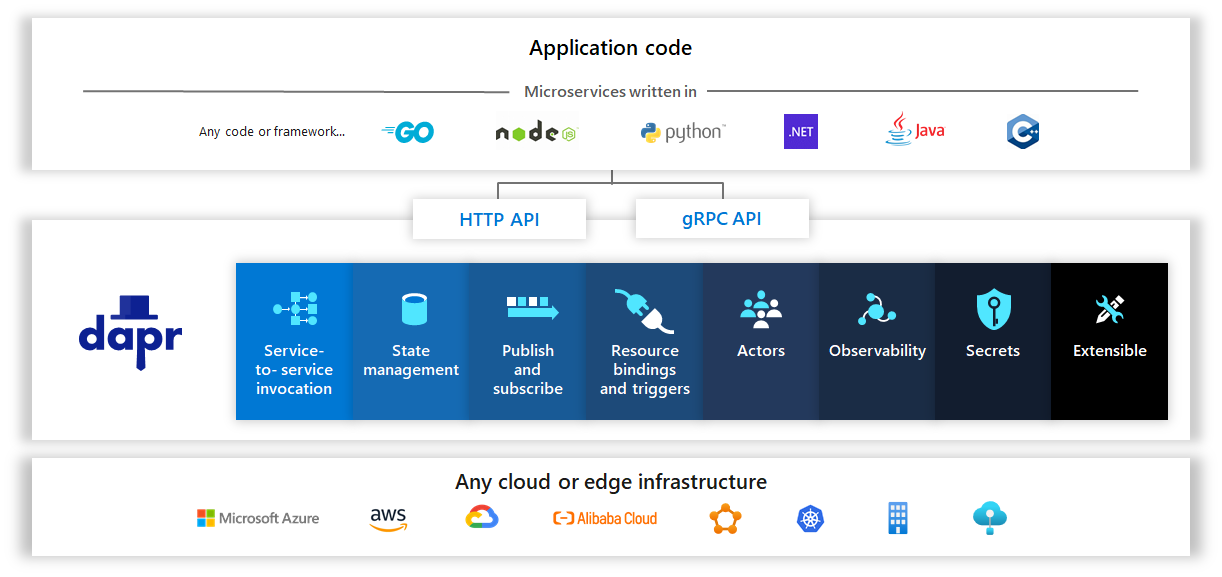
.net 5与dapr

**简介：** 分布式应用运行时Dapr目前已经发布了1.3.0版本，阿里云也在积极地为Dapr贡献代码和落地实践。

## **1 关于Dapr**

Dapr（**D**istributed **Ap**plication **R**untime）是一个开源、可移植、事件驱动的运行时。它使开发人员能够轻松地构建运行在云平台和边缘的弹性而微服务化的应用程序，无论是无状态还是有状态。Dapr 让开发人员能够专注于编写业务逻辑，而不是解决分布式系统的挑战，从而显著提高生产力并减少开发时间。此外，Dapr 也降低了大部分中小型企业基于微服务架构构建现代云原生应用的准入门槛。

Dapr 的核心构建模块 （或者说核心功能）如下：

* **服务调用：** 弹性服务与服务之间（service-to-service）调用可以在远程服务上启用方法调用，包括重试，无论远程服务在受支持的托管环境中运行在何处。
* **状态管理：**通过对键 / 值对的状态管理，可以很容易编写长时间运行、高可用性的有状态服务，以及同一个应用中的无状态服务。状态存储是可插入的，并且可以包括 Azure Cosmos 或 Redis，以及组件路线图上的其他组件，如 AWS DynamoDB 等。
* **在服务之间发布和订阅消息（Pub/Sub）：**使事件驱动的架构能够简化水平可扩展性，并使其具备故障恢复能力。
* **事件驱动的资源绑定：**资源绑定和触发器在事件驱动的架构上进一步构建，通过从任何外部资源（如数据库、队列、文件系统、blob 存储、webhooks 等）接收和发送事件，从而实现可扩展性和弹性。例如，你的代码可以由 Azure EventHub 服务上的消息触发，并将数据写入 Azure CosmosDB。
* **虚拟角色：**无状态和有状态对象的模式，通过方法和状态封装使并发变得简单。Dapr 在其虚拟角色（Virtual Actors）运行时提供了许多功能，包括并发、状态、角色激活 / 停用的生命周期管理以及用于唤醒角色的计时器和提醒。
* **服务之间的分布式跟踪：**使用 W3C 跟踪上下文（W3C Trace Context）标准，轻松诊断和观察生产中的服务间调用，并将事件推送到跟踪和监视系统。

## **2 准备工作**

## **一台Linux 主机**

为了后面的DEMO，在VMware Workstation中准备一个Linux虚拟机环境，这里我选择的是CentOS 7.6。

在此虚拟机中设定静态IP地址（本示例为 192.168.2.100），关闭防火墙，设定主机名等一系列基本操作。

## **安装.NET 5 SDK**

这里我的DEMO是基于local-host部署模式（也可以选择Kubernetes模式部署，但我没时间弄），因此给Linux安装一下.NET 5 SDK，命令如下：

添加受信源

sudo rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.rpm

安装.NET 5 SDK

sudo yum install dotnet-sdk-5.0

## **安装Dapr CLI**

官网提示直接在Linux下执行以下命令就可以将Dapr CLI下载到/usr/local/bin目录下：

wget -q https://raw.githubusercontent.com/dapr/cli/master/install/install.sh -O - | /bin/bash

通过输入 dapr 来验证是否安装成功：

此外，也可以通过 dapr --version 查看Dapr版本：

**初始化Dapr**

安装好Dapr CLI之后，就可以在Linux上初始化Dapr了，命令如下：

dapr init

这个命令会帮你做一些列的事情，包括但不限于 拉取一波docker镜像 & 运行一波docker容器，如下图所示：

可以看到，dapr, redis, zipkin都已经运行起来了。

为什么有redis？因为它会作为默认的pub/sub中间件为dapr提供具体的实现能力。

为什么会有zipkin？因为它会作为默认的tracing中间件为我们提供链路追踪的能力。

OK，到此为止，本地的Dapr运行时基础环境已基本就绪。

## **3 .NET 5 应用集成Dapr SDK**

## **准备三个.NET WebAPI**

这里我们准备了三个WebAPI项目，分别是订单服务、购物车服务 以及 商品服务。

## **为所有WebAPI项目添加集成**

为所有项目添加Dapr SDK的nuget包，这里是 Dapr.AspNetCore 组件。

## **为所有WebAPI项目注册Dapr**

在StartUp类中，对Dapr Client进行注册，这里的AddDapr背后的操作其实就是给IoC容器注入了一个单例的DaprClient对象。

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers()

.AddDapr();

......

}

## **4 服务调用示例**

这里假设CartService要和ProductService进行通信，通过REST获取商品数据。这里，就可以借助Dapr提供的服务间调用的功能进行通信。其工作原理如下图所示：

这里使用的方式是通过DaprClient直接InvokeMethod进行服务间的通信，传递了两个重要的参数，一个是依赖服务的app-id（根据你部署时设定的名字来写），另一个是依赖接口的route。

具体如下代码所示：

[ApiController]

[Route("[controller]")]

public class CartController : ControllerBase

{

private readonly ILogger<CartController> \_logger;

private readonly DaprClient \_daprClient;

public CartController(ILogger<CartController> logger, DaprClient daprClient)

{

\_logger = logger;

\_daprClient = daprClient;

}

[HttpGet]

public async Task<IEnumerable<SKU>> Get()

{

\_logger.LogInformation("[Begin] Query product data from Product Service");

var products = await \_daprClient.InvokeMethodAsync<IEnumerable<SKU>>

(HttpMethod.Get, "ProductService", "Product");

\_logger.LogInformation($"[End] Query product data from Product Service, data : {products.ToArray().ToString()}");

return products;

}

}

这里对应ProductService的接口默认返回一些假数据：

[ApiController]

[Route("[controller]")]

public class ProductController : ControllerBase

{

private static readonly string[] FakeProducts = new[]

{

"SKU1", "SKU2", "SKU3", "SKU4", "SKU5", "SKU6", "SKU7", "SKU8", "SKU9", "SKU10"

};

......

[HttpGet]

public IEnumerable<SKU> Get()

{

\_logger.LogInformation("[Begin] Query product data.");

var rng = new Random();

var result = Enumerable.Range(1, 5).Select(index => new SKU

{

Date = DateTime.Now.AddDays(index),

Index = rng.Next(1, 100),

Summary = FakeProducts[rng.Next(FakeProducts.Length)]

})

.ToArray();

\_logger.LogInformation("[End] Query product data.");

return result;

}

}

然后，将这两个服务发布到Linux服务器上，当然，我们要通过dapr来部署，让.net application和dapr sidecar形成一体。

部署命令如下所示，可以看到我们既要为.net application指定端口，也要为dapr sidecar指定端口（这里主要为dapr指定了http端口，也可以为其指定grpc端口）。

dapr run --app-id CartService --app-port 5000 --dapr-http-port 5005 -- dotnet EDT.EMall.Cart.API.dll --urls "http://\*:5000"

dapr run --app-id ProductService --app-port 5010 --dapr-http-port 5015 -- dotnet EDT.EMall.Product.API.dll --urls "http://\*:5010"

你会发现，当你run成功之后，会看到以下log，其中既有dapr的log，也有.net application的log，虽然他们是两个应用程序，但是你看到的它们是一体的。

最后，通过swagger来测试一下，结果如下，成功进行了服务调用。

## **5 消息发布及订阅示例**

发布订阅模式（Publish-Subscribe）是众所周知且广泛使用的消息模式。这里我们假设OrderService的某个接口完成后就发布一个消息，告知订阅方有新订单的事件产生。

具体代码示例如下，借助DaprClient的PublishEvent接口实现消息发布：

[ApiController]

[Route("[controller]")]

public class OrderController : ControllerBase

{

private const string DaprPubSubName = "pubsub";

private readonly ILogger<OrderController> \_logger;

private readonly DaprClient \_daprClient;

public OrderController(ILogger<OrderController> logger, DaprClient daprClient)

{

\_logger = logger;

\_daprClient = daprClient;

}

[HttpPost]

public async Task<Models.Order> Post(OrderDto orderDto)

{

\_logger.LogInformation("[Begin] Create Order.");

var order = new Models.Order()

{

// some mapping

Id = orderDto.Id,

ProductId = orderDto.ProductId,

Count = orderDto.Count

};

// some other logic for order

var orderStockDto = new OrderStockDto()

{

ProductId = orderDto.ProductId,

Count = orderDto.Count

};

await \_daprClient.PublishEventAsync(DaprPubSubName, "neworder", orderStockDto);

\_logger.LogInformation($"[End] Create Order Finished. Id : {orderStockDto.ProductId}, Count : {orderStockDto.Count}");

return order;

}

}

假设ProductService作为订阅方，需要消费这个事件，并扣减某个商品的库存。而基于Dapr，我们需要对ProductService添加一点配置：

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

......

app.UseCloudEvents(); // 标准化的消息传递格式

app.UseEndpoints(endpoints =>

{

endpoints.MapSubscribeHandler(); // 订阅消费处理

......

});

}

然后，在ProductService中添加一个方法/接口 来作为订阅处理。

具体代码示例如下，需要注意的就是：

（1）作为消息处理接口，需要指定为HttpPost方式。

（2）需要指定Topic特性，并标注pubsubname 和 事件名。

private const string DaprPubSubName = "pubsub";

[HttpPost]

[Topic(DaprPubSubName, "neworder")]

public Models.Product SubProductStock(OrderStockDto orderStockDto)

{

\_logger.LogInformation($"[Begin] Sub Product Stock, Stock Need : {orderStockDto.Count}.");

var product = \_productService.GetProductById(orderStockDto.ProductId);

if (orderStockDto.Count < 0 || orderStockDto.Count > product.Stock)

{

throw new InvalidOperationException("Invalid Product Count!");

}

product.Stock = product.Stock - orderStockDto.Count;

\_productService.SaveProduct(product);

\_logger.LogInformation($"[End] Sub Product Stock Finished, Stock Now : {product.Stock}.");

return product;

}

这里的DaprPubSubName是pubsub，这是因为Dapr默认的pubsub实现是基于Redis的，而在配置中为Redis设置的name就是 pubsub，因此对于我们入门的话，就不要去更改，或者和配置中的name保持一致。

[root@dapr-lab-server ~]# cat ~/.dapr/components/pubsub.yaml

apiVersion: dapr.io/v1alpha1

kind: Component

metadata:

name: pubsub

spec:

type: pubsub.redis

metadata:

- name: redisHost

value: localhost:6379

- name: redisPassword

value: ""

当然，我们也可以将默认的pubsub实现Redis换为熟悉的RabbitMQ。我们只需要更改上面的yml文件内容如下：

apiVersion: dapr.io/v1alpha1

kind: Component

metadata:

name: pubsub-rq

spec:

type: pubsub.rabbitmq

version: v1

metadata:

- name: host

value: "amqp://localhost:5672"

- name: durable

value: true

然后，将这两个服务发布到Linux服务器上，当然，我们要通过dapr来部署，让.net application和dapr sidecar形成一体。

dapr run --app-id OrderService --app-port 5020 --dapr-http-port 5025 -- dotnet OrderService.dll --urls "http://\*:5020"

dapr run --app-id ProductService --app-port 5010 --dapr-http-port 5015 -- dotnet ProductService.dll --urls "http://\*:5010"

run成功后，通过 dapr list 查看，可以看到三个服务都已经启动起来了，它们是三个由.net application + dapr sidecar 组成的“**合体应用**”。

最后，我们通过swagger来测试一下

这里的99其实是假总库存100 - 消息传递过来的商品数量得到的，具体可以参考代码示例。

## **6 小结**

本文总结了我试玩Dapr的一些经过，包括Dapr的Local环境搭建、.NET 5 Application与Dapr的集成 和 两个具体场景的小DEMO（服务调用 和 Pub/Sub）。

Dapr 本身是一种 Sidecar 模式（虽然Dapr也提供了SDK，但是个人认为这并不是Dapr以后的发展方向）。Sidecar 模式的意义在于， **解耦了基础设施和核心业务**。

简单来看，Dapr的意义在于：

* 对于小公司，甚至没有基础架构和中间件团队的公司，**Dapr 提供了开箱即用的基础设施功能，可以让小公司轻松构建弹性，分布式应用**。
* 对于中等单位，具备一定的基础架构能力，在使用Dapr的过程中，可能Dapr并不能完全满足需求，那么也**可以在Dapr框架体系下，花费较小的成本进行自定义扩展**。
* 对于大公司，**Dapr 提供了一种思路。相信基础架构团队会越来越倾向于通过交付Sidecar的形式来提供基础设施**。

长远来看，**Dapr背后的架构模式是符合未来架构趋势（多运行时架构）和云原生发展趋势的**。

dapr本地测试服多个docker镜像服务部署

1. 在对应的应用服务下面创建dockerfile 文件
2. 在对应的应用服务下面安装 Install-Package Dapr.AspNetCore 并注入到.net core services.AddControllers().AddDapr();服务中
3. 在解决方案中编写docker-compose编排/编译多个服务的镜像;
4. 写完docker-compose.yml之后在命令行执行 docker-compose up运行并编译镜像

需要注意事项

1.本地测试服多个镜像一定要用docker-compose来编排管理下, 否则镜像是互相隔离的。物理机器上的dapr运行时是发现不了镜像内dapr服务的.

2.本地测试服部署，一定要指定服务所属docker 镜像网络。否则服务之间无法正常通信. 如果是内网无需指定镜像端口。直接默认dockerfile里面编写的端口。如果这个服务需要对外开放用postman连接请求那么就需要指定开放的端口.

3.如果不指定 gprc或http api端口。dapr会默认分配端口.

4. 如果遇见无法连接到服务。则需要一步步排查是否镜像是在同一docker 网络内。是否dapr-app-port 端口正常,是否连接的服务id正确.是否docker 容器启动正常. 是否配置的dapr所需组件正确。查看docker 容器日志查看错误排查.

5. 本地开发时候无需docker 镜像。Docker 镜像只有部署到测试服或正式服的时候才需要编写dockerfile与docker-compose, 本地开发参考上一章.

docker与dapr一些常用命令

如何安装dapr请查阅官网, docker 同理

dapr init 初始化本地docker 环境的运行时与组件

dapr init --slim 初始化本地无需docker环境的运行时

dapr --version 查看dapr版本与运行时

dapr run --app-id OrderService --app-port 5020 --dapr-http-port 5025 -- dotnet OrderService.dll --urls "http://\*:5022

上面这串命令代表在本地运行一个dapr 服务; --appid 代表服务唯一标识号; --app-port 代表本地.net core 应用的端口号 --dapr-http-port代表服务对dapr的http 端口号。不填dapr默认给一个. -- dotnet OrderService.dll --urls “http://\*:5022”代表.net core应用启动指定端口号;

dapr uninstall -all 卸载本机所有dapr运行时

docker network create my-dapr-network 代表创建一个 名字叫my-dapr-network的docker网络

docker network ls 代表正在运行中的网络

docker images 已经编译的镜像

docker ps -a 运行中的所有容器

docker rm 容器id 删除指定容器

docker container stop 停止容器

docker container restart 重启容器

docker rmi -f 强制删除镜像

docker build ./DockerFile -t test/gejia 编译为指定的镜像名

docker run --name test -80:80 -d test/gejia 运行指定的镜像为容器. 并对外放开80端口

docker exec -it test bash 进入指定容器内部

docker-compose up 启动docker-compose 配置的容器镜像

docker-compose top 显示正在运行的镜像

docker-compose ps 显示所有正在运行的容器

docker-compose stop 停止容器

docker-compose down 停止docker-compose配置的容器并移除

docker-compose logs 查看容器日志

docker-compose build 编译 ,选项–compress 通过gzip压缩构建上下环境

–force-rm 删除构建过程中的临时容器

–no-cache 构建镜像过程中不使用缓存

–pull 始终尝试通过拉取操作来获取更新版本的镜像-m, –memory MEM为构建的容器设置内存大小

–build-arg key=val为服务设置build-time变量