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。

## 开发部署

开发时可采用两种模式运行服务：本地模式、k8s模式。本地模式节省资源、调试速度快、可以全程调试；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。生产环境一般采用k8s模式。

### 本地模式

本地模式下开发工具使用VS，服务直接以宿主方式运行，监听端口固定，默认监听协议为http2。单个服务调试时可访问<https://localhost:XXXX/.admin>、选择Api、填写完整参数值、点击调试按钮。多个服务联合调试时用Traefik做反向代理，windows版的Traefik使用方式和容器版类似，详细配置参见traefik.toml，服务对外提供的url地址形如：<https://localhost/appName/serviceName/.admin>。

### k8s模式

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BaiSui\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

### https证书

所有服务采用http2.0协议，主流浏览器要求http2.0协议必须采用https安全通信，需要在客户端将Dt.Core\Res\tls.pfx证书安装在“受信任的根证书颁发机构”，私钥密码dt。

# 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、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，内容包括数据库连接串、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)

## 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

缓存按照使用场景可分为服务端全局缓存、进程内缓存、客户端缓存，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；客户端缓存使用sqlite文件型数据库，存储静态或不常变化的信息，如各种字典表、菜单等，在服务器端生成，下载到客户端使用。

### 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。

### 客户端缓存

服务器端生成sqlite模型文件，文件中包含静态或不常变化的信息，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，不同时客户端下载最新版本。为保证服务有多个副本时版本号相同，刷新时采用远程事件的方式通知所有副本最新的版本号。

cm服务中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，一般不同类型的客户端使用不同的客户端缓存，如百岁客户端、百岁介护师端、百岁管理端分别使用各自的模型文件。

生成新模型文件只需要处理ModelRefreshEvent事件，如：



当调用SysMgr. RefreshModel刷新模型文件时，会通知所有的副本进行刷新，生成sqlite文件并压缩得到xxx.gz文件，其内容提供给客户端下载。

## 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，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。



客户端为UWP或基于mono的Android、iOS应用时，ClientStream和DuplexStream功能尚未实现，因这两种模式的应用场景非常少，基本不影响使用。



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。

## 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！

## 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，系统内部也提供9位+3位的方式，9位用于产生序列号(每毫秒最多支持512各)，3位用于存储id类型标志(0~7)，如userid采用该方式后可以根据userid可识别出用户类型。

获取新ID方法：Id.New()

获取含3位标志的新ID方法：Id.New(int p\_flag)

## 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》

## 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。



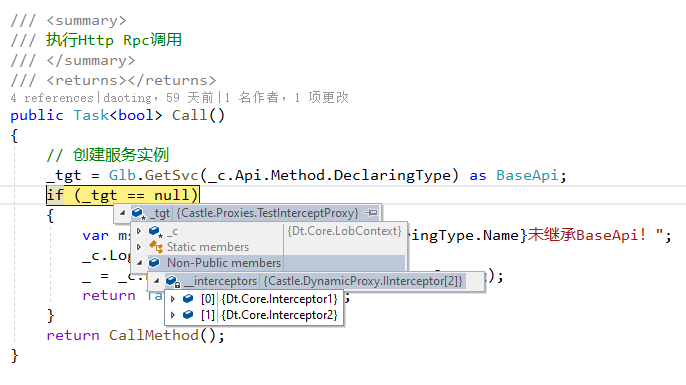
## 面向切面编程AOP

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。

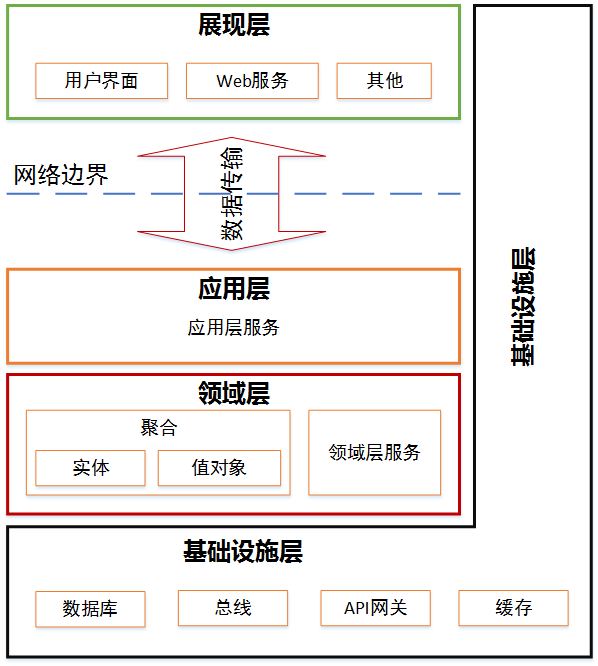
# 服务架构

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

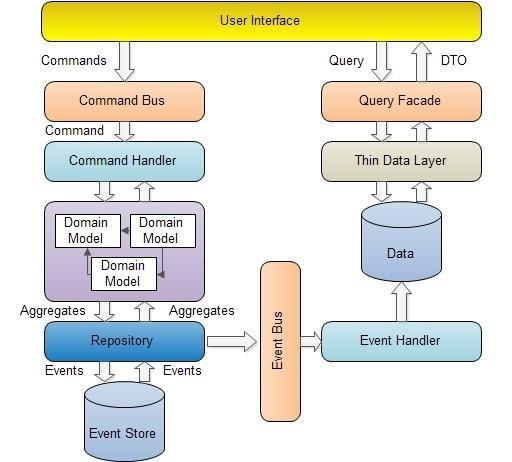
### 领域驱动设计

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。



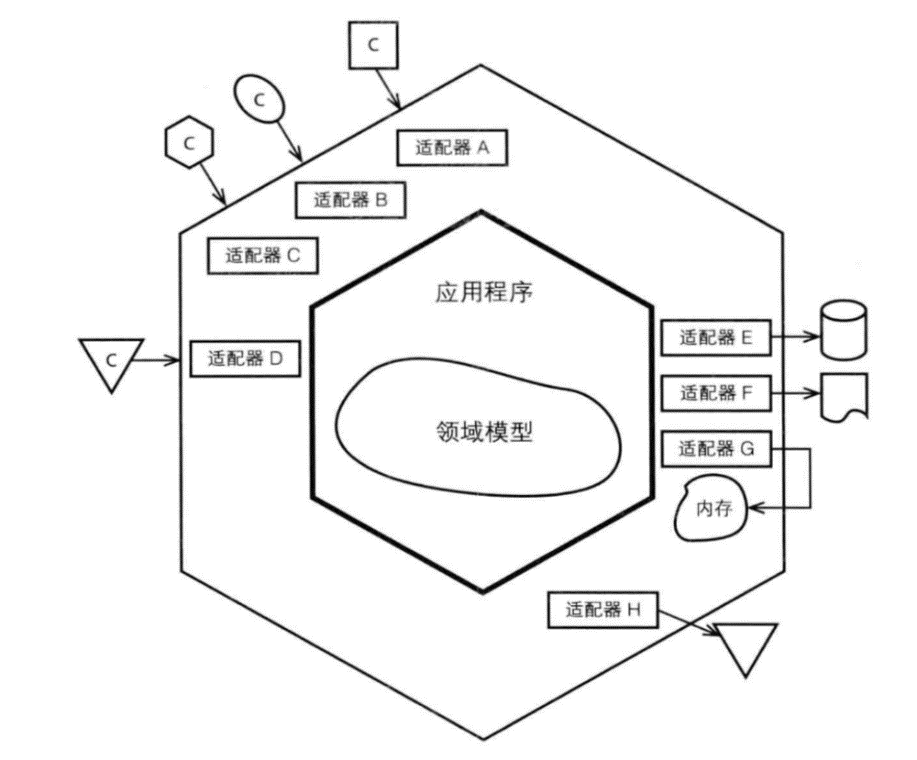
### 命令查询职责分离

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。



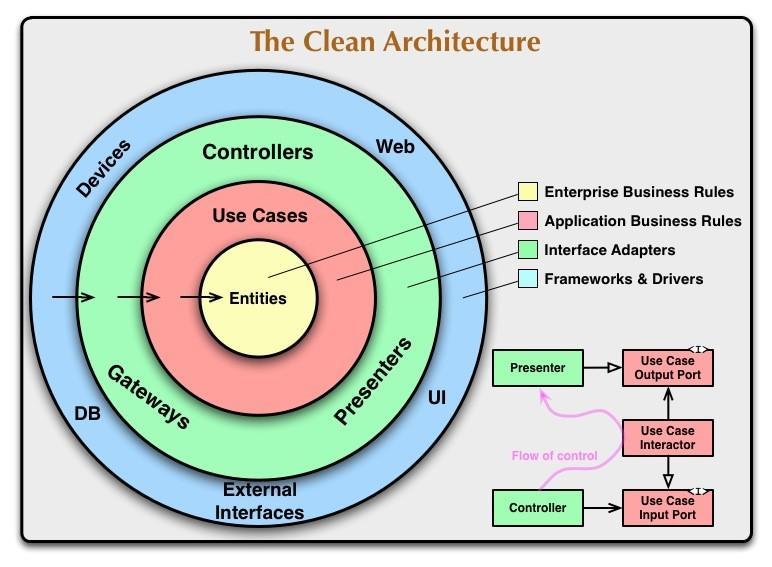
### 六边形架构

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。

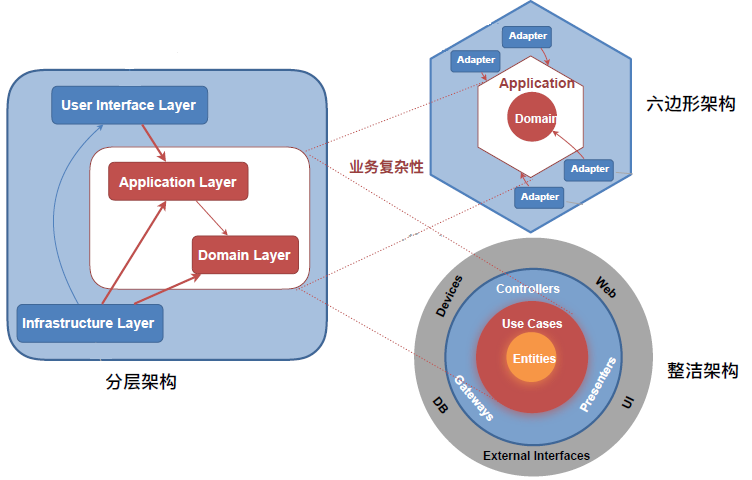


### 洋葱架构

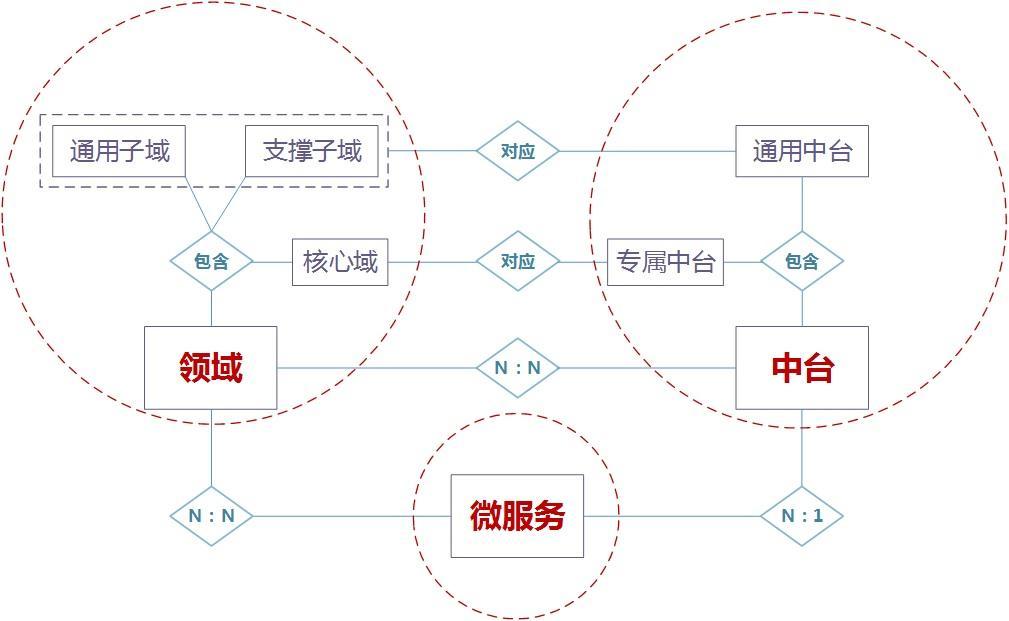
洋葱架构（又名整洁架构），同心圆代表应用软件的不同部分，从里到外依次是领域模型、领域服务、应用服务、最外围是容易变化的内容，如界面和基础设施（如数据存储等）。整洁架构是以领域模型为中心，不是以数据为中心。



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。

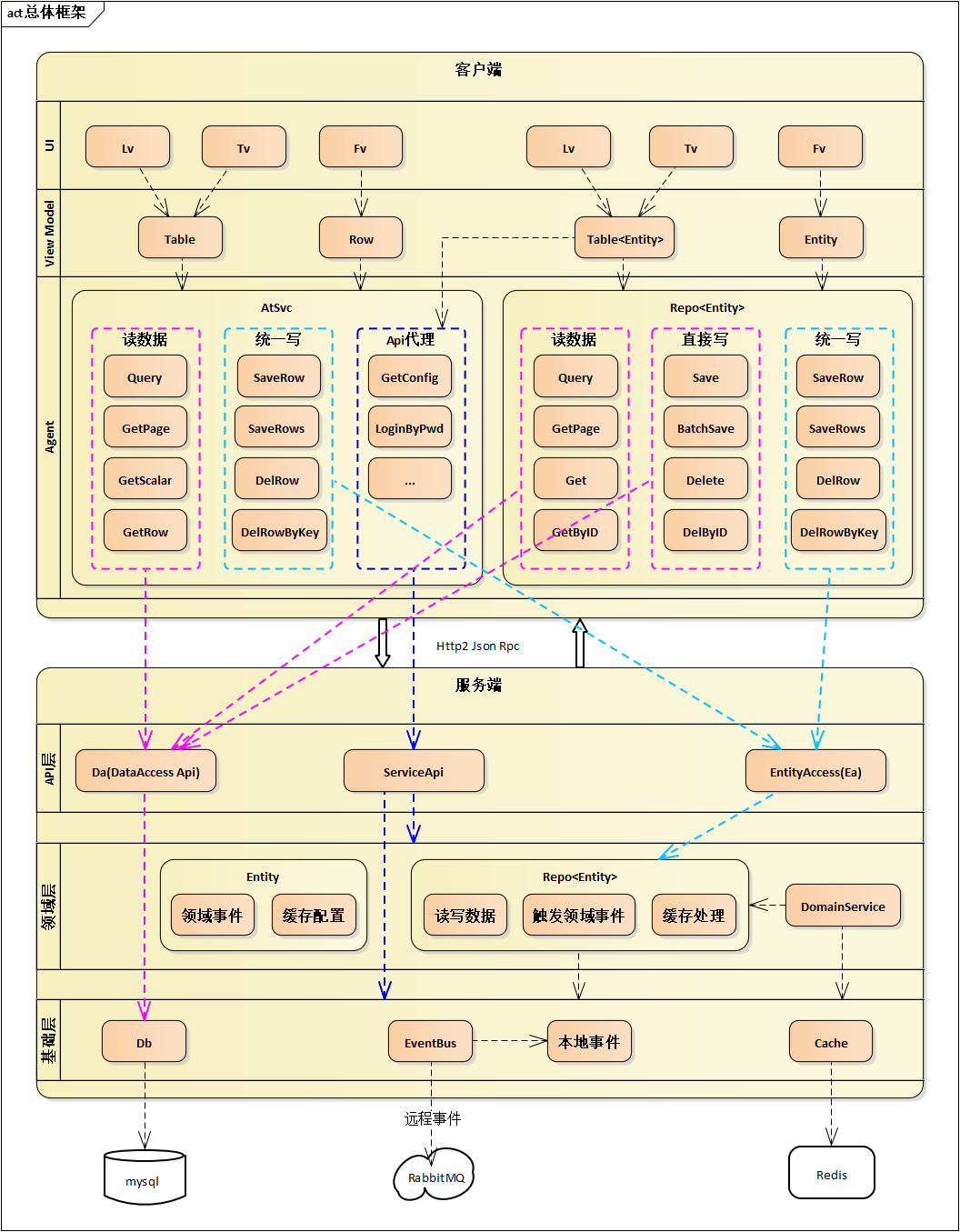


中台、领域以及微服务属于不同层面的内容，中台的本质是提炼各个业务线的共同需求，并将这些功能打造成组件化产品，然后以API接口的形式提供给前台各业务部门使用。领域的本质是问题域，问题域可能根据需要逐层细分，因此领域可分解为子域，子域或可继续分为子子域。微服务是技术实现和部署的范畴，实现领域或中台的业务逻辑，为前台应用提供服务。



## 总体架构

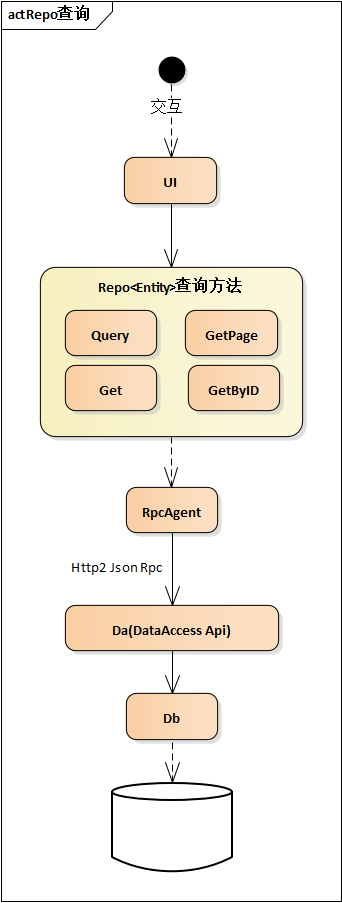
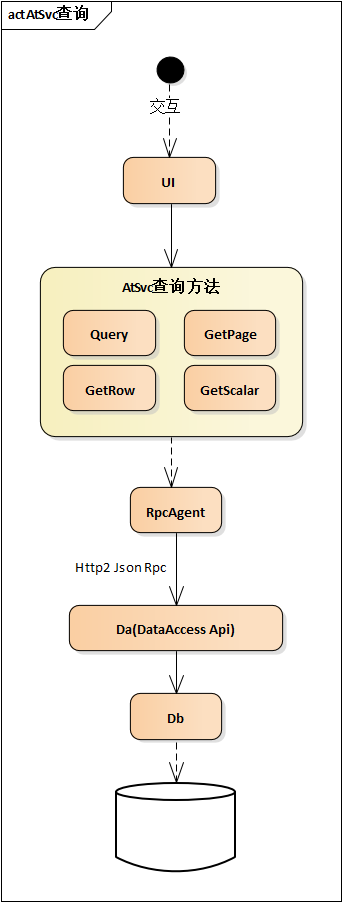
本系统基于DDD模式，吸取CQRS读写分离的优势，未严格按照以上某种具体的服务架构进行设计，而是结合自身整体架构的特点（比如增加客户端处理功能等），完美实现技术与业务解耦、业务逻辑解耦、业务封装复用等关键功能，为开发人员提供一种易懂易用的框架风格。总体框架图：



## 典型路径

### 读数据

借鉴CQRS读写分离的原则，客户端数据查询采用直接DataAccess方式，既提高查询速度，也和领域相关内容解耦，有两种查询方式，过程如下：

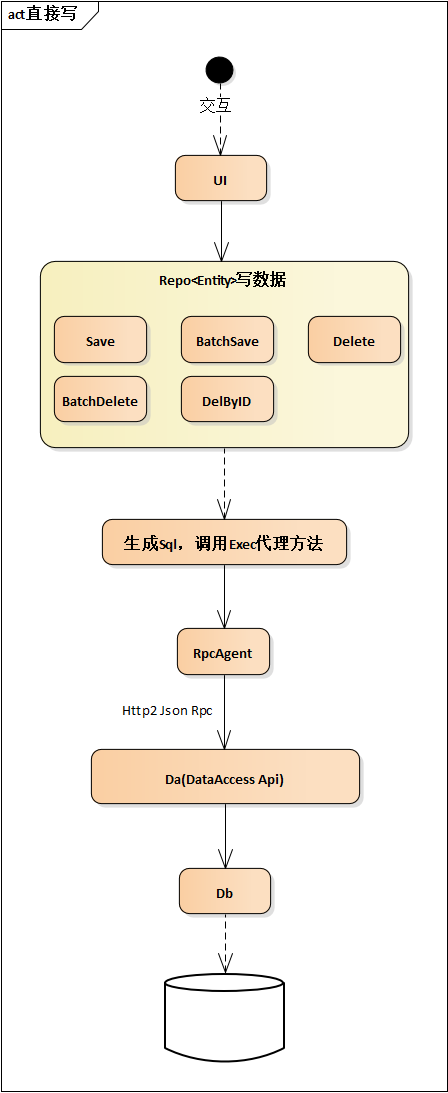
 

Repo<Entity>提供的查询方法和AtSvc中提供的查询方法经过Rpc代理，调用相同的服务器端Api，不同之处在于Repo<Entity>返回的结果为Entity或Table<Entity>，而AtSvc返回的为Row或Table。查询方法定义如下：

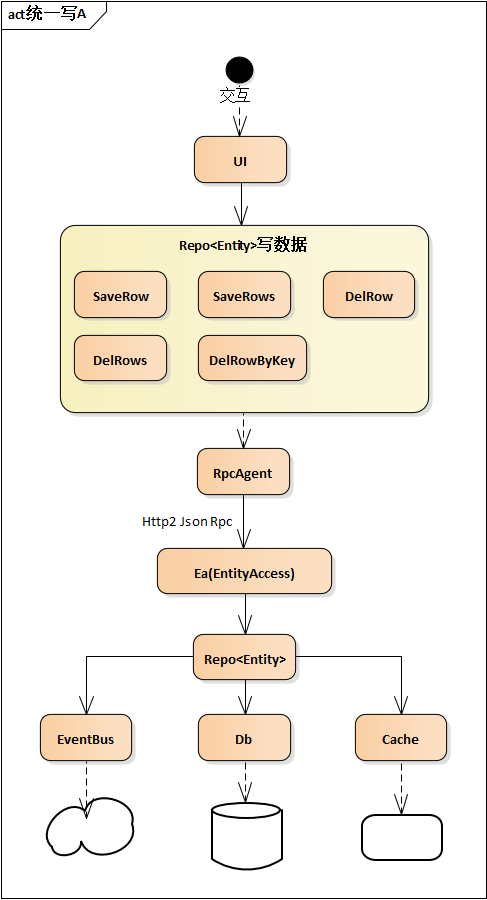
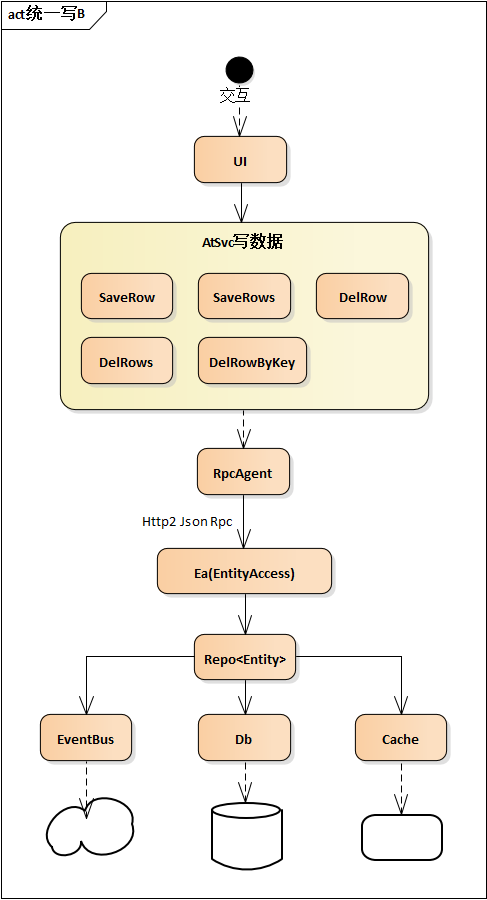


### 写数据

系统提供三种写数据方式，第一种为客户端直接写模式，通过Repo<Entity>的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，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，过程如下：



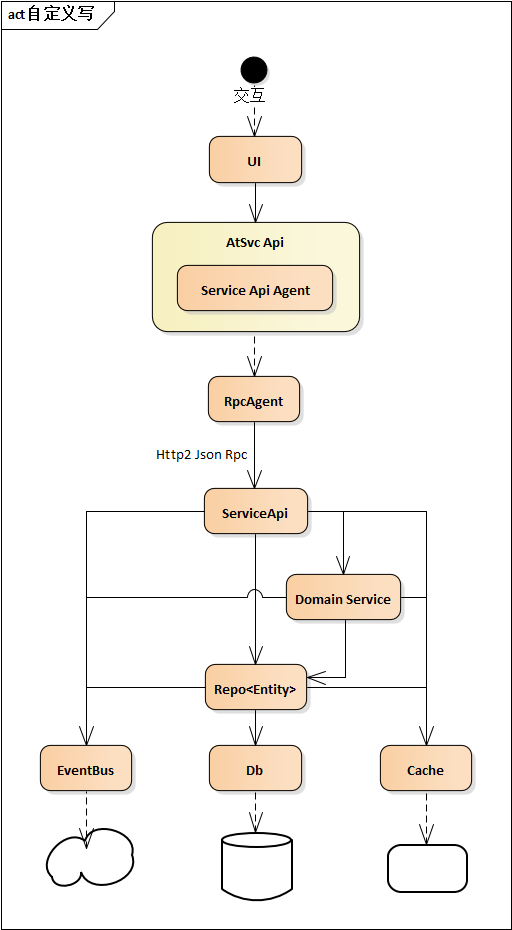
第二种为统一写模式，就是将数据直接传递到服务端，调用EntityAccess Api，Api内部再通过领域层的Repo写入数据，该模式利用了领域层的优势，可以触发领域事件或进行缓存处理，过程如下：

以上两种方式实际过程相同，只是为了书写方便，不同之处在于Repo方法不需提供实体对应的表名，AtSvc的静态方法需要提供。方法定义如下：



第三种为自定义写模式，就是普通的Api调用，将数据及相关参数提交给Api，按照DDD分层模式写入数据，该模式为典型的领域驱动模式，过程如下：

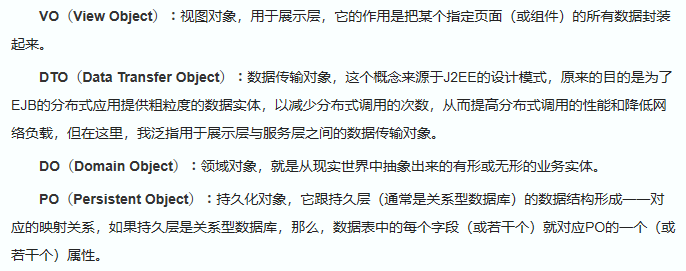


服务端Api遵循以下原则：业务方法与事务和业务用例一一对应，即每一个业务方法均构成了独立的事务边界，通过给Api方法添加Transaction标签，系统将整个方法封装到了一个事务中处理。

服务端Api层、领域层、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，Api层实际相当于领域层的领域服务，而领域服务只是从Api提取的可复用的公共功能，所以Api和领域服务对领域层和基础层的功能都可以直接调用，但Api和领域服务自身不应该包含业务逻辑，业务逻辑应该放在实体中实现，更准确的说是放在聚合根中实现，这一点严格遵循DDD的充血模式。

## 数据对象

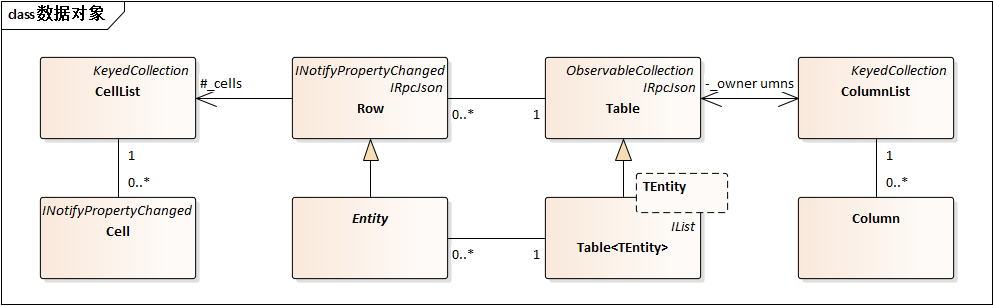
数据对象是整个系统的基础，标准的DDD设计包含多种数据对象，基本上每层一种类型，如：



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。



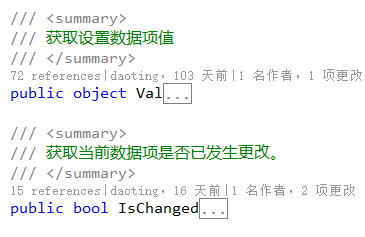
下面针对这些使用场景逐一说明：

### UI绑定

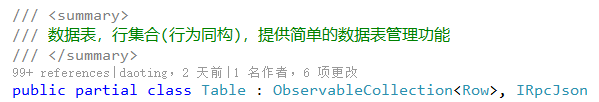
Fv的数据源为Row时，FvCell数据源对应Cell，通过Row.Cells[ID]确定，绑定路径为固定的Val，Row实现接口INotifyPropertyChanged，确保双向绑定有效，如IsChanged属性绑定到保存按钮的IsEnable属性实现保存控制。



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



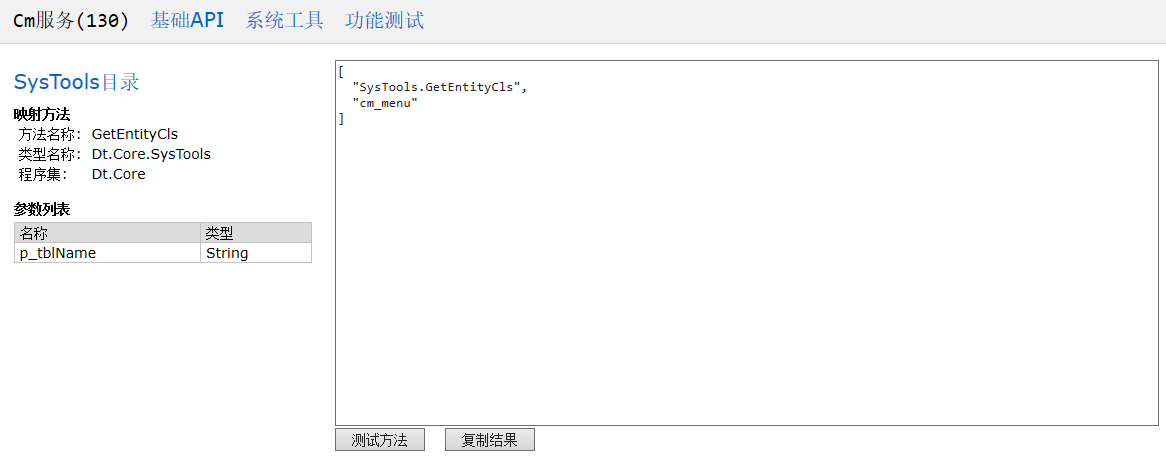
Lv或Tv的数据源为Table时，Row对应UI中的ViewItem，实际绑定到LvRow或TvPanelItem，Cell对应具体的可视元素，Table继承自ObservableCollection<Row>，能将行数的变化实时反应到UI上。



### 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。为方便管理和修改，两部分代码采取分文件管理方式。

自动生成实体代码需要提供实体类对应的表名，如下：



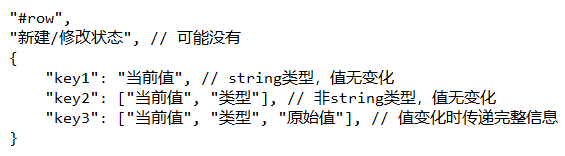
生成实体类代码如下：

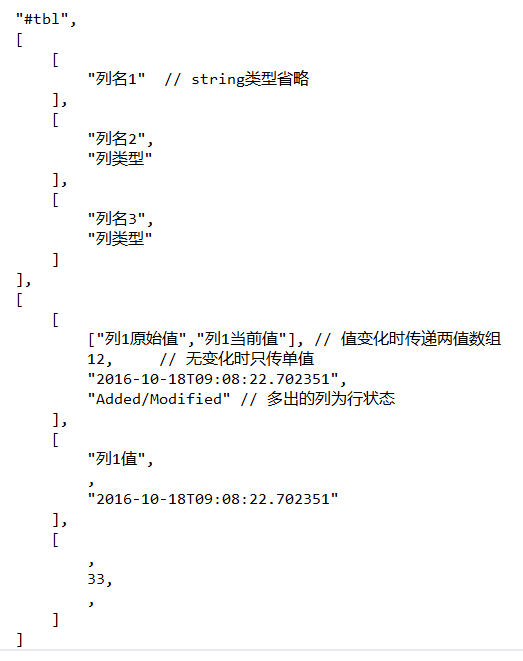


文件命名规范：生成的实体类文件名Entity.cs，业务逻辑部分EntityEx.cs，如User.cs, UserEx.cs。

### 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：





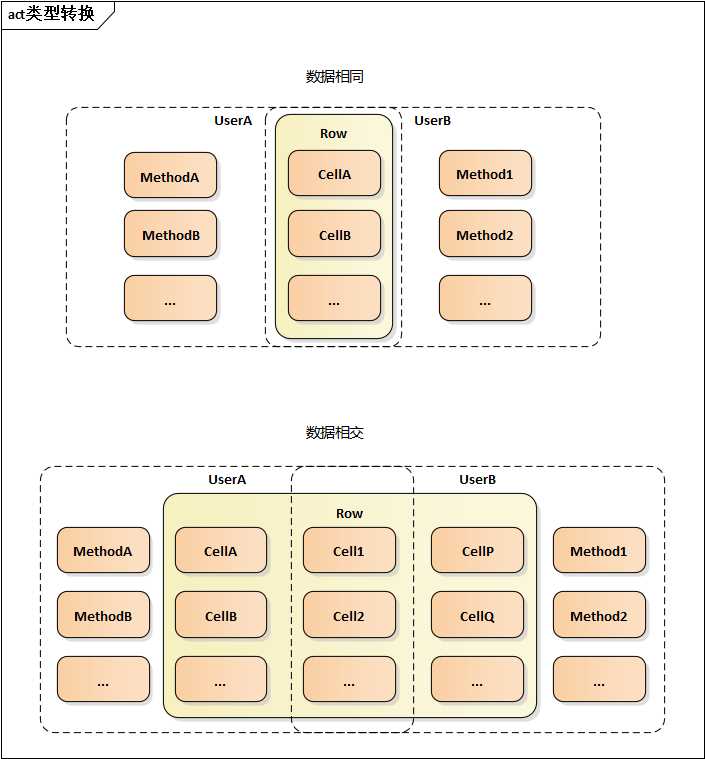
### 高性能加载数据

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

上文提到，对于每个实体类型为方便管理和修改，采取分文件管理方式，业务逻辑就放在EntityEx.cs中，自动生成的实体类型属性是不允许外部直接设置的，必须通过实体方法实现，这些都是DDD提倡的充血模式。

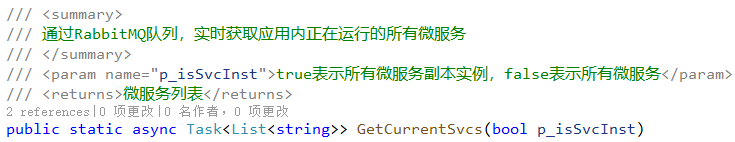
### 转换成Sql

在生成实体类型代码时通过Tbl标签已指定映射的表名/视图名和所属服务，实体对象在使用过程中记录数据变化过程，在传输过程未丢失任何信息，因此通过实体对象和实体类型配置生成要执行的Sql语句，生成Sql过程请参见TableSchema.cs文件。

## 典型案例

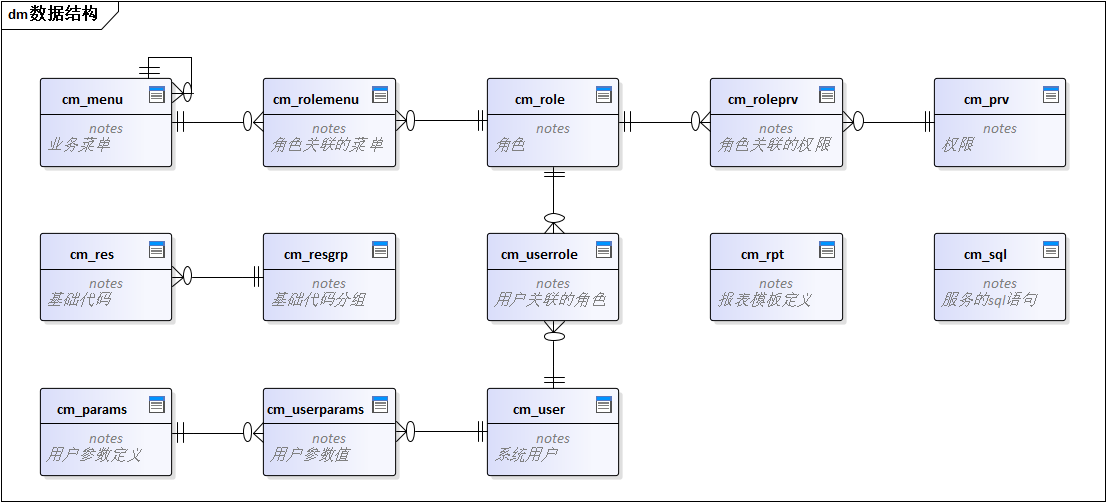
# 平台服务

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。



## Cm服务

CM是Core Model的缩写，为平台提供基础的内核模型服务，



## 服务配置

# 下一步

## 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。grpc基于http2实现4种通信模式参见：<https://github.com/grpc/grpc-dotnet>

## 创建pfx证书

## 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

### 相关术语

### 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