srv" sss/PostOrders
```

流程与接口




```
#Request:
method: POST
url:api/v1.0/orders
#data:
{
  "house_id": "1",
  "start_date": "2017-11-11 21:23:49",
  "end_date": "2017-11-12 21:23:49",
}

#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
```

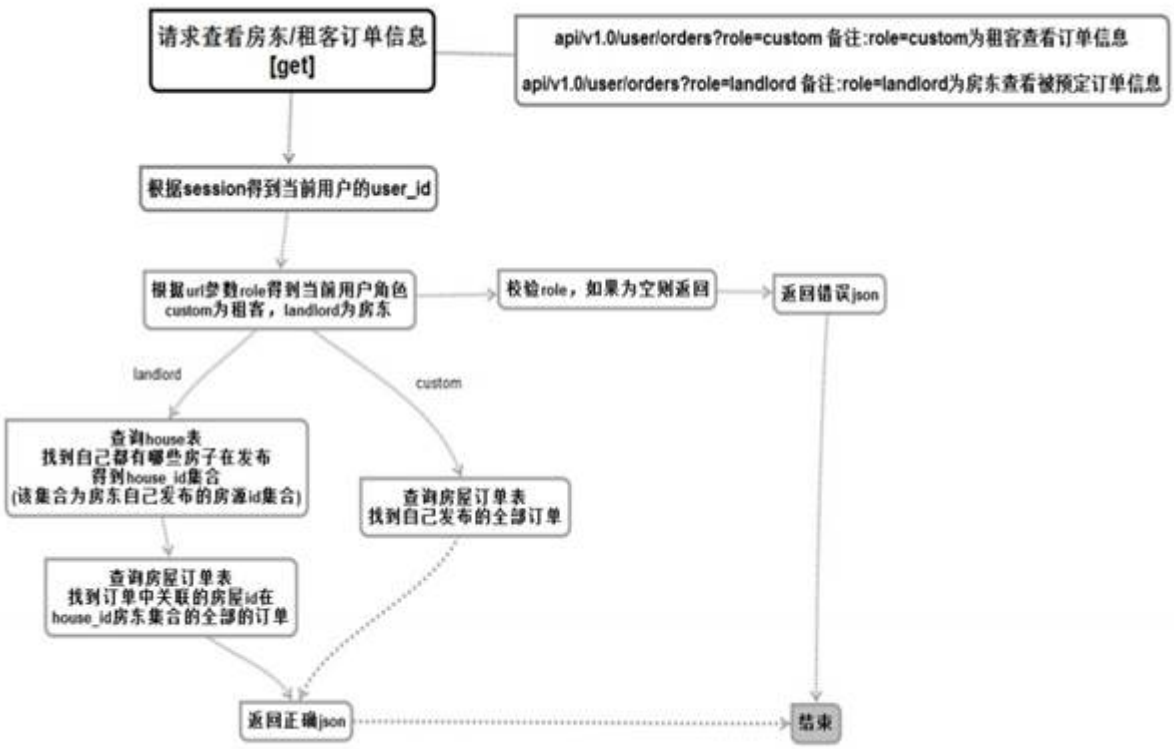
```
  "data": {
    "order_id":"1"
  }
}

#返回失败:
{
    "errno": "400x",    //状态码
    "errmsg":"状态错误信息"
}
```

21 请求查看房东/租客订单信息

创建命令

```
$ micro new --type "srv" sss/GetUserOrder
```

流程与接口




```
#Request:
method: GET
url:api/v1.0/user/orders?role=custom 备注:role=custom为租客查看订单信息
```

```
url:api/v1.0/user/orders?role=landlord 备注:role=landlord为房东查看被预定订单信息
#data:
no input data

#Response
#返回成功:
{
  "errno": "0",
  "errmsg": "成功",
  "data": {
    "orders": [
      {
        "amount": 200,
        "comment": "哈哈拒接",
        "ctime": "2017-11-11 21:23:49",
        "days": 2,
        "end_date": "2017-11-29 16:00:00",
        "img_url": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBJY-AL3m8AAS8K2x8TDE052.jpg",
        "order_id": 3,
        "start_date": "2017-11-28 16:00:00",
        "status": "REJECTED",//WAIT_ACCPET,WAIT_COMMENT,REJECTED,COMPLETE,CANCELED
        "title": "上奥世纪中心"
      },
      {
        "amount": 1500,
        "comment": "",
        "ctime": "2017-11-11 01:32:10",
        "days": 15,
        "end_date": "2017-11-24 16:00:00",
        "img_url": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBJY-AL3m8AAS8K2x8TDE052.jpg",
        "order_id": 2,
        "start_date": "2017-11-10 16:00:00",
        "status": "WAIT_COMMENT",
        "title": "上奥世纪中心"
      },
      {
        "amount": 300,
        "comment": "",
        "ctime": "2017-11-10 01:46:00",
        "days": 3,
        "end_date": "2017-11-11 16:00:00",
        "img_url": "http://101.200.170.171:9998/group1/M00/00/00/Zciqq1oBJY-AL3m8AAS8K2x8TDE052.jpg",
        "order_id": 1,
        "start_date": "2017-11-09 16:00:00",
        "status": "WAIT_COMMENT",
        "title": "上奥世纪中心"
      }
    ]
  }
}
```

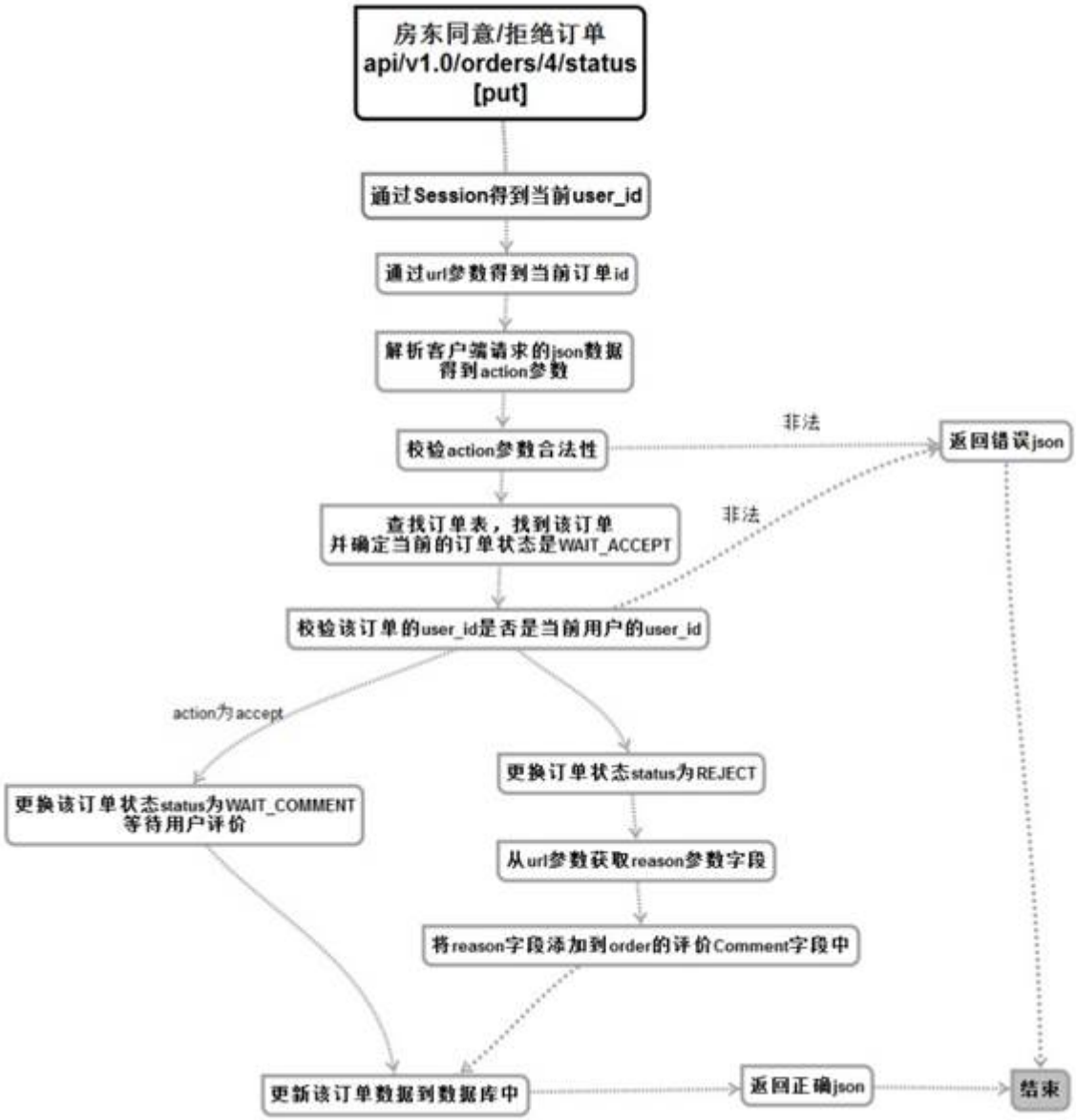
```
54
55
56 #返回失败:
57 {
58     "errno": "400x",   //状态码
59     "errmsg":"状态错误信息"
60 }
```

22 房东同意/拒绝订单

创建命令

```
1 $ micro new --type "srv" sss/PutOrders
```

流程与接口




```
#Request:
method: PUT
#4表示订单id
url:api/v1.0/orders/4/status
url:api/v1.0/orders/:id/status
#data:
#"accept"表示接受
#"reject"表示拒绝
{action: "accept"}

#Response
#返回成功:

{
  "errno": "0",
```

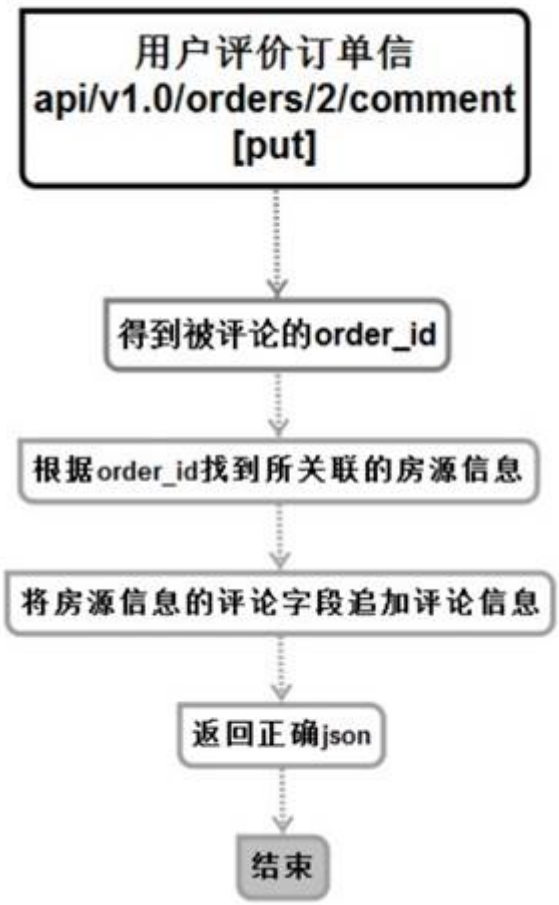
```
  "errmsg": "成功"
}

#返回失败:
{
    "errno": "400x",    //状态码
    "errmsg":"状态错误信息"
}
```

23 用户评价订单信息

创建命令

```
$ micro new --type "srv" sss/PutComment
```

流程与接口




```

```

```
#Request:
method: PUT
#2表示订单id
url:api/v1.0/orders/2/comment
url:api/v1.0/orders/:id/comment
#data:
{
    order_id: "2",
    comment: "烂房子！"
}
#Response
#返回成功:

{
  "errno": "0",
  "errmsg": "成功"
}

#返回失败:
{
    "errno": "400x",   //状态码
    "errmsg":"状态错误信息"
}
```

使用docker-compose进行单机集群启动

docker的安装

```
#安装基本软件
$apt-get update
$apt-get install apt-transport-https ca-certificates curl software-properties-common lrzsz -y
#使用阿里云的源{推荐}
$ sudo curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb_release -cs) stable"
#软件源升级
$ sudo apt-get update

#安装docker
$ sudo apt-get install docker-ce -y
#测试docker
docker version
#加速器配置
$ curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://f1361db2.m.daocloud.io
#修改配置文件
$ sudo vim /etc/docker/daemon.json
```

```
#文件内容
{"registry-mirrors": ["http://f1361db2.m.daocloud.io"], "insecure-registries": []}

#修改权限
#如果还没有 docker group 就添加一个:
$sudo groupadd docker
#将用户加入该 group 内。然后退出并重新登录就生效啦。
$sudo gpasswd -a ${USER} docker
#重启 docker 服务
$systemctl restart docker
#切换当前会话到新 group 或者重启 X 会话
$newgrp - docker
#注意:最后一步是必须的，否则因为 groups 命令获取到的是缓存的组信息，刚添加的组信息未能生效，
#所以 docker images 执行时同样有错。
```

docker-compose的安装

```
#安装依赖工具
sudo apt-get install python-pip -y
#安装编排工具
sudo pip install docker-compose
#查看编排工具版本
sudo docker-compose version
#查看命令帮助
docker-compose --help
#用pip安装依赖包时默认访问https://pypi.python.org/simple/,
#但是经常出现不稳定以及访问速度非常慢的情况，国内厂商提供的pipy镜像目前可用的有:

#在当前用户目录下创建.pip文件夹
mkdir ~/.pip
#然后在该目录下创建pip.conf文件填写:
[global]
trusted-host=mirrors.aliyun.com
index-url=http://mirrors.aliyun.com/pypi/simple/
```

部署前的项目修改

代码的修改

将所有读取conf文件夹内容部的程序的绝对路径修改为相对路径

```
//fastdfs中的读取client.conf
fdfs_client.NewFdfsClient("./conf/client.conf")
//utils中读取app.conf
config.NewConfig("ini", "./conf/app.conf")

```

将conf文件复制到各个微服务项目文件夹中

```
#仅将所需要的配置文件进行拷贝就可以了
.
├── app.conf    #项目配置信息
└── client.conf #fastdfs客户端配置信息
```

项目的编译

```
#二进制编译
$ CGO_ENABLED=0 GOOS=linux  /usr/local/go/bin/go build -a -installsuffix cgo -ldflags '-w' -i -o ihomeweb ./main.go
#编译需要在root账户下进行
#指明cgo工具是否可用的标识在这里表示禁用
CGO_ENABLED=0
#目标平台（编译后的目标平台）的操作系统（darwin、freebsd、linux、windows）
GOOS=linux
#由于没有在root下安装go所以我们需要使用go的绝对路径进行使用
/usr/local/go/bin/go build
#强制重新编译所有涉及的go语言代码包
-a
#为了使当前的输出目录与默认的编译输出目录分离，可以使用这个标记。此标记的值会作为结果文件的父目录名称的后缀。
-installsuffix

cgo
# 给 cgo指定命令
-ldflags
#关闭所有警告信息
'-w'
#标志安装目标的依赖包。
-i
#命名
-o ihomeweb
#编译的main.go地址
./main.go
```

服务容器化

web

```
FROM alpine:3.2
#拷贝文件
ADD conf /conf
#拷贝文件
ADD html /html
#拷贝二进制
ADD ihomeweb /ihomeweb
WORKDIR /

ENTRYPOINT [ "/ihomeweb" ]

EXPOSE 8999
```

srv

```
FROM alpine:3.2
ADD conf /conf

ADD getarea-srv /getarea-srv
ENTRYPOINT [ "/getarea-srv" ]
```

Compose编排

```
consul:
  #覆盖启动后的执行命令
  command: agent -server -bootstrap-expect=1  -node=node1 -client 0.0.0.0 -ui -bind=0.0.0.0 -join 127.0.0.2
  #command: agent -server -bootstrap -rejoin -ui
  #镜像：镜像名称:版本号
  image: consul:latest
  #主机名
  hostname: "registry"
  #暴露端口
  ports:
  - "8300:8300"
  - "8400:8400"
  - "8500:8500"
  - "8600:53/udp"

#web主页
web:
  #覆盖启动后的执行命令
  command: --registry_address=registry:8500 --register_interval=5 --register_ttl=10 web
  #镜像构建的dockerfile文件地址
  build: ./ihomeweb
  links:
  - consul
  ports:
```

```
  - "8999:8999"
#获取地区
getarea:
  #覆盖启动后的执行命令
  command: --registry_address=registry:8500 --register_interval=5 --register_ttl=10
web
  #镜像构建的dockerfile文件地址
  build: ./getarea
  links:
  - consul

#注册三部曲
getimagecd:
  #覆盖启动后的执行命令
  command: --registry_address=registry:8500 --register_interval=5 --register_ttl=10
web
  #镜像构建的dockerfile文件地址
  build: ./getimagecd
  links:
  - consul

getsmscd:
  #覆盖启动后的执行命令
  command: --registry_address=registry:8500 --register_interval=5 --register_ttl=10
web
  #镜像构建的dockerfile文件地址
  build: ./getsmscd
  links:
  - consul

postret:
  #覆盖启动后的执行命令
  command: --registry_address=registry:8500 --register_interval=5 --register_ttl=10
web
  #镜像构建的dockerfile文件地址
  build: ./postret
  links:
  - consul

```

```
```