Serverless Computing & Fass （author： Tab）

无服务器计算是在无需最终用户管理的基础设施上托管应用程序的新方式，是IaaS(基础设施即服务)演进的下一个阶段。它将**底层基础架构从开发人员中分离出来**，基本上虚拟化了运行时(虚拟机的一种，一般指进程级别的虚拟机)和运营管理。这通常被称为 FaaS(功能即服务)，**无服务器架构允许您执行给定的任务而不必担心服务器、虚拟机或底层计算资源**

无服务器计算是一种云服务，托管服务提供商会实时为你分配充足的资源，而不是让你预先为专用的服务器或容量付费。无服务器计算不是不需要服务器（无服务器字面上的意思是，不用去管服务器），只是立足于云基础设施之上建立新的抽象层，仅使用完成任务所需的非常精确的计算资源来执行开发人员编写的代码，不多也不少。当触发代码的预定义事件发生时，无服务器平台执行任务。

假设现在有下面的JavaScript代码：

```
module.exports = function(context, callback) { callback(200,
"Hello, world! "); }
```

显然它是一个函数，通过FaaS的方式，我们可以通过访问一个URL的方式调用这个函数。

```
$ curl -XGET localhost:8080
Hello, world!
```

FaaS需要借助于API Gateway将请求的路由和对应的处理函数进行映射，并将响应结果代理返回给调用方。

一些著名的Fass框架

- Severless Framework (https://serverless.com/)
- OpenFaas(https://github.com/openfaas/faas)
- Kubeless(https://kubeless.io/)
- OpenWhisk(https://github.com/apache/openwhisk)
- Fission
- knative(https://github.com/knative)

Openwhisk

Openwhisk是属于Apache基金会的开源Faas计算平台，由IBM在2016年公布并贡献给开源社区。IBM Cloud本身也提供完全托管的OpenWhisk Faas服务IBM Cloud Function。从业务逻辑来看，OpenWhisk同AWS Lambda一样，**为用户提供基于事件驱动的无状态的计算模型**，并直接支持多种编程语言。

> 1. 函数的代码及运行时全部在Docker容器中运行，利用Docker engine实现Faas函数运行的管理、负载均衡、扩展.
> 2. OpenWhisk所有其他组件(如：API网关，控制器，触发器等)也全部运行在 Docker容器中。这使得OpenWhisk全栈可以很容易的部署在任意IaaS/PaaS平台上

基于docker-compose进行openwhisk快速部署

docker-compose可以看作是docker一系列镜像的sequence操作。

1 环境配置

1.1 apt 更新

```
sudo apt update
sudo apt upgrade
```

1.2 node.js 以及 npm安装（因为后续openwhisk client打算用javascript，所以需要用node进行调试）

```
curl -sL https://deb.nodesource.com/setup_lts.x | sudo -E bash -

sudo apt-get install -y nodejs
```

1.3 docker安装

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
```

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-
key add -

# 添加stable库
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"

sudo apt install docker-ce docker-ce-cli containerd.io
```

1.4 docker-compose安装

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.28.2/docker-
compose-Linux-x86_64" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose # 添加权限
```

1.5 相关repository下载

```
mkdir openwhisk
cd openwhisk

git clone https://github.com/apache/openwhisk.git # openwhisk 核心源
码

git clone https://github.com/apache/openwhisk-devtools.git #
openwhisk-devtools docker-compose所需要的 yml文件 以及部署所用的
makefile文件就在该路径下的docker-compose文件夹中

# openwhisk-cli下载
wget https://github.com/apache/openwhisk-
cli/releases/download/1.1.0/OpenWhisk_CLI-1.1.0-linux-amd64.tgz
mkdir openwhisk-cli
tar zxvf OpenWhisk_CLI-1.1.0-linux-amd64.tgz -C ./openwhisk-cli
cp openwhisk-cli/wsk openwhisk/bin/wsk

# openwhisk-catalog下载
git clone https://github.com/apache/openwhisk-catalog.git
```

1.6 openwhisk相关配置

下载完以后的openwhisk-cli不是全局变量，需要设置

```
vim /etc/profile
# 在最后一行加入
PATH=$PATH:/root/Openwhisk/openwhisk/bin
# 关掉
source /etc/prifile
```

openwhisk admin（后续介绍，用来创建namespace）也需要配置

```
sudo apt install python2
sudo ln -s /usr/bin/python2 /usr/bin/python
```

1.7 docker-compose部署

```
cd ~/Openwhisk/openwhisk-devtools/docker-compose/

vim make.props
# 设置
OPENWHISK_PROJECT_HOME=/root/Openwhisk/openwhisk
OPENWHISK_CATALOG_HOME=/root/Openwhisk/openwhisk-catalog
WSK_CLI=/root/Openwhisk/openwhisk-cli/wsk

sudo ln -sf /bin/bash /bin/sh

cd ~/Openwhisk/openwhisk-devtools/docker-compose/
sudo make $(cat ./make.props) quick-start
```

按照上述流程，基于docker-compose的快速部署就完成了，需要注意的是，快速部署会有很多细节问题没有注意到，这里希望用户可以先浏览一遍Makefile文件，其中包含了对需要环境变量的定义，openwhisk部署需要哪些docker镜像，已经镜像的调用。

openwhisk-cli的基本操作

OpenWhisk提供了wsk命令行界面（CLI），可以轻松地创建、运行和管理OpenWhisk实体。

2.1 CLI config

There are two required properties to configure in order to use the CLI:

1. **API host** (name or IP address) for the OpenWhisk deployment you want to use.
2. **Authorization key** (username and password) which grants you access to the OpenWhisk API.

查看configure：（API host 和 Authorizationkey是需要个人设置的）

```
wsk property get -i
```

```
whisk API host      172.31.152.76
whisk auth      23bc46b1-71f6-4ed5-8c54-
816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyG
VCGuMDGIwP
whisk namespace     guest
client cert
Client key
whisk API version   v1
whisk CLI version   2020-10-02T00:33:38.010+0000
whisk API build     09/01/2016
whisk API build number  latest
```

API host指的就是服务器的本机ip地址，会在makefile中指定，whisk auth对应着一个namespace（通过wskadmin生成新的namespace就会对应生成一个新的auth key，后续讲）

如果需要修改API host或者是auth key 可以通过下述指令设定

```
wsk property set --apihost <值> --auth <值>
```

2.2 ACTION

（这里代码的中文介绍引用部分 谢磊的内容）

openwisk中的action就是Faas中的function，即一个action就是实现了一个服务，主要包括action的创建，运行，更新等。

action的函数接受字典作为输入，并生成字典作为输出。输入和输出字典是键-值对，其中键是字符串，值是任何有效的JSON值。当通过restapi或wskcli与操作接口时，字典被规范地表示为JSON对象。

必须调用main函数，否则必须显式导出以将其标识为入口点。根据您选择的语言，机制可能会有所不同，但通常在使用wsk CLI时，可以使用--main标志指定入口点。

action create

```
wsk action create [/namespace/package/] [function name] [file]
其中namespace*是可选的，默认为default。
--param [variable name] [variable value] 设置默认参数值
-web 可选true， false， raw。注意这个选项和--param一起用的时候， param不可被覆
盖。
```

```
wsk action create <action name> <code>
```

e.g. hello.py:

```
def main(params):
    if "name" in params:
        name = params["name"]
    else:
        name = "stranger"

    return {"payload": "hello" + name}
```

```
wsk action create hello hello.py -i
ok: created action hello
```

e.g.2 pee_search.py 实现数据库查询（这里需要用到docker镜像配置需要的库依赖 create --docker)

```
from datetime import date
from peewee import *

database = MySQLDatabase('test_db', user='lyz_leo', host='rm-
pOwodj1r55acz1932lo.mysql.australia.rds.aliyuncs.com',
password='sysAdmin1', port=3306)

class Person2(Model):
name = CharField()
phone = CharField()
class Meta:
database = database

def main(dict):
if "name" in dict:
name = dict['name']
p = Person2.get(Person2.name == name)
return {"name":p.name,"phone":p.phone}
else:
return {"error": "no people"}

root@openwhisk:~/Openwhisk/openwhisk/bin# ./wsk action invoke --
result MyDB/pee_search --param name tab -i
{
"name": "tab",
"phone": "1200"
}
```

```
wsk action invoke --result MyDB/pee_search --param name tab -i # 这
里用到了package，详情在下一节
{
"name": "tab",
"phone": "1200"
}
```

action update

```
wsk action update [/namespace/package/] [function name] [file]
其中namespace是可选的，默认为default。
可选参数：
--param [variable name] [variable value] 设置默认参数值
```

当函数文件被修改，或者是想调整action功能时，可以在原有的action name上直接进行update

```
wsk action update hello hello.py -i
ok: updated action hello
```

action invoke

```
wsk action invoke [/namespace/package/] [function name]
其中namespace是可选的，默认为default。
可选参数：
--blocking 阻塞invoke执行完毕，返回 entire action record。
--result 阻塞等待invoke执行完毕（--blocking），只获取执行结果(result)。
--param [variable name] [variable value] --入参，可多个。形如： --param
key1 value1 --param key2 value2
-p 用于传递一个对象概念的json串 eg： -p person '{"name": "Dorothy",
"place": "Kansas"}'
--param-file [filename.json] 传递一个json格式的参数文件。
```

激活一个action就是使一个action运行起来

```
wsk action invoke hello -i
ok: invoked /_/hello with id bbfa2536f8d5468cba2536f8d5d68cc4
```

在不加任何参数的情况下，会返回一个id，并不会直接显示结果，要检索激活记录，请使用wsk activation get命令

```
wsk activation get bbfa2536f8d5468cba2536f8d5d68cc4

{
    "namespace": "guest",
    "name": "hello",
    "version": "0.0.2",
```

```
    "subject": "guest",
    "activationId": "bbfa2536f8d5468cba2536f8d5d68cc4",
    "start": 1621477403329,
    "end": 1621477403689,
    "duration": 360,
    "statusCode": 0,
    "response": {
        "status": "success",
        "statusCode": 0,
        "success": true,
        "result": {
            "payload": "hellostranger"
        }
    },
    "logs": [],
    "annotations": [
        {
            "key": "path",
            "value": "guest/hello"
        },
        {
            "key": "waitTime",
            "value": 941
        },
        {
            "key": "kind",
            "value": "python:3"
        },
        {
            "key": "timeout",
            "value": false
        },
        {
            "key": "limits",
            "value": {
                "concurrency": 1,
                "logs": 10,
                "memory": 256,
                "timeout": 60000
            }
        },
        {
            "key": "initTime",
```

```
            "value": 356
        }
    ],
    "publish": false
}
# 返回信息在
"response": {
        "status": "success",
        "statusCode": 0,
        "success": true,
        "result": {
            "payload": "hellostranger"
        }
    },
```

或者可以直接添加参数：

```
wsk action invoke hello --result

{
    "payload": "hellostranger"
}
```

action 注释

```
  wsk action create echo echo.js \
     -a description 'An action which returns its input. Useful for
logging input to enable debug/replay.' \
     -a parameters '[{ "required":false, "description": "Any JSON
entity" }]' \
     -a sampleInput '{ "msg": "Five fuzzy felines"}' \
     -a sampleOutput '{ "msg": "Five fuzzy felines"}
```

```
wsk action get -i --summary MyDB/pee_search
action /guest/MyDB/pee_search: an action return the info you want
to search in DB.
       (parameters: *name)
```

2.3 PACKAGES

packages的作用就是把相关的action,trigger等打包在一起，方便管理。同时一个package还可以共享一组相同的参数

package create

```
wsk package create [package name]
可选参数：
--param [variable name] [variable value] 为package设置默认参数
```

```
wsk package create MyDB -i
ok: created package MyDB
```

package list

查看当前namespace下有哪些package

```
wsk package list -i
packages
/guest/MyDB
   private
```

在package下创建一个action，需要在action name前面加上package名

```
wsk action create -i MyDB/pee_search --docker 623344308/tabtry
pee_search.py
```

package get

get 获取package下所有的entities

```
wsk package get --summary MyDB

package /guest/MyDB
(parameters: none defined)
action /guest/MyDB/pee_search
(parameters: none defined)
```

package binding

binding用来定义一组参数，这组参数在package中的action invoke没有指定参数时，作为默认参数传入action中

```
wsk package bind [package name] [binding name]
可选参数：
--param [variable name] [variable value] 为binding设置默认参数
```

e.g.

```
root@openwhisk:~/Openwhisk/openwhisk/bin# wsk package bind MyDB
GuiDB --param name gui -i
ok: created binding GuiDB

root@openwhisk:~/Openwhisk/openwhisk/bin# wsk action invoke
GuiDB/pee_search --result -i
{
"name": "gui",
"phone": "123456"
}
```

2.4 TRIGGER & RULE

trigger可以实现“展示如何响应来自事件源的事件来自动执行操作“。 具体来说，trigger实现使用规则将具有输入参数的基本触发器创建，触发并将其关联到动作。trigger可以有如下的例子

- A trigger of location update events.
- A trigger of document uploads to a website.
- A trigger of incoming emails

  规则将一个触发器与一个操作相关联，每次触发触发器都会导致调用相应的操作，并将触发器事件作为输入。使用适当的规则集，单个触发器事件可以调用多个动作，或者一个动作可以作为对多个触发器事件的响应来调用。

trigger create

```
wsk trigger create [trigger name]
```

```
wsk trigger create hellotab -i
ok: created trigger hellotab
```

trigger fire

触发 trigger，可以传入参数，这个参数就会传入到rule连接的action中

```
wsk trigger fire [trigger name]
可选参数：
--parma 传入action的参数
```

rule create

```
wsk rule create myRule hellotab hello -i
ok: created rule myRule
```

rule 构造好以后，fire trigger就会invoke起来对应的action

```
wsk trigger fire hellotab --param name tab -i
ok: triggered /_/hellotab with id ef0f28ac8a294ff28f28ac8a29bff2c1
# 注意这里的id不是action的id
wsk activation list -i
root@iZ2ze6gj24x1tw3du24j61Z:~/Openwhisk# wsk activation list --
limit 1 -i
Datetime            Activation ID                    Kind     Start
Duration   Status  Entity
2021-05-20 11:32:37 832e4a9cd93d4c25ae4a9cd93d9c255e python:3 warm
5ms        success guest/hello:0.0.3
root@iZ2ze6gj24x1tw3du24j61Z:~/Openwhisk# wsk activation result
832e4a9cd93d4c25ae4a9cd93d9c255e -i
{
    "payload": "hellotab"
}
```

2.5 NAMESPACE&wskadmin

OpenWhisk action 、trigger 和 rules 属于一个名称空间，也可以是一个包。命名空间可以理解为 为一个organization创建一个空间，该organization只能使用该空间，保证隐私及安全。action trigger rules都属于 entities 因此 这些entities应该都包含在某一个命名空间下， namespace package也属于 entity。

命名规则：

第一个字符必须是字母数字字符或下划线。

后面的字符可以是字母数字、空格或以下任意字符：216;、@、.、-。

最后一个字符不能是空格。

相关limitations ：
https://github.com/apache/openwhisk/blob/master/docs/reference.md#openwhisk-entities，

一个action 容器 最长运行时间 60000毫秒（也就是一分钟）
一个action 容器 最大的内存 256Mb
一个namespace 最大只能有 100个action被执行
一个action 最大的code size 48MB

wskadmin实用工具对于针对OpenWhisk部署执行各种管理操作非常方便。它允许你创建一个新的主题，管理他们的命名空间，阻止一个主题或完全删除他们的记录。它还提供了一种方便的方法来转储资产、激活或主题数据库并查询它们的视图。这对于调试本地部署非常有用。最后，检查系统日志或检索特定于组件或事务id的日志是一种方便的方法。

wskadmin使用方法：

1. 配置：在wskadmin(openwhisk/bin/)的根目录下即bin文件夹中创建whisk.properties。内容如下：

```
WHISK_LOGS_DIR=/Users/nickgraziano/incubator-openwhisk/logs
DB_PROTOCOL=http
DB_PORT=5984
DB_HOST=127.0.0.1
DB_USERNAME=whisk_admin
DB_PASSWORD=some_passw0rd
DB_WHISK_AUTHS=local_subjects
DB_WHISK_ACTIONS=local_whisks
DB_WHISK_ACTIVATIONS=local_activations
```

2. 创建namespace：

```
wskadmin user create userA

b4501cba-5afd-4992-bff9-
2171174ecde6:Kd5H3cXdjhQAdYBEgUIwvQrqpp6eLRWjwDLu0WyaHU9Ta5T
QdsaGwCKiNsnNT099
```

上述返回的就是 wsk property中的auth。

3. 切换namespace：

```
wsk property get -i
whisk API host      172.31.152.76
whisk auth      23bc46b1-71f6-4ed5-8c54-
816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4V
rNZ9lyGVCGuMDGIwP
whisk namespace     guest
client cert
Client key
whisk API version   v1
whisk CLI version   2020-10-02T00:33:38.010+0000
whisk API build     09/01/2016
whisk API build number  latest

wsk property set --auth b4501cba-5afd-4992-bff9-
2171174ecde6:Kd5H3cXdjhQAdYBEgUIwvQrqpp6eLRWjwDLu0WyaHU9Ta5T
QdsaGwCKiNsnNT099 -i

whisk API host      172.31.152.76
```

```
whisk auth      b4501cba-5afd-4992-bff9-
2171174ecde6:Kd5H3cXdjhQAdYBEgUIwvQrqpp6eLRWjwDLuOWyaHU9Ta5T
QdsaGwCKiNsnNTO99
whisk namespace     userA
client cert
Client key
whisk API version   v1
whisk CLI version   2020-10-02T00:33:38.010+0000
whisk API build     09/01/2016
whisk API build number  latest

wsk package list -i
packages
```

可以看到namespace发生了变化，并且查看当前namespace下的package为空（之前创立的My_DB package在guest namespace中）

2.6 API 相关操作

https://github.com/apache/openwhisk/blob/master/docs/webactions.md#web-actions

https://github.com/apache/openwhisk/blob/master/docs/rest_api.md#using-rest-apis-with-openwhisk

这一部分内容，需要先把apihost的授权解决才能开展。

当前的问题是 wskcli任何操作都需要bypass certificate checking，所以需要签证授权。也就是要解决问题：

```
x509: cannot validate certificate for 172.31.152.76 because it
doesn't contain any IP SANs
```

```
sudo openssl req -subj '/CN=172.31.152.76 ' -newkey rsa:2048 -new -
nodes -x509 -days 3650 -keyout registry.key -out registry.crt

sudo cp ./registry.crt /usr/local/share/ca-certificates

sudo update-ca-certificates
```

openwhisk client （javascript）

openwhisk 支持用javascript脚本语言进行openwhisk相关entity的操作，并且利用API进行前后端交互。相关源码https://github.com/apache/openwhisk-client-js

3.1 client使用

如果在openwhisk platform中，client会自动根据环境变量配置，这里讲一下如何简单配置使用

3.1.1 进入nodejs，获取openwhisk依赖库，并根据auth apihost设置

```
node

> var options = {apihost:'172.31.152.76',api_key:'23bc46b1-71f6-
4ed5-8c54-
816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyG
VCGuMDGIwP', ignore_certs:true} # 这里设定ignoreCerT:true还是因为
2.6api相关问题

> var openwhisk = require('openwhisk')
> var ow = openwhisk(options);

> ow 可以看到相关操作也就是包括了 action activation rule trigger等等，类似
于cli
{
  actions: Actions {
    client: Client { options: [Object] },
    identifiers: [ 'name', 'actionName' ],
    qs_options: { invoke: [Array] },
    resource: 'actions'
  },
  activations: Activations { client: Client { options: [Object] }
},
  namespaces: Namespaces { client: Client { options: [Object] } },
  packages: Packages {
    client: Client { options: [Object] },
    identifiers: [ 'name', 'packageName' ],
    qs_options: {},
    resource: 'packages'
  },
  rules: Rules {
    client: Client { options: [Object] },
```

```
    identifiers: [ 'name', 'ruleName' ],
    qs_options: {},
    resource: 'rules'
  },
  triggers: Triggers {
    client: Client { options: [Object] },
    identifiers: [ 'name', 'triggerName' ],
    qs_options: {},
    resource: 'triggers'
  },
  feeds: Feeds {
    actions: Actions {
      client: [Client],
      identifiers: [Array],
      qs_options: [Object],
      resource: 'actions'
    },
    client: Client { options: [Object] }
  },
  routes: Routes { client: Client { options: [Object] } }
}
```

3.1.2 action相关操作

```
ow.actions

Actions {
  client: Client {
    options: {
      apiKey: '23bc46b1-71f6-4ed5-8c54-
816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyG
VCGuMDGIwP',
      api: 'https://172.31.152.76/api/v1/',
      apiVersion: 'v1',
      ignoreCerts: true,
      namespace: undefined,
      apigwToken: undefined,
      apigwSpaceGuid: undefined,
      authHandler: undefined,
      noUserAgent: undefined,
      cert: undefined,
```

```
      key: undefined,
      proxy: undefined,
      agent: undefined
    }
  },
  identifiers: [ 'name', 'actionName' ],
  qs_options: { invoke: [ 'blocking' ] },
  resource: 'actions'
}

# 相关变量设定
> const name = 'hello'
undefined
> const blocking = true, result = true
undefined
> const params = {name:'tab'}
undefined

# action invoke
> ow.actions.invoke({name, blocking, result, params}).then(result
=> {console.log(result)})
Promise { <pending> }
> { payload: 'hellotab' }
```

3.1.3 trigger相关操作

```
> ow.triggers
Triggers {
  client: Client {
    options: {
      apiKey: '23bc46b1-71f6-4ed5-8c54-
816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyG
VCGuMDGIwP',
      api: 'https://172.31.152.76/api/v1/',
      apiVersion: 'v1',
      ignoreCerts: true,
      namespace: undefined,
      apigwToken: undefined,
      apigwSpaceGuid: undefined,
      authHandler: undefined,
      noUserAgent: undefined,
      cert: undefined,
```

```
      key: undefined,
      proxy: undefined,
      agent: undefined
    }
  },
  identifiers: [ 'name', 'triggerName' ],
  qs_options: {},
  resource: 'triggers'
}

> const name = 'hellotab'
> ow.triggers.invoke({name, params}).then(result=>
{console.log(result)})
Promise { <pending> }
> { activationId: '9a84a4f3e6c6410384a4f3e6c641030b' }
```

trigger 被 fire 并且返回 trigger的id

```
>
ow.activations.get('9a84a4f3e6c6410384a4f3e6c641030b').then(result=
>{console.log(result)})
Promise { <pending> }
> {
  activationId: '9a84a4f3e6c6410384a4f3e6c641030b',
  annotations: [],
  logs: [

'{"statusCode":0,"success":true,"activationId":"0850b7183209461d90b
7183209a61dd8","rule":"guest/myRule","action":"guest/hello"}'
  ],
  name: 'hellotab',
  namespace: 'guest',
  publish: false,
  response: { result: { name: 'tab' }, status: 'success', success:
true },
  start: 1621494822688,
  subject: 'guest',
  version: '0.0.1'
}
```

3.14 package相关操作

```
> ow.packages.list().then(packages => packages.forEach(package =>
console.log(package.name)))
Promise { <pending> }
> MyDB
```