部署_nginx_pm2_docker

如何构建一个高可用的node环境

> 主要解决问题
>
> - 故障恢复
> - 多核利用
> - http://www.sohu.com/a/247732550_796914
> - 多进程共享端口

```
// app.js
const Koa = require('koa');

// 创建一个Koa对象表示web app本身:
const app = new Koa();

// 对于任何请求，app将调用该异步函数处理请求:
app.use(async (ctx, next) => {
    // 随机产生错误
    Math.random() > 0.9 ?  aaa() : '2'

    await next();
    ctx.response.type = 'text/html';
    ctx.response.body = '<h1>Hello, koa2!</h1>';
});

if (!module.parent) {
    app.listen(3000);
    console.log('app started at port 3000...');
} else {
    module.exports = app
}

// test.js
var http = require('http');
setInterval(async () => {
    try {
        await http.get('http://localhost:3000');
    } catch (error) {
    }
}, 1000)

// cluster.js
var cluster = require('cluster');
var os = require('os'); // 获取CPU 的数量
var numCPUs = os.cpus().length;
var process = require('process')

console.log('numCPUs:', numCPUs)
var workers = {};
if (cluster.isMaster) {
    // 主进程分支
    cluster.on('death', function (worker) {
        // 当一个工作进程结束时，重启工作进程 delete workers[worker.pid];
        worker = cluster.fork();
        workers[worker.pid] = worker;
    });
    // 初始开启与CPU 数量相同的工作进程
    for (var i = 0; i < numCPUs; i++) {
        var worker = cluster.fork();
        workers[worker.pid] = worker;
    }
} else {
    // 工作进程分支，启动服务器
    var app = require('./app');
    app.use(async (ctx, next) => {
        console.log('worker' + cluster.worker.id + ',PID:' + process.pid)
        next()
    })
    app.listen(3000);
}
// 当主进程被终止时，关闭所有工作进程
process.on('SIGTERM', function () {
    for (var pid in workers) {
        process.kill(pid);
    }
    process.exit(0);
});

require('./test')
```

文件上传服务器

- scp (最原始)

```
scp docker-compose.yml root@47.98.252.43:/root/source/ #文件
scp -r mini-01 root@47.98.252.43:/root/source/     #文件夹
```

- git (实际工作中)
- deploy插件 (debug)

PM2的应用

- 内建负载均衡

- 线程守护，keep alive
- 0秒停机重载，维护升级的时候不需要停机.
- 现在 Linux (stable) & MacOSx (stable) & Windows (stable).多平台支持
- 停止不稳定的进程（避免无限循环）
- 控制台检测 https://id.keymetrics.io/api/oauth/login#/register
- 提供 HTTP API

配置

```
npm install -g pm2
pm2 start app.js --watch -i 2
// watch 监听文件变化
// -i 启动多少个实例

pm2 stop all
pm2 list

pm2 start app.js -i max # 根据机器CPU核数，开启对应数目的进程
```

配置process.yml

```
apps:
  - script : app.js
    instances: 2
    watch  : true
    env    :
      NODE_ENV: production
```

- Keymetrics在线监控

  > https://id.keymetrics.io

```
pm2 link 8hxvp4bfrftvwxn uis7ndy58fvuf7l TARO-SAMPLE
```

pm2设置为开机启动

```
pm2 startup
```

Nginx 反向代理 + 前端打包Dist

安装

```
yum install nginx
-----
apt update
apt install nginx
```

添加静态路由

```
# /etc/nginx/sites-enable/taro

server {
    listen 80;
    server_name taro.josephxia.com;
    location / {
        root /root/source/taro-node/dist;
        index index.html index.htm;
    }
}
```

```
# 验证Nginx配置
nginx -t

# 重新启动Nginx
service nginx restart

nginx -s reload
```

```
# /etc/nginx/sites-enable
# taro

server {
    listen 80;
    server_name taro.josephxia.com;
    location / {
        root /root/source/taro-node/dist;
        index index.html index.htm;
    }
    location ~ \.(gif|jpg|png)$ {
        root /root/source/taro-node/server/static;
    }
    location /api {
            proxy_pass  http://127.0.0.1:3000;
            proxy_redirect     off;
            proxy_set_header   Host             $host;
            proxy_set_header   X-Real-IP        $remote_addr;
            proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
    }
}
```

```
# 查看配置文件位置
nginx -t
# nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
# nginx: configuration file /etc/nginx/nginx.conf test is successful

#重启
service nginx restart
```

Docker概念

- 操作系统层面的虚拟化技术
- 隔离的进程独立于宿主和其它的隔离的进程 - 容器
- GO语言开发

特点

- 高效的利用系统资源
- 快速的启动时间
- 一致的运行环境
- 持续交付和部署
- 更轻松的迁移

对比传统虚拟机总结

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 `MB`	一般为 `GB`
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

三个核心概念

- 镜像
- 容器
- 仓库

Docker基本使用

构建一个Nginx服务器

1. 拉取官方镜像

```
# 拉取官方镜像
docker pull nginx

# 查看
```

```
docker images nginx

# 启动镜像
mkdir www
echo 'hello docker!!' >> www/index.html

# 启动
# www目录里面放一个index.html
docker run -p 80:80 -v $PWD/www:/usr/share/nginx/html -d nginx

# 查看进程
docker ps
docker ps -a // 查看全部

# 伪终端 ff6容器的uuid
# -t 选项让Docker分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上，
# -i 则让容器的标准输入保持打开
docker exec -it ff6 /bin/bash

# 停止
docker stop ff6

# 删除镜像
docker rm ff6
```

Dockerfile定制镜像

```
#Dockerfile
FROM nginx:latest
RUN echo '<h1>Hello, Kaikeba!</h1>' > /usr/share/nginx/html/index.html
```

```
# 定制镜像
docker build -t mynginx .

# 运行
# -d 守护态运行
docker run -p 80:80 -d  mynginx
```

定制一个程序NodeJS镜像

```
npm init -y
npm i koa -s
```

```
// package.json
{
  "name": "myappp",
  "version": "1.0.0",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "myappp",
  "dependencies": {
    "koa": "^2.7.0"
  }
}
```

```
// app.js
const Koa = require('koa')
const app = new Koa()
app.use(ctx => {
    Math.random() > 0.8 ? abc() : ''
    ctx.body = 'Hello Docker'
})
app.listen(3000, () => {
    console.log('app started at http://localhost:3000/')
})
```

```
#Dockerfile
#制定node镜像的版本
FROM node:10-alpine
#移动当前目录下面的文件到app目录下
ADD . /app/
#进入到app目录下面，类似cd
WORKDIR /app
#安装依赖
RUN npm install
#对外暴露的端口
EXPOSE 3000
#程序启动脚本
CMD ["node", "app.js"]
```

```
# 定制镜像
docker build -t mynode .

# 运行
docker run -p 3000:3000  -d mynode
```

Pm2 - 利用多核资源

```
# .dockerignore
node_modules
```

```
// process.yml
apps:
  - script : app.js
    instances: 2
    watch  : true
    env    :
      NODE_ENV: production
```

```
# Dockerfile
FROM keymetrics/pm2:latest-alpine
WORKDIR /usr/src/app
ADD . /usr/src/app
RUN npm config set registry https://registry.npm.taobao.org/ && \
    npm i
EXPOSE 3000
#pm2在docker中使用命令为pm2-docker
CMD ["pm2-runtime", "start", "process.yml"]
```

```
# 定制镜像
docker build -t mypm2 .

# 运行
docker run -p 3000:3000  -d mypm2
```

Docker-Compose

```
#docker-compose.yml
app-pm2:
    container_name: app-pm2
    #构建容器
    build: .
    # volumes:
    #   - .:/usr/src/app
    ports:
      - "3000:3000"
```

```
// 强制重新构建并启
# --force-recreate 强制重建容器
# --build 强制编译
docker-compose up -d --force-recreate --build
```

```
#docker-compose.yml
version: '3.1'
services:
  nginx:
    image: nginx:kaikeba
    ports:
      - 80:80
```

```
# 运行
docker-compose up

# 后台运行
docker-compose up -d
```

部署Mongo + MongoExpress

```
#docker-compose.yml
version: '3.1'
services:
  mongo:
    image: mongo
    restart: always
    ports:
      - 27017:27017
  mongo-express:
    image: mongo-express
    restart: always
    ports:
      - 8081:8081
```

代码中添加Mongoose调用

```
// mongoose.js
const mongoose = require("mongoose");
// 1.连接
mongoose.connect("mongodb://mongo:27017/test", { useNewUrlParser: true });
const conn = mongoose.connection;
conn.on("error", () => console.error("连接数据库失败"));
```

```
// app.js

const mongoose = require('mongoose');
mongoose.connect('mongodb://mongo:27017/test', {useNewUrlParser: true});
const Cat = mongoose.model('Cat', { name: String });
Cat.deleteMany({})
const kitty = new Cat({ name: 'Zildjian' });
kitty.save().then(() => console.log('meow'));
```

```
app.use(async ctx => {

    ctx.body = await Cat.find()

})
```

Github WebHook实现CI持续集成

启动NodeJS监听

```
var http = require('http')
var createHandler = require('github-webhook-handler')
var handler = createHandler({ path: '/webhooks', secret: 'myHashSecret' })
// 上面的 secret 保持和 GitHub 后台设置的一致

function run_cmd(cmd, args, callback) {
    var spawn = require('child_process').spawn;
    var child = spawn(cmd, args);
    var resp = "";

    child.stdout.on('data', function (buffer) { resp += buffer.toString(); });
    child.stdout.on('end', function () { callback(resp) });
}

http.createServer(function (req, res) {
    handler(req, res, function (err) {
        res.statusCode = 404
        res.end('no such location')
    })
}).listen(3000)

handler.on('error', function (err) {
    console.error('Error:', err.message)
})


handler.on('*', function (event) {
    console.log('Received *', event.payload.action);
    //   run_cmd('sh', ['./deploy-dev.sh'], function(text){ console.log(text) });
})

handler.on('push', function (event) {
    console.log('Received a push event for %s to %s',
        event.payload.repository.name,
        event.payload.ref);
        // 分支判断
        if(event.payload.ref === 'refs/heads/master'){
            console.log('deploy master..')
        }
    //   run_cmd('sh', ['./deploy-dev.sh'], function(text){ console.log(text) });
})


handler.on('issues', function (event) {
    console.log('Received an issue event for % action=%s: #%d %s',
        event.payload.repository.name,
        event.payload.action,
        event.payload.issue.number,
        event.payload.issue.title)
})
```