NCC开源小组

# .Net core 分布式微服务引擎介绍

讲解：范亮

contents

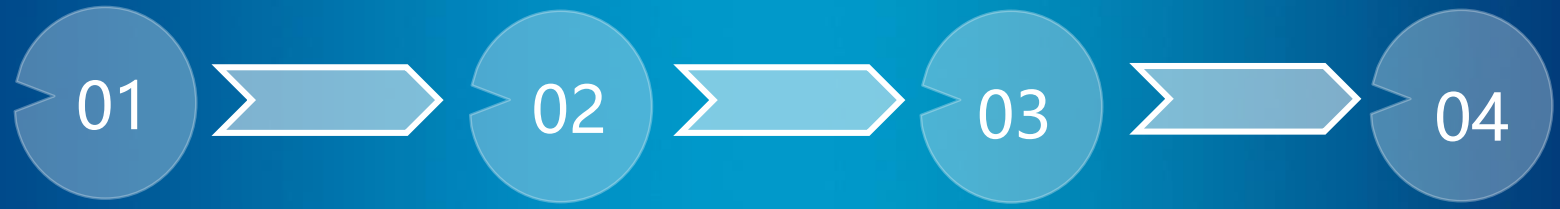
# 目录



# 入门篇

架构的演变过程
微服务如何设计

# 技术架构的演变




单体应用架构

垂直应用架构

分布式服务架构

分布式微服务架构

# 单体应用架构

## 一、什么单体应用架构

一般通过引用将所有功能在同一程序实现中的应用， 我们通常称之为单体应用。

## 二、单体应用简介

在面临各种业务需求时，通常会把功能堆积到同一个单体应用中去。比如：常见的ERP、CRM等系统都以单体应用进行架构，单体应用业务流程往往在同一个进程内部完成处理，不需要进行分布式协作。

# 一、单体架构优点

1. 易于开发：IDE针对于单体应用的开发、部署、调试而设计的，所以利用IDE就能短时间开发出单体应用。
2. 易于测试：单体应用不需要依赖其它接口运行，安装部署完成后就可以开始测试，简化了测试的过程，节约测试的时间
3. 易于部署：只需把文件复制或者安装到指定文件下，就已经部署成功

# 一、单体架构缺点

1. 逻辑复杂、模块耦合、代码臃肿，修改难度大，版本迭代效率低下。
2. 系统启动慢，一个进程包含了所有的业务逻辑，涉及到的启动模块过多，导致系统的启动、重启时间周期过长
3. 系统错误隔离性差、可用性差，任何一个模块的错误均可能造成整个系统的宕机
4. 可伸缩性差；系统的扩容只能只对这个应用进行扩容，不能做到对某个功能点进行扩容
5. 线上问题修复周期长；任何一个线上问题修复需要对整个应用系统进行全面升级

# 垂直应用架构

## 一、什么垂直应用架构

前端界面和业务逻辑分层拆分成独立部署的应用就叫做垂直应用。

## 二、垂直应用简介

当访问量逐渐增大，单体应用已经不能满足需求，然后将应用分层拆分独立部署，以提升效率。比如 常见的eshop、app应用等系统都以垂直应用进行架构。前端和后端业务分层拆分在不同的进程服务器中。

# 一、垂直架构优点

1. 更高的可用性:该特点是在于后端服务提供者和前端调用者的松散耦合关系上得以发挥与体现。前端无须了解提供者的具休实现细节。

2. 更好的伸缩性:依靠业务服务设计、开发和部署等所采用的架构模型实现伸缩性。使得服务提供者可以互相彼此独立地进行调整，以满足新的服务需求。

3. 更易维护:当需求发生变化的时候，不需要修改提供业务服务的接口，只需要调整业务实现即可，整个应用系统也更容易被维护

# 一、垂直架构缺点

1. 复杂应用开发的维护成本很高，部署效率低。
2. 团队协作效率差，功能重复开发。
3. 可靠性差，容易引起雪崩效应
4. 维护困难，随着功能越来越多，无法针对功能进行服务拆分，修改会造成其它功能性的错误

# 分布式服务架构

## 一、什么分布式服务架构

分布式服务架构是一种SOA粗粒度的单体应用，每个服务独立部署，并使用第三方服务治理中间件进行组装服务。

## 二、分布式服务架构简介

当访问压力越来越大，垂直应用已经不能满足需求，然后需要将服务分布式独立部署，以提升效率。比如 常见的大型的管理系统、电子商务平台都是分布式服务进行架构。通过把业务拆分成为多个服务部署在不同的服务器上。再通过第三方服务治理组件进行组合编排。

# 一、分布式服务架构优点

1. 高可用性，良好的伸缩性，更易维护，继承了垂直应用架构的优点
2. 团队能高效协作。各个团队互不干扰独自研发模块
3. 更好的可靠性，通过服务治理组件来保证服务的可靠运行

# 一、分布式服务架构缺点

1. 复杂应用开发的维护成本很高，部署效率低。
2. 维护困难，随着功能越来越多，无法针对功能进行服务拆分，修改会造成其它功能性的错误
3. 因依赖第三方服务治理组件，性能会有所降低
4. 底层耦合比较高，无法灵活配置

# 一、什么是分布式微服务服务架构

微服务架构是一种粒度更细小服务来开发单个应用，每个服务运行在自己的进程中，并使用TCP或HTTP进行通信，这些服务使用不同的编程语言实现， 采用网关集中式管理和外网访问。

# 二、分布式服务架构简介

1. 服务组件化：应用拆分服务运行在不同进程中，每个服务有明确的边界
2. 服务进程隔离：服务独立开发、编译、部署、测试、发布，有独立工程、独立版本、接口契约化
3. 去中心化：针对业务选择不同的编程语言，针对性的解决问题
4. 智能终端：业务逻辑在服务内部处理，服务之间通信使用轻量高性能通信机制
5. 更高容错能力：内部有一套完整的容错机制来进行熔断，以防止雪崩效应
6. 统一管理：通过网关统一访问，可以针对服务进行管理、数据监控、身份认证、流量控制、分流控制。

# 一、分布式服务架构缺点

1. 请求/响应：客户端向服务器端发起请求，同步等待响应，等待过程可能造成线程阻塞
2. 通知（也就是常说的单向请求）：客户端请求发送到服务端，服务端不返回请求响应
3. 请求/异步响应：客户端发送请求到服务端，服务端异步响应请求。客户端不会阻塞，而且被设计成默认响应不会立刻到达。
4. 发布/ 订阅模式：客户端发布通知消息，被零个或者多个订阅者服务消费。
4. 发布/异步响应模式：客户端发布请求消息，然后异步或者回调服务发回响应。

# 框架介绍

详细介绍surging

# 一、surging框架是什么

surging 是一个分布式微服务框架,提供高性能RPC远程服务调用，采用Zookeeper、Consul作为surging服务的注册中心，集成了哈希，随机，轮询、压力最小优先作为负载均衡的算法，RPC集成采用的是netty框架，采用异步传输..

# 二、能做什么

1.简化的服务调用，通过服务规则的指定，就可以做到服务之间的远程调用，无需其它方式的侵入

2.服务自动注册与发现，不需要配置服务提供方地址，注册中心基于ServiceId 或者RoutePath查询服务提供者的地址和元数据，并且能够平滑添加或删除服务提供者。

3.软负载均衡及容错机制，通过surging内部负载算法和容错规则的设定，从而达到内部调用的负载和容错

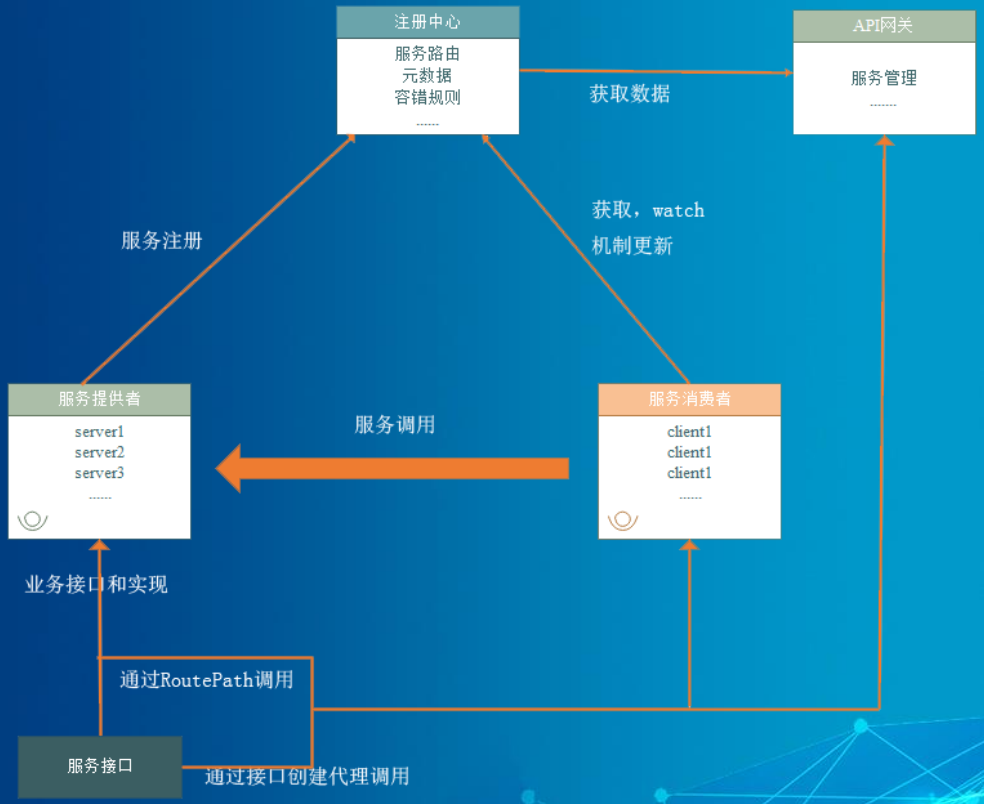
4.分布式缓存中间件：通过哈希一致性算法来实现负载，并且有健康检查能够平滑的把不健康的服务从列表中删除

5. 事件总线：通过对于事件总线的适配可以实现发布订阅交互模式

6.容器化持续集成与持续交付 ： 通过构建一体化Devops平台,实现项目的自动化构建、部署、测试和发布，从而提高生产环境的可靠性、稳定性、弹性和安全性。

7. 业务模块化驱动引擎，通过加载指定业务模块，能够更加灵活、高效的部署不同版本的业务功能模块

# 三、实现流程



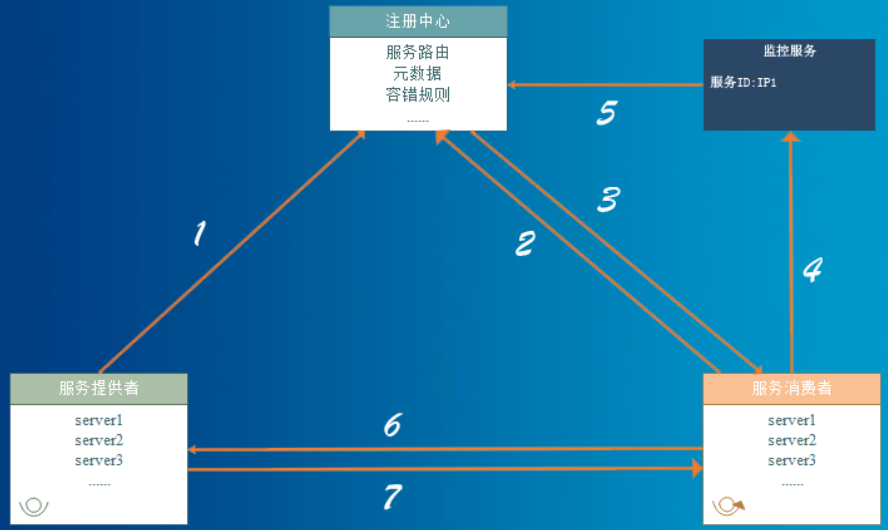

1 服务交互：
1.服务提供者和消费者基于接口契约进行交互
2.服务提供者和消费者基于RoutePath进行交互

2 服务提供者：
1.服务提供者实现接口提供业务应用服务
2.服务提供者启动时把服务注册到注册中心

3 服务消费者：
1.基于Watch或者心跳方式获取服务更新
2.服务调用获取业务应用服务

4 网关：
1.通过注册中心获取注册信息进行管理
2.提供统一入口访问

# 四、原理

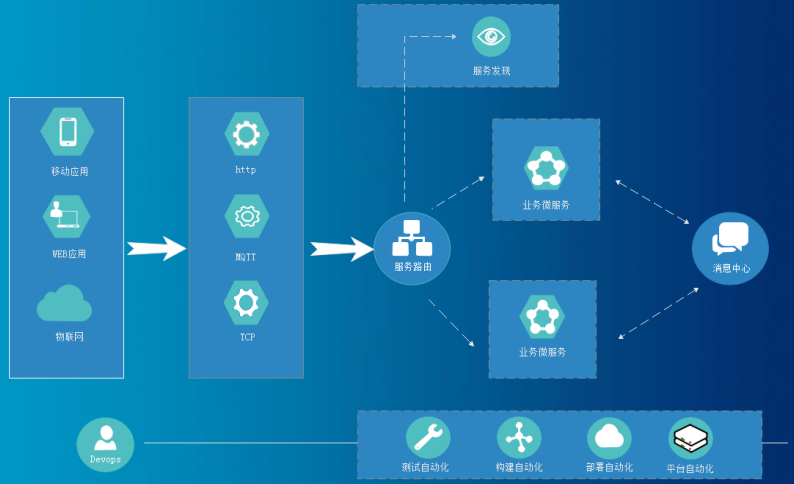



1. 服务提供者启动服务，然后通过向注册中心注册提供的服务和容错规则
2. 服务消费者通过向注册中心心跳获取服务和容错规则
3. 注册中心向服务消费者返回服务和容错规则存储到本地缓存
4. 服务者在调用过过程中通过负载算法定位服务地址放入到监控服务，并且返回健康的服务地址
5. 健康服务会针对地址列表进行健康检查，把多次不健康的服务从注册中心移除
6. 服务消费者通过设置的负载算法（哈希、轮询、随机、压力最小优先）来定位服务地址，从而通过服务地址去远程调用服务提供者
7. 服务消费者在调用过程中，如果成功会从服务提供者返回数据，如果失败会通过容错规则的设定而发生熔断，从而保证整个调用的稳定性。

# Surging框架介绍

## 五、架构

1. 通过surging支持的http、mqtt,tcp可以支持移动、web、物联网的消息调用和推送
2. 服务路由通过服务发现获取可用的服务地址进行访问，而业务微服务之间可以通过本地或者RPC远程调用
3. 消息中心通过rabbitmq 或者kafka进行消息推送
4. 构建DevOps



# 六、组件

1. 注册中心：支持consul 或者zookeeper K/V格式存储
2. 底层通信：支持dotnetty ,它是异步事件驱动框架,提供了对TCP、UDP和Http 支持
3. 序列化：支持messagepack, protobuffer, newtonsoft
4. 日志：支持nlog、log4net， 通过指定日志级别输出不同的信息
5. EventBus：支持rabbitmq、kafka 发布订阅
6. 缓存：支持redis、memorycache

# 七、负载均衡

1. 随机（Random）：通过生成随机数随机选择服务地址，调用量越大分布越均匀
2. 轮询（Polling）：通过轮询地址选择服务地址，存在比较慢的机器容易在这台机器的请求阻塞较多，默认使用此负载算法
3. 哈希（HashAlgorithm），通过生成的哈希值取模选择服务地址，对于相同参数的请求会定义到同一个服务提供者上。
4. 压力最小优先（FairPolling），通过轮询优先选择压力最小的服务地址，针对于压力比较小的服务器会分配更多的请求

# 八、容错策略

1. 故障转移策略（Failover)：通过设置故障转移群集数（FailoverCluster），从而服务故障自动转移到健康的服务提供者
2. 脚本注入策略（Injection）：通过设置脚本注入（Injection），服务发生错误时会返回所定义运行的脚本结果
3. 回退策略（FallBack），通过设置回退的实例名（FallBackName），服务发生错误时通过FallBackName 去调用依赖注入的接口IFallbackInvoker

# 九、服务熔断

1. 错误率熔断：通过设置错误率（BreakeErrorThresholdPercentage），当失败调用数/远程调用数大于错误率，会启用熔断
2. 超时熔断：通过设置执行超时时间（ ExecutionTimeoutInMilliseconds），当服务调用超过执行时间会启用熔断。
3. 并发熔断（FallBack），通过设置信号量最大并发度（MaxConcurrentRequests），在多线程环境下超过设置的信号量，会启用熔断
4. 错误数熔断（FallBack），通过设置调用失败错误数（BreakerRequestVolumeThreshold），在10秒钟范围内超过设置的调用失败错误数，会启用熔断

## 如何功能性的调用

# 十、开发

1. 针对于业务开发需要采用领域驱动设计，需要按照规则接口继承IServiceKey，领域服务继承ProxyServiceBase，仓储继承BaseRepository
2. 在一个接口多个实例情况下，需要针对于领域服务标识ModuleName特性
3. 通过接口标识ServiceBundle特性，用来标识可以注册为服务
4. 通过RpcContext隐性传参，可以向客户端向服务器端传递参数

## 如何功能性的调用

# 十一、开发

1. 针对于业务开发需要采用领域驱动设计，需要按照规则接口继承IServiceKey，领域服务继承ProxyServiceBase，仓储继承BaseRepository
2. 在一个接口多个实例情况下，需要针对于领域服务标识ModuleName特性
3. 通过接口标识ServiceBundle特性，用来标识可以注册为服务
4. 通过RpcContext隐性传参，可以向客户端向服务器端传递参数

# 十二、调用

1.基于接口创建代理进行调用

```
ServiceLocator.GetService<IServiceProxyFactory>().CreateProxy<T>()
```

2.基于RoutePath远程调用

```
ServiceLocator.GetService<IServiceProxyProvider>().Invoke<object>(new Dictionary<string, object>(),
"api/user/GetDictionary", "User").Result;
```

3.服务与服务之间的调用,使用继承的抽象类ProxyServiceBase中的GetService方法

```
This.GetService<T>()
```

# 十二、配置

1. 通过针对于配置ServiceHostBuilder来集成需要的组件
2. 通过ServiceHostBuilder 的Configure方法来加载配置文件
3. 通过配置规则“${环境变量名}|默认名”用来设置环境变量

# 服务部署

如何部署服务

# 简单实例

2018 NCC开源小组

# 感谢观看 THANK YOU

讲解人：范亮