如鹏网
《.Net Core 微服务》
第二版
课件

购买配套课程请访问如鹏网：http://www.rupeng.com

感谢：张善友、朱永光、Mike、蒋金楠、Lemon 等的支持、修改、建议、帮助；

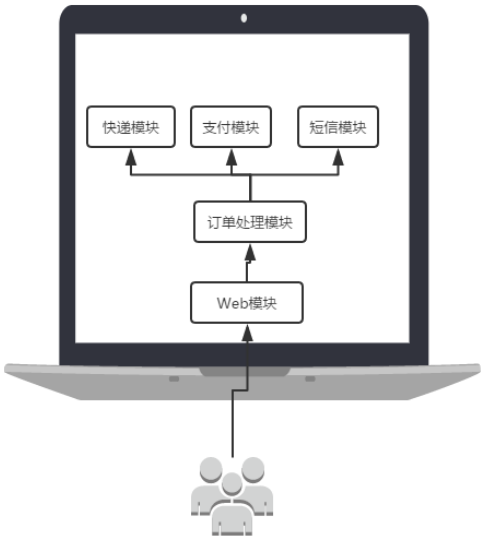
本次直播课程的盈利将会全部捐献给.Net Core 微服务相关开源社区。

1、微服务架构：“一解释就懂，一问就不知，一讨论就吵架”。才疏学浅，我讲的可能都是错的，欢迎讨论，请勿谩骂。
2、本课程适合哪些人群？
本课程适合熟悉 ASP.Net MVC 项目开发但是不熟悉微服务的开发者，特别是需要对 lambda、异步编程、反射、AOP 等熟悉。由于不是针对初学者，所以不需要所有地方都手把手敲，有一些地方是参考文档或者给出扩展资料。
如果您还是.Net 初学者或者没有项目开发经验，请报名如鹏就业训练营 http://www.rupeng.com
如果您已经是微服务高手，本课程也不适合您。
3、和如鹏系统培训的学生不一样，我不知道大家学过什么、没学过什么，所以有的地方可能你感觉讲太快，有的地方太啰嗦。请理解。
4、本课程不讲什么？
本课程主要目的是全面讲解“微服务”，会讲解涉及到的.Net Core 基础，即使您不熟悉.Net Core 也可以学习，但是不会深入讲解.Net Core 细节。如鹏网.Net 提高班系统讲解了.Net Core 等技术。
本课程中会讲解基于 Identity Server 实现 Ocelot 的服务授权，但是不会讲解 Identity Server 的其他内容。
5、我讲课的风格是循序渐进的，因此别急“你怎么不用***”，比如 HttpClient 最佳用法、Ocelot+Consul 等。
6、受限于篇幅，不可能事无巨细，比如有的地方服务器等是写死在代码中的，你根据需要写到配置中即可；没有对参数做合法性校验等。
7、软件更新较快，所以讲课时候的版本可能和大家练习的时候不一样的兼容性问题，因此练习的时候尽可能指定组件版本。
8、开发工具：VS2017+，安装.Net Core2.1+插件(https://www.microsoft.com/net/learn/get-started/windows)；安装时候没有安装的可以到控制面板中修改。
9、正版服务：答疑、方案相关指导、其他课程、社区、资源。

目录

什么是微服务架构？

下面是传统单体结构，整个系统运行在一个进程中：

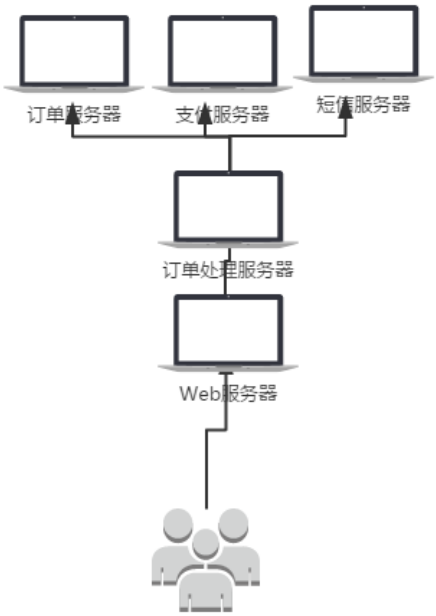


缺点：

1) 只能采用同一种技术，很难用不同的语言或者语言不同版本开发不同模块；
2) 系统耦合性强，一旦其中一个模块有问题，整个系统就瘫痪了；一旦升级其中一个模块，整个系统就停机了；
3) 要上线必须一起上线，互相等待，无法快速响应需求；
4) 集群只能是复制整个系统，即使只是其中一个模块压力大。

随着现在 IT 系统规模的扩大、模块的剧增，传统的系统架构已经难以满足要求，因此近几年微服务架构开始流行。

下面是微服务（Micro-Service）架构的示意图，不同模块放到不同的进程/服务器上，模块之间通过网络通讯进行协作。



优点：

1) 可以用不同的语言或者语言不同版本开发不同模块；
2) 系统耦合性弱，其中一个模块有问题，可以通过“降级熔断”等手段来保证系统不雪崩；
3) 可以独立上线，能够迅速响应需求；
4) 可以对不同模块用不同的集群策略，哪里慢集群哪里。

缺点：

1) 开发难度大，系统结构更复杂；
2) 运行效率低；

不仅是阿里、腾讯等大公司在大面积使用微服务架构，很多中小公司的 IT 系统架构也是微服务架构流行了。但是不是所有项目都适合微服务，没有“银弹”。

微服务架构要处理哪些问题：服务间通讯；服务治理与服务发现；网关和安全认证；限流与容错；监控等；

第一代微服务：Dubbo(Java)、Orleans(.Net)等；第二代微服务：Spring Cloud 等；第三代微服务：Service Mesh(Service Fabric、Istio、Conduit 等)。第一代微服务和语言绑定紧密；第二代微服务适合混合开发，正当年；第三代微服务目前还在快速发展中，更新迭代比较快。

.Net Core 微服务选型

为什么是.Net Core？虽然.Net Framework 也可以实现微服务，但是.Net Core 是为云而生，用来实现微服务更方便，而且.Net Core 可以跨平台。

.Net Framework 不会再有.Net 5.x，下一代就是.Net Core。

不是说第二代适合混合开发吗？为什么还要特别说.Net Core。因为不同的二三代微服务平台虽然可以混合开发，但是还是有“嫡庶之分”。

Spring Cloud 是使用 Java 开发的，使用 Java 在 Spring Cloud 下开发最爽，使用 steelto（https://steeltoe.io/）这个开发包也是可以使用.Net Core 开发 Spring Cloud 下的微服务，但是开发体验没有 Java 那么效率高，而且支持的软件版本更新没有 Java 那么快，使用 Zuul 等的时候还可能需要写一些 Java 代码。

因此，如果整个项目的技术栈是 Spring Cloud 的，那么.Net Core 开发者可以借助于 steelto “寄居”。如果自己有技术栈的选择权，那么可以自己搭建更亲近.Net Core 的微服务框架。

Service Fabric 是微软开源的微软内部使用的第三代微服务框架，可以使用.Net Core 开发，也可以使用 Java 开发。但是目前开源的版本还不太适合普通厂商使用（难度；跨平台；通用性；支持私有云，但是难度大），慎用。愿意花钱在 Azure 云上用 SF，可以用。但是如果想搭建 SF 私有云，慎用。

如果选择.Net Core 技术栈的第二代微服务框架，推荐使用腾讯（微信支付清算网关）在使用架构：Consul+Ocelot+.Net Core+Polly+……。注意腾讯只是一部分系统是使用.Net Core 开发的。大厂实践验证的东西，大家更放心。

腾讯的.Net 大队长张善友把腾讯内部的架构实战整理出一个开源项目 NanoFabric（https://github.com/geffzhang/NanoFabric）。NanoFabric 不是一个独立的技术，它只是帮我们把这些组件帮我们配置好了，脚手架。NanoFabric 文档不全，而且只是“胶水项目”，仅供我们参考，所以我这次课就是参考 NanoFabric 帮大家自己搭建其中最核心的部分。

在 Spring Cloud 中：Eureka Server 做服务治理和服务发现、Hystrix 做熔断降级、Zuul 做网关；

在 NanoFabric 中：Consul 做服务治理和服务发现、Polly 做熔断降级、Ocelot 做网关；

在微服务中，服务之间的通讯有两种主要形式：

1) Restful，也就是传输 Json 格式数据。.Net 中就是对应 WebAPI 技术，不精通 WebAPI 也没关系，和 ASP.Net MVC 差不多，可以使用 PostMan 方便的调试 Restful 接口。

2) 二进制 RPC：二进制传输协议，比 Restful 用的 Http 通讯效率更高，但是耦合性更强。技术有 Thrift、gRPC 等。

后面讲解这两者的区别，下面的课程中先讲 Restful 方式通讯。

.Net Core 基础

一、 .Net 跨平台历史

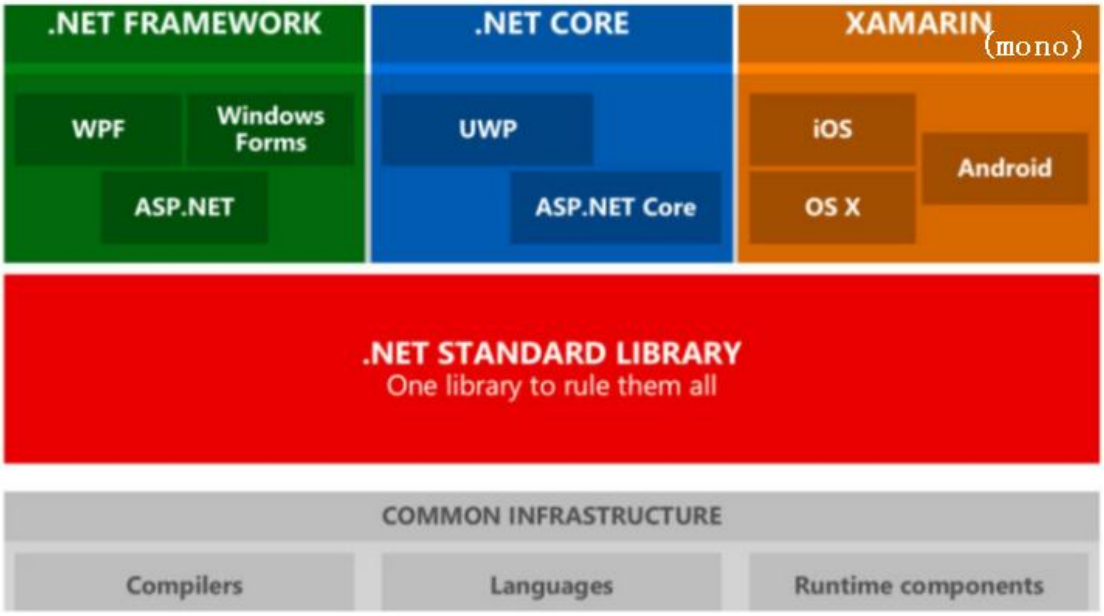
.net 设计之初就是考虑像 java 一样跨平台的，.net framework 是在 windows 下运行的，大部分类是可以兼容移植到 linux 下的，但是没人做这个工作。2001 年米格尔为 Gnome 寻找桌面开发技术，在研究了微软的.net framework 之后发起了 mono 项目。后来 novell 公司收购了 mono，进一步完善了 mono，把大部分.net framework 功能移植到 linux 下。mono 也成为 xamarin（使用.net 开发 Android、IOS app 的技术）和 unity3d（使用.Net 开发 android、ios 游戏的技术）的基础。

.Net core 是微软开发的另外一个可以跨 Linux、windows、mac 等平台的.Net。

2016 年初，微软收购 mono 的公司 xamarin。为什么微软已经收购了 mono，还要搞出来一个.net core？因为 mono 完全兼容.net framework，架构太陈旧，不有利于现在云计算、集群等现在新的架构理念。因此微软推翻重写了.net core。

二、 .net framework、.net core、mono 的关系

.net framework、.net core、xamarin 有通用的类，也有特有的。为了保证代码通用，微软定义了公共的.Net Standard Library(.net 标准库，像 FileStream、List 等这些)，按照.Net Standard Library 编写的代码可以在几个平台下通用。

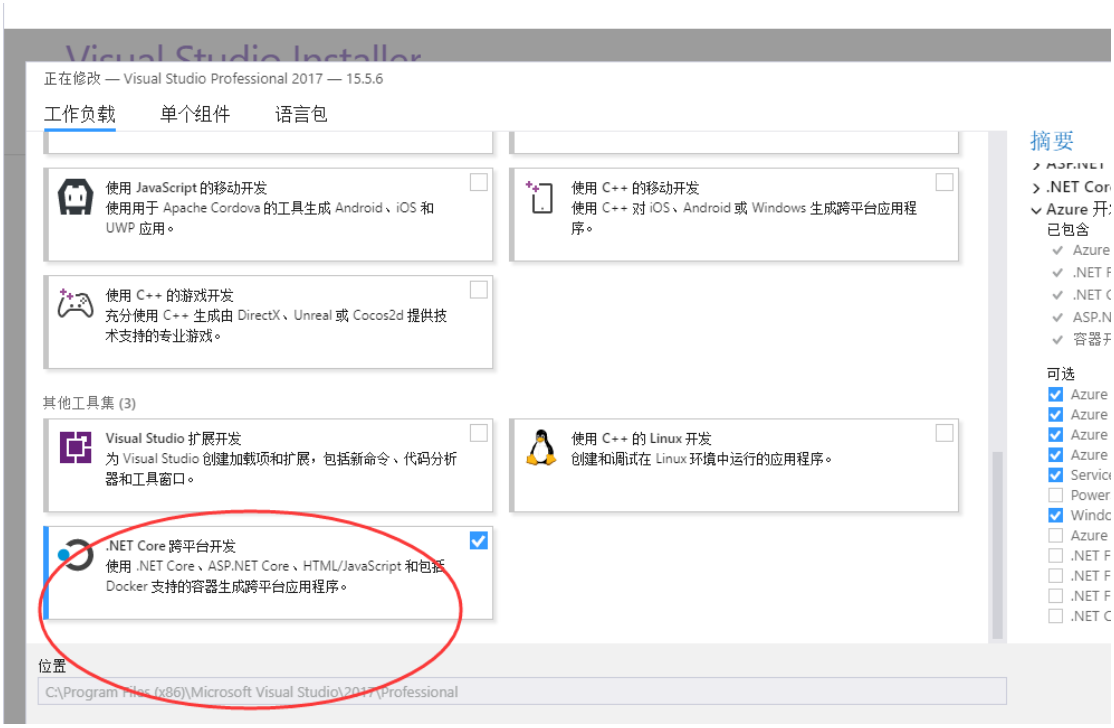


大部分.net framework 中的类在.net core 中还有，方法也还有，只是 namespace 可能变了，有些方法也有不一样，部分 api 缺失（注册表等 windows 平台特有的 api），后面会讲区别。

之前那些在.net framework 中调用的 dll，不一定都能用在.net core(linux)中。

如何判断 dll 是否支持.net core，举例：https://www.nuget.org/packages/NPOI/ 、https://www.nuget.org/packages/Npoi.Core/ 、https://www.nuget.org/packages/ImageProcessor/

三、 .net core 在 Windows 下的安装配置



1、如何看、修改自己的项目的.Net Core 版本。

2、控制台、Web 程序部署到 Linux 等服务器的统一方式：服务器上先要安装.net core，然后在开发环境发布，然后把发布包上传到服务器，然后到目录下执行“dotnet 主程序 dll”即可。.Net core 时代“命令行”才是正道，不要还是想着“傻瓜化图形化界面”

3、https 是保证在不安全网络环境下通讯安全的必要的步骤。从.Net core 2.1 开始默认启用 https，但是对调试程序不方便，新建项目的时候去掉【配置 Https】的钩。(*)生产环境加上 app.UseHsts();app.UseHttpsRedirection();即可。

四、 ASP.Net Core 的基本配置

在 VS 中调试的时候有很多修改 Web 应用运行端口的方法。但是在开发、调试微服务应用的时候可能需要同时在不同端口上开启多个服务器的实例，因此下面主要看看如何通过命令行指定 Web 应用的端口（默认 5000）

可以通过设置临时环境变量 ASPNETCORE_URLS 来改变默认的端口、域名，也就是执行 dotnet xxx.dll 之前执行 set ASPNETCORE_URLS=http://127.0.0.1:5001 来设置环境变量。也可以修改配置文件、server.urls 等其他方式，但是这种方式启动多个实例更方便。

如果需要在程序中读取端口、域名（后续服务治理会用到），用 ASPNETCORE_URLS 环境变量就不太方便，可以自定义配置文件，自己读取设置。

修改 Program.cs（注意 asp.net core 2.0 产生的代码不一样）

```
public static IWebHostBuilder CreateWebHostBuilder (string[] args)
{
    return WebHost.CreateDefaultBuilder(args)
```

```
                .UseStartup<Startup>()
                .UseUrls("http://127.0.0.1:5002");
    }
```

当然还要修改从命令行或者配置文件中读取。

```
public static IWebHost BuildWebHost(string[] args)
{
    var config = new ConfigurationBuilder()
        .AddCommandLine(args)
        .Build();
    String ip = config["ip"];
    String port = config["port"];
    Console.WriteLine($"ip={ip},port={port}");
    return WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseUrls($"http://{ip}:{port}")
        .Build();
}
```

然后启动的时候：dotnet WebApplication5.dll --ip 127.0.0.1 --port 8889 即可

.Net Core 因为跨平台，所以可以不依赖于 IIS 运行了。后续课程中统一用.Net Core 内置的 kestrel 服务器运行网站，当然真正面对终端用户访问的时候一般通过 Nginx 等做反向代理。

Ctrl+C 终止程序。

五、 讲 WebAPI 基础

编写几个有意义的接口，多搞几个不同的服务，包含 get、post，方便后面测试、演示。

调试 WebAPI 项目的时候把项目属性中的【启动浏览器】勾掉，这样就不会启动浏览器了，讲解使用 PostMan 调试 Http 接口。

WebAPI 就是 Restful 风格，请求响应都最好是 json 格式，虽然请求也可以是表单格式，但是最好都用 json 格式请求（contenttype=application/json）的方法体：{phoneNum:"110",msg:"aaaaaaaaaaaa"}，因此这里只讲 json 格式请求的方法。

和.Net Framework 中的 WebAPI 不一样，如果[HttpPost]、[HttpGet]等标记不加参数，则表示匹配“没有 Action”，比如 http://localhost:5000/api/SMS/。如果指定[HttpPost("Send_MI")]，则匹配 Action 的名字，比如 http://localhost:5000/api/SMS/ Send_MI。如果方法名字和 Action 名字一样，建议用 nameof。

参数：

1） 正确：public void Send_MI(dynamic model)

2） 正确：public void Send_HW(SendSMSRequest model)

3） 错误：public void Send_LX(string phoneNum,string msg)

信息服务项目 MsgService

有两个文件

EmailController.cs

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace MsgService.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class EmailController : ControllerBase
    {
        [HttpPost(nameof(Send_QQ))]
        public void Send_QQ(SendEmailRequest model)
        {
            Console.WriteLine($"通过QQ邮件接口向{model.Email}发送邮件，标题{model.Title}，内容：{model.Body}");
        }

        [HttpPost(nameof(Send_163))]
        public void Send_163(SendEmailRequest model)
        {
            Console.WriteLine($"通过网易邮件接口向{model.Email}发送邮件，标题{model.Title}，内容：{model.Body}");
        }

        [HttpPost(nameof(Send_Sohu))]
        public void Send_Sohu(SendEmailRequest model)
        {
            Console.WriteLine($"通过Sohu邮件接口向{model.Email}发送邮件，标题{model.Title}，内容：{model.Body}");
        }
    }

    public class SendEmailRequest
    {
        public string Email { get; set; }
        public string Title { get; set; }
        public string Body { get; set; }
    }
}
```

SMSController.cs

```
using System;
using Microsoft.AspNetCore.Mvc;
```

```
namespace MsgService.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class SMSController : ControllerBase
    {
        //发请求，报文体为{phoneNum:"110",msg:"aaaaaaaaaaaa"}，
        [HttpPost(nameof(Send_MI))]
        public void Send_MI(dynamic model)
        {
            Console.WriteLine($"通过小米短信接口向{model.phoneNum}发送短信{model.msg}");
        }

        [HttpPost(nameof(Send_LX))]
        public void Send_LX(SendSMSRequest model)
        {
            Console.WriteLine($"通过联想短信接口向{model.PhoneNum}发送短信{model.Msg}");
        }

        [HttpPost(nameof(Send_HW))]
        public void Send_HW(SendSMSRequest model)
        {
            Console.WriteLine($"通过华为短信接口向{model.PhoneNum}发送短信{model.Msg}");
        }
    }

    public class SendSMSRequest
    {
        public string PhoneNum { get; set; }
        public string Msg { get; set; }
    }
}
```

产品信息服务项目 ProductService

ProductController.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Net;
using Microsoft.AspNetCore.Mvc;

namespace ProductService.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
```

```
public class ProductController : ControllerBase
{
    //这显然是为了demo，这样放到内存中不能“集群”
    private static List<Product> products = new List<Product>();
    static ProductController()
    {
        products.Add(new Product { Id=1,Name="T430笔记本",Price=8888,Description="CPU i7标压版，1T硬盘"});
        products.Add(new Product { Id = 2, Name = "华为Mate10", Price = 3888, Description = "大猩猩屏幕，多点触摸" });
        products.Add(new Product { Id = 3, Name = "天梭手表", Price = 9888, Description = "瑞士经典款，可好了" });
    }

    [HttpGet]
    public IEnumerable<Product> Get()
    {
        return products;
    }

    [HttpGet("{id}")]
    public Product Get(int id)
    {
        var product = products.SingleOrDefault(p=>p.Id==id);
        if(product==null)
        {
            Response.StatusCode = 404;
        }
        return product;
    }

    [HttpPost]
    public void Add(Product model)
    {
        if(products.Any(p=>p.Id==model.Id))
        {
            Response.StatusCode = (int)HttpStatusCode.Conflict;//通过状态码而非响应体报错，是restful风格
            return;
        }
        products.Add(model);
    }
```

```
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
            var product = products.SingleOrDefault(p => p.Id == id);
            if(product != null)
            {
                products.Remove(product);
            }
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ProductService.Controllers
{
    public class Product
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public double Price { get; set; }
        public string Description { get; set; }
    }
}
```

然后再把两个项目配置通过命令行读取ip、port自定义监听的ip、端口。

六、 .Net Core 深入学习参考

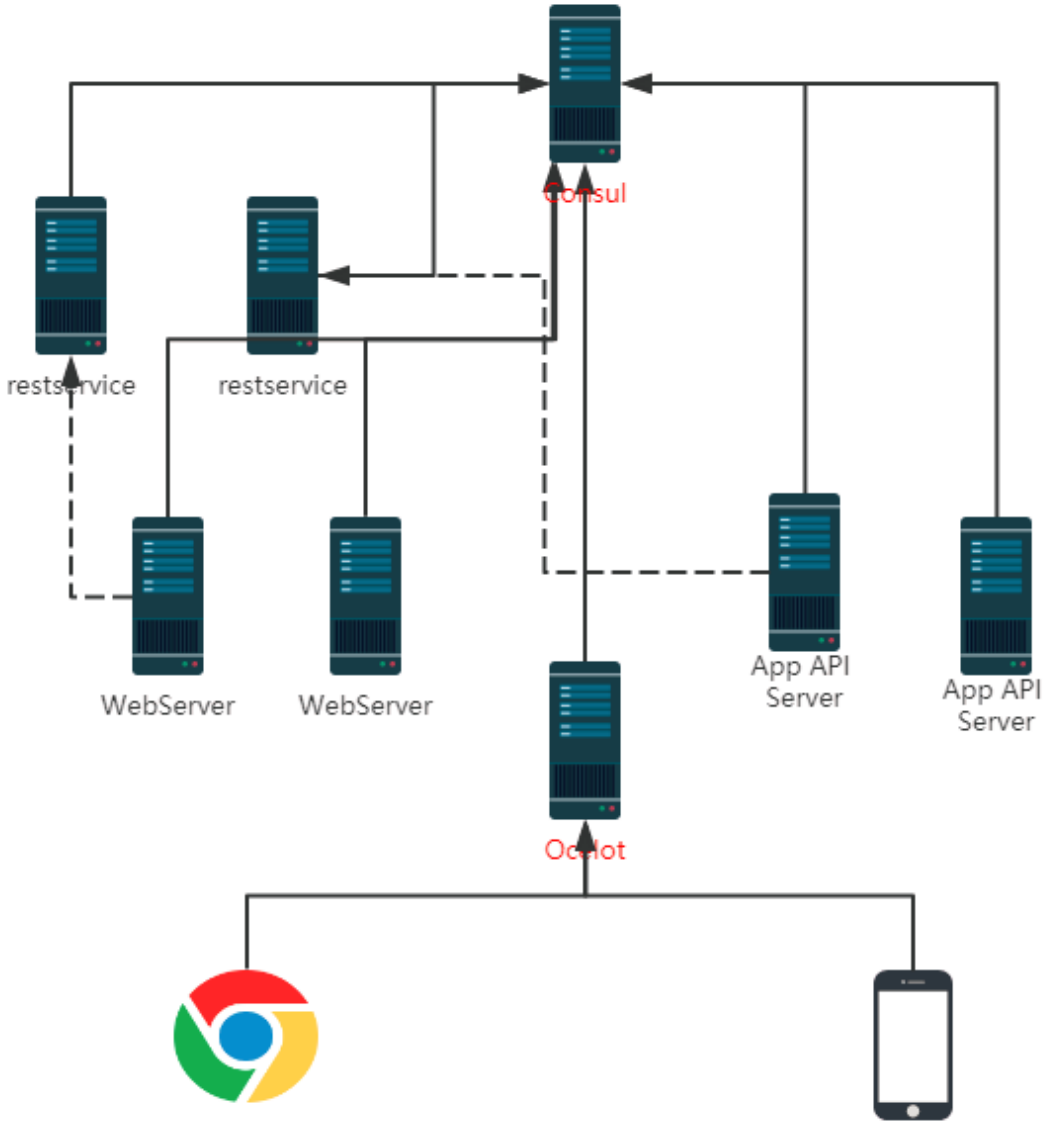
文档 https://github.com/dotnet/docs.zh-cn/tree/master/docs/core
如鹏网.Net 提高班 http://www.rupeng.com

Consul 服务治理发现

Consul（['kɒnsl]，康搜）是注册中心，服务提供者、服务消费者等都要注册到 Consul 中，这样就可以实现服务提供者、服务消费者的隔离。

除了 Consul 之外，还有 Eureka、Zookeeper 等类似软件。

用 DNS 举例来理解 Consul。consul 是存储服务名称与 IP 和端口对应关系的服务器。



一、 consul 服务器安装

consul 下载地址
运行 consul.exe agent -dev
这是开发环境测试，生产环境要建集群，要至少一台 Server，多台 Agent。
开发环境中 consul 重启后数据就会丢失。
consul 的监控页面 http://127.0.0.1:8500/

consult 主要做三件事：提供服务到 ip 地址的注册；提供服务到 ip 地址列表的查询；对提供服务方的健康检查（HealthCheck）；

二、 .Net Core 连接 consul

Install-Package Consul

三、 Rest 服务的准备

先使用使用默认生成的 ValuesController 做测试
再提供一个 HealthController.cs

```
[Route("api/[controller]")]
public class HealthController : Controller
{
    [HttpGet]
    public IActionResult Get()
    {
        return Ok("ok");
    }
}
```

服务器从命令行中读取 ip 和端口。

四、 服务注册 Consul 及注销

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, IApplicationLifetime applicationLifetime)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseMvc();

    string ip = Configuration["ip"];
    int port = Convert.ToInt32(Configuration["port"]);

    string serviceName = "MsgService";
    string serviceId = serviceName + Guid.NewGuid();
    using (var client = new ConsulClient(ConsulConfig))
    {
        //注册服务到 Consul
        client.Agent.ServiceRegister(new AgentServiceRegistration()
```

```
        {
            ID = serviceId,//服务编号，不能重复，用 Guid 最简单
            Name = serviceName,//服务的名字
            Address = ip,//服务提供者的能被消费者访问的 ip 地址(可以被其他应用访问的地址，本地测试可以用 127.0.0.1，机房环境中一定要写自己的内网 ip 地址)
            Port = port,// 服务提供者的能被消费者访问的端口
            Check = new AgentServiceCheck
            {
                DeregisterCriticalServiceAfter = TimeSpan.FromSeconds(5),//服务停止多久后反注册(注销)
                Interval = TimeSpan.FromSeconds(10),//健康检查时间间隔，或者称为心跳间隔
                HTTP = $"http://{ip}:{port}/api/health",//健康检查地址
                Timeout = TimeSpan.FromSeconds(5)
            }
        }).Wait();//Consult 客户端的所有方法几乎都是异步方法，但是都没按照规范加上 Async 后缀，所以容易误导。记得调用后要 Wait()或者 await
    }

    //程序正常退出的时候从 Consul 注销服务
    //要通过方法参数注入 IApplicationLifetime
    applicationLifetime.ApplicationStopped.Register(()=> {
        using (var client = new ConsulClient(ConsulConfig))
        {
            client.Agent.ServiceDeregister(serviceId).Wait();
        }
    });
}

private void ConsulConfig(ConsulClientConfiguration c)
{
    c.Address = new Uri("http://127.0.0.1:8500");
    c.Datacenter = "dc1";
}
```

也支持 tcp 探测，很显然也可以把普通 TCP 服务注册到 Consul，因为 Consul 中注册的只是服务名字、ip 地址、端口号，具体服务怎么实现、怎么调用 Consul 不管。

注意不同实例一定要用不同的 Id，即使是相同服务的不同实例也要用不同的 Id，上面的代码用 Guid 做 Id，确保不重复。相同的服务用相同的 Name。Address、Port 是供服务消费者访问的服务器地址（或者 IP 地址）及端口号。Check 则是做服务健康检查的（解释一下）。

在注册服务的时候还可以通过 AgentServiceRegistration 的 Tags 属性设置额外的标签。

通过命令行启动两个实例：

```
dotnet WebApplication4.dll --ip 127.0.0.1 --port 5001
dotnet WebApplication4.dll --ip 127.0.0.1 --port 5002
```

打开 Consul 的 Web 页面服务已经注册进来了，注意刚开始启动的时候，有短暂的 Failing 是正常的。服务正常结束（Ctrl+C）会触发 ApplicationStopped，正常注销。即使非正常结束也没关系，Consul 健康检查过一会发现服务器死掉后也会主动注销。

如果服务器刚刚崩溃，但是还买来得及注销，消费的使用者可能就会拿到已经崩溃的实例，这个问题通过后面讲的重试等策略解决。

服务只会注册 ip、端口，consul 只会保存服务名、ip、端口这些信息，至于服务提供什么接口、方法、参数，consul 不管，需要消费者知道服务的这些细节。

多个服务应用就注册多个就可以。Consul 中可能注册多个服务，一个服务有多个服务器实例。

上面讲的就是服务治理：服务的注册、注销、健康检查。

下面讲：服务发现

五、 编写服务消费者

这里用控制台测试，真实项目中服务消费者同时也可能是另外一个 Web 应用（比如 Web 服务器调用短信服务器发短信）。

下面就是打印出所有 Consul 登记在册的服务实例

```
using (var consulClient = new ConsulClient(c => c.Address = new Uri("http://127.0.0.1:8500")))
{
    var services = consulClient.Agent.Services().Result.Response;
    foreach(var service in services.Values)
    {

    Console.WriteLine($"id={service.ID},name={service.Service},ip={service.Address},port={servi
ce.Port}");
    }
}
```

下面的代码使用当前 TickCount 进行取模的方式达到随机获取一台服务器实例的效果，这叫做“客户端负载均衡”：

```
using (var consulClient = new ConsulClient(c => c.Address = new Uri("http://127.0.0.1:8500")))
{
    var services = consulClient.Agent.Services().Result.Response.Values
        .Where(s => s.Service.Equals("MsgService", StringComparison.OrdinalIgnoreCase));
    if(!services.Any())
    {
        Console.WriteLine("找不到服务的实例");
    }
    else
    {
```

```
            var service = services.ElementAt(Environment.TickCount%services.Count());
            Console.WriteLine($"{service.Address}:{service.Port}");
        }
}
```

当然在一个毫秒之类会所有请求都压给一台服务器，基本就够用了。也可以自己写随机、轮询等客户端负载均衡算法，也可以自己实现按不同权重分配（注册时候 Tags 带上配置、权重等信息）等算法。

首先编写一个 RestTemplateCore 类库项目（模仿 Spring Cloud 中的 RestTemplate）
GitHub 地址：https://github.com/yangzhongke/RuPeng.RestTemplateCore
Nuget 地址：https://www.nuget.org/packages/RestTemplateCore

nuget 安装：Consul、Newtonsoft.Json

```
using System.Net;
using System.Net.Http.Headers;

namespace RestTemplateCore
{
    /// <summary>
    /// Rest响应结果
    /// </summary>
    public class RestResponse
    {
        /// <summary>
        /// 响应状态码
        /// </summary>
        public HttpStatusCode StatusCode { get; set; }

        /// <summary>
        /// 响应的报文头
        /// </summary>
        public HttpResponseHeaders Headers { get; set; }
    }
}

namespace RestTemplateCore
{
    /// <summary>
    /// 带响应报文的Rest响应结果，而且json报文会被自动反序列化
    /// </summary>
    /// <typeparam name="T"></typeparam>
    public class RestResponseWithBody<T>: RestResponse
    {
        /// <summary>
```

```
        /// 响应报文体json反序列化的内容
        /// </summary>
        public T Body { get; set; }
    }
}

using Consul;
using Newtonsoft.Json;
using System;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;


namespace RestTemplateCore
{
    /// <summary>
    /// 会自动到 Consul 中解析服务的 Rest 客户端，能把"http://ProductService/api/Product/"这样的虚拟地址
    /// 按照客户端负载均衡算法解析为 http://192.168.1.10:8080/api/Product/这样的真实地址
    /// </summary>
    public class RestTemplate
    {
        public String ConsulServerUrl { get; set; }= "http://127.0.0.1:8500";
        private HttpClient httpClient;

        public RestTemplate(HttpClient httpClient)
        {
            this.httpClient = httpClient;
        }

        /// <summary>
        /// 获取服务的第一个实现地址
        /// </summary>
        /// <param name="consulClient"></param>
        /// <param name="serviceName"></param>
        /// <returns></returns>
        private async Task<String> ResolveRootUrlAsync(String serviceName)
        {
            using (var consulClient = new ConsulClient(c => c.Address = new Uri(ConsulServerUrl)))
            {
```

```
                var services = (await consulClient.Agent.Services()).Response.Values
                    .Where(s => s.Service.Equals(serviceName, StringComparison.OrdinalIgnoreCase));
                if (!services.Any())
                {
                    throw new ArgumentException($"找不到服务{serviceName }的任何实例");
                }
                else
                {
                    //根据当前时钟毫秒数对可用服务个数取模，取出一台机器使用
                    var service = services.ElementAt(Environment.TickCount % services.Count());
                    return $"{service.Address}:{service.Port}";
                }
            }
        }

        //把 http://apiservice1/api/values 转换为 http://192.168.1.1:5000/api/values
        private async Task<String> ResolveUrlAsync(String url)
        {
            Uri uri = new Uri(url);
            String serviceName = uri.Host;//apiservice1
            String realRootUrl = await ResolveRootUrlAsync(serviceName);// 查询出来 apiservice1 对应的服务器地址 192.168.1.1:5000

//uri.Scheme=http,realRootUrl =192.168.1.1:5000,PathAndQuery=/api/values
            return uri.Scheme + "://" + realRootUrl + uri.PathAndQuery;
        }

        /// <summary>
        /// 发出 Get 请求
        /// </summary>
        /// <typeparam name="T">响应报文反序列类型</typeparam>
        /// <param name="url">请求路径</param>
        /// <param name="requestHeaders">请求额外的报文头信息</param>
        /// <returns></returns>
        public async Task<RestResponseWithBody<T>> GetForEntityAsync<T>(String url, HttpRequestHeaders requestHeaders = null)
        {
            using (HttpRequestMessage requestMsg = new HttpRequestMessage())
            {
                if (requestHeaders != null)
                {
```

```
                    foreach (var header in requestHeaders)
                    {
                        requestMsg.Headers.Add(header.Key, header.Value);
                    }
                }
                requestMsg.Method = System.Net.Http.HttpMethod.Get;
                //http://apiservice1/api/values 转换为 http://192.168.1.1:5000/api/values
                requestMsg.RequestUri = new Uri(await ResolveUrlAsync(url));
                RestResponseWithBody<T> respEntity = await SendForEntityAsync<T>(requestMsg);
                return respEntity;
            }
        }

        /// <summary>
        /// 发出 Post 请求
        /// </summary>
        /// <typeparam name="T">响应报文反序列类型</typeparam>
        /// <param name="url">请求路径</param>
        /// <param name="body">请求数据，将会被 json 序列化后放到请求报文体中</param>
        /// <param name="requestHeaders">请求额外的报文头信息</param>
        /// <returns></returns>
        public async Task<RestResponseWithBody<T>> PostForEntityAsync<T>(String url,object body=null, HttpRequestHeaders requestHeaders = null)
        {
            using (HttpRequestMessage requestMsg = new HttpRequestMessage())
            {
                if (requestHeaders != null)
                {
                    foreach (var header in requestHeaders)
                    {
                        requestMsg.Headers.Add(header.Key, header.Value);
                    }
                }
                requestMsg.Method = System.Net.Http.HttpMethod.Post;
                //http://apiservice1/api/values 转换为 http://192.168.1.1:5000/api/values
                requestMsg.RequestUri = new Uri(await ResolveUrlAsync(url));
                requestMsg.Content = new StringContent(JsonConvert.SerializeObject(body));
                requestMsg.Content.Headers.ContentType = new MediaTypeHeaderValue("application/json");

                RestResponseWithBody<T> respEntity = await
```

```
SendForEntityAsync<T>(requestMsg);
                return respEntity;
            }
        }

        /// <summary>
        /// 发出 Post 请求
        /// </summary>
        /// <param name="url">请求路径</param>
        /// <param name="requestHeaders">请求额外的报文头信息</param>
        /// <returns></returns>
        public async Task<RestResponse> PostAsync(String url, object body = null, HttpRequestHeaders requestHeaders = null)
        {
            using (HttpRequestMessage requestMsg = new HttpRequestMessage())
            {
                if (requestHeaders != null)
                {
                    foreach (var header in requestHeaders)
                    {
                        requestMsg.Headers.Add(header.Key, header.Value);
                    }
                }
                requestMsg.Method = System.Net.Http.HttpMethod.Post;
                //http://apiservice1/api/values 转换为 http://192.168.1.1:5000/api/values
                requestMsg.RequestUri = new Uri(await ResolveUrlAsync(url));
                requestMsg.Content = new StringContent(JsonConvert.SerializeObject(body));
                requestMsg.Content.Headers.ContentType = new MediaTypeHeaderValue("application/json");

                var resp = await SendAsync(requestMsg);
                return resp;
            }
        }

        /// <summary>
        /// 发出 Put 请求
        /// </summary>
        /// <typeparam name="T">响应报文反序列类型</typeparam>
        /// <param name="url">请求路径</param>
        /// <param name="body">请求数据，将会被 json 序列化后放到请求报文体中</param>
        /// <param name="requestHeaders">请求额外的报文头信息</param>
```

```
        /// <returns></returns>
        public async Task<RestResponseWithBody<T>> PutForEntityAsync<T>(String url, object body = null, HttpRequestHeaders requestHeaders = null)
        {
            using (HttpRequestMessage requestMsg = new HttpRequestMessage())
            {
                if (requestHeaders != null)
                {
                    foreach (var header in requestHeaders)
                    {
                        requestMsg.Headers.Add(header.Key, header.Value);
                    }
                }
                requestMsg.Method = System.Net.Http.HttpMethod.Put;
                //http://apiservice1/api/values 转换为 http://192.168.1.1:5000/api/values
                requestMsg.RequestUri = new Uri(await ResolveUrlAsync(url));
                requestMsg.Content = new StringContent(JsonConvert.SerializeObject(body));
                requestMsg.Content.Headers.ContentType = new MediaTypeHeaderValue("application/json");

                RestResponseWithBody<T> respEntity = await SendForEntityAsync<T>(requestMsg);
                return respEntity;
            }
        }

        /// <summary>
        /// 发出 Put 请求
        /// </summary>
        /// <param name="url">请求路径</param>
        /// <param name="body">请求数据，将会被 json 序列化后放到请求报文体中</param>
        /// <param name="requestHeaders">请求额外的报文头信息</param>
        /// <returns></returns>
        public async Task<RestResponse> PutAsync(String url, object body = null, HttpRequestHeaders requestHeaders = null)
        {
            using (HttpRequestMessage requestMsg = new HttpRequestMessage())
            {
                if (requestHeaders != null)
                {
                    foreach (var header in requestHeaders)
                    {
```

```
                    requestMsg.Headers.Add(header.Key, header.Value);
                }
            }
            requestMsg.Method = System.Net.Http.HttpMethod.Put;
            //http://apiservice1/api/values 转换为 http://192.168.1.1:5000/api/values
            requestMsg.RequestUri = new Uri(await ResolveUrlAsync(url));
            requestMsg.Content = new StringContent(JsonConvert.SerializeObject(body));
            requestMsg.Content.Headers.ContentType = new MediaTypeHeaderValue("application/json");

            var resp = await SendAsync(requestMsg);
            return resp;
        }
    }

    /// <summary>
    /// 发出 Delete 请求
    /// </summary>
    /// <typeparam name="T">响应报文反序列类型</typeparam>
    /// <param name="url">请求路径</param>
    /// <param name="requestHeaders">请求额外的报文头信息</param>
    /// <returns></returns>
    public async Task<RestResponseWithBody<T>> DeleteForEntityAsync<T>(String url, HttpRequestHeaders requestHeaders = null)
    {
        using (HttpRequestMessage requestMsg = new HttpRequestMessage())
        {
            if (requestHeaders != null)
            {
                foreach (var header in requestHeaders)
                {
                    requestMsg.Headers.Add(header.Key, header.Value);
                }
            }
            requestMsg.Method = System.Net.Http.HttpMethod.Delete;
            //http://apiservice1/api/values 转换为 http://192.168.1.1:5000/api/values
            requestMsg.RequestUri = new Uri(await ResolveUrlAsync(url));
            RestResponseWithBody<T> respEntity = await SendForEntityAsync<T>(requestMsg);
            return respEntity;
        }
    }
```

```
        /// <summary>
        /// 发出 Delete 请求
        /// </summary>
        /// <param name="url">请求路径</param>
        /// <param name="requestHeaders">请求额外的报文头信息</param>
        /// <returns></returns>
        public async Task<RestResponse> DeleteAsync(String url, HttpRequestHeaders requestHeaders = null)
        {
            using (HttpRequestMessage requestMsg = new HttpRequestMessage())
            {
                if (requestHeaders != null)
                {
                    foreach (var header in requestHeaders)
                    {
                        requestMsg.Headers.Add(header.Key, header.Value);
                    }
                }
                requestMsg.Method = System.Net.Http.HttpMethod.Delete;
                //http://apiservice1/api/values 转换为 http://192.168.1.1:5000/api/values
                requestMsg.RequestUri = new Uri(await ResolveUrlAsync(url));
                var resp = await SendAsync(requestMsg);
                return resp;
            }
        }

        /// <summary>
        /// 发出 Http 请求
        /// </summary>
        /// <typeparam name="T">响应报文反序列类型</typeparam>
        /// <param name="requestMsg">请求数据</param>
        /// <returns></returns>
        public async Task<RestResponseWithBody<T>> SendForEntityAsync<T>(HttpRequestMessage requestMsg)
        {
            var result = await httpClient.SendAsync(requestMsg);
            RestResponseWithBody<T> respEntity = new RestResponseWithBody<T>();
            respEntity.StatusCode = result.StatusCode;
            respEntity.Headers = respEntity.Headers;
            String bodyStr = await result.Content.ReadAsStringAsync();
            if(!string.IsNullOrWhiteSpace(bodyStr))
            {
                respEntity.Body = JsonConvert.DeserializeObject<T>(bodyStr);
            }
```

```
            return respEntity;
        }

        /// <summary>
        ///  发出 Http 请求
        /// </summary>
        /// <param name="requestMsg">请求数据</param>
        /// <returns></returns>
        public async Task<RestResponse> SendAsync(HttpRequestMessage requestMsg)
        {
            var result = await httpClient.SendAsync(requestMsg);
            RestResponse response = new RestResponse();
            response.StatusCode = result.StatusCode;
            response.Headers = result.Headers;
            return response;
        }
    }

}
```

编写控制台项目做测试：

```
using RestTemplateCore;
using System;
using System.Net.Http;

namespace consultest1
{
    class Program
    {
        static void Main(string[] args)
        {
            using (HttpClient httpClient = new HttpClient())
            {
                RestTemplate rest = new RestTemplate(httpClient);

                Console.WriteLine("---查询数据---------");
                var ret1 =
rest.GetForEntityAsync<Product[]>("http://ProductService/api/Product/").Result;
                Console.WriteLine(ret1.StatusCode);
                if(ret1.StatusCode== System.Net.HttpStatusCode.OK)
                {
                    foreach (var p in ret1.Body)
                    {
```

```
                    Console.WriteLine($"id={p.Id},name={p.Name}");
                }
            }

            Console.WriteLine("---新增数据---------");
            Product newP = new Product();
            newP.Id = 888;
            newP.Name = "辛增";
            newP.Price = 88.8;
            var ret = rest.PostAsync("http://ProductService/api/Product/", newP).Result;
            Console.WriteLine(ret.StatusCode);
        }
        Console.ReadKey();
    }
    class Product
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public double Price { get; set; }
        public string Description { get; set; }
    }
}
}
```

参考资料：

1） 关于.Net Core 2.1中新增的管理HttpClient实例的HttpClientFactory：
http://www.cnblogs.com/dudu/p/9127115.html

解析RestTemplate代码。主要作用：

1） 根据url到Consul中根据服务的名字解析获取一个服务实例，把路径转换为实际连接的服务器；负载均衡，这里用的是简单的随机负载均衡，这样服务的消费者就不用自己指定要访问那个服务提供者了，解耦、负载均衡。

2) RestTemplate还负责把响应的json反序列化返回结果。

服务的注册者、消费者都是网站内部服务器之间的事情，对于终端用户是不涉及这些的。终端用户是不访问consul的。对终端用户来讲是对的Web服务器，Web服务器是服务的消费者。

一个服务实例注册一次，可以注册同服务的多个实例，也可以注册多个服务。

todo：报文头添加的有 bug。

熔断、降级等

一、 什么是熔断降级

熔断就是“保险丝”。当出现某些状况时，切断服务，从而防止应用程序不断地尝试执行可能会失败的操作给系统造成“雪崩”，或者大量的超时等待导致系统卡死。

降级的目的是当某个服务提供者发生故障的时候，向调用方返回一个错误响应或者替代响应。举例子：调用联通接口服务器发送短信失败之后，改用移动短信服务器发送，如果移动短信服务器也失败，则改用电信短信服务器，如果还失败，则返回“失败”响应；在从推荐商品服务器加载数据的时候，如果失败，则改用从缓存中加载，如果缓存中也加载失败，则返回一些本地替代数据。

二、 Polly 简介

.Net Core 中有一个被.Net 基金会认可的库 Polly，可以用来简化熔断降级的处理。主要功能：重试（Retry）；断路器（Circuit-breaker）；超时检测（Timeout）；缓存（Cache）；降级（FallBack）；

官网：https://github.com/App-vNext/Polly

介绍文章：https://www.cnblogs.com/CreateMyself/p/7589397.html

Install-Package Polly -Version 6.0.1

Polly 的策略由“故障”和“动作”两部分组成，“故障”包括异常、超时、返回值错误等情况，“动作”包括 FallBack（降级）、重试（Retry）、熔断（Circuit-breaker）等。

策略用来执行可能会有有故障的业务代码，当业务代码出现“故障”中情况的时候就执行“动作”。

由于实际业务代码中故障情况很难重现出来，所以 Polly 这一些都是用一些无意义的代码模拟出来。

Polly 也支持请求缓存“数据不变化则不重复自行代码”，但是和新版本兼容不好，而且功能局限性很大，因此这里不讲。

由于调试器存在，看不清楚 Polly 的执行过程，因此本节都用【开始执行（不调试）】

三、 Polly 简单使用

使用Policy的静态方法创建ISyncPolicy实现类对象，创建方法既有同步方法也有异步方法，根据自己的需要选择。下面先演示同步的，异步的用法类似。

举例：当发生ArgumentException异常的时候，执行Fallback代码。

```
Policy policy = Policy
.Handle<ArgumentException>() //故障
.Fallback(() =>//动作
{
    Console.WriteLine("执行出错");
});
policy.Execute(() => {//在策略中执行业务代码
```

```
    //这里是可能会产生问题的业务系统代码
    Console.WriteLine("开始任务");
    throw new ArgumentException("Hello world!");
    Console.WriteLine("完成任务");
});
Console.ReadKey();
```

如果没有被Handle处理的异常，则会导致未处理异常被抛出。

还可以用Fallback的其他重载获取异常信息：

```
Policy policy = Policy
.Handle<ArgumentException>() //故障
.Fallback(() =>//动作
{
    Console.WriteLine("执行出错");
},ex=> {
    Console.WriteLine(ex);
});
policy.Execute(() => {
    Console.WriteLine("开始任务");
    throw new ArgumentException("Hello world!");
    Console.WriteLine("完成任务");
});
```

如果Execute中的代码是带返回值的，那么只要使用带泛型的Policy<T>类即可：

```
Policy<string> policy = Policy<string>
.Handle<Exception>() //故障
.Fallback(() =>//动作
{
    Console.WriteLine("执行出错");
    return "降级的值";
});
string value = policy.Execute(() => {
    Console.WriteLine("开始任务");
    throw new Exception("Hello world!");
    Console.WriteLine("完成任务");
    return "正常的值";
});
Console.WriteLine("返回值："+value);
```

FallBack的重载方法也非常多，有的异常可以直接提供降级后的值。

(*)异常中还可以通过lambda表达式对异常判断“满足***条件的异常我才处理”，简单看看试试重载即可。还可以多个Or处理各种不同的异常。

(*)还可以用HandleResult等判断返回值进行故障判断等，我感觉没太大必要。

四、 重试处理

```
Policy policy = Policy
.Handle<Exception>()
.RetryForever();
policy.Execute(() => {
    Console.WriteLine("开始任务");
    if (DateTime.Now.Second % 10 != 0)
    {
        throw new Exception("出错");
    }
    Console.WriteLine("完成任务");
});
```

RetryForever()是一直重试直到成功

Retry()是重试最多一次；

Retry(n) 是重试最多n次；

WaitAndRetry()可以实现“如果出错等待100ms再试还不行再等150ms秒。。。。”，重载方法很多，不再一一介绍。还有WaitAndRetryForever。

五、 短路保护 Circuit Breaker

出现N次连续错误，则把“熔断器”（保险丝）熔断，等待一段时间，等待这段时间内如果再Execute则直接抛出BrokenCircuitException异常，根本不会再去尝试调用业务代码。等待时间过去之后，再执行Execute的时候如果又错了（一次就够了），那么继续熔断一段时间，否则就恢复正常。

这样就避免一个服务已经不可用了，还是使劲的请求给系统造成更大压力。

```
Policy policy = Policy
            .Handle<Exception>()
            .CircuitBreaker(6,TimeSpan.FromSeconds(5));//连续出错6次之后熔断5秒(不会再去尝试执行业务代码）。
```

```
while(true)
{
    Console.WriteLine("开始Execute");
    try
    {
        policy.Execute(() => {
            Console.WriteLine("开始任务");
            throw new Exception("出错");
            Console.WriteLine("完成任务");
        });
    }
    catch(Exception ex)
```

```
    {
        Console.WriteLine("execute出错"+ex);
    }
    Thread.Sleep(500);
}
```

其计数的范围是policy对象，所以如果想整个服务器全局对于一段代码做短路保护，则需要共用一个policy对象。

https://andrewlock.net/when-you-use-the-polly-circuit-breaker-make-sure-you-share-your-policy-instances-2/

六、 策略封装

可以把多个ISyncPolicy合并到一起执行：

policy3= policy1.Wrap(policy2);

执行policy3就会把policy1、policy2封装到一起执行

policy9=Policy.Wrap(policy1, policy2, policy3, policy4, policy5);把更多一起封装。

七、 超时处理

这些处理不能简单的链式调用，要用到Wrap。例如下面实现“出现异常则重试三次，如果还出错就FallBack”这样是不行的

```
Policy policy = Policy
.Handle<Exception>()
.Retry(3)
.Fallback(()=> { Console.WriteLine("执行出错"); });//这样不行
policy.Execute(() => {
    Console.WriteLine("开始任务");
    throw new ArgumentException("Hello world!");
    Console.WriteLine("完成任务");
});
```

注意Wrap是有包裹顺序的，内层的故障如果没有被处理则会抛出到外层。

下面代码实现了“出现异常则重试三次，如果还出错就FallBack”

```
Policy policyRetry = Policy.Handle<Exception>()
    .Retry(3);
Policy policyFallback = Policy.Handle<Exception>()
    .Fallback(()=> {
        Console.WriteLine("降级");
    });
//Wrap：包裹。policyRetry在里面，policyFallback裹在外面。
//如果里面出现了故障，则把故障抛出来给外面
Policy policy = policyFallback.Wrap(policyRetry);
policy.Execute(()=> {
    Console.WriteLine("开始任务");
    if (DateTime.Now.Second % 10 != 0)
    {
        throw new Exception("出错");
    }
    Console.WriteLine("完成任务");
});
```

Timeout是定义超时故障。

Policy policy = Policy.Timeout(3, TimeoutStrategy.Pessimistic);// 创建一个3秒钟（注意单位）的超时策略。

Timeout生成的Policy要和其他Policy一起Wrap使用。

超时策略一般不能直接用，而是和其他封装到一起用：

```
Policy policy = Policy
```

```
.Handle<Exception>()     //定义所处理的故障
.Fallback(() =>
{
    Console.WriteLine("执行出错");
});
policy = policy.Wrap(Policy.Timeout(2, TimeoutStrategy.Pessimistic));
policy.Execute(()=> {
    Console.WriteLine("开始任务");
    Thread.Sleep(5000);
    Console.WriteLine("完成任务");
});
```

上面的代码就是如果执行超过2秒钟，则直接Fallback。

这个的用途：请求网络接口，避免接口长期没有响应造成系统卡死。

八、 Polly 的异步用法

所有方法都用Async方法即可，Handle由于只是定义异常，所以不需要异常方法：

带返回值的例子：

```
Policy<byte[]> policy = Policy<byte[]>
.Handle<Exception>()
.FallbackAsync(async c => {
    Console.WriteLine("执行出错");
    return new byte[0];
},async r=> {
    Console.WriteLine(r.Exception);
});

policy = policy.WrapAsync(Policy.TimeoutAsync(20, TimeoutStrategy.Pessimistic,
async(context, timespan, task) =>
{
    Console.WriteLine("timeout");
}));
var bytes = await policy.ExecuteAsync(async () => {
    Console.WriteLine("开始任务");
    HttpClient httpClient = new HttpClient();
    var result = await
httpClient.GetByteArrayAsync("http://static.rupeng.com/upload/chatimage/20183/07EB793A4
C247A654B31B4D14EC64BCA.png");
    Console.WriteLine("完成任务");
    return result;
});
Console.WriteLine("bytes长度"+bytes.Length);
```

没返回值的例子

```
Policy policy = Policy
.Handle<Exception>()
.FallbackAsync(async c => {
    Console.WriteLine("执行出错");
},async ex=> {//对于没有返回值的，这个参数直接是异常
    Console.WriteLine(ex);
});

policy = policy.WrapAsync(Policy.TimeoutAsync(3, TimeoutStrategy.Pessimistic,
async(context, timespan, task) =>
{
    Console.WriteLine("timeout");
}));
await policy.ExecuteAsync(async () => {
    Console.WriteLine("开始任务");
    await Task.Delay(5000);//注意不能用Thread.Sleep(5000);
    Console.WriteLine("完成任务");

});
```

九、 AOP 框架基础

要求懂的知识：AOP、Filter、反射（Attribute）。

如果直接使用 Polly，那么就会造成业务代码中混杂大量的业务无关代码。我们使用 AOP（如果不了解 AOP，请自行参考网上资料）的方式封装一个简单的框架，模仿 Spring cloud 中的 Hystrix。

需要先引入一个支持.Net Core 的 AOP，目前我发现的最好的.Net Core 下的 AOP 框架是 AspectCore（国产，动态织入），其他要不就是不支持.Net Core，要不就是不支持对异步方法进行拦截。MVC Filter

GitHub：https://github.com/dotnetcore/AspectCore-Framework

```
Install-Package AspectCore.Core -Version 0.5.0
```


蒋金楠
除了您的这个包，还有其他支持异步的 aop 库吗
好像有一个叫什么 aspect.net
谢谢，明天我再调调，也麻烦您帮着看看。希望用国产
aspectcore
刚刚试了试，可以捕捉
等我这两周抽时间升级一下

这里只介绍和我们相关的用法：

1、编写拦截器 CustomInterceptorAttribute

一般继承自 AbstractInterceptorAttribute

```
public class CustomInterceptorAttribute : AbstractInterceptorAttribute
{
   //每个被拦截的方法中执行
    public async override Task Invoke(AspectContext context, AspectDelegate next)
    {
        try
        {
            Console.WriteLine("Before service call");
            await next(context);//执行被拦截的方法
        }
        catch (Exception)
        {
            Console.WriteLine("Service threw an exception!");
            throw;
        }
        finally
        {
```

```
            Console.WriteLine("After service call");
        }
    }
}
```

AspectContext 的属性的含义：

Implementation 实际动态创建的 Person 子类的对象。

ImplementationMethod 就是 Person 子类的 Say 方法

Parameters 方法的参数值。

Proxy==Implementation：当前场景下

ProxyMethod==ImplementationMethod：当前场景下

ReturnValue 返回值

ServiceMethod 是 Person 的 Say 方法

2、编写需要被代理拦截的类

在要被拦截的方法上标注 CustomInterceptorAttribute 。类需要是 public 类，方法需要是虚方法，支持异步方法，因为动态代理是动态生成被代理的类的动态子类实现的。

```
public class Person
{
    [CustomInterceptor]
    public virtual void Say(string msg)
    {
        Console.WriteLine("service calling..."+msg);
    }
}
```

3、通过 AspectCore 创建代理对象

```
ProxyGeneratorBuilder proxyGeneratorBuilder = new ProxyGeneratorBuilder();
using (IProxyGenerator proxyGenerator = proxyGeneratorBuilder.Build())
{
    Person p = proxyGenerator.CreateClassProxy<Person>();
    p.Say("rupeng.com");
}
Console.ReadKey();
```

注意 p 指向的对象是 AspectCore 生成的 Person 的动态子类的对象，直接 new Person 是无法被拦截的。

十、 创建简单的熔断降级框架

要达到的目标是：

```
public class Person
{
```

```
    [HystrixCommand("HelloFallBackAsync")]
    public virtual async Task<string> HelloAsync(string name)
    {
        Console.WriteLine("hello"+name);
        return "ok";
    }
    public async Task<string> HelloFallBackAsync(string name)
    {
        Console.WriteLine("执行失败"+name);
        return "fail";
    }
}
```

参与降级的方法参数要一样。

当 HelloAsync 执行出错的时候执行 HelloFallBackAsync 方法。

1、编写 HystrixCommandAttribute

```
using AspectCore.DynamicProxy;
using System;
using System.Threading.Tasks;

namespace hystrixtest1
{
    [AttributeUsage(AttributeTargets.Method)]
    public class HystrixCommandAttribute : AbstractInterceptorAttribute
    {
        public HystrixCommandAttribute(string fallBackMethod)
        {
            this.FallBackMethod = fallBackMethod;
        }

        public string FallBackMethod { get; set; }
        public override async Task Invoke(AspectContext context, AspectDelegate next)
        {
            try
            {
                await next(context);//执行被拦截的方法
            }
            catch (Exception ex)
            {
                //context.ServiceMethod 被拦截的方法。context.ServiceMethod.DeclaringType
被拦截方法所在的类
                //context.Implementation 实际执行的对象 p
                //context.Parameters 方法参数值
                //如果执行失败，则执行 FallBackMethod
                var                          fallBackMethod                          =
```

```
context.ServiceMethod.DeclaringType.GetMethod(this.FallBackMethod);
                Object  fallBackResult  =  fallBackMethod.Invoke(context.Implementation,
context.Parameters);
                context.ReturnValue = fallBackResult;
            }
        }
    }
}
```

2、编写类

```
public class Person//需要 public 类
{
    [HystrixCommand(nameof(HelloFallBackAsync))]
    public virtual async Task<string> HelloAsync(string name)//需要是虚方法
    {
        Console.WriteLine("hello"+name);
        String s = null;
       // s.ToString();
        return "ok";
    }
    public async Task<string> HelloFallBackAsync(string name)
    {
        Console.WriteLine("执行失败"+name);
        return "fail";
    }

    [HystrixCommand(nameof(AddFall))]
    public virtual int Add(int i,int j)
    {
        String s = null;
      //  s.ToArray();
        return i + j;
    }
    public int AddFall(int i, int j)
    {
        return 0;
    }
}
```

3、创建代理对象

```
ProxyGeneratorBuilder proxyGeneratorBuilder = new ProxyGeneratorBuilder();
using (IProxyGenerator proxyGenerator = proxyGeneratorBuilder.Build())
```

```
{
    Person p = proxyGenerator.CreateClassProxy<Person>();
    Console.WriteLine(p.HelloAsync("yzk").Result);
    Console.WriteLine(p.Add(1, 2));
}
```

上面的代码还支持多次降级，方法上标注[HystrixCommand]并且 virtual 即可：

```
public class Person//需要 public 类
{
    [HystrixCommand(nameof(Hello1FallBackAsync))]
    public virtual async Task<string> HelloAsync(string name)//需要是虚方法
    {
        Console.WriteLine("hello" + name);
        String s = null;
         s.ToString();
        return "ok";
    }

    [HystrixCommand(nameof(Hello2FallBackAsync))]
    public virtual async Task<string> Hello1FallBackAsync(string name)
    {
        Console.WriteLine("Hello 降级 1" + name);
        String s = null;
        s.ToString();
        return "fail_1";
    }

    public virtual async Task<string> Hello2FallBackAsync(string name)
    {
        Console.WriteLine("Hello 降级 2" + name);

        return "fail_2";
    }

    [HystrixCommand(nameof(AddFall))]
    public virtual int Add(int i, int j)
    {
        String s = null;
        s.ToString();
        return i + j;
    }
    public int AddFall(int i, int j)
    {
        return 0;
```

```
    }
}
```

十一、 细化框架

上面明白了了原理，然后直接展示写好的更复杂的 HystrixCommandAttribute，讲解代码，不现场敲了。

这是我会持续维护的开源项目

github 最新地址 https://github.com/yangzhongke/RuPeng.HystrixCore

Nuget 地址：https://www.nuget.org/packages/RuPeng.HystrixCore

重试：MaxRetryTimes 表示最多重试几次，如果为 0 则不重试，RetryIntervalMilliseconds 表示重试间隔的毫秒数；

熔断：EnableCircuitBreaker 是否启用熔断，ExceptionsAllowedBeforeBreaking 表示熔断前出现允许错误几次，MillisecondsOfBreak 表示熔断多长时间（毫秒）；

超时：TimeOutMilliseconds 执行超过多少毫秒则认为超时（0 表示不检测超时）

缓存：CacheTTLMilliseconds 缓存多少毫秒（0 表示不缓存），用“类名+方法名+所有参数值 ToString 拼接”做缓存 Key（唯一的要求就是参数的类型 ToString 对于不同对象一定要不一样）。

由于 CircuitBreaker 要求同一段代码必须共享同一个 Policy 对象。而方法上标注的 Attribute 对于这个方法来讲就是唯一的对象，一个方法对应一个方法上标注的 Attribute 对象。一般我们熔断控制是针对一个方法，一个方法无论是通过几个 Person 对象调用，无论是谁调用，只要全局出现 ExceptionsAllowedBeforeBreaking 次错误，就会熔断，这是我框架的实现，你如果认为不合理，你自己改去。我们在 Attribute 上声明一个 Policy 的成员变量，这样一个方法就对应一个 Policy 对象。

```
Install-Package Microsoft.Extensions.Caching.Memory
```

```
using System;
using AspectCore.DynamicProxy;
using System.Threading.Tasks;
using Polly;

namespace RuPeng.HystrixCore
{
    [AttributeUsage(AttributeTargets.Method)]
    public class HystrixCommandAttribute : AbstractInterceptorAttribute
    {
        /// <summary>
        /// 最多重试几次，如果为0则不重试
        /// </summary>
        public int MaxRetryTimes { get; set; } = 0;

        /// <summary>
```

```
        /// 重试间隔的毫秒数
        /// </summary>
        public int RetryIntervalMilliseconds { get; set; } = 100;

        /// <summary>
        /// 是否启用熔断
        /// </summary>
        public bool EnableCircuitBreaker { get; set; } = false;

        /// <summary>
        /// 熔断前出现允许错误几次
        /// </summary>
        public int ExceptionsAllowedBeforeBreaking { get; set; } = 3;

        /// <summary>
        /// 熔断多长时间（毫秒）
        /// </summary>
        public int MillisecondsOfBreak { get; set; } = 1000;

        /// <summary>
        /// 执行超过多少毫秒则认为超时（0表示不检测超时）
        /// </summary>
        public int TimeOutMilliseconds { get; set; } = 0;

        /// <summary>
        /// 缓存多少毫秒（0表示不缓存），用“类名+方法名+所有参数ToString拼接”做缓存Key
        /// </summary>

        public int CacheTTLMilliseconds { get; set; } = 0;

        private Policy policy;

        private static readonly Microsoft.Extensions.Caching.Memory.IMemoryCache memoryCache
= new Microsoft.Extensions.Caching.Memory.MemoryCache(new
Microsoft.Extensions.Caching.Memory.MemoryCacheOptions());

        /// <summary>
        ///
        /// </summary>
        /// <param name="fallBackMethod">降级的方法名</param>
        public HystrixCommandAttribute(string fallBackMethod)
        {
            this.FallBackMethod = fallBackMethod;
        }
```

```
        public string FallBackMethod { get; set; }

        public override async Task Invoke(AspectContext context, AspectDelegate next)
        {
            //一个HystrixCommand中保持一个policy对象即可
            //其实主要是CircuitBreaker要求对于同一段代码要共享一个policy对象
            //根据反射原理，同一个方法就对应一个HystrixCommandAttribute，无论几次调用，
            //而不同方法对应不同的HystrixCommandAttribute对象，天然的一个policy对象共享
            //因为同一个方法共享一个policy，因此这个CircuitBreaker是针对所有请求的。
            //Attribute也不会在运行时再去改变属性的值，共享同一个policy对象也没问题
            lock (this)//因为Invoke可能是并发调用，因此要确保policy赋值的线程安全
            {
                if (policy == null)
                {
                    policy = Policy.NoOpAsync();//创建一个空的Policy
                    if (EnableCircuitBreaker)
                    {
                        policy =
policy.WrapAsync(Policy.Handle<Exception>().CircuitBreakerAsync(ExceptionsAllowedBeforeBreaking,
TimeSpan.FromMilliseconds(MillisecondsOfBreak)));
                    }
                    if (TimeOutMilliseconds > 0)
                    {
                        policy = policy.WrapAsync(Policy.TimeoutAsync(() =>
TimeSpan.FromMilliseconds(TimeOutMilliseconds), Polly.Timeout.TimeoutStrategy.Pessimistic));
                    }
                    if (MaxRetryTimes > 0)
                    {
                        policy =
policy.WrapAsync(Policy.Handle<Exception>().WaitAndRetryAsync(MaxRetryTimes, i =>
TimeSpan.FromMilliseconds(RetryIntervalMilliseconds)));
                    }
                    Policy policyFallBack = Policy
                    .Handle<Exception>()
                    .FallbackAsync(async (ctx, t) =>
                    {
                       //这里拿到的就是ExecuteAsync(ctx => next(context), pollyCtx);这里传的
pollyCtx
                        AspectContext aspectContext = (AspectContext)ctx["aspectContext"];
                        var fallBackMethod =
context.ServiceMethod.DeclaringType.GetMethod(this.FallBackMethod);
                        Object fallBackResult = fallBackMethod.Invoke(context.Implementation,
context.Parameters);
```

```
                    //不能如下这样，因为这是闭包相关，如果这样写第二次调用Invoke的时候context指向的
                    //还是第一次的对象，所以要通过Polly的上下文来传递AspectContext
                    //context.ReturnValue = fallBackResult;
                    aspectContext.ReturnValue = fallBackResult;
                }, async (ex, t) => { });

                policy = policyFallBack.WrapAsync(policy);
            }
        }

        //把本地调用的AspectContext传递给Polly，主要给FallbackAsync中使用，避免闭包的坑
        Context pollyCtx = new Context();//Context是polly中通过Execute给FallBack、Execute等回调方法传上下文对象使用的
        pollyCtx["aspectContext"] = context;//context是aspectCore的上下文

        //Install-Package Microsoft.Extensions.Caching.Memory
        if (CacheTTLMilliseconds > 0)
        {
            //用类名+方法名+参数的下划线连接起来作为缓存key
            string cacheKey = "HystrixMethodCacheManager_Key_" + context.ServiceMethod.DeclaringType
                                                                + "." + context.ServiceMethod + string.Join("_", context.Parameters);
            //尝试去缓存中获取。如果找到了，则直接用缓存中的值做返回值
            if (memoryCache.TryGetValue(cacheKey, out var cacheValue))
            {
                context.ReturnValue = cacheValue;
            }
            else
            {
                //如果缓存中没有，则执行实际被拦截的方法
                await policy.ExecuteAsync(ctx => next(context), pollyCtx);
                //存入缓存中
                using (var cacheEntry = memoryCache.CreateEntry(cacheKey))
                {
                    cacheEntry.Value = context.ReturnValue;//返回值放入缓存
                    cacheEntry.AbsoluteExpiration = DateTime.Now + TimeSpan.FromMilliseconds(CacheTTLMilliseconds);
                }
            }
        }
        else//如果没有启用缓存，就直接执行业务方法
        {
```

```
                await policy.ExecuteAsync(ctx => next(context), pollyCtx);
            }
        }
    }
}
```

没必要、也不可能把所有 Polly 都封装到 Hystrix 中。框架不是万能的，不用过度框架，过度框架带来的复杂度陡增，从人人喜欢变成人人恐惧。

十二、 结合 asp.net core 依赖注入

在 asp.net core 项目中，可以借助于 asp.net core 的依赖注入，简化代理类对象的注入，不用再自己调用 ProxyGeneratorBuilder 进行代理类对象的注入了。

```
Install-Package AspectCore.Extensions.DependencyInjection
```

修改 Startup.cs 的 ConfigureServices 方法，把返回值从 void 改为 IServiceProvider

```
using AspectCore.Extensions.DependencyInjection;
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddSingleton<Person>();
    return services.BuildAspectCoreServiceProvider();
}
```

其中 services.AddSingleton<Person>(); 表示把 Person 注入。BuildAspectCoreServiceProvider 是让 aspectcore 接管注入。

在 Controller 中就可以通过构造函数进行依赖注入了：

```
public class ValuesController : Controller
{
    private Person p;
    public ValuesController(Person p)
    {
        this.p = p;
    }
}
```

当然要通过反射扫描所有 Service 类，只要类中有标记了 CustomInterceptorAttribute 的方法都算作服务实现类。为了避免一下子扫描所有类，所以 RegisterServices 还是手动指定从哪个程序集中加载。

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    RegisterServices(this.GetType().Assembly, services);
    return services.BuildAspectCoreServiceProvider();
}
```

```
private static void RegisterServices(Assembly asm, IServiceCollection services)
{
    //遍历程序集中的所有 public 类型
    foreach (Type type in asm.GetExportedTypes())
    {
        //判断类中是否有标注了 CustomInterceptorAttribute 的方法
        bool hasCustomInterceptorAttr = type.GetMethods()
          .Any(m => m.GetCustomAttribute(typeof(CustomInterceptorAttribute)) != null);
        if (hasCustomInterceptorAttr)
        {
            services.AddSingleton(type);
        }
    }
}
```

RestTemplate+Hystrix+之前测试用的两个微服务，结合到一起组合演示一下。

Ocelot API 网关(API GateWay)

现有微服务的几点不足：

1） 对于在微服务体系中、和 Consul 通讯的微服务来讲，使用服务名即可访问。但是对于手机、web 端等外部访问者仍然需要和 N 多服务器交互，需要记忆他们的服务器地址、端口号等。一旦内部发生修改，很麻烦，而且有时候内部服务器是不希望外界直接访问的。

2） 各个业务系统的人无法自由的维护自己负责的服务器；

3） 现有的微服务都是“我家大门常打开”，没有做权限校验。如果把权限校验代码写到每个微服务上，那么开发工作量太大。

4） 很难做限流、收费等。

ocelot 中文文档：https://blog.csdn.net/sD7O95O/article/details/79623654

资料：http://www.csharpkit.com/apigateway.html

官网：https://github.com/ThreeMammals/Ocelot

腾讯.Net 大队长“张善友”是项目主力开发人员之一。

一、 Ocelot 基本配置

Ocelot 就是一个提供了请求路由、安全验证等功能的 API 网关微服务。

建一个空的 asp.net core 项目。

Install-Package Ocelot

项目根目录下创建 configuration.json

ReRoutes 下就是多个路由规则

```
{
```

```
  "ReRoutes": [
    {
      "DownstreamPathTemplate": "/api/{url}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/MsgService/{url}",
      "UpstreamHttpMethod": [ "Get", "Post" ]
    },
    {
      "DownstreamPathTemplate": "/api/{url}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5003
        }
      ],
      "UpstreamPathTemplate": "/ProductService/{url}",
      "UpstreamHttpMethod": [ "Get","Post" ]
    }
  ]
}
```

Program.cs的CreateWebHostBuilder中

```
.UseUrls("http://127.0.0.1:8888")
.ConfigureAppConfiguration((hostingContext, builder) => {
    builder.AddJsonFile("configuration.json",false, true);
})
```

在ConfigureAppConfiguration 中AddJsonFile是解析json配置文件的方法。为了确保直接在bin下直接dotnet运行的时候能找到配置文件，所以要在vs中把配置文件的【复制到输出目录】设置为【如果如果较新则复制】。

Startup.cs中

通过构造函数注入一个private IConfiguration Configuration;

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddOcelot(Configuration);
```

```
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseOcelot().Wait();//不要忘了写Wait
}
```

这样当访问http://127.0.0.1:8888/MsgService/Send?msg=aaa的时候就会访问
http://127.0.0.1:5002/api/email/Send?msg=aaa
UpstreamHttpMethod表示对什么样的请求类型做转发。

二、 Ocelot+Consul

上面的配置还是把服务的ip地址写死了，Ocelot可以和Consul通讯，通过服务名字来配置。
只要改配置文件即可

```
{
  "ReRoutes": [
    {
      "DownstreamPathTemplate": "/api/{url}",
      "DownstreamScheme": "http",
      "UpstreamPathTemplate": "/MsgService/{url}",
      "UpstreamHttpMethod": [ "Get", "Post" ],
      "ServiceName": "MsgService",
      "LoadBalancerOptions": {
        "Type": "RoundRobin"
      },
      "UseServiceDiscovery": true
    }
  ],

  "GlobalConfiguration": {
    "ServiceDiscoveryProvider": {
      "Host": "localhost",
      "Port": 8500
    }
  }
}
```

有多个服务就 ReRoutes 下面配置多组即可

访问 http://localhost:8888/MsgService/SMS/Send_MI 即可，请求报文体{phoneNum:"110",msg:"aaaaaaaaaaaa"}。

表示只要是/MsgService/开头的都会转给后端的服务名为" MsgService "的一台服务器，转发的路径是"/api/{url}"。LoadBalancerOptions 中"LeastConnection"表示负载均衡算法是“选择当前最少连接数的服务器”，如果改为 RoundRobin 就是“轮询”。ServiceDiscoveryProvider 是 Consul 服务器的配置。

Ocelot 因为是流量中枢，也是可以做集群的。

(*) 也支持 Eureka 进行服务的注册、查找（http://ocelot.readthedocs.io/en/latest/features/servicediscovery.html），也支持访问 Service Fabric 中的服务（http://ocelot.readthedocs.io/en/latest/features/servicefabric.html）。

三、 Ocelot 其他功能简单介绍

1、 限流：

文档：http://ocelot.readthedocs.io/en/latest/features/ratelimiting.html

需要和 Identity Server 一起使用，其他的限速是针对 clientId 限速，而不是针对 ip 限速。

比如我调用微博的 api 开发了一个如鹏版新浪微博，我的 clientid 是 rpwb，然后限制了 1 秒钟只能调用 1000 次，那么所有用如鹏版微博这个 app 的所有用户加在一起，在一秒钟之内，不能累计超过 1000 次。目前开放式 api 的限流都是这个套路。

如果要做针对 ip 的限速等，要自己在 Ocelot 前面架设 Nginx 来实现。

2、 请求缓存

http://ocelot.readthedocs.io/en/latest/features/caching.html

只支持 get，只要 url 不变，就会缓存。

3、 QOS（熔断器）

http://ocelot.readthedocs.io/en/latest/features/qualityofservice.html

JWT 算法

一、 JWT 简介

内部 Restful 接口可以“我家大门常打开”，但是如果要给 app 等使用的接口，则需要做权限校验，不能谁都随便调用。

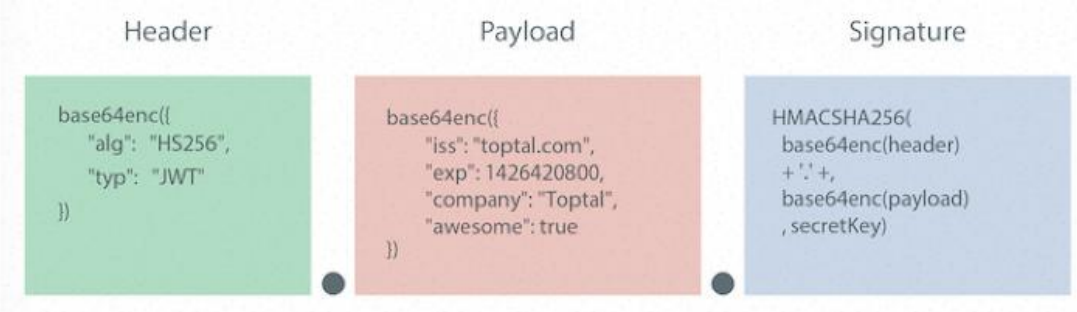
Restful 接口不是 web 网站，App 中很难直接处理 SessionId，而且 Cookie 有跨域访问的限制，所以一般不能直接用后端 Web 框架内置的 Session 机制。但是可以用类似 Session 的机制，用户登录之后返回一个类似 SessionId 的东西，服务器端把 SessionId 和用户的信息对应关系保存到 Redis 等地方，客户端把 SessionId 保存起来，以后每次请求的时候都带着这个 SessionId。

用类似 Session 这种机制的坏处：需要集中的 Session 机制服务器；不可以在 nginx、CDN

等静态文件处理服务器上校验权限；每次都要根据 SessionId 去 Redis 服务器获取用户信息，效率低；

JWT（Json Web Token）是现在流行的一种对 Restful 接口进行验证的机制的基础。JWT 的特点：把用户信息放到一个 JWT 字符串中，用户信息部分是明文的，再加上一部分签名区域，签名部分是服务器对于“明文部分+秘钥”加密的，这个加密信息只有服务器端才能解析。用户端只是存储、转发这个 JWT 字符串。如果客户端篡改了明文部分，那么服务器端解密时候会报错。

JWT 由三块组成，可以把用户名、用户 Id 等保存到 Payload 部分



注意 Payload 和 Header 部分都是 Base64 编码，可以轻松的 Base64 解码回来。因此 Payload 部分约等于是明文的，因此不能在 Payload 中保存不能让别人看到的机密信息。虽然说 Payload 部分约等于是明文的，但是不用担心 Payload 被篡改，因为 Signature 部分是根据 header+payload+secretKey 进行加密算出来的，如果 Payload 被篡改，就可以根据 Signature 解密时候校验。

用 JWT 做权限验证的好处：无状态，更有利于分布式系统，不需要集中的 Session 机制服务器；可以在 nginx、CDN 等静态文件处理服务器上校验权限；获取用户信息直接从 JWT 中就可以读取，效率高；

二、 .Net 中使用 JWT 算法

1） 加密

```
Install-Package JWT
var payload = new Dictionary<string, object>
{
    { "UserId", 123 },
    { "UserName", "admin" }
};
var secret = "GQDstcKsx0NHjPOuXOYg5MbeJ1XT0uFiwDVvVBrk";//不要泄露

IJwtAlgorithm algorithm = new HMACSHA256Algorithm();
IJsonSerializer serializer = new JsonNetSerializer();
IBase64UrlEncoder urlEncoder = new JwtBase64UrlEncoder();
IJwtEncoder encoder = new JwtEncoder(algorithm, serializer, urlEncoder);

var token = encoder.Encode(payload, secret);
```

```
Console.WriteLine(token);
```

2） 解密

```
var token = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJVc2VySWQiOjEyMywiVXNlck5hbWUiOiJhZG1pbiJ9.Qjw1epD5P6p4Yy2yju3-fkq28PddznqRj3ESfALQy_U";
var secret = "GQDstcKsx0NHjPOuXOYg5MbeJ1XT0uFiwDVvVBrk";
try
{
    IJsonSerializer serializer = new JsonNetSerializer();
    IDateTimeProvider provider = new UtcDateTimeProvider();
    IJwtValidator validator = new JwtValidator(serializer, provider);
    IBase64UrlEncoder urlEncoder = new JwtBase64UrlEncoder();
    IJwtDecoder decoder = new JwtDecoder(serializer, validator, urlEncoder);

    var json = decoder.Decode(token, secret, verify: true);
    Console.WriteLine(json);
}
catch(FormatException)
{
    Console.WriteLine("Token format invalid");
}
catch (TokenExpiredException)
{
    Console.WriteLine("Token has expired");
}
catch (SignatureVerificationException)
{
    Console.WriteLine("Token has invalid signature");
}
```

试着篡改一下 Payload 部分。

3） 过期时间

在 payload 中增加一个名字为 exp 的值，值为过期时间和 1970/1/1 00:00：00 相差的秒数

```
double exp = (DateTime.UtcNow.AddSeconds(10) - new DateTime(1970, 1, 1)).TotalSeconds;
```

4） 不用秘钥解析数据 payload

因为 payload 部分是明文的，所以在不知道秘钥的时候也可以用 Decode、DecodeToObject 等不需要秘钥的方法把 payload 部分解析出来。

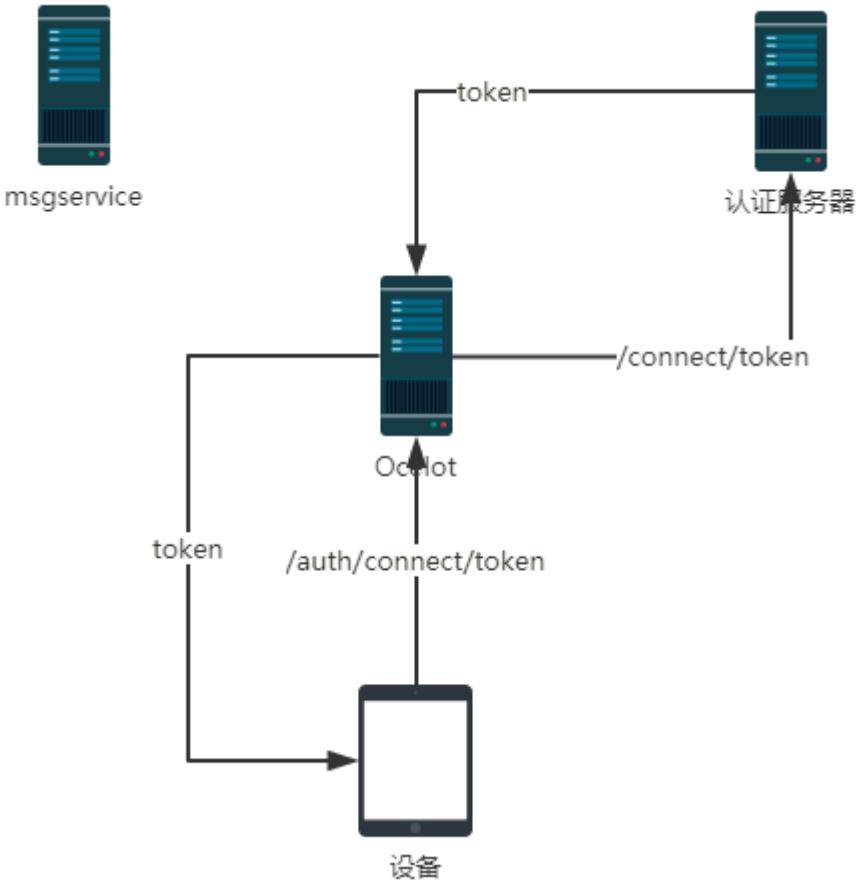
```
var token = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJVc2VySWQiOjEyMywiVXNlck5hbWUiOiJhZG1pbiJ9.Qjw1epD5P6p4Yy2yju3-fkq28PddznqRj3ESfALQy_U";
try
{
    IJsonSerializer serializer = new JsonNetSerializer();
```

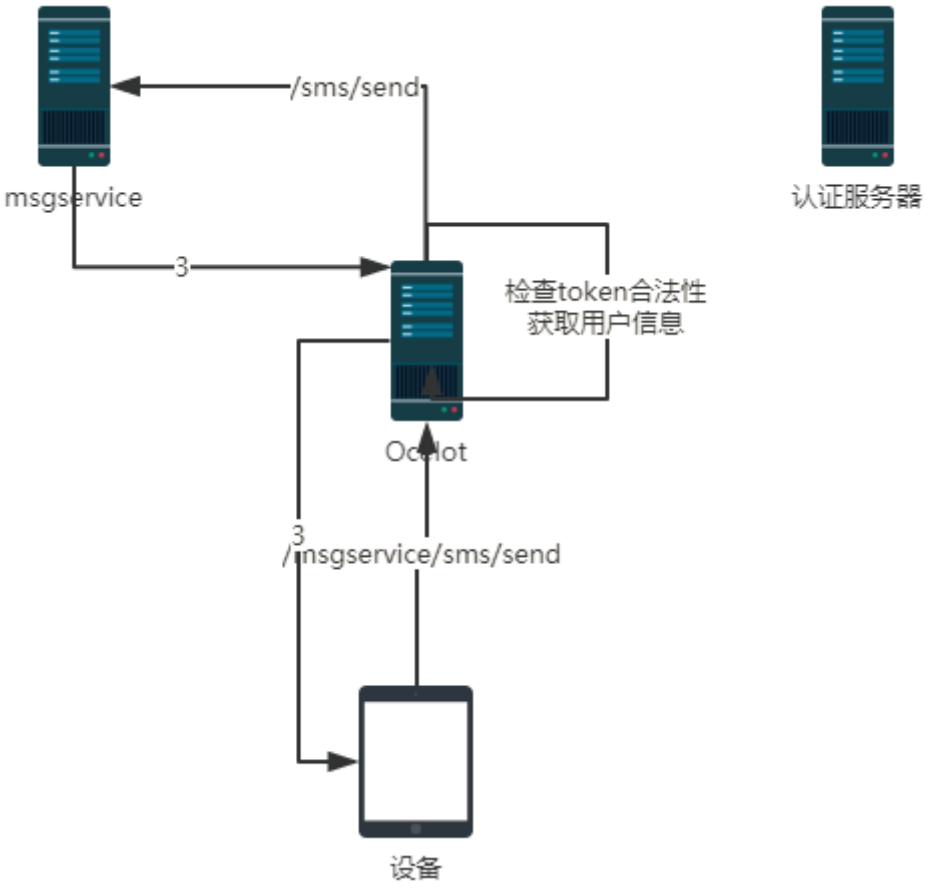
```
    IDateTimeProvider provider = new UtcDateTimeProvider();
    IJwtValidator validator = new JwtValidator(serializer, provider);
    IBase64UrlEncoder urlEncoder = new JwtBase64UrlEncoder();
    IJwtDecoder decoder = new JwtDecoder(serializer, validator, urlEncoder);

    var json = decoder.Decode(token);
    Console.WriteLine(json);
}
catch (FormatException)
{
    Console.WriteLine("Token format invalid");
}
catch (TokenExpiredException)
{
    Console.WriteLine("Token has expired");
}
```

Ocelot+Identity Server

用 JWT 机制实现验证的原理如下图：

认证服务器负责颁发 Token（相当于 JWT 值）和校验 Token 的合法性。


msgservice
token
认证服务器
/connect/token
Ocelot
token
/auth/connect/token
设备

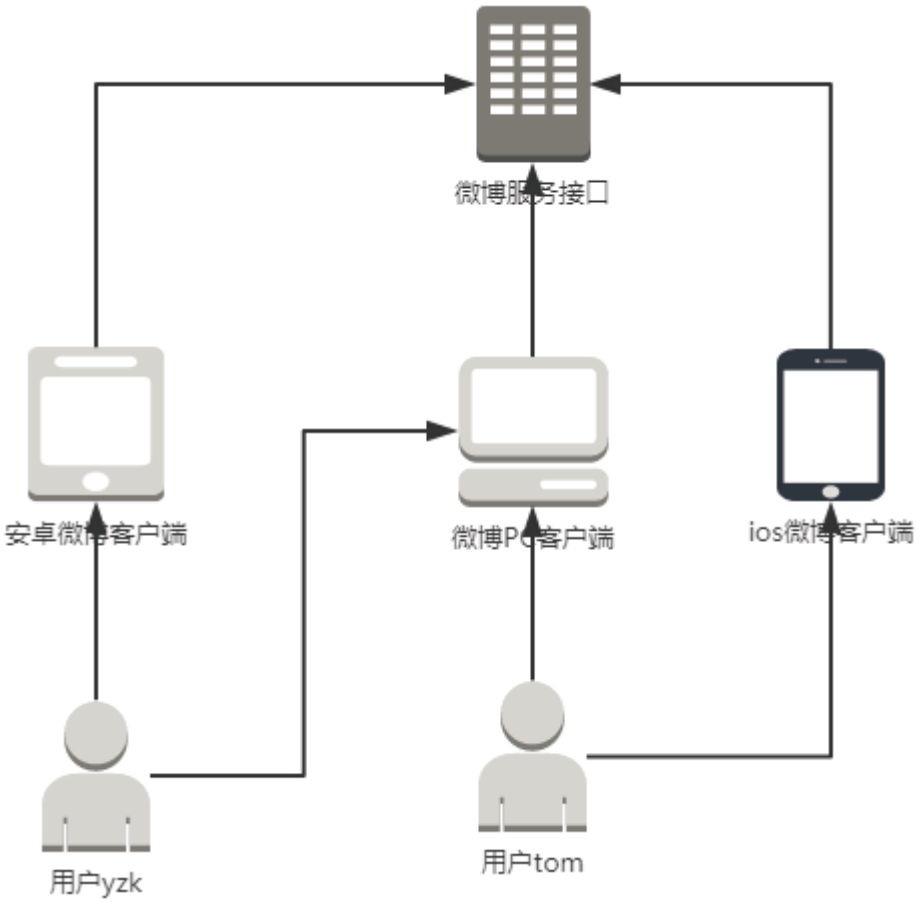


一、 相关概念

API 资源（API Resource）：微博服务器接口、斗鱼弹幕服务器接口、斗鱼直播接口就是 API 资源。

客户端（Client）：Client 就是官方微博 android 客户端、官方微博 ios 客户端、第三方微博客户端、微博助手等。

身份资源（Identity Resource）：就是用户。



一个用户可能使用多个客户端访问服务器；一个客户端也可能服务多个用户。
封禁了一个客户端，所有用户都不能使用这个这个客户端访问服务器，但是可以使用其他客户端访问；
封禁了一个用户，这个用户在所有设备上都不能访问，但是不影响其他用户。

二、 搭建 identity server 认证服务器

新建一个空的 web 项目 ID4.IdServer
Install-Package IdentityServer4
首先编写一个提供应用列表、账号列表的 Config 类

```
using IdentityServer4.Models;
using System.Collections.Generic;

namespace ID4.IdServer
{
    public class Config
    {
        /// <summary>
        /// 返回应用列表
        /// </summary>
        /// <returns></returns>
```

```
        public static IEnumerable<ApiResource> GetApiResources()
        {
            List<ApiResource> resources = new List<ApiResource>();
            //ApiResource第一个参数是应用的名字，第二个参数是描述
            resources.Add(new ApiResource("MsgAPI", "消息服务API"));
            resources.Add(new ApiResource("ProductAPI", "产品API"));
            return resources;
        }

        /// <summary>
        /// 返回账号列表
        /// </summary>
        /// <returns></returns>
        public static IEnumerable<Client> GetClients()
        {
            List<Client> clients = new List<Client>();
            clients.Add(new Client
            {
                ClientId = "clientPC1",//API账号、客户端Id
                AllowedGrantTypes = GrantTypes.ClientCredentials,
                ClientSecrets =
                {
                    new Secret("123321".Sha256())//秘钥
                },
                AllowedScopes = { "MsgAPI","ProductAPI" }//这个账号支持访问哪些应用
            });
            return clients;
        }
    }
}
```

如果允许在数据库中配置账号等信息，那么可以从数据库中读取然后返回这些内容。疑问待解。

修改 Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentityServer()
        .AddDeveloperSigningCredential()
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients());
}
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
```

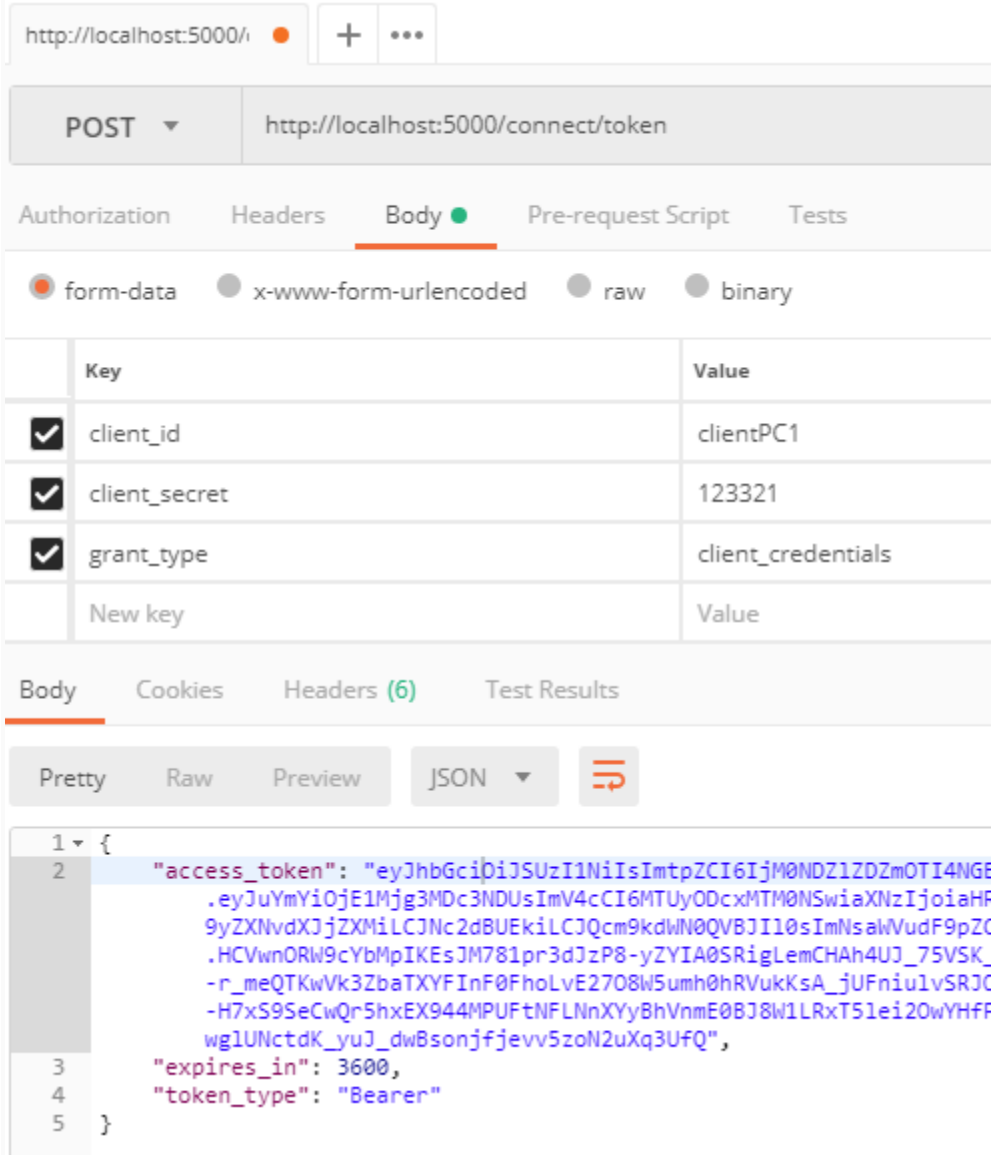
```
    app.UseIdentityServer();
}
```

然后在 9500 端口启动

在 postman 里发出请求，获取 token

http://localhost:9500/connect/token，发 Post 请求，表单请求内容（注意不是报文头）：

client_id=clientPC1 client_secret=123321 grant_type=client_credentials



把返回的 access_token 留下来后面用（注意有有效期）。

注意，其实不应该让客户端直接去申请 token，这只是咱演示，后面讲解正确做法。

三、 搭建 Ocelot 服务器项目

空 Web 项目，项目名 ID4.Ocelot1

nuget 安装 IdentityServer4、Ocelot

编写配置文件 Ocelot.json（注意设置【如果较新则】）

```
{
  "ReRoutes": [
    {
      "DownstreamPathTemplate": "/api/{url}",
      "DownstreamScheme": "http",
      "UpstreamPathTemplate": "/MsgService/{url}",
      "UpstreamHttpMethod": [ "Get", "Post" ],
      "ServiceName": "MsgService",
      "LoadBalancerOptions": {
        "Type": "RoundRobin"
      },
      "UseServiceDiscovery": true,
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "MsgKey",
        "AllowedScopes": []
      }
    },
    {
      "DownstreamPathTemplate": "/api/{url}",
      "DownstreamScheme": "http",
      "UpstreamPathTemplate": "/ProductService/{url}",
      "UpstreamHttpMethod": [ "Get", "Post" ],
      "ServiceName": "ProductService",
      "LoadBalancerOptions": {
        "Type": "RoundRobin"
      },
      "UseServiceDiscovery": true,
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "ProductKey",
        "AllowedScopes": []
      }
    }
  ],

  "GlobalConfiguration": {
    "ServiceDiscoveryProvider": {
      "Host": "localhost",
      "Port": 8500
    }
  }
}
```

把/MsgService 访问的都转给消息后端服务器（使用 Consul 进行服务发现）。也可以把 Identity Server 配置到 Ocelot，但是我们不做，后边会讲为什么不放。

Program.cs 的 CreateWebHostBuilder 中加载 Ocelot.json

```
.ConfigureAppConfiguration((hostingContext, builder) =>
{
    builder.AddJsonFile("Ocelot.json",false, true);
})
```

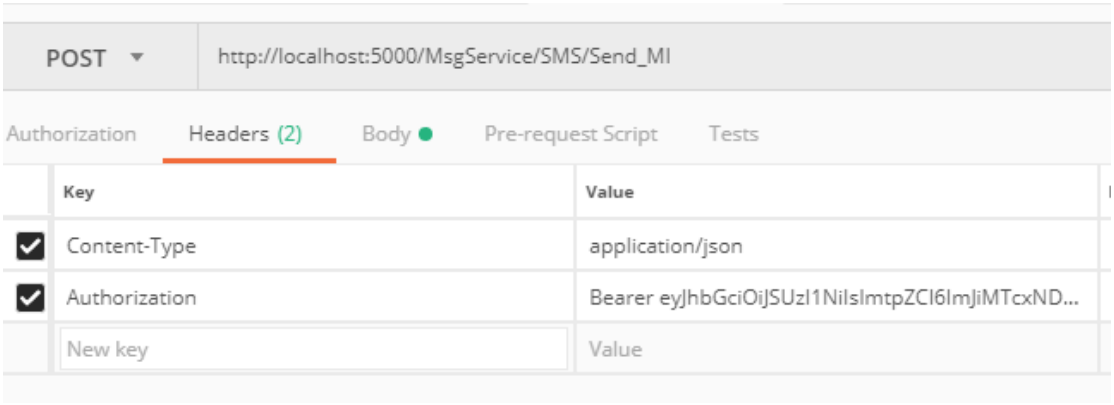
修改 Startup.cs 让 Ocelot 能够访问 Identity Server 进行 Token 的验证

```
using System;
using IdentityServer4.AccessTokenValidation;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Ocelot.DependencyInjection;
using Ocelot.Middleware;

namespace ID4.Ocelot1
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            //指定Identity Server的信息
            Action<IdentityServerAuthenticationOptions> isaOptMsg = o => {
                o.Authority = "http://localhost:9500";
                o.ApiName = "MsgAPI";//要连接的应用的名字
                o.RequireHttpsMetadata = false;
                o.SupportedTokens = SupportedTokens.Both;
                o.ApiSecret = "123321";//秘钥
            };
            Action<IdentityServerAuthenticationOptions> isaOptProduct = o => {
                o.Authority = "http://localhost:9500";
                o.ApiName = "ProductAPI";//要连接的应用的名字
                o.RequireHttpsMetadata = false;
                o.SupportedTokens = SupportedTokens.Both;
                o.ApiSecret = "123321";//秘钥
            };
            services.AddAuthentication()
            //对配置文件中使用ChatKey配置了AuthenticationProviderKey=MsgKey
            //的路由规则使用如下的验证方式
            .AddIdentityServerAuthentication("MsgKey", isaOptMsg)
            .AddIdentityServerAuthentication("ProductKey", isaOptProduct);
            services.AddOcelot();

        }
```

```
            // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
            public void Configure(IApplicationBuilder app, IHostingEnvironment env)
            {
                if (env.IsDevelopment())
                {
                    app.UseDeveloperExceptionPage();
                }

                app.UseOcelot().Wait();
            }
        }
}
```

很显然我们可以让不同的服务采用不同的Identity Server。

启动 Ocelot 服务器，然后向 ocelot 请求/MsgService/SMS/Send_MI（报文体还是要传 json 数据），在请求头（不是报文体）里加上：
Authorization="Bearer "+上面 identityserver 返回的 accesstoken



如果返回 401，那就是认证错误。

Ocelot 会把 Authorization 值传递给后端服务器，这样在后端服务器可以用 IJwtDecoder 的这个不传递 key 的重载方法 IDictionary<string, object> DecodeToObject(string token)，就可以在不验证的情况下获取 client_id 等信息。

也可以把 Identity Server 通过 Consul 进行服务治理。

Ocelot+Identity Server 实现了接口的权限验证，各个业务系统不需要再去做验证。

四、 不能让客户端请求 token

上面是让客户端去请求 token，如果项目中这么搞的话，就把 client_id 特别是 secret 泄露给普通用户的。正确的做法应该是，开发一个 token 服务，由这个服务来向 identity Server 请求 token，客户端向 token 服务发请求，把 client_id、secret 藏到这个 token 服务器上。当然这个服务器也要经过 Ocelot 转发。

这个做起来很简单，就不演示了。放到下面一起演示。

五、 用户名密码登录

如果 Api 和用户名、密码无关（比如系统内部之间 API 的调用），那么上面那样做就可以了，但是有时候需要用户身份验证的（比如 Android 客户端）。也就是在请求 token 的时候还要验证用户名密码，在服务中还可以获取登录用户信息。

修改的地方：

1、 ID4.IdServer 项目中

增加类 ProfileService.cs

```
using IdentityServer4.Models;
using IdentityServer4.Services;
using System.Linq;
using System.Threading.Tasks;

namespace ID4.IdServer
{
    public class ProfileService : IProfileService
    {
        public async Task GetProfileDataAsync(ProfileDataRequestContext context)
        {
            var claims = context.Subject.Claims.ToList();
            context.IssuedClaims = claims.ToList();
        }

        public async Task IsActiveAsync(IsActiveContext context)
        {
            context.IsActive = true;
        }
    }
}
```

增加类 ResourceOwnerPasswordValidator.cs

```
using IdentityServer4.Models;
using IdentityServer4.Validation;
using System.Security.Claims;
using System.Threading.Tasks;

namespace ID4.IdServer
{
    public class ResourceOwnerPasswordValidator : IResourceOwnerPasswordValidator
    {
        public async Task ValidateAsync(ResourceOwnerPasswordValidationContext context)
        {
            //根据context.UserName和context.Password与数据库的数据做校验，判断是否合法
```

```
            if (context.UserName == "yzk" && context.Password == "123")
            {
                context.Result = new GrantValidationResult(
                  subject: context.UserName,
                  authenticationMethod: "custom",
                  claims:new Claim[] { new Claim("Name",context.UserName), new Claim("UserId",
"111"), new Claim("RealName", "杨中科"), new Claim("Email", "yzk365@qq.com") });
            }
            else
            {
                //验证失败
                context.Result = new GrantValidationResult(TokenRequestErrors.InvalidGrant, "invalid
custom credential");
            }
        }
    }
}
```

当然这里的用户名密码是写死的，可以在项目中连接自己的用户数据库进行验证。claims 中可以放入多组用户的信息，这些信息都可以在业务系统中获取到。

Config.cs

修改一下，主要是把GetClients中的AllowedGrantTypes属性值改为GrantTypes.ResourceOwnerPassword，并且在AllowedScopes中加入IdentityServerConstants.StandardScopes.OpenId, //必须要添加，否则报forbidden错误

IdentityServerConstants.StandardScopes.Profile

修改后的 Config.cs

```
using System.Collections.Generic;
using IdentityServer4;
using IdentityServer4.Models;


namespace ID4.IdServer
{
    public class Config
    {
        /// <summary>
        ///  返回应用列表
        /// </summary>
        /// <returns></returns>
        public static IEnumerable<ApiResource> GetApiResources()
        {
            List<ApiResource> resources = new List<ApiResource>();
            //ApiResource第一个参数是应用的名字，第二个参数是描述
            resources.Add(new ApiResource("MsgAPI", "消息服务API"));
```

```
            resources.Add(new ApiResource("ProductAPI", "产品API"));
            return resources;
        }

        /// <summary>
        /// 返回客户端账号列表
        /// </summary>
        /// <returns></returns>
        public static IEnumerable<Client> GetClients()
        {
            List<Client> clients = new List<Client>();
            clients.Add(new Client
            {
                ClientId = "clientPC1",//API账号、客户端Id
                AllowedGrantTypes = GrantTypes.ResourceOwnerPassword,
                ClientSecrets =
                {
                    new Secret("123321".Sha256())//秘钥
                },
                AllowedScopes = { "MsgAPI",
"ProductAPI",IdentityServerConstants.StandardScopes.OpenId, //必须要添加，否则报forbidden错误
                    IdentityServerConstants.StandardScopes.Profile }//这个账号支持访问哪些应用
            });
            return clients;
        }
    }
}
```

Startup.cs 的 ConfigureServices 修改为

```
public void ConfigureServices(IServiceCollection services)
{
    var idResources = new List<IdentityResource>
    {
        new IdentityResources.OpenId(), //必须要添加，否则报无效的 scope 错误
        new IdentityResources.Profile()
    };
    services.AddIdentityServer()
        .AddDeveloperSigningCredential()
        .AddInMemoryIdentityResources(idResources)
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients())//
        .AddResourceOwnerValidator<ResourceOwnerPasswordValidator>()
        .AddProfileService<ProfileService>();
}
```

主要是增加了 AddInMemoryIdentityResources、AddResourceOwnerValidator、AddProfileService

2、修改业务系统

以 MsgService 为例

Install-Package IdentityServer4.AccessTokenValidation

然后 Startup.cs 的 ConfigureServices 中增加

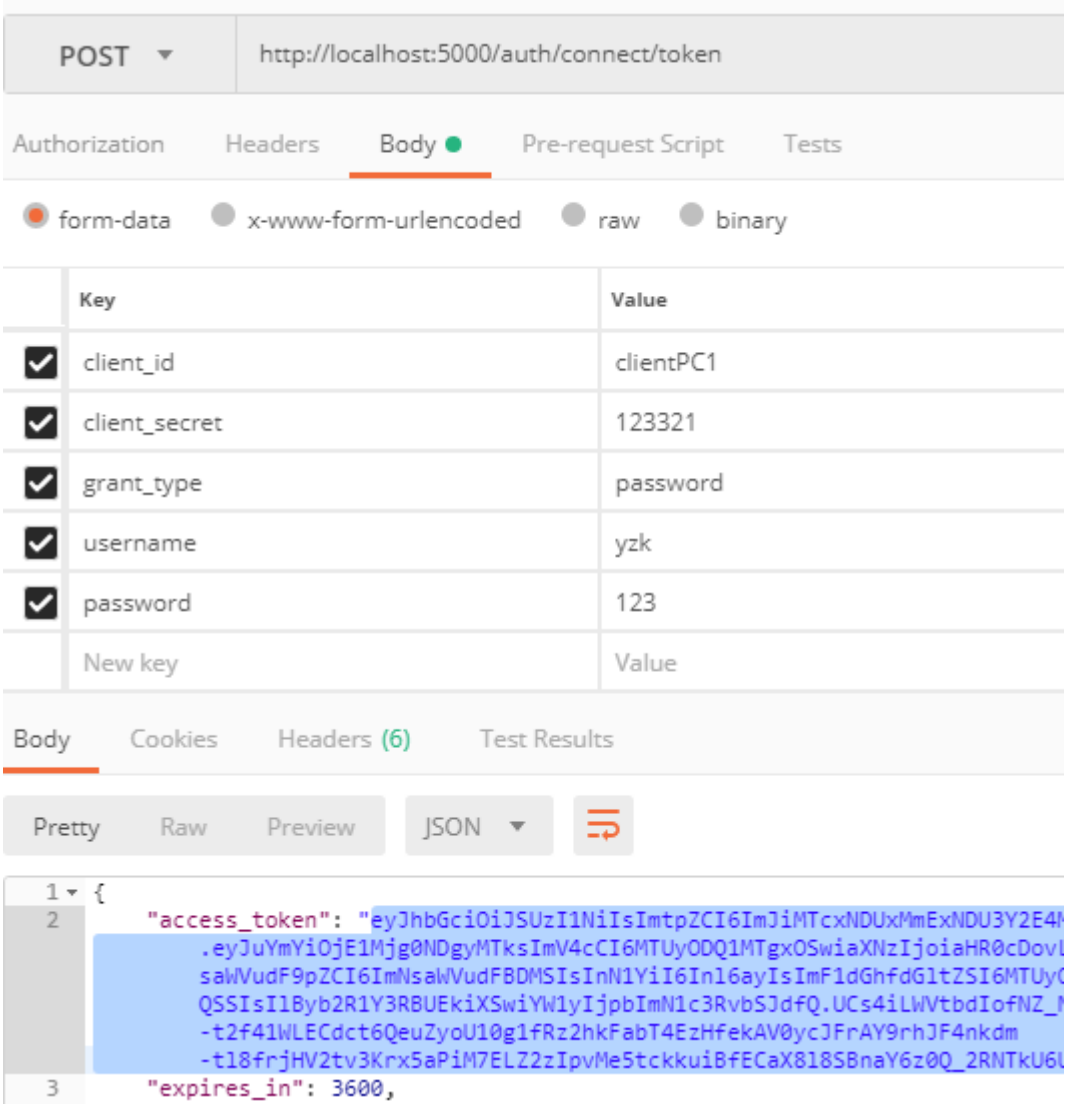
```
services.AddAuthentication("Bearer")
    .AddIdentityServerAuthentication(options =>
    {
        options.Authority = "http://localhost:9500";//identity server 地址
        options.RequireHttpsMetadata = false;
    });
```

Startup.cs 的 Configure 中增加

```
app.UseAuthentication();
```

3、请求 token

把报文头中的 grant_type 值改为 password，报文头增加 username、password 为用户名、密码。



像之前一样用返回的 access_token 传递给请求的 Authorization 中，在业务系统的 User 中就可以获取到 ResourceOwnerPasswordValidator 中为用户设置的 claims 等信息了。

```
public void Send_MI(dynamic model)
{
    string name = this.User.Identity.Name;//读取的就是"Name"这个特殊的 Claims 的值
    string userId = this.User.FindFirst("UserId").Value;
    string realName = this.User.FindFirst("RealName").Value;
    string email = this.User.FindFirst("Email").Value;
    Console.WriteLine($"name={name},userId={userId},realName={realName},email={email}");
    Console.WriteLine($"通过小米短信接口向{model.phoneNum}发送短信{model.msg}");
}
```

4、独立登录服务器

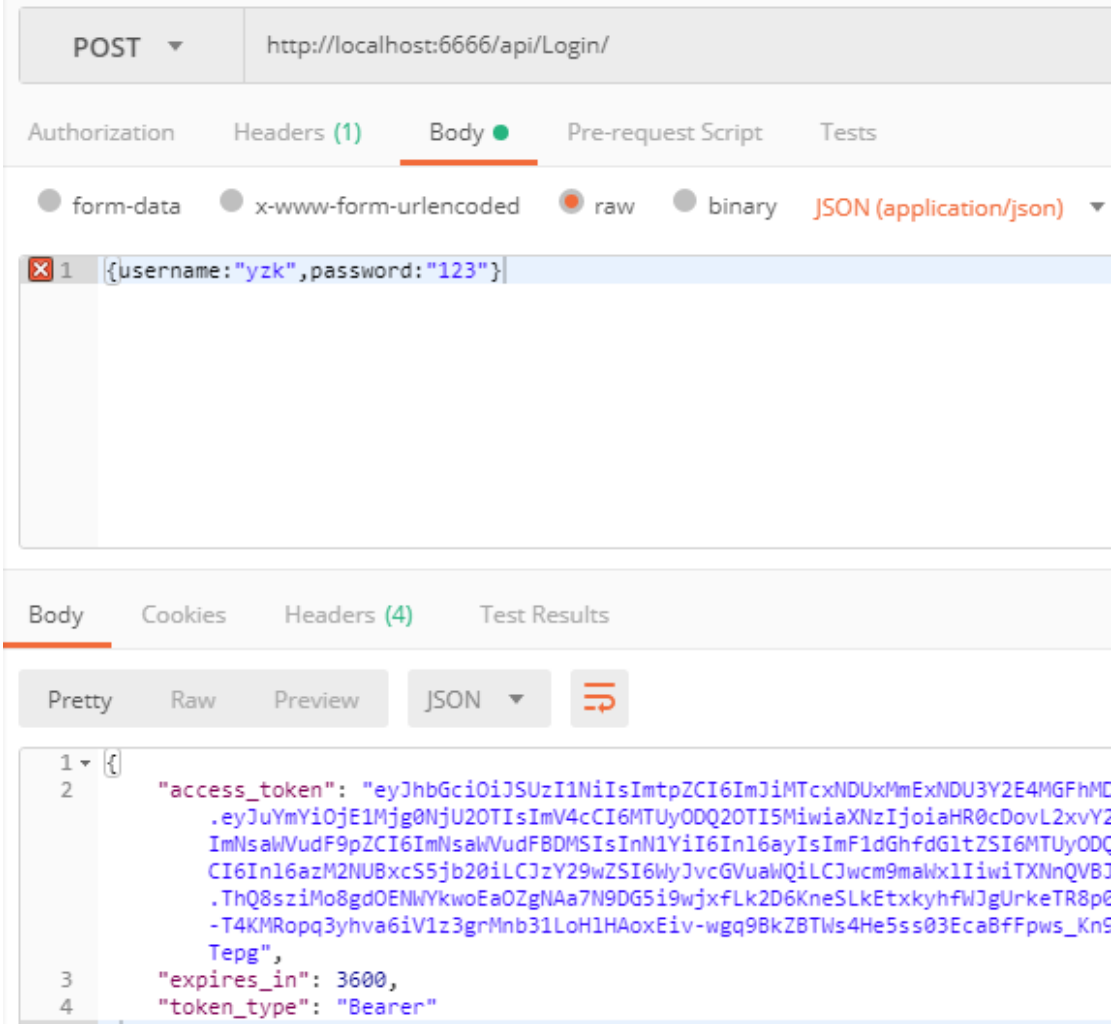
解决上面提到的“不能让客户端接触到 client_id、secret 的问题”

开发一个服务应用 LoginService

```
public class RequestTokenParam
```

```
{
        public string username { get; set; }
        public string password { get; set; }
}
using System.Collections.Generic;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace LoginService.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class LoginController : ControllerBase
    {
        [HttpPost]
        public async Task<ActionResult> RequestToken(RequestTokenParam model)
        {
            Dictionary<string, string> dict = new Dictionary<string, string>();
            dict["client_id"] = "clientPC1";
            dict["client_secret"] = "123321";
            dict["grant_type"] = "password";
            dict["username"] = model.username;
            dict["password"] = model.password;

             //由登录服务器向IdentityServer发请求获取Token
            using (HttpClient http = new HttpClient())
            using (var content = new FormUrlEncodedContent(dict))
            {
                var msg = await http.PostAsync("http://localhost:9500/connect/token", content);
                string result = await msg.Content.ReadAsStringAsync();
                return Content(result, "application/json");
            }
        }
    }
}
```

这样客户端只要向 LoginService 的/api/Login/发请求带上 json 报文体{username:"yzk",password:"123"}即可。客户端就不知道 client_secret 这些机密信息了。



把 LoginService 配置到 Ocelot 中。

参考文章：https://www.cnblogs.com/jaycewu/p/7791102.html

Thrift 高效通讯

一、 什么是 RPC

Restful 采用 Http 进行通讯，优点是开放、标准、简单、兼容性升级容易；缺点是性能略低。在 QPS 高或者对响应时间要求苛刻的服务上，可以用 RPC（Remote Procedure Call），RPC 由于采用二进制传输、TCP 通讯，所以通常性能更好。

.Net Core 下的 RPC（远程方法调用）框架有 gRPC、Thrift 等，都支持主流的编程语言。

RPC 虽然效率略高，但是耦合性强，如果兼容性处理不好的话，一旦服务器端接口升级，客户端就要更新，即使是增加一个参数，而 rest 则比较灵活。

最佳实践：对内一些性能要求高的场合用 RPC，对内其他场合以及对外用 Rest。比如 web 服务器和视频转码服务器之间通讯可以用 restful 就够了，转账接口用 RPC 性能会更高

一些。

二、 Thrift 基本使用

参考资料：https://www.cnblogs.com/focus-lei/p/8889389.html

1、 下载 thrift http://thrift.apache.org/
把 thrift-***.exe 解压到磁盘，改名为 thrift.exe（用起来方便一些）
2、 编写一个 UserService.thrift 文件（IDL）

```
namespace csharp RuPeng.ThriftTest1.Contract

service UserService{
    SaveResult Save(1:User user)
    User Get(1:i32 id)
    list<User> GetAll()
}

enum SaveResult {
  SUCCESS = 0,
  FAILED = 1,
}

struct User {
    1: required i64 Id;
    2: required string Name;
    3: required i32 Age;
    4:  optional bool IsVIP;
    5: optional string Remark;
}
```

service 定义的是服务类，enum 是枚举，struct 是传入或者传出的复杂数据类型（支持对象级联）。语法规范
http://thrift.apache.org/docs/idl

根据 thrift 语法生成 C#代码

```
thrift.exe -gen csharp UserService.thrift
```

创建一个类库项目 ThriftTest1.Contract，作为客户端和服务器之间的共用协议，把上一步生成的代码放进项目。

项目 nuget 安装 apache-thrift-netcore：

```
Install-Package apache-thrift-netcore
```

3、 创建服务器端项目 ThriftTest1.Server，建一个控制台项目（放到 web 项目中或者在 Linux 中用守护进程运行起来(SuperVisor 等，类似 Windows 下的“Windows 服务”)也可以）。
ThriftTest1.Server 项目引用 ThriftTest1.Contract
创建项目：ApplicationExtenssion.cs
编写实现类
UserServiceImpl.cs

```
public class UserServiceImpl : UserService.Iface
{
    public User Get(int id)
    {
        User u = new User();
        u.Id = id;
        u.Name = "用户" + id;
        u.Age = 6;
        return u;
    }

    public List<User> GetAll()
    {
        List<User> list = new List<User>();
        list.Add(new User { Id = 1, Name = "yzk", Age = 18, Remark = "hello" });
        list.Add(new User { Id = 2, Name = "rupeng", Age = 6 });
        return list;
    }

    public SaveResult Save(User user)
    {
        Console.WriteLine($"保存用户，{user.Id}");
        return SaveResult.SUCCESS;
    }
}

TServerTransport transport = new TServerSocket(8800);
var processor = new RuPeng.ThriftTest1.Contract.UserService.Processor(new
UserServiceImpl());
TServer server = new TThreadPoolServer(processor, transport);
server.Serve();
```

监听 8800 端口

客户端项目也引用 ThriftTest1.Contract 项目

```
using (TTransport transport = new TSocket("localhost", 8800))
using (TProtocol protocol = new TBinaryProtocol(transport))
using (var clientUser = new UserService.Client(protocol))
{
    transport.Open();
    User u = clientUser.Get(1);
    Console.WriteLine($"{u.Id},{u.Name}");
}
```

三、 一个服务器中放多个服务

0.9.1 之前只支持一个服务器一个服务，这也是建议的做法。之后支持多路服务

在 thrift 中增加一个服务

```
service CalcService{
    i32 Add(1:i32 i1,2:i32 i2)
}
```

服务器

```
TServerTransport transport = new TServerSocket(8800);
var processorUserService = new RuPeng.ThriftTest1.Contract.UserService.Processor(new UserServiceImpl());
var processorCalcService = new RuPeng.ThriftTest1.Contract.CalcService.Processor(new CalcServiceImpl());

var processorMulti = new TMultiplexedProcessor();
processorMulti.RegisterProcessor("userService", processorUserService);
processorMulti.RegisterProcessor("calcService", processorCalcService);

TServer server = new TThreadPoolServer(processorMulti, transport);

server.Serve();
```

客户端：

```
using (TTransport transport = new TSocket("localhost", 8800))
using (TProtocol protocol = new TBinaryProtocol(transport))
using (var protocolUserService = new TMultiplexedProtocol(protocol, "userService"))
using (var clientUser = new UserService.Client(protocolUserService))
using (var protocolCalcService = new TMultiplexedProtocol(protocol, "calcService"))
using (var clientCalc = new CalcService.Client(protocolCalcService))
{
    transport.Open();
    User u = clientUser.Get(1);
    Console.WriteLine($"{u.Id},{u.Name}");
    Console.WriteLine(clientCalc.Add(1,2));
}
```

https://www.cnblogs.com/focus-lei/p/8889389.html

(*)新版：thrift.exe -gen netcore UserService.thrift

貌似支持还不完善（http://www.cnblogs.com/zhaiyf/p/8351361.html ）还不能用，编译也有问题，值得期待的是：支持异步。

四、 Java 等其他语言的融入

和使用 Restful 做服务一样，Java 也可以调用、也可以做 Thrift 服务，演示一下 java 调用 c#写的 Thrift 服务的例子

Java 编译器版本需要>=1.6

Maven（thrift maven 版本一定要和生成代码的 thrift 的版本一致）：

```
<dependency>
  <groupId>org.apache.thrift</groupId>
  <artifactId>libthrift</artifactId>
  <version>0.11.0</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.5</version>
</dependency>
```

在 thrift 的 IDL 文件中加入一行（各个语言的 namespace 等参数可以共存）

```
namespace java com.rupeng.thriftTest1.contract
```

就可以控制生成的 java 类的报名，最好按照 java 的命名规范来。

```
thrift.exe -gen java UserService.thrift 产生 java 代码
```

Java 代码：

```
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TProtocol;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;

public class Main {

    public static void main(String[] args) throws Exception {
        System.out.println("客户端启动....");
        TTransport transport = new TSocket("localhost", 8800, 30000);
        TProtocol protocol = new TBinaryProtocol(transport);
        UserService.Client client = new UserService.Client(protocol);
        transport.open();
        User result = client.Get(1);

System.out.println(result.getAge()+result.getName()+result.getRemark());
    }
}
```

也可以用 Java 写服务器，C#调用。当然别的语言也可以。

接口设计原则“API design is like sex: Make one mistake and support it for the rest of your life”

五、 Thrift+Consul 服务发现

注册和发现和 Rest 方式没有什么区别。

consul 支持 tcp 健康监测：https://www.consul.io/docs/agent/checks.html

A TCP check:

```
{
  "check": {
    "id": "ssh",
    "name": "SSH TCP on port 22",
    "tcp": "localhost:22",
    "interval": "10s",
    "timeout": "1s"
  }
}
```

因为 Thrift 一般不对外，所以一般不涉及和 API 网关结合的问题

gRPC 的优势：支持异步；支持 Http2。

回顾一下我们学到的微服务：服务治理和服务发现、熔断降级、API 网关。

不是所有项目都适合微服务架构，互联网项目及结构复杂的企业信息系统才可以考虑微服务架构。

设计微服务架构，模块拆分的原则：可以独立运行，尽量服务间不要依赖，即使依赖层级也不要太深，不要想着还要 join。按业务划分、按模块划分。

把 MsgService 等用上面讲的所有 RestTemplate、Hystrix 等包装起来展示一下一个 Demo。

主要就是搞一个 Web 端、一个 App 服务器端，各自调用封装的 ProductService、MsgService。用 winform 模仿客户端。

扩展知识：

1、分布式跟踪、日志服务、监控等对微服务来说非常重要

2、gRPC 另外一个 RPC 框架，gRPC 的.Net Core 支持异步。百度“.net core grpc”https://www.jianshu.com/p/f5e1c002047a

3、https://github.com/neuecc/MagicOnion 可以参考下这位日本 mvp 写的 grpc 封装，不需要定义接口文件。

4、nanofabric https://github.com/geffzhang/NanoFabric 简单分析

5、Surging https://github.com/dotnetcore/surging

6、service fabric https://azure.microsoft.com/zh-cn/documentation/learning-paths/service-fabric/

7、Spring Cloud 入门视频：http://www.rupeng.com/Courses/Chapter/755

8、steeltoe http://steeltoe.io/ 参考文章 https://mp.weixin.qq.com/s/g9w-qgT2YHyDX8OE5q-OHQ

9、限流算法 https://mp.weixin.qq.com/s/bck0Q2lDj_J9pLhFEhqm9w

10、https://github.com/PolicyServer/PolicyServer.Local 认证 + 授权 是两个服务，identityserver 解决了认证 ，PolicyServer 解决授权

11、Using Polly with HttpClient factory from ASPNET Core 2.1 https://github.com/App-vNext/Polly/wiki/Polly-and-HttpClientFactory

12、CSharpKit 微服务工具包 http://www.csharpkit.com/

13、如鹏网.Net 提高班 http://www.rupeng.com