搬运工服务端手册

# 引言

## 概述

搬运工服务端是搬运工平台的中后端，全面采用基于.NET Core的云原生应用开发，所有应用服务使用Kubernetes 进行调度，面向Kubernetes 编程，采用DevOps模式进行研发和运维。

## 名词解释

* Dt：Data Transfer的缩写，搬运工平台的简称，“我们是数据的搬运工”。
* Docker：一个开源的容器，里面装有一系列文件系统、网络、依赖包等应用程序运行需要的环境，属于操作系统层面的虚拟化技术。
* Kubernetes：简称K8s，用于管理云平台中多个主机上的容器化的应用，提供了应用部署，规划，更新，维护的一种机制。
* DevOps：是一组完整的实践，可以自动化软件开发和 IT 团队之间的流程，以便他们可以更快、更可靠地构建、测试和发布软件。
* 云原生：符合以下十二原则的应用开发。
* DDD：Domain-Driven Design，领域驱动设计，统一分析、设计、编程，使软件能灵活快速跟随需求变化。
* RPC：Remote Procedure Call，远程过程调用。



# 基础功能

## 服务配置





系统配置包括全局配置和服务配置两部分，所有配置文件的存储路径为/etc/config /。文件global.json中存储全局配置，全局配置包括应用名称、所有涉及的数据库连接串、rabbitmq配置、redis配置等，是系统内所有微服务需要的公共配置。文件service.json中存储当前服务的配置，内容包括默认数据库、各种控制参数等。logger.json是单独提供给日志的配置文件。系统配置未包含环境变量。

当服务部署在K8s时，路径/etc/config /下的json配置文件作为集群的configMap资源，以Volume的方式加载到Pod中，最后释放到服务的app/etc/config目录下，支持动态更新配置文件，响应时间10多秒。

获取配置的统一方法：Glb.GetCfg (“key”, defaultVal)，可以获取两个配置文件及环境变量中的值，定义如下

/// <summary>

/// 获取系统配置中指定键的值

/// </summary>

/// <typeparam name="T">值的类型</typeparam>

/// <param name="p\_key">键名</param>

/// <param name="p\_defaultValue">键不存在时的默认值</param>

/// <returns></returns>

public static T GetCfg<T>(string p\_key, T p\_defaultValue)

如global.json：



service.json：



## 服务日志

### 内置日志

微软扩展库Microsoft.Extensions.Logging提供内置的日志支持，其工作过程为：内部创建ILoggingBuilder实例，管理外部传入的容器IServiceCollection，以单实例方式向容器中注入要输出的日志类型提供者ILoggerProvider，如ConsoleLoggerProvider、DebugLoggerProvider；注入单实例ILoggerFactory，内部实现类型为LoggerFactory，当容器Build()后，日志输出只需调用单实例ILoggerFactory进行。

默认类型LoggerFactory的实现方式为：内部维护provider列表和logger字典(键为类名)，每增加一个provider，在logger字典中所有logger的LogInformation[]数组中增加LogInformation，其结构如下：



日志输出过程为：遍历所有LogInformation，对IsEnable的调用logger.Log(...)。

### Serilog

系统采用第三方开源日志Serilog，Serilog中将日志分为六级：Verbose(冗余详细级)、Debug(内部调试级)、Information(普通信息级)、Warning(警告级)、Error(错误级)、Fatal(崩溃级)，以上为递增顺序，可以通过设置最小级别控制输出内容。系统使用logger.json文件作为外部配置，支持运行时动态修改文件。

通常使用内置的静态类Log输出日志，根据级别的不同提供对应方法，如：Log.Information(...), Log.Warning(...), Log.Error(...)。支持外部接收器(sink)处理日志事件，可以把日志输出到Console,Debug,File,Db，日志事件的内容记录在LogEvent对象，包含：事件发生时的时间戳[timestamp]、描述何时应该捕获事件的级别[level]、记录事件的消息[message]内容]、描述事件的命名属性[properties]、还可能有一个Exception对象，本质上日志事件由消息模板+属性字典组成。

日志中含动态属性时尽可能使用消息模板的方式，内部会对消息模板hash处理，节省资源、方便过滤等，如：

推荐 Log.Information("The time is {Time}", DateTime.Now);

不推荐 Log.Information("The time is " + DateTime.Now);

日志附加属性：给特定类型日志事件附加属性，用来识别相关事件集合，按照作用域分为两类：全局附加属性、上下文附加属性。

全局附加属性：

Log.Logger = new LoggerConfiguration()

.Enrich.WithProperty("Application", "Demo")

.WriteTo.Console()

.CreateLogger();

通过Enrich添加的属性会自动附加在所有日志事件中，与源自消息模板的属性相同。

上下文标记：

var orderLog = Log.ForContext("OrderId", order.Id);

orderLog.Information("Looking up product codes");

// ...

orderLog.Information("Product lookup took {Elapsed} ms", elapsed.TotalMilliseconds);

这样通过orderLog输出的日志都包含OrderId属性，还有其它方式实现上下文标记：

ILogger \_log = Log.ForContext<HomeController>()

日志事件会附加SourceContext= MyApp.Controllers.HomeController属性

Log.Logger = new LoggerConfiguraition()

.Enrich.FromLogContext()

// ...

using (LogContext.PushProperty("MessageId", message.Id))

{

Log.Debug("Dispatching message of type {MessageType}", message.GetType());

await handler.HandleAsync(message);

}

.Enrich.FromLogContext()允许日志事件从LogContext继承属性， 这样using内的日志事件都会附加MessageId属性。

初始化日志：



在Api方法内部记录日志时请使用\_.Log对象，输出时自动添加Api方法名等附加属性，方便bug位置的跟踪，直接使用静态类Log时无附加属性。

## EventBus

系统中包含两种类型的EventBus，一种是进程内事件LocalEventBus，一种是基于RabbitMQ实现的远程事件RemoteEventBus。

### LocalEventBus

其中LocalEventBus支持两种类型的事件：请求/响应模式的事件、通知模式的事件。请求/响应模式的事件需实现接口IRequest<out TResponse>，TResponse为响应时的返回值类型，事件处理类型需实现接口IRequestHandler<TRequest, TResponse>。通知模式的事件需实现普通接口IEvent，事件处理类型需实现接口ILocalHandler<TEvent>。两种模式的事件定义和处理如下：



### RemoteEventBus

RemoteEventBus是微服务之间的事件总线。每个服务在启动时打开一个连接(Conntection)、三个通道(Channel)、两个消费者队列(Queue)、一个交换机(exchange)。三个通道其中一个用来发布事件，另两个提供给消费者队列，交换机名称使用当前应用的名称，只提供给第二队列绑定。

第一消费者队列名称为AppName.SvcName，采用work模式，未绑定交换机，只支持接收和队列名称完全匹配的投递，当有多个服务副本时采用均衡算法只投递给其中一个。

第二消费者队列名称为AppName.SvcName.SvcID，绑定交换机，路由规则采用topic，按正则表达式对RoutingKey匹配，提供两种匹配方式的RoutingKey，第一种 AppName.SvcName.\* 接收对服务所有副本的投递，第二种 #.SvcID 接收对当前副本的投递，两种匹配实现接收对所有副本的组播和指定副本的投递。因每次重启服务id不同，队列采用自动删除模式。

两个消费者队列满足以下常见的消息发布场景：

* 向应用内的所有服务的所有副本或单个副本进行投递，调用Broadcast方法
* 向应用内的多个服务的所有副本进行投递，调用Broadcast方法
* 向应用内的多个服务进行广播，当一个服务有多个副本时只投递给其中一个副本，调用Broadcast方法
* 向某个服务的所有副本进行组播，调用Multicast方法
* 只向某个服务发布事件，有多个服务副本时采用均衡算法将消息投递给其中一个，调用Push方法
* 只向某个服务的固定副本发布事件，使用场景少，调用PushFixed方法

方法定义如下：



定义事件



定义事件处理



LocalEventBus和RemoteEventBus都是单例模式，服务启动时已注入，业务开发过程中通过 \_.LocalEB 和 \_.RemoteEB获得实例。

## 缓存

为了更有效的利用机器资源，平台在服务端和客户端都使用了缓存，总体原则是榨取客户端、节约服务器，按照使用场景可分为服务端全局缓存、服务进程内缓存、客户端模型缓存、客户端用户数据缓存，全局缓存使用redis内存数据库，以key-value形式存储要缓存的内容，供所有服务使用，使用常见的过期策略；进程内缓存只在服务内部使用，在生成环境缓存的内容不经常变化，如Sql语句缓存就属于进程内缓存；客户端模型缓存使用sqlite文件型数据库，存储系统模型和静态(不常变化)信息，如表结构、系统模型数据、各种字典表等，在服务器端生成，下载到客户端使用；客户端用户数据缓存是将用户频繁使用但又很少变化的数据保存在状态库(state.db)，按照数据版本进行更新。

### 全局缓存

全局缓存使用redis内存数据库，所有服务都可访问，缓存内容以key-value形式存储，key只能为字符串类型，value的数据类型有5种：String、list、hash、set、sorted set。

系统以value数据类型作为区别，提供5个类实现不类型的相关操作：StringCache、ListCache、HashCache、SetCache、SortedSetCache，同时提供方便使用的静态类Cache，Cache内容使用以上类型实现，适合于单独一次读写操作，相关操作包括：



比如 new StringCache("Test:Str").Set(p\_key, p\_val) 在redis库中增加了key为Test:Str:p\_key，值为p\_val的缓存，new StringCache("Test:Long").Get<long>(p\_key) 获取key为Test:Long:p\_key的long类型的值，以上操作可以简写为：Cache.StringGet<long>("Test:Str", p\_key)。

实际业务开发过程中通常将以上5个类型内嵌到具体缓存类中实现，如：



### 进程内缓存

进程内缓存比较自由，常使用Dictionary或List<T>类型缓存数据，不需要网络通信，是性能最好的缓存方法，不常变动的业务数据也可以使用该类缓存，一般缓存过程为：查询缓存中是否有目标数据，存在时直接取出使用，不存在时查询数据库，将查询结果缓存到本地待下次使用。

### 模型缓存

合理利用模型缓存能大幅减轻服务端压力，平台原则是榨取客户端、节约服务器。

客户端启动时会根据模型文件版本号判断是否需要下载新版本文件，然后打开模型库，过程可参见《客户端手册》的启动过程一节，如下图所示，模型库是只读的，禁止程序中写数据。



模型文件是sqlite数据库格式的文件，在服务器端生成，文件中包含系统模型和静态(不常变化)信息，文件名作为版本号，存储在etc/model/目录下，用来供客户端识别模型文件是否一致，不相同时客户端下载最新版本。



cm服务提供系统默认的模型文件，每个服务也可以定义新的模型文件，按照模型文件的不同，客户端主要分为两类：管理端、内容端，管理端可以在cm模型的基础上增加些缓存，而内容端则可以完全自定义模型文件，如百岁管理端、百岁客户端、百岁介护师端分别属于以上两类，每种客户端通过AtApp.OpenModelDb提交不同的服务下载各自的模型文件。



任何服务通过以下过程都可以支持生成模型文件：

首先在Stub中注册并初始化SqliteModelHandler单例对象



然后在service.json中定义模型文件的表结构及数据，生成的模型文件中将包含配置的所有表及数据。



通过以上两步的注册和配置，服务初次启动时并未生成模型文件，可通过.admin页->系统管理->更新模型->测试方法 生成模型文件。





如上图所示，“更新模型”的内容包括：

* 创建新版本号
* 刷新服务端的表结构缓存
* 导出global.json中配置的所有数据库的表结构信息，导入OmColumn表，用于客户端生成sql
* 根据service.json中配置创建表结构、导入数据，若service.json中未提供任何模型文件的内容，则模型文件只包含OmColumn
* 生成sqlite文件并压缩成xxx.gz文件
* 刷新版本号，缓存xxx.gz文件等待下载

由此可见，在修改数据库表结构后或service.json配置的模型数据变化后都需要“更新模型”，cm服务的默认模型文件内容包括：

* OmOption基础选项，数据多，基本不修改，在客户端多用作下拉选择项；
* OmReport报表模板定义，模板定义内容长，修改频次低；
* OmMenu菜单定义，修改频次低，属基础模型；

这些基础数据在客户端修改后会提醒“更新模型”，可以批量修改后一次性“更新模型”，客户端和.admin页都提供“更新模型”功能。





为保证服务有多个副本时版本号相同，“更新模型”时采用远程事件的方式通知所有副本最新的版本号并各自重新生成模型文件。



当“更新模型”时，会通知当前服务的所有副本进行更新，最终由ModelRefreshHandler.cs完成所有功能。

### 用户数据缓存

客户端成功登录后除了返回当前用户的基本信息外，还返回当前用户各类缓存数据的版本号，用于和保存在状态库的各数据版本号比较，版本号不同的删除缓存，以便下次用到数据时进行更新。这类用户数据如：用户可访问的菜单、用户具有的权限、用户参数值等，这些数据使用频繁但又很少变化，缓存在客户端能大大减少不必要的查询，同时在数据变化时能自动更新，建议也缓存具有这些特点的业务数据，虽然程序上复杂些但大大减轻了服务端的压力。数据缓存及更新过程见下图：



## RPC

RPC（Remote Procedure Call）全称为远程过程调用，是客户端与服务之间、服务与服务之间进行网络通信的基础。服务之间可以通过消息总线RemoteEventBus实现远程事件的发布/订阅，但无法实现远程调用功能。服务之间Rpc和客户端与服务之间Rpc方法相同，都是基于http2协议、使用Post方法、传输json内容的方式调用服务端的Api，返回json格式的结果。未使用http3因底层基于udp，目前当UDP流量过大的时候基本上都会触发运营商随机丢包，这个问题连Google自己都没有解决。



### 未使用grpc

protobuf序列化复杂类型困难，且传输的类型复用到业务中效果不好，已有的Dict,Table等类型无法实现，从.proto原型生成代码对系统影响大且拦截、权限控制等功能无实现思路，并且iOS Android客户端对grpc暂不支持，故未使用grpc。grpc基于http2实现4种通信模式参见：<https://github.com/grpc/grpc-dotnet>

虽然未直接使用grpc，但底层的通信方法和通信模式都参考grpc，保证了灵活性和稳定性。

### 通信模式

参照grpc实现4种通信模式：



最常使用的为Unary模式，就是普通的请求/响应，Api调用基本使用这种模式。其余三种模式统称为流模式，ServerStream实现了服务器推送功能，ClientStream实现了上传功能，DuplexStream支持请求/响应双向流，所有模式都是基于http2协议建立的长连接，以帧(Frame)的方式发送接收，可以多路复用，网络传输的json内容长度默认超过4K时进行gzip压缩。

服务之间Rpc完全支持上述四种模式，客户端为UWP、Android、iOS、WebAssembly应用时，只支持Unary和ServerStream两种模式，因客户端HttpClient. SendAsync方法在发送请求时内部使用的WriteStream类型不同，无法实现双工通信，如下图。



客户端与服务端进行Rpc时，有三种方式可中断请求，一种是客户端HttpClient. SendAsync方法未结束时可通过CancellationToken取消请求；第二种是客户端收到HttpResponseMessage但Response未结束时(如ServerStream模式)可以Response.Dispose；第三种是服务端通过HttpContext.Abort()取消请求；切记切记不要使用第二种方式，因该方式会被服务端断开连接，其余两种方式只是结束请求并未断开连接！如客户端登录后会调用Msg服务的Register方法来注册服务器推送，该Rpc为ServerStream模式的长连接，当多个客户端通过同一连接连接到Msg服务时，任何一个客户端若采用第二种方式取消请求，会造成同一连接的所有客户端断开！所以ServerStream模式只能通过第三种方式取消请求！

### Rpc代理方法

系统提供客户端Rpc代理方法的自动生成，过程为：完成Api方法 -> 打开.admin页面 -> 点击要生成代理的类名 -> 复制生成的代理方法到客户端。如：

完成Api方法



打开.admin页面



点击要生成代理的类名



## Sql语句

服务中使用Sql语句的原则：不推荐在代码中硬编码的方式，而是将Sql语句统一管理，在生成环境中采用缓存成key-value的方式，客户端只使用key配合参数进行查询，既提高安全性、性能，也实现不同用户存在个性化查询的问题。Sql语句保存在当前服务默认库的xxx\_sql表中，xxx为服务名，开发时为调试方便可以将service.json中的"CacheSql"设置为false，这样Sql语句不采用缓存方式每次都去表中获取。

代码中通过key获取Sql的方法为：Glb.Sql(string p\_key)

调试时经常需要把Sql语句和Dict参数拼成完整Sql的场景，可以将service.json中的"TraceSql"设置为true，完整Sql语句会输出日志。

含有in的Sql语句比较特殊，可通过mysql的FIND\_IN\_SET函数替代以便使用参数，如：

select \* from cm\_pubfile where isfolder=1 and not FIND\_IN\_SET(id, @id)

@id为拼接的字符串，逗号隔开。

## MySql

Db为MySql数据库访问类，所有外部方法全部采用异步操作，基于开源项目 MySqlConnector 和 Dapper，MySqlConnector实现标准的Ado.net，高性能、原生异步I/O，Dapper提供精简ORM功能，主打高性能，性能接近Ado.net。

Db默认构造方法无参数，采用配置中的默认连接串、调用每个公共方法后自动关闭连接，可以自定义构造方法：

/// <summary>

/// 其他数据源连接串 + 设置是否自动关闭连接

/// </summary>

/// <param name="p\_dbKey">数据源键名，在json配置DbList节</param>

/// <param name="p\_autoClose">调用每个公共方法后是否自动关闭连接，false时切记最后手动关闭！</param>

public Db(string p\_dbKey, bool p\_autoClose = true)

方法中若多次调用Db方法时建议采用net Db(false)的方式构造，避免调用每个公共方法后自动关闭连接，但方法结束时需要手动关闭连接！

设计表时常用tinyint类型，不同长度对应不同的C#类型，tinyint(1)对应bool类型，只用来保存bool值0和1，不要保存其他值；如果要保存多个值，就用tinyint(4)，无符号对应byte类型，有符号对应SByte类型，无符号使用场景多，当为byte时，生成代理类时支持自动将#EnumName#开始的注释当作枚举类型，增加代码的可读性。

## 分布式ID生成方案

通常使用数据库自增长字段、序列、Guid等方法生成ID，对ID的理想要求是：全局唯一、趋势递增、自由生成、存储空间小，以上方法都不能满足，系统采取Twitter的Snowflake算法生成8字节的long类型ID，mysql中为bigint型无符号字段，实现以上理想要求。算法如下：



1位标识，long类型最高位是符号位，正数是0，负数是1，最高位始终为0。

41位时间戳(毫秒级)，存储时间差的值（当前时间戳 - 开始时间戳)，41位可以使用69年保证不重复，开始时间一般设置为生成器开始使用的时间，系统为2019-10-01。

10位工作机器id根据当前服务器IP生成，IPV6时将每个 Bit 位的后6位相加，IPV4时采用IP段数值相加生成唯一的workerId，满足 < 1024，重复可能性非常小，凑合用！

12位序列支持每个节点每毫秒(同一机器，同一时间戳)产生4096个序列号。

获取新ID方法：Id.New()

## Api授权控制

通过Rpc方式调用服务Api时，系统内部执行公共的授权验证，授权验证标签为AuthAttribute，可以标记在Api的类或方法上，按照就近原则，若方法上有Auth标签就按方法的Auth进行授权验证，否则按类的Auth标签进行授权验证，类和方法上都无Auth标签时不进行授权验证，都可访问。

提供两种格式的Auth标签，一种为 [Auth] 表示所有登录用户都可访问(只限制匿名用户)，一种为 [Auth(Type)] 表示外部自定义校验授权方法，Type需要实现ICustomAuth接口，可以按照需求自定义授权验证，如按权限判断、按用户所属分类等等。



## Table操作

和客户端功能相同，参见《搬运工客户端手册》

## 异常处理

和客户端功能相同，参见《搬运工客户端手册》

## 序列化

和客户端功能相同，参见《搬运工客户端手册》

## 依赖注入/控制反转

依赖注入DI(Dependency Injection)和控制反转IoC(Inversion of Control)是从不同角度描述的同一件事情，指通过引入IoC容器，利用依赖关系注入的方式，实现对象之间的解耦。

常用的IoC容器有Autofac、Castle.Windsor、Spring.Core、DependencyInjection(DI)，asp.net core原生使用DI，本系统采用Autofac容器以应对复杂的注册、注入，配合Castle.Core实现拦截。



## 面向切面编程AOP

通过预编译方式或运行期动态代理方式实现在不修改源代码的情况下给程序动态统一添加功能的一种技术，系统内部使用Castle.Core在运行时动态生成代理类的方式实现AOP编程。

首先如下图所示在需要拦截的Api类型上指定Interceptors，支持多个拦截器，再将需要拦截的方法指定为virtual。



通过Http Rpc方式调用Api时，内部以容器方式实例化Api对象，得到的实例为继承自Api类的代理类，如下图所示：



Api实际类型为TestIntercept，通过容器实例化得到的对象类型为TestInterceptProxy，该类型继承TestIntercept为动态生成的代理类，所有需要拦截的方法必须为virtual以便override，可以注册中指定多个拦截器(类型)执行不同的拦截内容，如上图。拦截器需要实现IInterceptor接口，参见下图：



建议：对于被拦截的virtual方法，当其内部调用其它被拦截的virtual方法时，只在最外层拦截一次；若外层方法为非virtual方法，内部调用的virtual方法仍然被拦截，实际使用时避免该情况；总体可理解为只拦截最先调用的virtual方法。

# 服务架构

架构模型非常重要，俗话说：架构设计不合规，累死开发和运维，设计原则数不清，高内聚低耦合最要命。大多的业务逻辑是无法通过推理得到的，有时甚至是被臆想出来的，所以业务逻辑是很没有逻辑的逻辑。而在具体编码实现时，除了应付业务上的复杂性，技术上的复杂性也不能忽略，比如要讲究技术上的分层，要遵循软件开发的基本原则，又要考虑到性能和安全等等，这样的结果使得原本已经很复杂的业务变得更加复杂而难以理解。在很多项目中，技术复杂度与业务复杂度相互交错纠缠不清，这种火上浇油的做法是很多软件项目无法继续维护的主要原因。然而，在合理的设计下，技术和业务是可以分离开来或者至少它们之间的耦合度是可以降低的。

## 常见服务架构

**1 领域驱动设计**

DDD(Domain Driven Design)领域驱动设计，作为一种软件建模方法，通过其自有的原则与套路来解决软件的复杂性问题，DDD设计的目的是使得业务能够从技术中分离并突显出来，让代码直接表达业务内容，其中包含了聚合根、领域服务、实体、值对象、仓库、工厂等概念。



**2 命令查询职责分离**

CQRS(Command Query Responsibility Segregation)命令查询职责分离，就是读写分离，读写分离的主要目的是为了提高查询性能，同时达到读、写解耦。而DDD和CQRS结合，可以分别对读和写建模。



**3 六边形架构**

六边形架构（又名端口适配器架构），它的核心理念是：应用是通过端口与外部进行交互的，这也是微服务架构下API网关盛行的主要原因。六边形架构中，内部业务逻辑（应用层和领域模型）与外部资源（APP，WEB应用以及数据库资源等）完全隔离，仅通过适配器进行交互。它解决了业务逻辑与用户界面的代码交错的主要问题，从而可以很好的实现前后端分离。



**4 洋葱架构**

洋葱架构（又名整洁架构），同心圆代表应用软件的不同部分，从里到外依次是领域模型、领域服务、应用服务、最外围是容易变化的内容，如界面和基础设施（如数据存储等）。整洁架构是以领域模型为中心，不是以数据为中心。



虽然整洁架构、六边形架构以及DDD分层架构三种架构模型展现方式以及解决问题的出发点不一样，但其架构思想与微服务架构高内聚低耦合的设计原则高度一致。



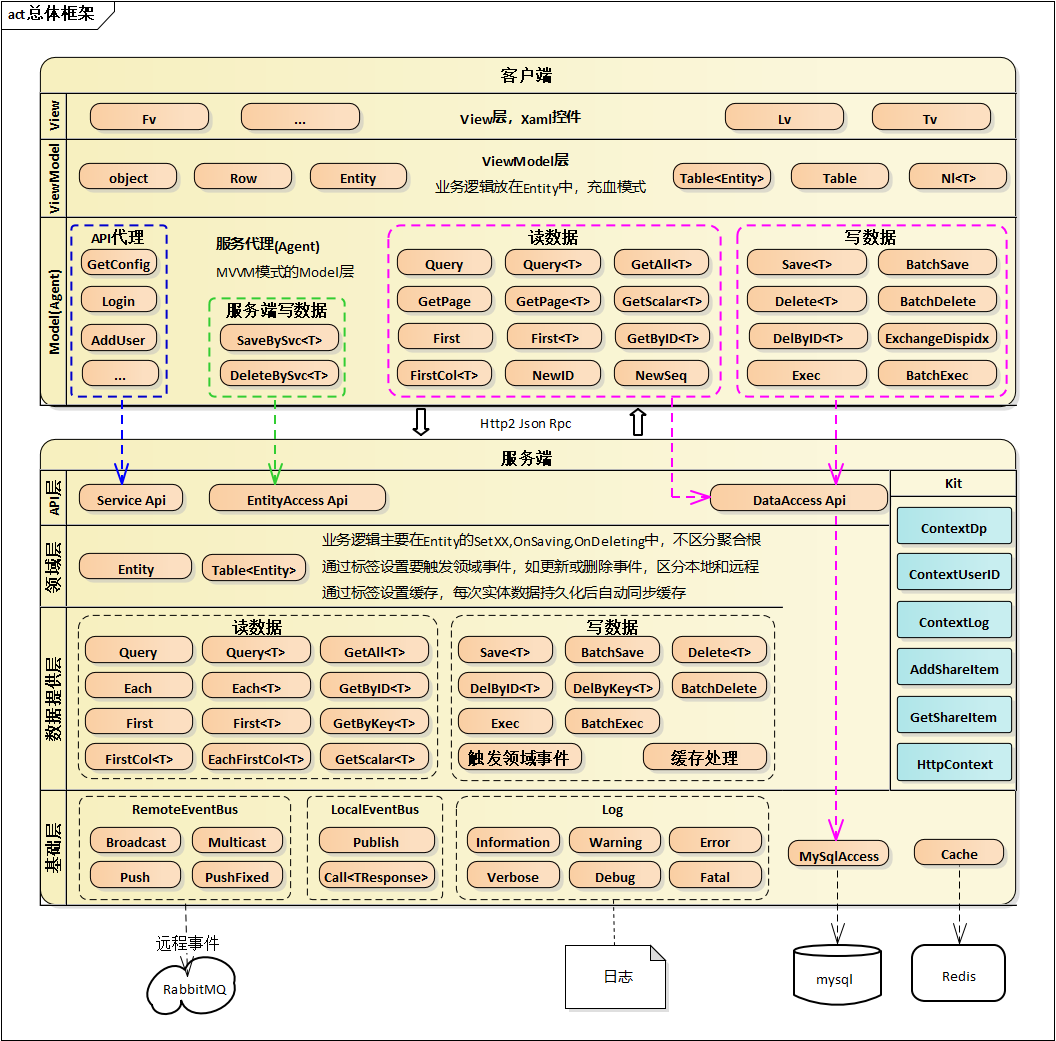
**5 中台、领域及微服务**

中台、领域以及微服务属于不同层面的内容，中台的本质是提炼各个业务线的共同需求，并将这些功能打造成组件化产品，然后以API接口的形式提供给前台各业务部门使用。领域的本质是问题域，问题域可能根据需要逐层细分，因此领域可分解为子域，子域或可继续分为子子域。微服务是技术实现和部署的范畴，实现领域或中台的业务逻辑，为前台应用提供服务。



## 总体架构

本系统未严格按照以上某种具体的服务架构进行设计，而是根据自身整体架构的特点，借鉴以上各架构的思想，比如DDD模式的领域事件、充血模式、数据仓库，CQRS架构的读写分离，六边形架构的微服务API设计等，结合客户端xaml的优势，使用MVVM(Model-View-ViewModel)模式开发客户端控件，完美实现技术与业务解耦、业务逻辑解耦、业务封装复用等关键功能，为开发人员提供一种易懂易用的框架风格。总体框架图：



目前平台只支持所有微服务连接同一RabbitMq、同一Redis，但每个微服务可以连接不同的数据库，详细配置见“3.1服务配置”。

### 客户端

客户端采用MVVM模式，属于富客户端，支持App和Web，可以看作单页模式的UI，常用的xaml控件Lv Fv Tv FileList等请参见《搬运工客户端手册》，这些控件都支持多种数据源类型，如Fv支持普通对象作为数据源，和Row、Entity数据源一样也实现了Fv.Changed事件、FvCell.Changed事件、淡黄色修改状态样式、支持撤销等功能；Lv Tv支持所有实现INotifyList接口的数据源，常用数据源类型有Table、Table<TEntity>、Nl<T>等。

用户通过与这些控件的交互修改了数据，如在Fv中修改了某格的选项，在Lv中删除了某行等，但控件不包含数据持久化的功能，实际上控件和数据源只是View层和ViewModel层，Model层的功能由服务代理实现，负责数据加载和数据持久化，是基本的数据提供者。

服务代理的主要工作是序列化请求、采用http2协议、Post方法、传输json内容的方式调用服务端的Api，返回json结果并将其转换成需要的类型提供给调用者，每个服务代理对应一个微服务，如AtCm和cm服务、AtMsg和msg服务，服务代理约定的命名规则为“At + 首字母大写的服务名”，At为Agent的缩写，每个Api在服务代理类中都对应一个原型相同的静态方法，该静态方法由所属服务自动生成：





由上图可见，生成的代码已自动标记服务名、Api、Api参数，同时带有注释内容，”Entry.GetConfig”是Api的完整名称，其中Entry在服务端是一个类，GetConfig是它的方法，每个微服务可以包含多个Api类，用于区分Api的不同功能范围，如上图中Entry负责登录相关，UserRoleApi负责用户角色相关。

以上生成的静态方法属于架构图中左侧蓝色框的Api，右侧红框的“读数据”和“写数据”是客户端的数据提供者，属于公共Api，即每个服务代理都提供的Api，为了使用方便，这部分Api实现的比较诡异，以AtCm为例看一下实现过程：

首先这部分Api作为静态方法放在泛型抽象类DataProvider<T>中：



AtCm继承DataProvider<T>，传递cm类型作为泛型参数：



这样在调用AtCm.Query时就能提交给cm服务，AtMsg.Query时就能提交给msg服务，使用非常方便。

这些公共Api是客户端的数据提供者，它屏蔽了sql生成、序列化反序列化、网络通信等细节，为客户端提供数据加载和持久化功能。参见架构图右侧的红箭头，它们在服务端调用的是DataAccess Api，直接数据访问，未涉及业务逻辑、领域事件、缓存等内容，所以这部分的业务逻辑只能放在客户端处理，对于复杂的业务尽可能放在实体Entity中处理，也就是DDD推荐的充血模式，可参见Dt.App中的流程设计部分，这种模式适用于大部分场景，服务端的数据访问对于前端完全透明，类似传统的直连数据库方式，系统主要功能都在客户端完成，这些公共Api按功能又可分为读数据和写数据，详细内容在后面陈述。

总体而言，客户端能处理大部分业务功能，对于涉及数据缓存、复杂业务逻辑、触发消息、领域事件的功能还需要服务端实现。客户端也提供了SaveBySvc<T>和DeleteBySvc<T>方法应对简单的服务端处理，它们只是将实体对象传输到服务端，调用服务端\_dp.Save<T>或\_dp.Delete<T>方法，见上图绿框。

### 服务端

服务端可粗略分为四层：Api层、领域层、数据提供层、基础层。

Api层是客户端提交请求的入口，因采用富客户端单页模式，所以请求参数和返回结果都为数据，无UI相关内容，因此Api层对应于DDD模式的应用层+领域服务，可以包含参数校验、实体加载、多种类型实体之间的业务交互、数据持久化等内容。在实现上Api类继承自BaseApi并带有Api标签，如下图：



每个Api类在服务的.admin页面都可以查看到：



点击Api方法进入测试页面，输入参数可以手动测试：

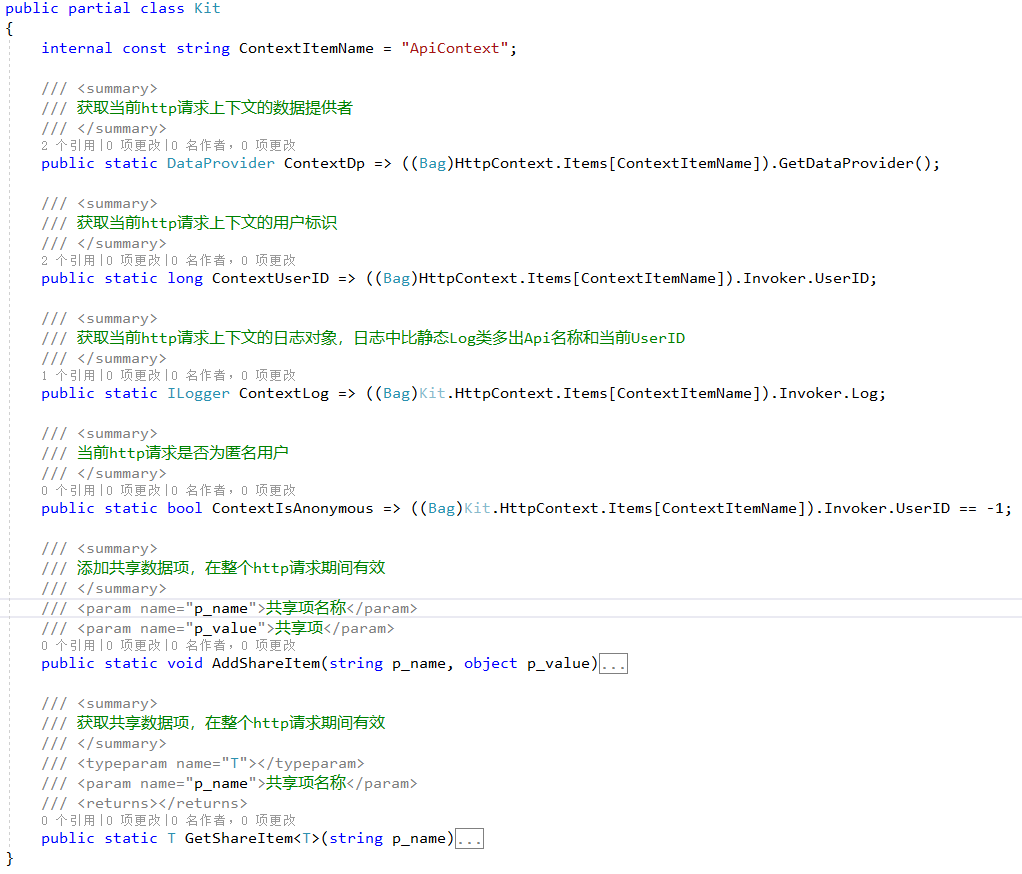


Api遵循以下原则：业务方法与事务和业务用例一一对应，即每一个业务方法均构成了独立的事务边界，通过给Api方法添加Transaction标签，系统将整个方法封装到了一个事务中处理。Api层、领域层、数据提供层、基础层的调用关系上未严格遵循DDD模式，因不存在各层之间数据格式的转换。

架构图右侧垂直放置的Kit的静态对象，在Api调用的整个生命周期都存在，它存放在当前HttpContext的字典中，在程序的任意位置都能获取，它的创建及释放过程见以下代码，在没有进入Api方法以前该对象就已存在，并且在Api调用结束后，根据是否成功处理后续工作，如提交或回滚事务、关闭数据库连接、发布领域事件等。



为方便使用放在Kit类，所有属性和方法都为静态：



领域层主要由实体组成，和DDD模式不同，不再区分聚合根、值对象等类型，实体之间的关系可自定义调整，实体中主要包含业务逻辑、修改属性时的校验和附加操作、保存或删除前的校验，实体可以增加静态方法用来创建或加载实体及子实体，但始终不应包含持久化的功能，这和客户端的规则相同，详细使用方法请参见“数据对象”一节。

数据提供层的方法和客户端的公共Api基本相同，但该层还负责触发领域事件、同步缓存处理等功能，该层由DataProvider类实现，吸取CQRS架构读写分离的优势，读数据的方法都采用sql参数化查询的方式，配合Sql语句的管理，既灵活又有性能优势，写数据的方法负责根据实体状态生成sql语句对数据库进行插入、更新和删除操作，原则上整个系统不允许出现含insert update delete的sql语句，sql查询语句只放在xxx\_sql表中统一管理。DataProvider内部针对数据库的操作由MySqlAccess完成，开发时避免直接使用MySqlAccess，每个http请求对应一个Bag(私有类)，在Api子类中通过\_dp变量获取当前http请求的数据提供者，其他位置可通过Kit.ContextDp获取同一对象，为何每个http请求对应一个固定的DataProvider，因为DataProvider内部可能已启动事务或打开连接，在请求结束时需要提交或回滚事务、关闭连接等操作，这些处理都已封装在DataProvider中，所以涉及\_dp操作的Task必须await，否则会出现请求结束\_dp已释放，仍有Task使用\_dp的情况。Api方法需要启动事务时请添加Transaction标签：



这样在DataProvider会自动启动、提交或回滚事务：





Api类中可通过变量\_dp获取当前http请求的数据提供者，其他位置可通过Kit.ContextDp获取(如实体类内部)。

基础层主要包括日志、本地事件、远程事件、数据访问、缓存处理等功能，在开发时可通过Kit的静态属性获得。

## 数据对象

数据对象是整个系统的基础，标准的DDD设计包含多种数据对象，基本上每层一种类型，如：



它的优势是各层之间可以自定义数据内容，降低各层之间的耦合，缺点是类型转换带来的资源消耗、系统的分裂感、代码及名称的繁琐，总体感觉太过于循规蹈矩。

本系统的数据对象需要满足以下使用场景：支持UI层的绑定(ViewMode)、自动生成实体代码、自定义序列化内容、从db加载数据时高性能不使用反射、反序列化时自动转换实体类型、充血模式的实体对象、数据持久化时转换成Sql。

为满足以上所有使用场景，采用Row作为基类，Row负责装载所有数据，内部通过数据项(Cell)集合的方式进行管理，Table是Row的集合类，并包含Columns属性，为满足领域中用到的实体对象，抽象类Entity继承自Row，所有实体类需要从Entity派生，泛型类Table<TEntity>继承自Table，泛型参数约束为实体类。



下面针对这些使用场景逐一说明：

### UI绑定

Fv的数据源为Row时，FvCell数据源对应Cell，通过Row.Cells[ID]确定，绑定路径为固定的Val，Row实现接口INotifyPropertyChanged，确保双向绑定有效，如IsChanged属性绑定到保存按钮的IsEnable属性实现保存控制。



同样Cell也实现接口INotifyPropertyChanged，常用的绑定属性有：



Lv或Tv的数据源为Table时，Row对应UI中的ViewItem，实际绑定到LvRow或TvPanelItem，Cell对应具体的可视元素，Table继承自ObservableCollection<Row>，能将行数的变化实时反应到UI上。



### 生成实体代码

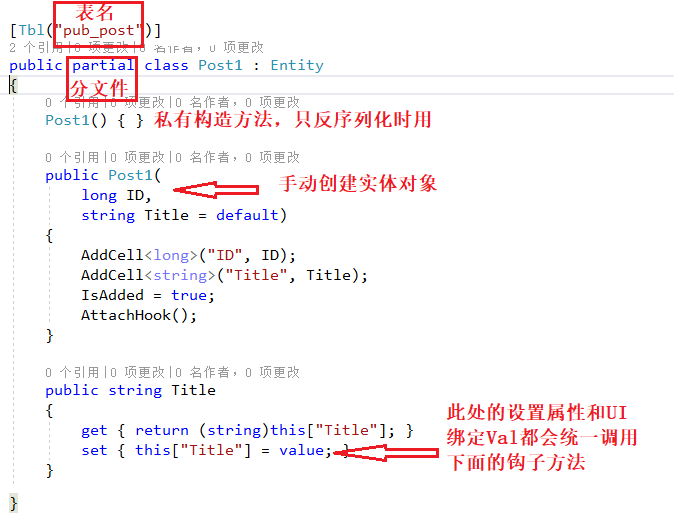
参见上述读写数据过程，业务开发时既可以使用Table/Row，也可以使用Table<Entity>/Entity，一般简单数据读写时使用Table/Row，包含业务逻辑时使用Table<Entity>/Entity，实体类代码分成两部分，一部分通过系统自动生成代码，主要为构造方法和属性，另一部分为业务逻辑，根据需求完成。为方便管理，自动生成的代码禁止修改，只修改业务逻辑的partial classs，通常两个类放在一个文件中，如：



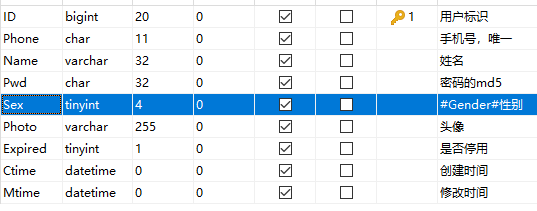
自动生成实体代码需要提供实体类对应的表名，如下：

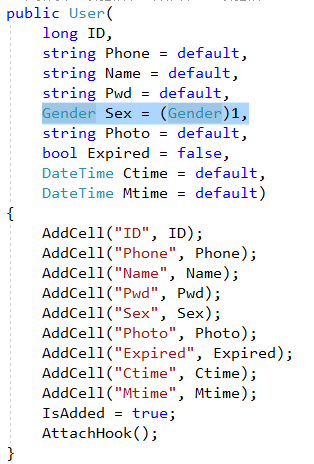


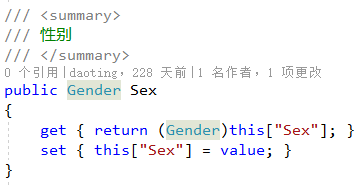
生成实体类代码如下：



该实体类主要包括构造方法和属性，公共构造方法用于手动创建实体对象，方法参数除主键外其他参数都有默认值，默认值是数据库表结构中字段的默认值；属性和字段一一对应，所以属性类型都是System命名空间的简单类型，如string long DateTime bool int byte等，但不包括枚举类型。mysql中tinyint(1)对应C#的bool，无符号的tinyint(4) 对应byte，有符号tinyint(4)对应SByte(使用场景少)，为了编码、查询、显示方便，特殊处理了enum类型的情况，将无符号tinyint(4)设计为enum类型，最多能存储256个成员在程序中已够用，生成实体类时支持自动将#EnumName#开始的注释当作枚举类型，增加方便性和代码的可读性。如Gender(性别)枚举类型：

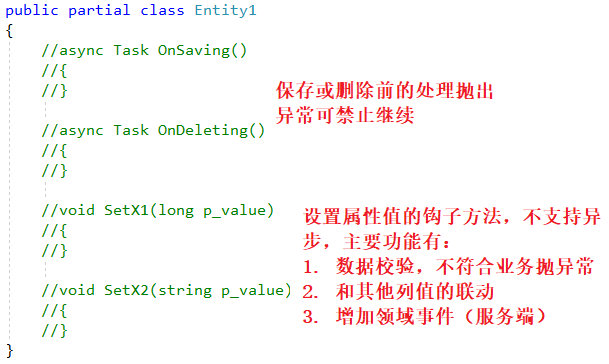






这样所有涉及对该实体类的sql查询、序列化反序列化，底层都会自动将byte类型转为对应的enum类型，减少很多重复的工作量，如sql查询中不再需要case语句，程序用于显示的地方显示enum成员名，保存时为byte值，编辑时下拉选项为所有enum成员。

业务逻辑类型可以通过VS的扩展工具添加代码框架，内容主要是保存、删除前的校验，属性值改变时的钩子方法(主要UI用)，如：



OnSaving和OnDeleting在写数据前调用，返回Task，通过抛出异常来禁止写数据，OnSaving方法使用场景较多，保存前的校验(如不可为空、重名、数据不符合业务规则等)、数据完善(如修改时间)、服务端还可触发领域事件。

SetXXX方法是属性值(Cell值)改变时的钩子方法，若实体类中包含某属性的钩子方法，所有对Cell.Val的赋值都会调用钩子方法，包括UI的绑定。

钩子方法规范：

* 私有方法，命名SetXXX的XXX为cell.ID；
* 一个入参，和cell.Type相同；
* 无返回值；
* 方法内部只包含业务逻辑和数据处理，不涉及UI。

钩子方法的主要功能：

* 数据校验，不符合业务抛异常使赋值失败；
* 和其他列值的联动；
* 增加领域事件（服务端），如单据状态变化时触发领域事件发送短信提醒

### 自定义序列化

数据对象需要在客户端与服务端之间、服务与服务之间传输，序列化时需要保证数据的完整、简洁，以下为Table, Row序列化为json时的结构：





### 高性能加载数据

普通ORM从数据库读取数据、加载数据时一般通过反射创建实体对象、进行属性赋值，本系统采用的方式不同，因基类Row通过数据项集合的方式管理数据，实体类属性并不直接对应变量，所以添加数据时不需要通过反射的方式对属性赋值，而是向集合增加数据项即可，参见以下：



### 自动类型转换

为避免数据对象在客户端与服务端之间、各层之间的耦合性，采用降型传输的方式，即将实体类型降低为Row传输，反序列化时再根据实际情况创建新的实体对象，比如客户端实体类UserA，在保存实体数据时以Row类型传输到服务端，反序列化时创建实体对象UserB，类型UserA和UserB分属两端，各自独立使用，无任何代码上的耦合，但数据内容是相同的或相交的，这样既降低实体类型的耦合性也减少实体类型转换带来的冗余操作。



由上图可见，两个实体类型不论数据相同还是相交，Row都应该提供两个实体的完整数据。

### 充血模式

实体中的SetXXX方法是属性值改变时的钩子方法，主要用来数据校验，不符合业务抛异常使赋值失败；或可以和其他属性值联动；或触发领域事件（服务端），如单据状态变化时触发领域事件发送短信提醒等。

OnSaving和OnDeleting在数据持久化前调用，返回Task，通过抛出异常来禁止写数据。

也可以增加静态方法用来创建或加载实体及子实体。

但实体中始终不应包含持久化的功能，不应涉及UI，这些规则服务端和客户端相同。

### 转换成Sql

在生成实体类型代码时通过Tbl标签已指定映射的表名/视图名，实体对象在使用过程中自动记录数据变化及状态(如IsAdded, IsChanged)，在传输过程未丢失任何信息，客户端和服务端都可以将实体对象生成要执行的Sql语句，生成Sql过程请参见TableSchema.cs文件，客户端公共Api的“写数据”就是在客户端生成的Sql语句。

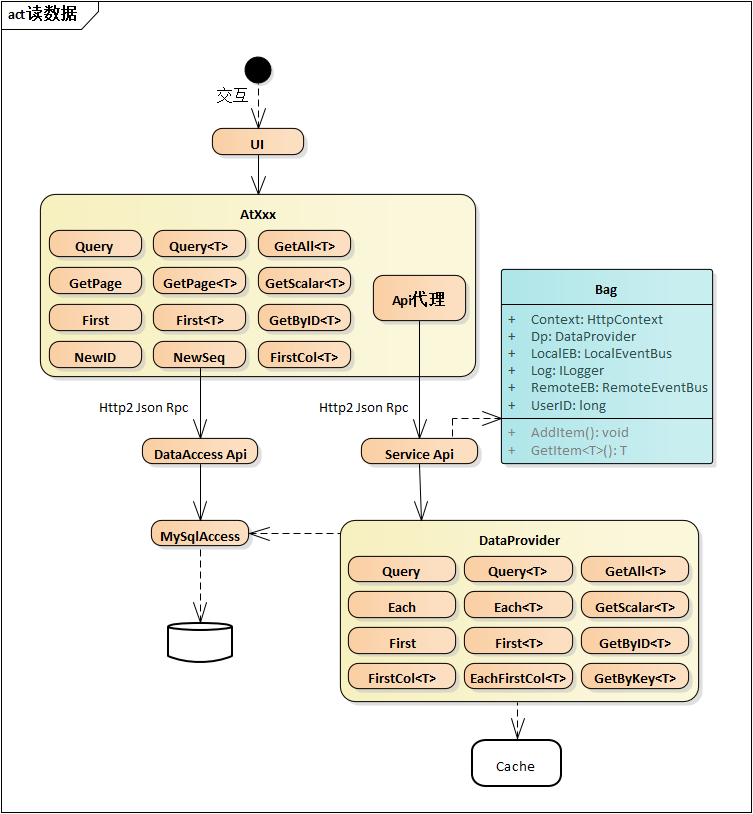
## 数据提供者

数据提供者和DDD模式中数据仓库(Repository) 类似，介于领域层与数据访问层之间，负责数据映射、根据实体状态生成Sql、数据持久化等功能。它的存在让领域层感觉不到数据访问层的存在，领域层需要什么只需查询数据仓库，不需要知道数据实际放在哪里。

客户端的数据提供者已合并在服务代理类中，作为公共Api使用；服务端的数据提供者已绑定在当前http请求的Bag(私有类)中，总体架构中都已介绍，以下按照读写数据的分类进行详细说明。

### 读数据

借鉴CQRS读写分离的原则，数据查询采用sql参数化查询的方式，既灵活、查询速度快，也和领域相关内容解耦，过程如下：



读数据按位置分为两种，客户端公共Api读数据和服务端DataProvider读数据，两处的方法及参数基本相同，下面以客户端为例介绍这些方法。

客户端公共查询Api分三类：

一种返回实体列表，参见下图有五个方法，Query和Query<T>两方法参数相同，Sql键名+Sql参数，返回类型为Table和Table<T>，GetPage和GetPage<T>是按页查询，除页面参数外，其他参数和Query相同，GetAll获取单表的所有数据，是一个辅助方法，使用场景少。



第二种返回单个实体对象，主要有三个方法，GetRow和Get<T>的参数和Query相同，只是返回符合条件的第一行数据，GetByID<T>是通过主键获取实体对象，不需要Sql查询语句，服务端DataProvider的GetByID<T>不同之处主要是在获取单个实体时首先从缓存中查询。



最后一种是查询单个值，GetScalar<T>是通过Sql参数查询返回第一单元格数据，NewID获取新ID，详见“分布式ID生成方案”一节，NewSeq获取新序列号，类似于oracle的sequence，自增长的序列号，一般用来存储显示顺序。



整个读数据的核心就是Sql参数化查询，它是最灵活、最直观、性能最优的查询方式，远比其他ORM生成的查询好控制。

需要特殊说明的是，通过Sql参数查询的方式，结果不管是实体列表还是单个实体，Sql语句中的列都无需和实体属性一一对应，按需查询即可，前面“数据对象”中已介绍，因为Entity继承自Row，最终使用单元格Cell存储数据，而实体属性只是对Cell的封装，当Sql查询语句中缺少或多出某些列时，能正常进行序列化、反序列化、类型转换，只是在访问缺少的属性时异常，这样增加了很大的灵活性，比如：某列内容多但不需要显示，查询时为提高性能减少数据量就可以不查询该列，但前端其他列的绑定都不受影响。Sql查询语句中也可以比实体属性多出一些列，比如通过多表联合查询的翻译列等，这些多出的列在绑定显示时同样不受影响。

### 写数据

系统提供两种写数据方式，见下图，第一种为通过客户端AtXxx的写数据模式，通过这些写数据方法在客户端生成Sql，交由代理方法调用服务端DataAccess Api，最终由MySqlAccess将数据持久化到数据库，该模式可以理解为传统的sql方式，只是sql语句由程序自动生成；第二种是客户端将需要持久化的数据通过Api代理交由服务端Service Api，再由DataProvider根据实体模型及状态生成sql语句，最终也是通过MySqlAccess将数据持久化到数据库，同时触发领域事件、同步缓存。

所以两种写数据方式都是根据实体模型及状态生成sql语句对数据库进行插入、更新和删除操作，只是生成sql语句的位置不同，一种在客户端生成，一种在服务端生成。原则上整个系统不允许出现含insert update delete的sql语句，增删改的sql语句都由系统自动生成。



保存单个实体数据使用Save<T>方法，该方法根据实体状态IsAdded确定是insert还是update，当update时，只将变化的Cell(根据IsChanged状态)生成sql语句，服务端还会根据实体的缓存设置进行处理：若已设置按主键缓存，则删除该缓存数据以便下次能从数据库读取最新数据；收集领域事件，在保存成功后统一触发。



批量保存实体使用BatchSave方法，它不是泛型方法，待保存的列表仅需IList，为各种保存提供可能，如单表数据保存参数可以为Table<Entity>或List<Entity>，其中Table<Entity>根据DeletedRows属性还支持删除操作，List<Entity>只支持单表的增改；多表保存时参数为List<object>，列表中每项可以为Entity, Table<Entity>或List<Entity>，这就满足了批量保存的所有场景，并且生成sql语句的执行顺序严格按照List<object>中的顺序，避免存在主外键关系的数据批量保存时的异常。服务端DataProvider保存成功后会对所有实体触发领域事件和同步缓存操作。



删除方法包括删除单个实体、批量删除实体，删除单个实体DelByID可以只给定id，但内部未通过OnDeleting校验直接删除，所以要统一经过业务校验请使用Delete<T>或BatchDelete方法。使用BatchDelete的参数和BatchSave参数相同，支持单表或多表批量删除。服务端DataProvider同样会在删除成功后对所有实体触发领域事件和同步缓存操作。



Exec和BatchExec是直接执行sql语句，业务开发时禁止使用。

### 领域事件

领域事件底层使用的EventBus技术实现细节请参见“基础功能”的相关内容，这里主要描述业务开发过程中的应用。

领域事件是实体触发的事件，一般在修改属性、保存或删除后触发，类似于数据库触发器的概念，领域事件能减少业务之间的耦合，比如用户手机号码修改后需要发送email通知，实现时若将发送email通知的功能和保存手机号码写在一起，那么再添加“向新号码发送通知”的功能就会变动非常大，甚至跨服务添加该功能时无法实现，关键因为和“保存手机号码”耦合的过紧，这时采用领域事件就能解决该问题，“保存手机号码”后触发一个手机号码变化事件，该事件可以是本地事件或远程事件，后续的其他功能就可以订阅该事件执行相应处理即可，增加或停用后续功能都不会对“保存手机号码”造成影响，并且他们之间也互不影响，还可以跨服务处理。系统内置一些常用的领域事件：



这些事件可以通过实体标签进行配置，并且支持同时触发多个事件：



也可以通过Entity的AddDomainEvent方法添加自定义事件，比如在某属性修改时调用AddDomainEvent添加新事件，Api调用结束后会触发这些事件。





以下为实体的增删改和属性值变化的领域事件处理，增删改的基类都是泛型，支持不同实体类型分别处理，如继承InsertEventHandler<User>的类只负责处理User的插入事件，继承InsertEventHandler<Role>的类只负责处理Role的插入事件，在同一程序集的同一事件类型的处理类只支持一个，如继承InsertEventHandler<User>的类若有多个，则只有一个有效，这样也避免多个同类处理造成的混乱和冲突。



### 缓存

服务端DataProvider的缓存原则为：当实体已配置缓存标签时，读取单个实体时首先从Redis缓存查询，缓存中没有再从数据库中查询并将结果缓存以备下次查询，的保存或删除实体后从缓存中也删除这些实体，避免从缓存中查询出脏数据的情况。需要特殊说明的是：查询方法中只有GetByID<T>和GetByKey<T>在查询单个实体时首先从缓存中查询，其他sql参数化查询都不支持从缓存中取数据，但Save BatchSave Delete BatchDelete都支持在保存成功后删除缓存。

实体缓存配置：



以上配置表示在缓存User的键名为：



每个User的缓存内容为json串：



# 平台服务

服务名称取自服务所属程序集默认命名空间的末尾段，小写格式。通过Glb.GetCurrentSvcs实时获取应用内正在运行的所有微服务。



目前平台只支持所有微服务连接同一RabbitMq、同一Redis，但每个微服务可以连接不同的数据库，详细配置见“3.1服务配置”。

## 内核模型服务(Cm)

CM是Core Model的缩写，为平台提供基础的内核模型服务，



## 消息服务(Msg)

平台内置的基础消息服务，支持多副本部署，所有已注册的客户端在启动时调用服务端的Register方法，该Api采用基于http2协议的ServerStream模式，即：客户端发送一个请求，服务端返回数据流响应，相当于在服务端和客户端建立了长连接，服务端可以实时向客户端推送信息。



### 推送过程

服务端向客户端发送的消息主要包括三类：即时消息(聊天内容或系统消息)、订阅消息、指令消息，其中即时消息和指令消息按userid发送，订阅消息只推送给订阅了该消息类型的用户，所有类型消息的发送都遵循“在线直接发送”、“离线按照用户设置进行推送”的原则。其推送过程如下图所示：



### 客户端解析

客户端通过PushHandler.Call方法解析推送内容并调用相应的处理方法，处理方法所属类型需要指定[PushApi]属性，这样生成代码时能包含到Stub.PushHandlers字典。



系统内置推送处理



推送内容的PushHandler.Call方法内容如下：



## 文件服务(Fsm)

文件服务统一管理应用范围内使用的文件，在k8s中支持多副本部署，支持分卷存储文件，可通过ConfigMap将数据卷Volume挂载到指定目录，该服务只负责文件的上传下载和文件管理功能，不涉及具体的业务问题，在功能和Api上较稳定。

### 卷

服务支持挂载多个卷，每个卷有不同的用途，系统将卷划分为固定卷和普通卷，固定卷是指客户端在上传文件时可根据卷名访问的卷，该卷一般存储特殊用途的文件，如头像文件、聊天文件等，使用场景较少。普通卷是除固定卷以外的卷，上传文件时根据普通卷的使用情况动态选择要存储的卷，服务启动时在Redis缓存中创建了SortedSet类型的volume键，该SortedSet列表中的每行对应一个卷，值为这个卷的“正在使用数”，正在使用数就是当前正在写入卷的请求数，该数值能正确反映一个卷的闲置状态，每次有新的上传请求时都选择“正在使用数”最小的卷进行存储。



### 上传

文件上传采用流式上传方式，流式上传和使用IFormFile在效率上没有太大的差异，IFormFile的缺点主要是客户端上传过来的文件首先会缓存在服务器内存中，任何超过 64KB 的单个缓冲文件会从 RAM 移动到服务器磁盘上的临时文件中。文件上传所用的资源（磁盘、RAM）取决于并发文件上传的数量和大小，流式上传主要解决内存占用过高和磁盘空间不足问题。

Http Post请求时内容有三种编码类型，文件上传使用multipart/form-data：

1. application/x-www-urlencoded，Form默认的编码类型，消息内容会经过 URL 格式编码，就像在 GET请求时 URL 里的 QueryString 那样。





1. multipart/form-data,它可以包含了多个 Parts，每个 Part 都包含头信息部分，Part 头信息中必须包含一个 Content-Disposition 头，其他的头信息则为可选项，比如 Content-Type 等，Content-Disposition 包含 type 和一个名字为 name 的 parameter，type是 form-data，name 参数的值则为表单控件（也即 field）的名字，如果是文件，那么还有一个 filename 参数，值就是文件名。





1. text-plain，空格转换为 "+" 加号，但不对特殊字符编码。

文件上传的Section结构如下：



当使用固定卷时第一Section用于指定固定卷名，当使用普通卷时所有Section都是FileSection。

文件上传过程：



### 下载

下载文件时需要提供文件的完整路径，记录文件下载次数。下载缩略图时，若缩略图不存在，则下载原始图像，客户端可通过Api先判断缩略图是否存在。

缩略图命名规则：原文件名添加后缀"-t.jpg"，如 143203944767549440.wmv 的缩略图文件名为143203944767549440.wmv-t.jpg。

### 浏览

和下载类似，文件服务还支持按路径直接浏览的方式，方便图片或视频等嵌入到网页中。

如：https://localhost/baisui/fsm/v0/21/06/80819234523705344.jpg

同样浏览缩略图时若无缩略图自动取原图。

## 网页发布服务(Pub)

Pub是静态网页服务，包括html富文本编辑器和动态生成的网页两部分，html编辑器用于编辑网页内容，编辑后的网页内容可以保存在数据库或根据模板生成完整网页。

### 编辑器

html编辑器使用Froala Editor(V3.1.1)，功能强大界面美观但收费，在froala\_editor.min.js或froala\_editor.pkgd.min.js文件中搜索new Image，删除以下代码：

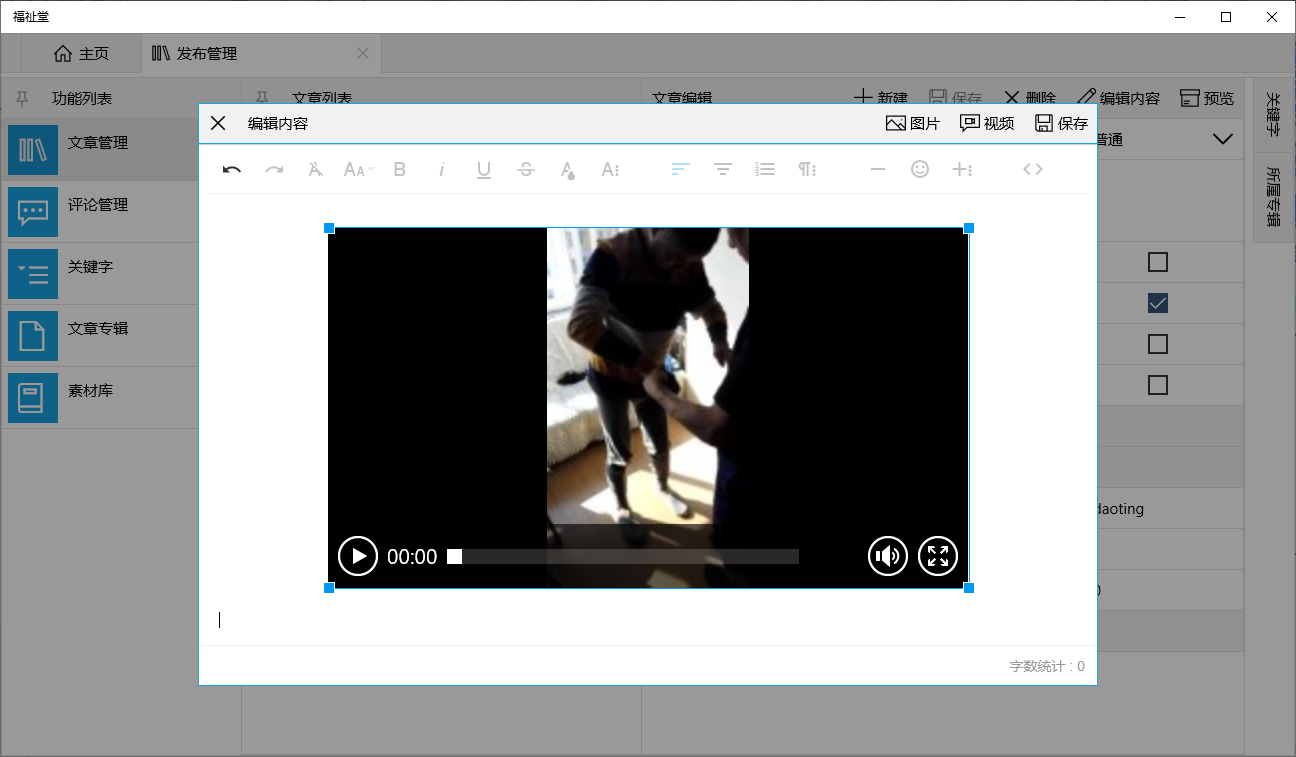


搜索POWERED\_BY，删除左下角logo链接：



### 发布过程

打开客户端“发布管理”-> 编辑文章内容

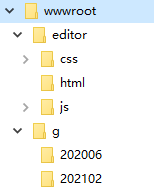


->编辑结束后保存文章时pub服务自动生成该文章的静态网页 –> 返回该网页的地址、保存，以备浏览用。

可通过目录 <https://localhost/fz/pub/g/> 浏览生成的静态网页。

### 文件目录

pub服务的编辑器和生成的静态网页都部署在wwwroot目录下，如下图：



editor目录包含froala编辑器的所有文件，默认编辑器在wwwroot\editor\html\default.html页面。g目录下保存生成的静态网页，以“年月”格式的目录作为网页的二级分类。因生成的网页中采用相对路径，如插入的图片或视频为“../../../fsm/v0/26/2D/91789217609150464.mp4”，并且为了保证生成的网页和froala中编辑的内容都能正常显示，禁止修改目录结构。

# 服务部署

服务端环境的安装部署主要包括平台服务、mysql库、RabbitMQ、Redis、k8s、IIS等，本章主要介绍平台服务的两种部署方式：IIS部署、k8s部署，最初开发时使用Traefik做反向代理，windows版的Traefik使用方式和容器版类似，可以按照服务的url规则进行配置，详细配置参见traefik.toml，还特意做了“搬运工助手”工具，用于启动、停止服务和查看日志。

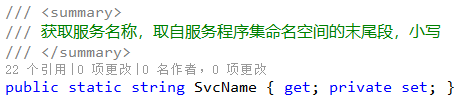
所有部署都需遵循以下url规则：

## URL规则

平台服务的url规则模板为：<https://ip/appName/svcName/xxx>，如：<https://ip/baisui/cm/xxx>。

一级目录appName为应用名称，必须和svc\etc\config\global.json中的App值相同；二级目录svcName为服务名，必须和服务的Stub类所在程序集命名空间的末尾段相同；所有名称都采用小写，服务管理页面地址为：<https://ip/appName/svcName/.admin>。



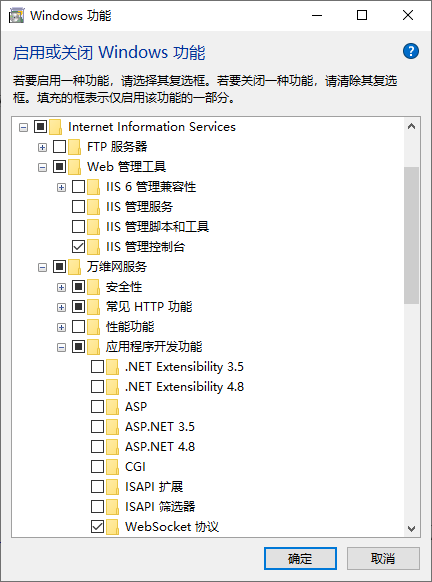


## IIS部署

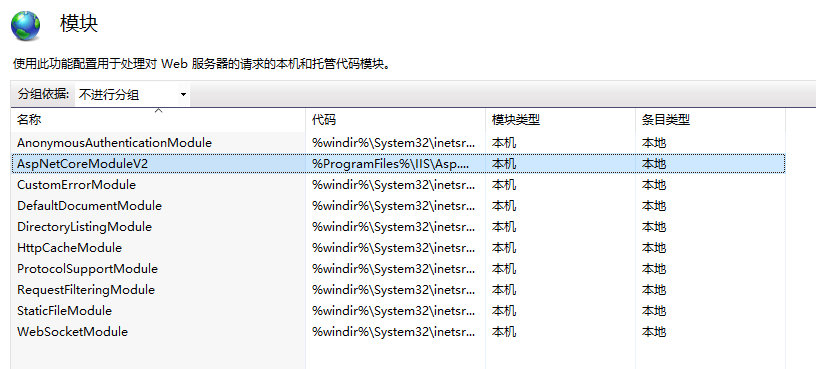
在开发或部署到windows server时首选IIS，和传统asp.net应用不同，IIS只作为反向代理使用。

### 安装IIS

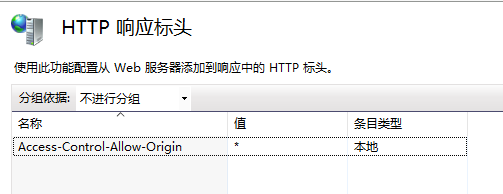
若无asp.net旧应用，无需安装“应用开发功能”，若调试wasm应用需要安装WebSocket协议，如下图：



安装AspNetCoreModuleV2，<https://dotnet.microsoft.com/download/dotnet/5.0> 点击对应版本的Hosting Bundle进行下载安装。



为避免浏览器跨域访问的问题，在根节点为IIS添加”http响应标头”： Access-Control-Allow-Origin，值根据实际填写，本地调试时可为\*，如图：



### https配置

平台所有服务都采用http2.0协议，http2.0协议和https之间本来没有依赖关系，但主流浏览器要求http2.0协议必须采用https通信，要使用https需要在网站的服务器上配置https证书，IIS和k8s的traefik都需要配置该证书。

证书可以向专门的https证书提供商进行购买，也可以自己生成，两种的区别是自己生成的证书不被浏览器信任，需要点击信任或将证书安装在“受信任的根证书颁发机构”之后才能继续访问，安装方法是将Dt\Service\iis\tls.pfx证书安装在“受信任的根证书颁发机构”，私钥密码dt。

默认情况下，IIS中的网站只提供了http绑定，要支持https绑定，首先需要pfx格式的服务器证书，以下为生成https证书的过程：

在windows上安装Win64OpenSSL\_Light-1\_1\_1b.exe，然后打开PowerShell，cd到openssl.exe所在的目录

生成私钥tls.key, 密钥位数是 2048

.\openssl genrsa -out tls.key 2048

使用server.key 生成自签证书，域名localhost，有效期20年

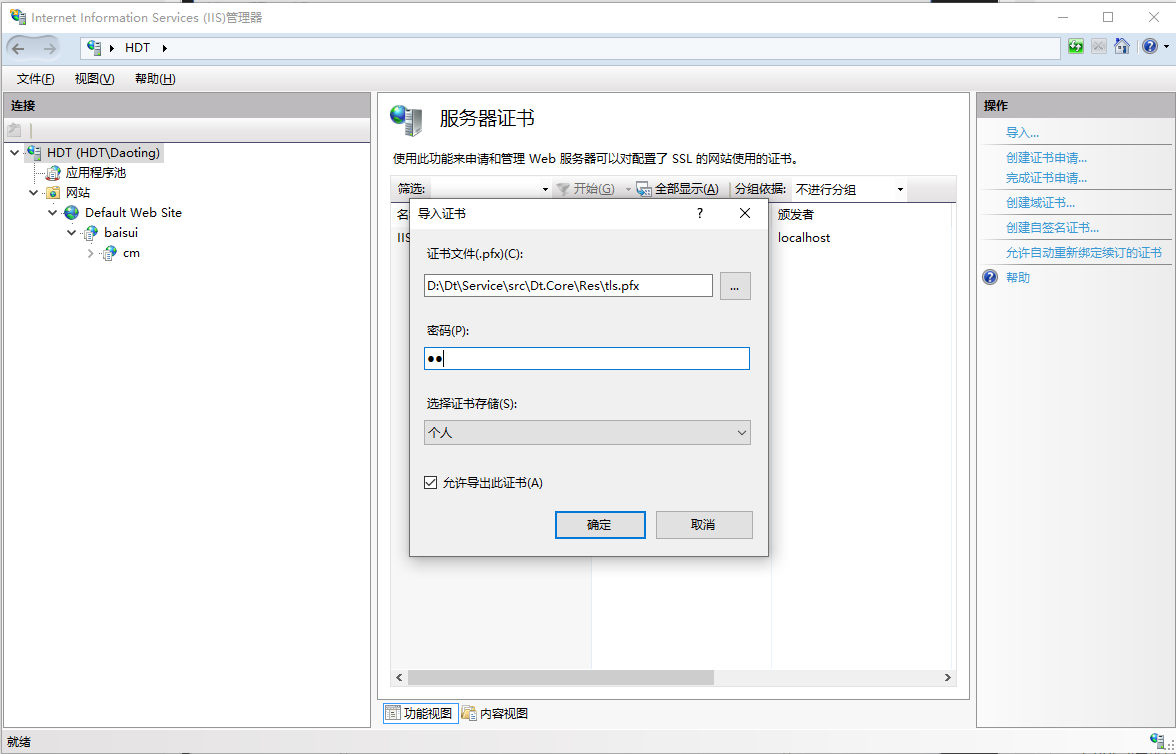
.\openssl req -new -x509 -days 7300 -key tls.key -out tls.crt -subj /CN=localhost -addext "subjectAltName=DNS:localhost"

生成tls.key, tls.crt文件用于以上secret资源，crt存储公钥，key存储私钥

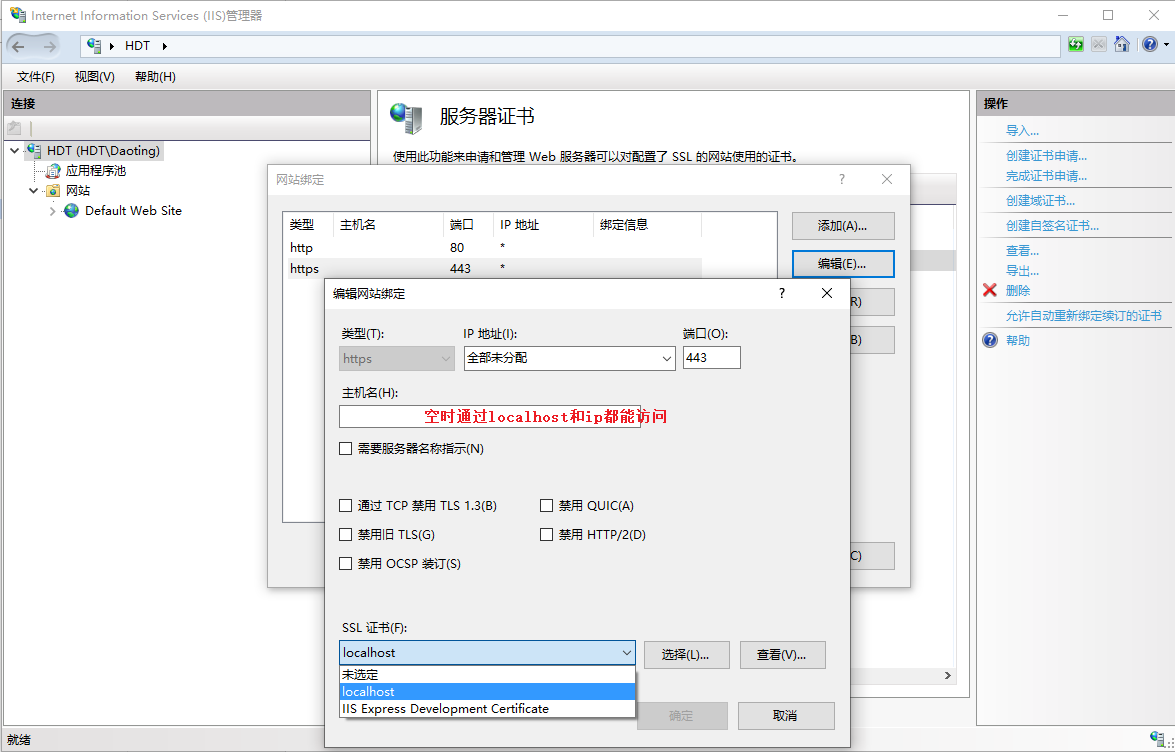
将私钥和公钥合并成pfx证书，密码为dt，用于服务的x509认证

.\openssl pkcs12 -export -in tls.crt -inkey tls.key -out tls.pfx -name "Dt Platform"

导入服务器证书，证书位置Dt\Service\iis\ tls.pfx，私钥密码dt，此证书签名为localhost，只在开发时使用，生成环境可替换，导入新证书即可。



然后右键“Default Web Site”->编辑绑定->添加->选择https->选择ssl证书，也可以使用IIS Express提供的证书。

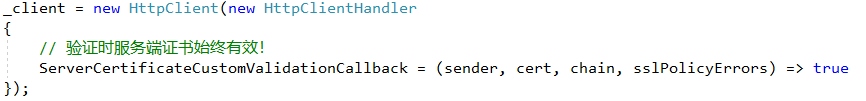


当使用KestrelServer时需要将证书复制到服务的根目录：

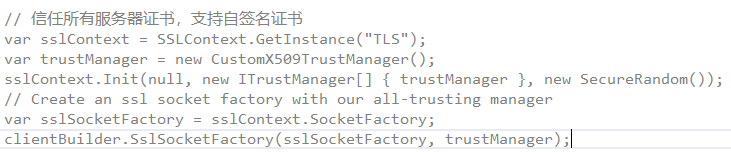


客户端提供对证书的支持：

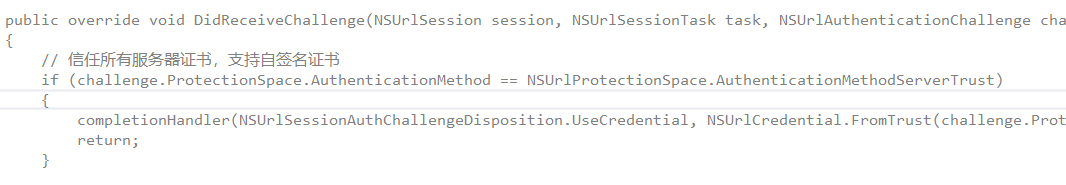
uwp



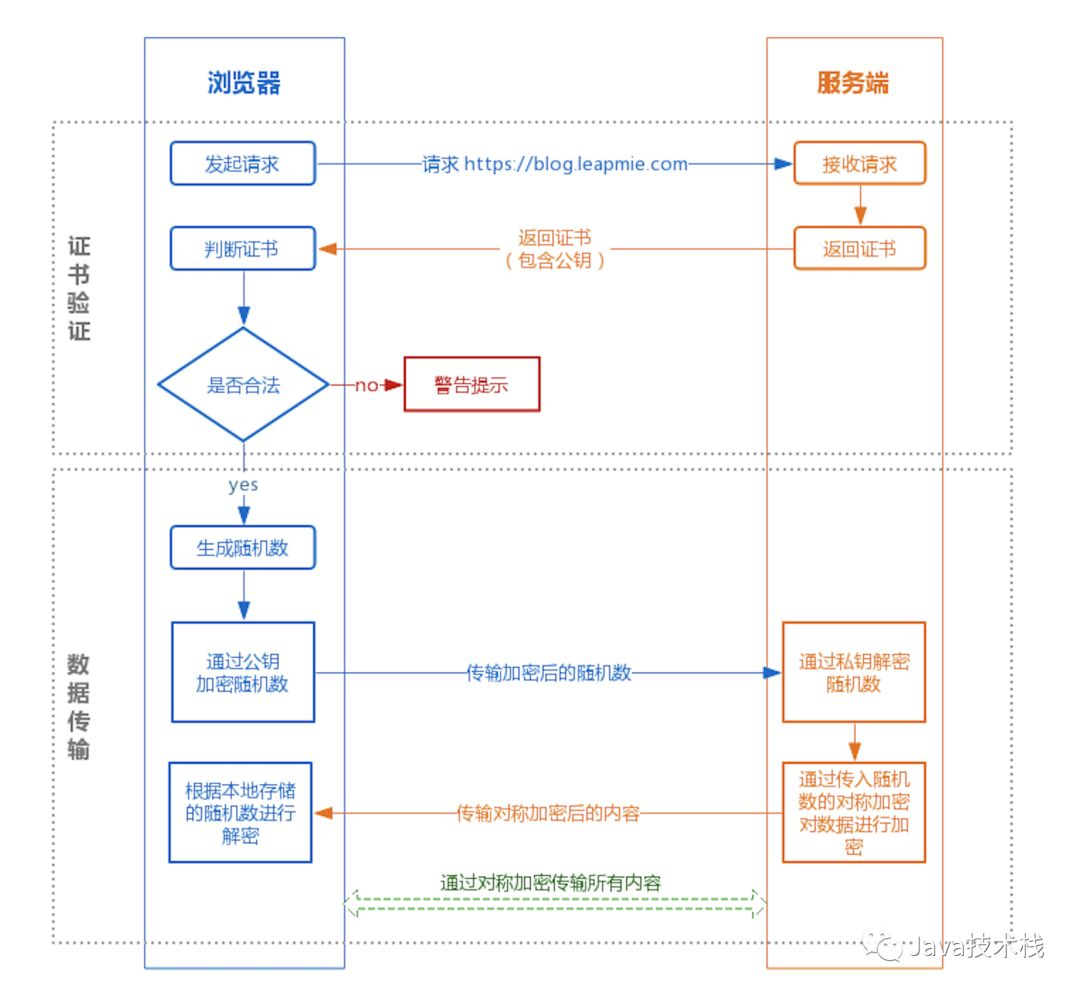
android



iOS



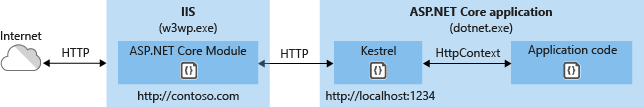
对称加密是加密和解密使用同一个密钥；非对称加密使用一对非对称的私有密钥和公开密钥，发送密文的一方使用对方的公开密钥进行加密处理，对方收到被加密的信息后，再使用自己的私有密钥进行解密，对称密钥的好处是解密的效率比较快，非对称加密的好处是传输的内容不能被破解。HTTPS采用对称加密和非对称加密两者并用的混合加密机制，过程如下



### 承载模式

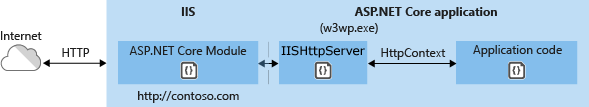
将服务部署在IIS时有两种承载模式：进程(w3wp)外承载和进程内承载。

进程外承载时CreateDefaultBuilder 将 Kestrel 服务器配置为 Web 服务器，如下图：

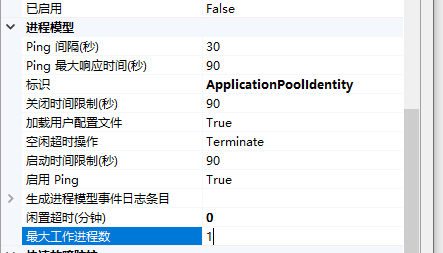


进程内承载时CreateDefaultBuilder 将IISHttpServer配置为 Web 服务器，相较进程外承载提供更优的性能，因为请求并不通过环回适配器进行代理，可提供明显更高的请求吞吐量，默认采用进程内承载，参见：

<https://docs.microsoft.com/zh-cn/aspnet/core/host-and-deploy/iis/in-process-hosting?view=aspnetcore-5.0>



IIS同样支持启动多个服务进程提高响应，对于无状态服务非常有效。



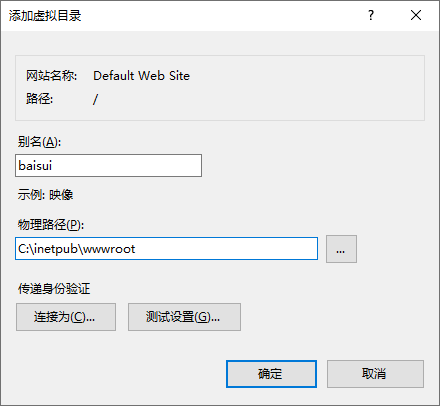
### 添加服务

可以运行批处理文件 \Dt\Service\iis\xxx.bat自动添加服务：

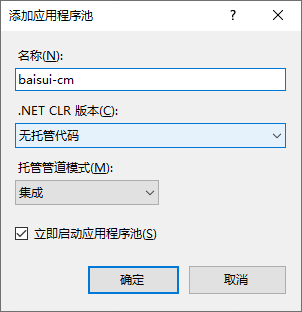


也可以手动添加服务，过程如下：

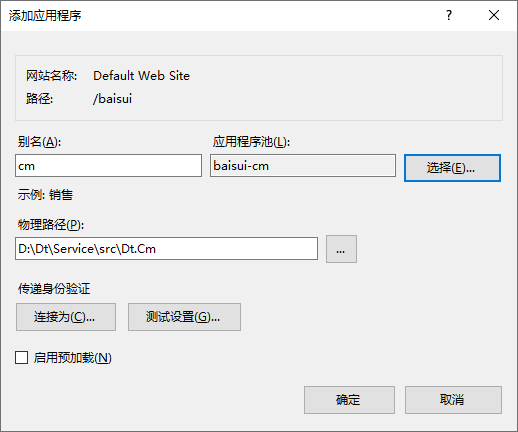
根据平台服务的URL规则：<https://ip/appName/svcName>，在网站下先添加一级目录：网站->添加虚拟目录->别名为appName，路径任意：



因每个asp.net core应用需要占用独立应用池，先创建应用池，命名规则appName-svcName：

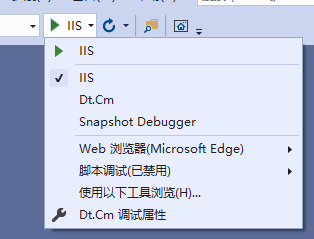
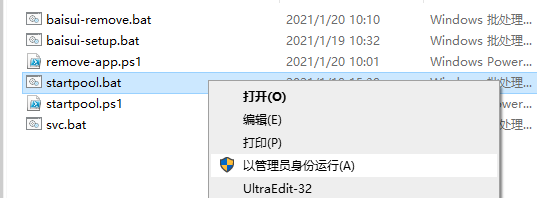


baisui右键->添加应用程序->别名为svcName，路径为服务的发布目录，开发时为项目目录，注意web.config文件：

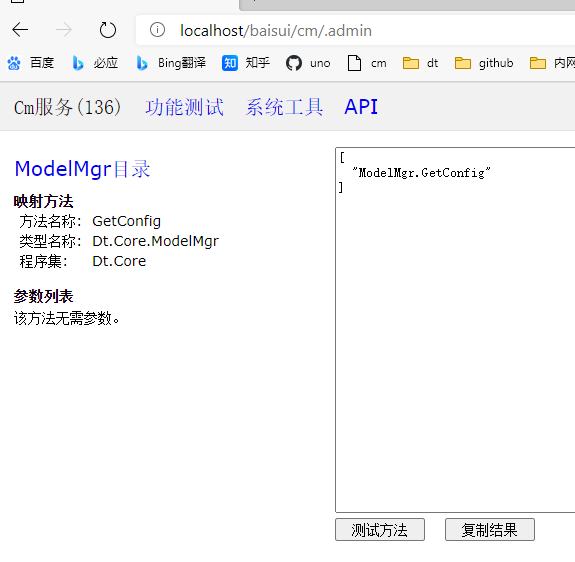


### 调试服务

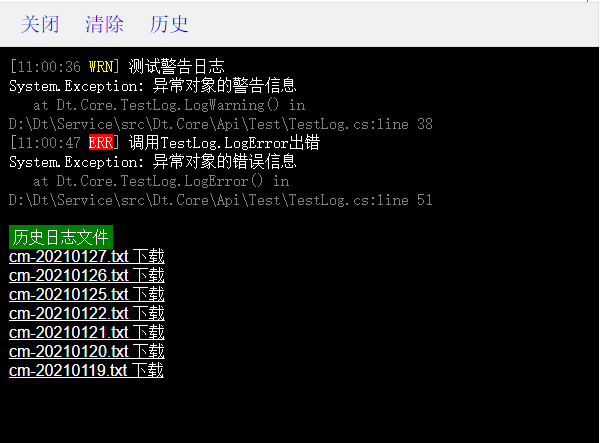
VS中主要有两种调试服务方式，一种是直接启动应用的方式，另一种是部署到IIS或IIS Express进行调试，因为受到服务URL规则的限制(存在二级目录)，系统采用部署到IIS的调试方法。老版本的VS调试IIS上的asp.net应用时需要采用“附加到进程 w3wp”的方式，新版本更方便，初次启动调试时自动创建应用池和应用，代码修改后重新编译时能自动停止应用池，再次启动调试时会同步启动应用池，并且支持调试服务的启动部分。因应用池命名的原因，请使用上节的bat添加服务。注意在关闭VS时会自动停止所有调试过的应用池，可以手动运行startpool.bat启动所有应用池。

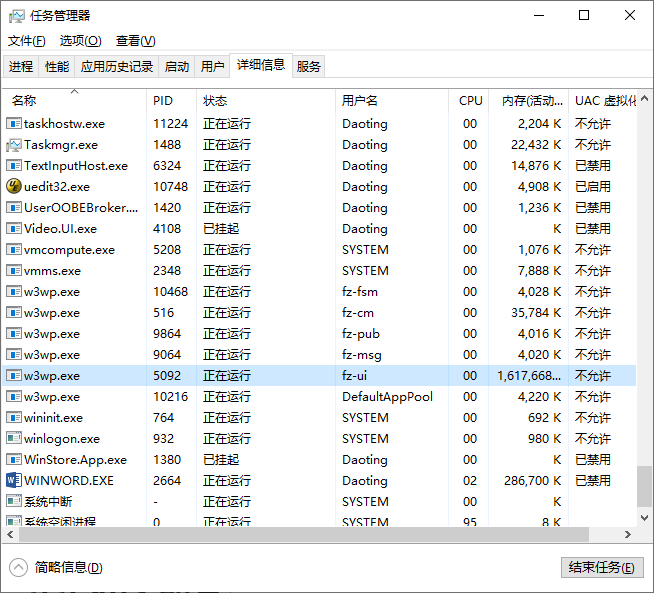
调试服务时可访问https://localhost/baisui/xxx/.admin、选择Api、填写完整参数值、点击调试按钮。



该页面可以查看日志，日志内容及样式和输出到控制台的一样，还可以查看历史日志文件，点击文件链接可以下载日志文件：



以下为所有服务资源使用情况，在调试wasm时内存占用特别大：



## k8s部署

k8s的概念和环境安装过程请参见下一章介绍。

在开发环境和生产环境都可以使用k8s模式部署服务，k8s模式是通过helm->k8s->docker在容器内运行服务，该模式启动慢、耗资源、附加到远程调试无法调试服务的启动部分，运行前需要生成镜像、使用Helm部署到容器、附加到远程pod等过程，所以该模式部署繁琐耗时，不适合在开发过程中使用，适合于生产环境的云部署。

k8s模式下的服务需要一系列步骤才能运行：编译生成docker镜像、运行Helm打包脚本部署到k8s、附加到远程pod进行调试等，为方便部署和远程调试，开发工具使用VS Code，利用VS Code提供的任务，将一系列步骤打包成一个任务运行，任务定义参见.vscode\tasks.json，详细步骤参见dt\Service\k8s目录下的debug.ps1脚本文件、chart模板、服务配置。

k8s远程调试参见.vscode\launch.json，容器内运行时的远程调试配置如下：

{

"name": "cm-k8s",

"type": "coreclr",

"request": "attach",

// 前置任务

//"preLaunchTask": "cm-k8s",

// ${command:pickRemoteProcess} 出错，通常为1

"processId": "1",

"justMyCode": true,

"pipeTransport": {

"pipeProgram": "kubectl",

"pipeArgs": [

"exec",

"-i",

// 使用Deployment部署时pod名称变化，无法自动获取！

"baisui-cm",

"--"

],

// 管道工作目录

"pipeCwd": "${workspaceFolder}",

// 调试工具位置

"debuggerPath": "/root/vsdbg/vsdbg",

"quoteArgs": false

},

"sourceFileMap": {

// 设置源文件映射，容器中的/src映射到当前工作区的/src

"/src": "${workspaceFolder}/src"

}

}

运行命令kubectl exec –i baisui-cm – 附加到远程pod(baisui-cm)内，通过容器中的vsdbg调试工具建立与VS Code的通信，将调试控制和输出信息通过管道传输。sourceFileMap用来设置容器中编译时的源代码路径和当前工作区路径的映射关系，debuggerPath指定vsdbg的路径，quoteArgs设置命令后缀是字符串还是命令，默认true添加引号按字符串处理。

vsdbg是命令行方式的调试器，获取方式和VS2017的相同，因vsdbg下载过慢，为避免重复下载通常将其复制到asp.net基础镜像内的/root/vsdbg目录后制作新镜像作为基础镜像。

vsdbg下载方式如下

在VS2017中调试Docker容器的.NET Core程序比较简单，F5运行即可，初次调试时出现以下错误输出：

1>Info: Using vsdbg version '15.7.20425.2'

1>Info: Using Runtime ID 'linux-x64'

1>Info: C:\Users\ Daoting\vsdbg\vs2017u5 exists, deleting.

因为下载压缩包慢造成的，在浏览器里打开以下链接，

https://vsdebugger.azureedge.net/vsdbg-16-0-20419-1/vsdbg-linux-x64.zip

<https://vsdebugger.azureedge.net/vsdbg-15-7-20425-2/vsdbg-linux-x64.zip>

将链接中的15-7-20425-2替换为第一行输出中的版本，将 "." 替换为 "-"；

将链接中的linux-x64替换为第二行输出中的id。

然后将压缩包解压到C:\Users\Daoting\vsdbg\vs2017u5 （如果文件夹不存在，则创建），之后回到VS，开启调试，一切运行正常。

手动下载最新vsdbg的方法：打开<https://aka.ms/getvsdbgsh>，自动跳转的路径为最新版本号，如：<https://vsdebugger.blob.core.windows.net/vsdbg-16-0-20322-2/GetVsDbg.sh> ，将上部url替换版本号即可。

在VS2019中下载

<https://vsdebugger.azureedge.net/vsdbg-16-0-20419-1/vsdbg-linux-x64.zip>

解压到C:\Users\hdt\vsdbg\vs2017u5

# K8S 环境

## Docker

### 与虚拟机的区别





### 体系结构

主要由Docker Client、Docker Daemon、Docker Registry、Driver、Docker Container五个模块组成，另外还有Graph和Libcontainer两个辅助模块，如下图所示：





Docker Client：用户使用可执行文件docker + 参数的命令行形式与Docker Daemon建立通信，并发送请求→处理请求→返回结果。

Docker Daemon：是Docker架构中一个常驻在后台的系统进程，接受并处理Docker Client发送的请求，然后由Engine执行Docker内部的一系列工作，每一项工作都是以一个Job的形式存在，Job运行过程中需要容器镜像时，则从Docker Registry中下载镜像，并通过镜像管理驱动graphdriver将下载镜像以Graph的形式存储；当需要为Docker创建网络环境时，通过网络管理驱动networkdriver创建并配置Docker容器网络环境；当需要限制Docker容器运行资源或执行用户指令等操作时，则通过execdriver来完成。

Docker Registry：存储容器镜像的仓库，在Docker的运行过程中，Docker Daemon会与Docker Registry通信，并实现搜索镜像、下载镜像、上传镜像三个功能，这三个功能对应的job名称分别为"search"，"pull" 与 "push"。

Driver： Driver是Docker架构中的驱动模块。通过Driver驱动，Docker可以实现对Docker容器执行环境的定制。包含管理容器镜像的graphdriver驱动，配置容器内网络环境的networkdriver驱动，execdriver用来创建和维护容器。

Docker Container：运行应用程序的特定容器，是容器服务的交付实体。

### 常用命令

docker pull 从镜像仓库中拉取或者更新指定镜像

docker images列出本地镜像

docker ps 列出所有在运行的容器信息，-a 显示所有的容器，包括未运行的

docker build 使用 Dockerfile 创建镜像

docker run 创建新容器并启动容器

docker start 启动容器，docker stop 停止容器

docker rm 删除容器

docker rmi 删除本地镜像，先删除容器才能删除镜像

docker exec 在容器中执行命令，-i可附加调试

docker exec –it 容器id bash 进入容器内运行bash

进入容器后安装ping, ifconfig命令：

apt-get update 更新了apt的资源列表，没有真正的对系统执行更新

apt install iputils-ping 安装ping命令

apt install net-tools 安装ifconfig命令

删除坏的<none>:<none>镜像，节省存储空间

docker rmi $(docker images -f "dangling=true" -q)

### 创建镜像

镜像是分层叠加后的文件系统，如图：



一个docker镜像由多个只读的镜像层组成，作为容器运行时的会在镜像上面多加一个可写层，任何对文件的更改都只在此层进行，不会影响到镜像。

制作Docker image 有两种方式：从Dockerfile创建、从正在运行的容器创建 (docker commit)。

最常使用从Dockerfile创建的方式，该方式将所有动作记录在Dockerfile文件中，再 build 成 image，创建镜像文件的命令为：

docker build -t name:tag .

name是打包后的镜像文件名，tag为标签，最后的.表示当前目录为打包的上下文目录，执行该命令时首先将当前目录的所有内容打包传递给Docker daemon，由Docker daemon生成镜像文件。

一般Dockerfile 分为四部分：基础镜像信息、环境信息、镜像操作指令、容器启动时执行指令，# 为 Dockerfile 中的注释。

Dockerfile 文件同时存在多个 FROM 指令时为多阶段构建(multi-stage)，每个 stage 可以理解为构建一个容器镜像，后面的 stage 可以引用前面 stage 中创建的镜像。每个 FROM 指令代表一个 stage 的开始，可以将通过 as 将stage命名，如FROM microsoft/aspnetcore:2.0 AS base。只生成最后FROM指令创建的镜像，所有需要保留的内容通过COPY --from=stagename 命令从前阶段复制，丢弃其它任何中间文件。

以下是Dockerfile常用的指令：

# 基础镜像，版本是可选的，如果空则使用latest版本

FROM microsoft/aspnetcore:2.0 AS base

# 为RUN,CMD,ENTRYPOINT,COPY,ADD等设置工作目录，相当于cd，不存在时会创建，可设置多次。

WORKDIR /app

# 指定容器要打开的端口，容器启动时通过-P指定主机与容器的端口映射关系。

EXPOSE 80

# 复制本地文件(以 Dockerfile 所在目录为工作目录)到容器中的指定目录

COPY <src> <dest>

# 构建镜像时执行的命令，每一个指令都会建立一层镜像

RUN dotnet build WebApplication1.csproj -c Release -o /app

# 容器启动时要运行的命令，在构造镜像时并不运行

CMD ["executable","param1","param2"]

# 容器启动时的默认命令，只能有一个

ENTRYPOINT ["dotnet", "WebApplication1.dll"]

# 定义环境变量，并在容器运行时保持。

ENV <key> <value>

### 导入导出镜像

因网络原因下载大的镜像非常耗时，通常把已下载的本地镜像导出为文件保存，需要时再将文件导入为镜像。

将镜像导出为文件的命令：

docker save -o file.tar img:tag

在powershell下不要使用含 > 的命令

如：docker save -o aspnet-2.1.tar mcr.microsoft.com/dotnet/core/aspnet:2.1-stretch-slim

将文件导入为镜像的命令：

docker load –i file.tar

如：docker load –i aspnet-2.1.tar

## Kubernetes



### 核心概念

**Pod：**最小部署单元，一个Pod有一个或多个容器组成，Pod中容器共享存储和网络，在同一台Host主机上运行。

**Service：**是真实应用服务的抽象，定义了Pod逻辑集合和访问这个Pod集合的策略。对外表现为一个单一访问入口，分配一个集群IP地址，来自这个IP的请求将负载均衡转发后端Pod中的容器，通过Lable Selector选择一组Pod提供服务。

**Volume：**数据卷，共享Pod中容器使用的数据。

**Lable：**键/值对标签，用于区分对象（比如Pod、Service），每个对象可以有多个标签，通过标签关联对象。

**ReplicaSet：**下一代Replication Controller。确保任何给定时间指定的Pod副本数量，并提供声明式更新等功能。RC与RS唯一区别就是lable selector支持不同，RS支持新的基于集合的标签，RC仅支持基于等式的标签。

**Deployment：**一个更高层次的API对象，它管理ReplicaSets和Pod，并提供声明式更新功能。官方建议使用Deployment管理ReplicaSets，而不是直接使用ReplicaSets，这就意味着可能永远不需要直接操作ReplicaSet对象。

**DaemonSet：**确保所有（或一些）节点运行同一个Pod。当节点加入Kubernetes集群中，Pod会被调度到该节点上运行，当节点从集群中移除时，DaemonSet的Pod会被删除。删除DaemonSet会清理它所有创建的Pod。如在每个 Node 上运行日志收集 daemon，在每个 Node 上运行监控 daemon。

**StatefulSet：**RS、Deployment、DaemonSet都是面向无状态的服务，它们所管理的Pod的IP、名字，启停顺序等都是随机的，而StatefulSet是有状态的集合，管理所有有状态的服务，比如MySQL、MongoDB集群等。有唯一的网络标识符（IP），持久存储，有序的部署、扩展、删除和滚动更新。

### 架构及组件



**kubectl：**Command-line interface命令行接口。

**Etcd**：分布式键值存储系统，用于保持集群状态，比如Pod、Service等对象信息。

**Master组件**

**kube-apiserver：**集群的统一入口，各组件协调者，提供基于HTTP或者HTTPS的RESTful API，如kubectl, the scheduler, RS, 和etcd 数据存储，及运行在kubernetes nodes上的kubelet 和kube-proxy。

**kube-controller-manager：**处理集群中常规后台任务，如管理kubernetes nodes，创建和更新kubernetes内部信息。

**kube-scheduler：**根据调度算法为新创建的Pod选择一个Node节点。

**Node组件**

**Kubelet：**是Master在Node节点上的Agent，管理本机运行容器的生命周期，比如创建容器、设置容器的环境变量、给容器绑定 Volume和Port、Pod挂载数据卷、获取容器和节点状态、在容器中运行命令等工作。

**kube-proxy：**为外部网络能够访问跨机器集群中的容器，每创建一种 Service，Proxy 根据Services 和 Endpoints 的配置信息，在 Node 上启动一个 Proxy 的进程并监听相应的服务端口，当外部请求发生时，Proxy 会根据 Load Balancer 将请求分发到后端正确的容器处理。

**docker：**运行容器。

### 外部访问

从外部网络访问 K8s集群中 Pod的几种常见方式：hostNetwork、hostPort、ClusterIP、NodePort、LoadBalancer、Ingress。

1直接Pod方式，hostNetwork、hostPort是直接暴露Pod的方式，如果在 Pod 中配置 hostNetwork:true的话，容器和宿主在同一网络环境，外部通过宿主机IP:containerPort访问Pod；hostPort 是直接将容器的端口与所调度的节点上的端口路由，外部通过宿主机IP:hostPort访问Pod。因Pod需要动态创建、销毁、漂移等操作，宿主机IP是不固定的，所以此方法不适合在生成环境中使用。

2暴露服务方式，服务有三种类型：ClusterIP、NodePort 、LoadBalancer，ClusterIP只可集群内访问，通过ClusterIP:port访问，为service的默认类型；NodePort支持集群外访问服务，通过宿主机IP:nodePort访问，因占用宿主机端口，为避免重复，规定范围为30000-32767；LoadBlancer也支持暴露到集群外部，需要特定的云服务商支持。ClusterIP和NodePort类型都需要经过节点的反向代理 Kube-Proxy 流入后端Pod里容器的端口，从而到达容器内。

3 Ingress方式，适合在生成环境中使用，Ingress是外部流量的入口，实现反向代理、负载均衡、自定义路由规则转发、暴露服务等功能，包含两部分：Ingress Controller、Ingress，常用的组件有ingress-nginx、Traefik等，如下图：



平台选用Traefik，它用Go开发，虽然相比于Nginx是后起之秀，但是它天然拥抱kubernetes，直接与集群的Api Server通信，反应非常迅速，实时感知集群中Ingress定义的路由规则集合和后端Service、Pod的变化，自动热更新Traefik后端配置，根本不用创建Ingress controller对象，同时还提供了友好的控制面板和监控界面，如图：



### 内部访问

k8s为内部之间互相访问提供一致的 DNS 服务，每个 service 都会有对应的 DNS 记录，格式如下：

<service\_name>.<namespace>.svc.<domain>

每个部分的内容：

service\_name: 服务名称，就是定义 service 的时候取的名字

namespace：service 所在 namespace 的名字

domain：提供的域名后缀，比如默认的 cluster.local

在 pod 中可以通过 service\_name.namespace.svc.domain 来访问任何的服务，也可以使用缩写 service\_name.namespace，如果 pod 和 service 在同一个 namespace，甚至可以直接使用 service\_name。

### Yaml文件

Yaml是一种直观的数据序列化格式，比json更方便。K8s中使用Yaml带来的好处：不必添加大量的参数到命令行中执行命令、可以创建比命令行更加复杂的结构。语法规则：

大小写敏感；

使用缩进表示层级关系；

缩进时不允许使用Tal键，只允许使用空格；

缩进的空格数目不重要，只要相同层级的元素左侧对齐即可；

#表示注释，从这个字符一直到行尾，都会被解析器忽略；

K8s的Yaml格式中包含以下公共内容：

apiVersion 版本号，不是写死的

kind 资源类型可以是Pod、Service、Ingress、Deployment等

Metadata 元数据信息，如name、labels、namespace等信息

Spe 详细定义内容

以下按照资源类型列出Yaml格式

### Pod

apiVersion: v1

kind: Pod

metadata:

name: myapp

labels:

app: myapp

spec:

containers: #必选，Pod中容器列表

- name: myapp #必选，容器名称

image: #必选，容器的镜像名称

imagePullPolicy: [Always | Never | IfNotPresent]

#默认为IfNotPresent，但:latest标签的镜像默认为Always，

#Always：不管镜像是否存在都会进行一次拉取，

#Never表示仅使用本地镜像，

#IfNotPresent表示优先使用本地镜像，否则下载镜像

resources:

limits:

memory: "128Mi" #内存限制，单位可以为Mib/Gib，将用于docker run --memory参数

cpu: "500m" #Cpu的限制，单位为core数，将用于docker run --cpu-shares参数

ports: #需要暴露的端口号列表

- containerPort: 80 #容器需要监听的端口号

hostPort: int #容器所在主机需要监听的端口号，默认为虚拟端口号，和容器端口号相同，只集群内部用！

### Deployment

apiVersion: apps/v1

kind: Deployment

metadata:

name: dtapp

spec:

replicas: 3 #副本数量，默认为 1

selector: #圈定Deployment管理的pod范围

matchLabels:

app: dtapp #此标签和Pod模板内的标签相同，且不要和其它Deploy, Pod等的定义重复，应用后不要修改！

template: #更新Pod的目标属性

metadata:

labels:

app: dtapp

spec:

containers:

- name: dtapp

image: <Image>

imagePullPolicy: Never

ports:

- containerPort: 80

livenessProbe: #存活探针，定时检查容器内应用的运行状态，在Pod有问题的时候reload Pod

httpGet: #向容器内应用的80端口发送HTTP GET请求，返回码大于200小于400为应用运行正常，其他码都为失败

path: /

port: http

initialDelaySeconds: 5 #第一次执行probe之前要的等待5秒钟

periodSeconds: 5 #每隔5秒执行一次存活探针

readinessProbe: #就绪探针，确定容器是否已经就绪可以接受流量，规则同存活探针

httpGet:

path: /

port: http

initialDelaySeconds: 5 #第一次执行probe之前要的等待5秒钟

periodSeconds: 5 #每隔5秒执行一次探针

resources:

limits:

memory: "128Mi"

cpu: "500m"

Deployment是最常使用的部署方式，其yaml定义格式参见以上，初次执行yaml文件时以name为标识创建deployment，输出为deployment.apps "nginx-deployment" created，修改完yaml再次apply时，输出为deployment.apps "nginx-deployment" configured，只有在Pod模板内容更新时触发重新上线(rollout)，只修改副本数量时不触发，k8s通过name识别deployment，通过selector的matchLabels和Pod模板中的labels识别Pod，这三处的内容在整个生命周期不要修改！

### Service

apiVersion: v1

kind: Service

metadata:

name: myapp

spec:

selector: #选择具有指定Label标签的Pod作为管理范围

app: myapp

type:[ ClusterIP | NodePort | LoadBalancer] #默认ClusterIP

ports:

- port: <Port> #集群内部服务之间访问的虚拟端口号， clusterIP:port

targetPort: #Pod里容器的端口号，默认和port相同

nodePort: #当type=NodePort时有效，可从集群外部通过nodeIP:nodePort访问，规定范围为30000-32767

clusterIP: None # Headless Service无头服务，因为没有ClusterIP，不支持负载均衡或代理，主要与StatefulSet配合

### Ingress

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: cm

annotations:

kubernetes.io/ingress.class: "traefik"

ingress.kubernetes.io/ssl-redirect: "false"

traefik.frontend.rule.type: # PathPrefix 或 PathPrefixStrip，默认PathPrefix会传递配置中path内容到后端服务，系统中使用PathPrefixStrip，不然会出现404错误！

traefik.ingress.kubernetes.io/frontend-entry-points: # http或https或同时支持，默认http

traefik.ingress.kubernetes.io/priority: # 优先级，无特殊需求不需要指定

spec:

rules:

- host: dt.com

http:

paths:

- path: /cm #访问路径，标签中设置PathPrefixStrip时不将路径传递到后端服务

backend:

serviceName: cm #服务名称

servicePort: 80 #服务内的port值，集群内部服务之间访问的虚拟端口号

### ConfigMap

apiVersion: v1

kind: ConfigMap

metadata:

name: special-config

data:

special.how: very

special.type: charm

通过环境变量的方式直接传递到pod

apiVersion: v1

kind: Pod

metadata:

name: dapi-test-pod

spec:

containers:

- name: test-container

image: gcr.io/google\_containers/busybox

env:

- name: SPECIAL\_LEVEL\_KEY

valueFrom:

configMapKeyRef:

name: special-config

key: special.how

使用volume的方式挂载入到pod内，挂载至pod的/etc/config目录下

apiVersion: v1

kind: Pod

metadata:

name: dapi-test-pod

spec:

containers:

- name: test-container

image: gcr.io/google\_containers/busybox

command: [ "/bin/sh", "-c", "cat /etc/config/special.how" ]

volumeMounts:

- name: config-volume

mountPath: /etc/config

volumes:

- name: config-volume

configMap:

name: special-config

当ConfigMap以数据卷的形式挂载进Pod时，会覆盖容器中相同的目录，更新ConfigMap（或删掉重建ConfigMap），Pod内挂载的配置信息会热更新，但使用环境变量方式加载到pod，则不会自动更新。

### StatefulSet

apiVersion: v1

kind: Service

metadata:

name: nginx

labels:

app: nginx

spec:

ports:

- port: 80

name: web

clusterIP: None # Headless Service无头服务

selector:

app: nginx

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

selector:

matchLabels:

app: nginx #和以下 .spec.template.metadata.labels 匹配

serviceName: "nginx" #声明它属于哪个Headless Service.

replicas: 3 #默认1

template:

metadata:

labels:

app: nginx #和以上 .spec.selector.matchLabels 匹配

spec:

terminationGracePeriodSeconds: 10

containers:

- name: nginx

image: k8s.gcr.io/nginx-slim:0.8

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumeClaimTemplates: #可看作pvc的模板

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: "gluster-heketi" #存储类名，改为集群中已存在的

resources:

requests:

storage: 1Gi

也可以提前手动创建PVC，手动创建的PVC名称必须符合之后创建的StatefulSet命名规则：(volumeClaimTemplates.name)-(pod\_name)，如上，Statefulset名称为web，三个Pod副本: web-0，web-1,web-2，volumeClaimTemplates名称为：www，那么自动创建出来的PVC名称为www-web-0，www-web-1，www-web-2，为每个Pod创建一个PVC。Pod重新调度后其PodName和HostName不变，还能访问到相同的持久化数据，命名规则：

* Pod命名规则：(statefulset名称)-(序号)
* 访问Pod的DNS域名：(podname).(headless service name)
* pvc的命名规则：(volumeClaimTemplates.name)-(pod\_name)，删除Pod不会删除其pvc，只能手动删除

Statefulset的启停顺序：

* 有序部署：部署StatefulSet时，如果有多个Pod副本，它们会被顺序地创建（从0到N-1）并且，在下一个Pod运行之前所有之前的Pod必须都是Running和Ready状态。
* 有序删除：当Pod被删除时，它们被终止的顺序是从N-1到0。
* 有序扩展：当对Pod执行扩展操作时，与部署一样，它前面的Pod必须都处于Running和Ready状态。

StatefulSet一般不暴露到外网，能不用StatefulSet，就不要用。





### 数据持久化

可以将数据挂载到宿主机，但是pod重启之后有可能到另外一个节点，这样数据虽然不会丢但还是有可能会找不到。一般采用如下方法：

* 创建一个卷PV，不属于任何namespaces，可以限制大小，读写权限

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv0001

labels:

app: "my-nfs"

spec:

capacity:

storage: 5Gi

accessModes:

- ReadWriteMany

persistentVolumeReclaimPolicy: Recycle

nfs:

path: "/data/disk1"

server: 192.168.20.47

readOnly: false

* 在对应的namespace下面创建PVC。apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: nfs-pvc

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 1Gi

selector:

matchLabels:

app: "my-nfs"

* 最后在应用用使用该PVC，这样可以方便的限制每个pvc所在的子目录，同时万一nfs迁移后，只需要更改pv中的url即可

apiVersion: v1

kind: Pod

metadata:

name: test-nfs-pvc

labels:

name: test-nfs-pvc

spec:

containers:

- name: test-nfs-pvc

image: registry:5000/back\_demon:1.0

ports:

- name: backdemon

containerPort: 80

command:

- /run.sh

volumeMounts:

- name: nfs-vol

mountPath: /home/laizy/test/nfs-pvc

volumes:

- name: nfs-vol

persistentVolumeClaim:

claimName: nfs-pvc

### 常用命令

kubectl get [–o json|yaml|wide] TYPE[/NAME]

获取一个或多个资源的信息，常用TYPE有：pods,svc(services),deploy(deployments),nodes，可以后面跟Name进行具体查询。

kubectl describe (TYPE [NAME\_PREFIX | -l label] | TYPE/NAME)

输出指定的一个/多个资源的详细信息，TYPE同get，Name可以是前缀或精确匹配

kubectl proxy [--address='10.10.1.18' --accept-hosts='^\*$']

API server在本地8001端口上监听，可以指定监听的IP，'^\*$'表示接收所有

kubectl apply -f deploy\_nginx.yaml

创建或更新yaml文件中的资源

kubectl delete -f deploy\_nginx.yaml

删除yaml文件中描述的资源

与deploy相关的常用命令：

kubectl rollout undo deploy/nginx-deployment --to-revision=2

回滚到指定版本，无--to-revision时回滚到上次

kubectl get deploy

查看deployment列表

kubectl rollout status deploy/nginx-deployment

查看” nginx-deployment”的上线状态

kubectl get rs

查看副本列表

kubectl describe deploy nginx-deployment

查看” nginx-deployment”的内容及副本缩放日志

kubectl rollout history deploy/nginx-deployment

查看” nginx-deployment”的所有历史版本

kubectl rollout history deploy/nginx-deployment --revision=2

查看版本2的详细内容

## Helm

### 基本概念

Helm是K8s的一个包管理工具，用来简化K8s应用的部署和管理。采用客户端/服务器架构，客户端helm和服务端tiller，helm客户端是一个命令行工具，可在本地运行，tiller运行在K8s集群上，管理chart安装的release。三个基本概念：

* Chart: 是Helm管理的安装包，采用tar压缩格式，里面包含需要部署的安装包资源。每个Chart包含下面两部分：包的基本描述文件Chart.yaml、放在templates目录中的一个或多个Kubernetes manifest文件模板。Dt服务的Chart.Name命名规则为：AppName-SvcName。
* Release：是chart的部署实例，一个chart在一个K8s集群上可以有多个release，即这个chart可以被安装多次。Dt服务部署时一个chart只被部署一次，Release.Name=Chart.Name，没有部署多次的需求。
* Repository：chart的仓库，用于发布和存储chart。

### 编写Chart

为服务创建初始文件：helm create svcName

目录结构如下：

svcName

Chart.yaml # Yaml文件，用于描述Chart的基本信息，包括名称版本等

values.yaml # Yaml文件，定义模板中需要引用的值

chart # [可选]: 该目录中放置当前Chart依赖的其它Chart

templates # [可选]: 部署文件模版目录，模版使用的值来自values.yaml和由Tiller提供的值

deployment.yaml # 部署，Go模板语法

service.yaml # 服务，Go模板语法

ingress.yaml # 反向代理，Go模板语法

\_helpers.tpl # 定义子模板

Chart.yaml内容

apiVersion: v1

appVersion: "1.0"

description: [可选] Chart的简要描述

name: [必须] Chart的名称

version: [必须] Chart的版本号，版本格式：主版本号.次版本号.修订号，必须符合：<http://semver.org/>

Go模板语法

Go语言模板其实就是具有一些变量和函数的文本文件，Go的变量和函数包含在{{ }}中

定义变量：{{ $article := "hello" }}

引用变量：{{ $article }}

调用函数，使用空格来分隔参数：{{ add 1 2 }}

当前上下文：.

通过.符号访问方法和字段：{{ .Params.bar }}

管道：|，每个管道的输出会作为下一个管道的输入，只可传输单一的值

条件判断，有三种值视为false：bool类型的false、0、任何长度为0的array, slice, map或string，if可以用with替代

{{if .condition1}}

{{else if .contition2}}

{{end}}

定义子模板

\_helpers.tpl中定义了一些需要复用的子模板，如下定义了名称为test.name的子模板：

{{- define "test.name" -}}

{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" -}}

{{- end -}}

引用子模板

{{ include "test.name" . }}

{{- template "test.name" . }}

template方式是直接用子模板内容替换，include是一个引用模板的函数，可以指定缩进，缩进在yaml中非常重要如引用test.name且每一行缩进４字符：

{{ include "test.name" . | indent 4 }}

### 常用命令

Chart Install 过程



Helm 从指定的目录或者 TAR 文件中解析出 Chart 结构信息。

Helm 将指定的 Chart 结构和 Values 信息通过 gRPC 传递给 Tiller。

Tiller 根据 Chart 和 Values 生成一个 Release。

Tiller 将 Release 发送给 Kubernetes 用于生成 Release。

Release.Name为唯一值，”Release.Name-Chart.Name”作为deployment、service、ingress的Name。

## 环境安装

### Docker

安装Hyper-V。

运行Docker for Windows Installer.exe，选择linux容器，安装后在Hyper-V中自动创建MobyLinuxVM虚拟机。

Docker for Windows的linux容器版，实质是在Hyper-V安装定制版linux的虚拟机MobyLinuxVM，IP为10.0.75.2，windows本机IP为10.0.75.1，所有的镜像及容器都在虚拟机中，windows中的docker命令是docker client模块，通过管道方式和linux中的Docker daemon通信。内部容器的IP自动分配为172.17.0.X，掩码255.255.0.0，可以直接连接到宿主windows所在的网络。虚拟机文件位置C:\Users\Public\Documents\Hyper-V\Virtual Hard Disks。

托盘->Docker->Settings->Daemon设置Docker registry为国内地址(<https://registry.docker-cn.com>)，否则pull镜像时速度太慢。Settings->Shared Drives设置共享用到的分区，如：C, D



安装Kitematic，一个优秀的 Docker 可视化管理工具。初次点击时下载程序包，将Kitematic-Windows.zip解压到C:\Program Files\Docker\Kitematic 目录即可。

### Kubernetes

安装单节点K8s集群，K8s以容器方式运行在Docker内，可通过K8s部署服务，即使K8s不可用其部署的独立容器仍然有效。安装过程需要下载镜像文件和kubectl.exe，因无法连接google服务，先使用阿里云下载镜像文件，在<https://github.com/AliyunContainerService/k8s-for-docker-desktop>给出了下载的脚本，切换到需要的版本进行下载，使用powershell切换到目录，如：E:\Software\Docker\k8s-for-docker-desktop-2.0.0.2，运行命令 .\load\_images.ps1，等待镜像文件下载完毕，再设置K8s如下图：

若powershell禁止执行脚本先设置允许策略：set-executionpolicy remotesigned。



执行 kubectl cluster-info，输出以下，表示正常启动：

Kubernetes master is running at https://localhost:6445

KubeDNS is running at https://localhost:6445/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

### Dashboard

Dashboard是k8s集群的Web UI，集合了所有可以操作的命令，访问Dashboard有四种方式：NodePort、API Server、kubectl proxy、Ingress，本地开发时使用NodePort方式，省去启动代理和麻烦的登录过程。

* NodePort方式（开发环境）

修改官方的kubernetes-dashboard.yaml文件，Service类型为NodePort，对外开放一个31000的nodePort端口，实现基于ip:31000的方式来访问。

1 修改dashboard.yaml文件如下：

2安装

kubectl apply -f dashboard.yaml

3 访问<http://localhost:31000/>

* kubectl proxy方式

1 默认安装

kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/recommended/kubernetes-dashboard.yaml

2 启动API Server监听8001

kubectl proxy

要在固定IP上监听并可集群外访问

kubectl proxy --address='10.10.1.18' --accept-hosts='^\*$'

3 打开furl，出现登录页面

<http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/>

4获取token

创建admin

kubectl apply -f dashboard-admin.yaml

查询账户信息，复制Tokens值

kubectl describe ServiceAccount admin -n kube-system

查询Tokens内容，admin-token-kjx9j替换为上面查到的结果

kubectl describe secret admin-token-kjx9j -n kube-system



复制token到上述网页中



查看创建的pod

kubectl get pods -n kube-system

查看创建的service

kubectl describe svc kubernetes-dashboard -n kube-system

在Docker中查看k8s已运行的容器时，需要在设置中开启显示系统容器，如图：



### Traefik

平台选用Traefik 作为Ingress组件，它是外部流量的入口，实现反向代理、负载均衡、自定义路由规则转发、暴露服务等功能。由于客户端与服务之间、服务与服务之间基于http2通信，Traefik需要配置成https，安装需要的文件有：证书tls.key和tls.crt（生成过程见后）、配置文件traefik.toml、部署文件traefik.yaml，命名空间统一为kube-system。

将证书作为k8s的secret资源

kubectl create secret tls traefik-cert --key tls.key --cert tls.crt -n kube-system

将traefik.toml配置生成为k8s的configmap资源

kubectl create configmap traefik-conf --from-file=traefik.toml -n kube-system

部署traefik

kubectl apply -f traefik.yaml

安装成功后普通端口30000 <https://localhost:30000> , traefik UI端口30001 <http://localhost:30001>

traefik.toml内容

# 访问https后端时可以忽略TLS证书验证错误，从而使得后端像http一样直接通过 "traefik" 透出

insecureSkipVerify = true

defaultEntryPoints = ["http","https"]

[entryPoints]

[entryPoints.http]

address = ":80"

[entryPoints.http.redirect]

entryPoint = "https"

[entryPoints.https]

address = ":443"

[entryPoints.https.tls]

[[entryPoints.https.tls.certificates]]

# 部署时两文件装载到ssl目录

certFile = "/ssl/tls.crt"

keyFile = "/ssl/tls.key"

生成https证书，默认情况下，ingress只提供了http服务，而没有https服务，要部署一个https服务，首先得有https证书，生成证书需要在windows上安装Win64OpenSSL\_Light-1\_1\_1b.exe，然后cd到openssl.exe所在的目录：

生成私钥tls.key, 密钥位数是 2048

openssl genrsa -out tls.key 2048

使用server.key 生成自签证书，域名localhost，有效期10年

openssl req -new -x509 -days 3650 -key tls.key -out tls.crt -subj /CN=localhost

生成tls.key, tls.crt文件用于以上secret资源，crt存储公钥，key存储私钥

将私钥和公钥合并成pfx证书，密码为dt，用于服务的x509认证

openssl pkcs12 -export -in tls.crt -inkey tls.key -out tls.pfx

### Helm

安装客户端：从<https://github.com/helm/helm/releases>下载2.12.3版，zip在google服务器可能无法下载，将压缩包的两个exe释放到C:\Program Files\Docker\Docker\resources\bin，和docker.exe同目录省去配置Path，运行helm version查看版本号。

安装服务器端：

1创建服务端

helm init --upgrade -i registry.cn-hangzhou.aliyuncs.com/google\_containers/tiller:v2.12.3 --stable-repo-url <https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts>

2因Tiller需要在k8s中创建和删除应用，必须授权

kubectl apply -f rbac-config.yaml

3查看安装是否正确

helm version

kubectl -n kube-system get pod

### RabbitMQ

安装otp\_win64\_22.0.exe，新建环境变量：ERLANG\_HOME=C:\Program Files\erl10.4，添加到PATH：%ERLANG\_HOME%\bin。

安装rabbitmq-server-3.7.15.exe，新建环境变量：RABBITMQ\_SERVER=C:\Program Files\RabbitMQ Server\rabbitmq\_server-3.7.15，添加到PATH：%RABBITMQ\_SERVER%\sbin，RabbitMQ已作为服务运行。

安装web管理插件C:\Program Files\RabbitMQ Server\rabbitmq\_server-3.7.15\sbin\rabbitmq-plugins.bat enable rabbitmq\_management，默认地址：http://localhost:15672，账号密码都是guest。

防火墙允许5672, 15672通过。

常用命令：

rabbitmqctl list\_users 查看已有用户及用户的角色

rabbitmqctl add\_user <username> <password> 添加一个用户

rabbitmqctl delete\_user <username>删除一个用户，只需指定用户名

rabbitmqctl set\_user\_tags <username> <tag> 设置用户角色，执行命令时会清空原有角色，角色可以设置多个或零个

rabbitmqctl set\_permissions -p / <username> .\* .\* .\* 设置用户权限

rabbitmqctl list\_queues 查看所有队列

rabbitmqctl reset 清除所有队列

rabbitmqctl stop\_app 关闭应用

rabbitmqctl start\_app 启动应用

清除所有队列需要关闭应用、清除、启动应用。

# 其他

## Auth服务(废弃)

提供整个平台的认证(Authentication)、授权(Authorization)服务，基于开源中间件Identity Server4，支持单点登录注销、第三方应用登录、Api访问控制等功能，涉及两种协议OpenID Connect(简称OIDC)和OAuth2，OIDC是认证，就是证实某个用户的真实性；OAuth2是授权，决定一个认证过的用户能够访问哪些资源。

### 相关术语

* User：用户是指已经注册的人。
* Client：一个向IdentityServer请求token的客户端软件，可以是各类App或微服务，客户端在请求token前必须在IdentityServer上进行注册。
* Resources：资源是想通过IdentityServer保护的东西，如APIs，每个资源通过唯一的名字进行识别。
* Identity Token：是代表认证结果的身份令牌，包含用户id（也做sub)和一些额外身份信息。
* Access Token：是代表授权结果的访问令牌，可以携带令牌去请求已授权的资源。
* JWT：Json Web Token，是一种基于Json的Token编码方式，有三部分组成<header>.<payload>.<signature>。

### 认证授权框架

客户端采用用户名密码模式获得访问令牌，API资源服务通过验证JWT格式的Access Token保护API资源，其调用过程及框架结构如下图所示：



ASP.NET Core 采用基于声明的认证(claims-based)，框架代码参见Dt.Core项目的Authentication.txt 文件。认证准备的主要内容如下图：



认证过程如下图：



Cliam类表示用户身份中的一项信息，键/值都是字符串，ClaimsIdentity类表示一个用户身份，ClaimsPrincipal类包含一个或多个用户身份，是HttpContext的属性User，为了序列化和安全，需要生成用户票据AuthenticationTicket

### Client应用场景

GrantTypes.ClientCredentials：客户端模式，主要授权服务与服务之间直接交互访问资源的场景。

GrantTypes.ResourceOwnerPassword：密码模式，客户端发送用户名和密码获取访问令牌，为避免密码泄露仅适用于受信任的客户端

## 未使用Dapr

微软的技术迭代：Orleans => Service Fabric => Dapr，Kubernetes的普及使Service Fabric逐渐没落，2019年微软在github上发布了[Dapr](https://github.com/dapr/dapr)，详细参见[Dapr 文档库](https://docs.dapr.io/zh-hans/)。