```
ls
      MYSQL_ROOT_PASSWORD: xxxxxx
    networks:
      - common-network

  redis-container:
    image: redis
    volumes:
      - /www/apps/docker-data/redis:/data
    networks:
      - common-network

networks:
  common-network:
    driver: bridge
```

配置解析

版本

```
version: "3.8"
```

这是 Docker Compose 文件的版本。`3.8` 是 Docker Compose 文件格式的版本，确保 Docker 引擎和 Docker Compose 版本兼容。

服务（Services）

Nest.js 应用服务（nest-app）

```
nest-app:
  build:
    context: ./
```

```
    dockerfile: ./Dockerfile
  depends_on:
    - mysql-container
    - redis-container
  ports:
    - 3005:3005
  networks:
    - common-network

```

1. **build**:

* **context**: `./` 表示构建镜像的上下文是当前目录。
* **dockerfile**: `./Dockerfile` 表示使用当前目录下的 Dockerfile 文件构建镜像。
2. **depends_on**:

* `mysql-container` 和 `redis-container` 表示 nest-app 容器依赖这两个容器，确保它们先启动。
3. **ports**:

* `3005:3005` 表示将宿主机的 3005 端口映射到容器的 3005 端口。**提醒一下，如果自己的 nest 服务跑的不是 3005，需要把前面的端口改成自己实际的端口。**
4. **networks**:

* `common-network` 指定该服务加入到 common-network 网络中。

MySQL 容器（mysql-container）


```
mysql-container:
  image: mysql
  volumes:
    - /www/apps/docker-data/mysql:/var/lib/mysql
  environment:
    MYSQL_DATABASE: picals
    MYSQL_ROOT_PASSWORD: xxxxxx
  networks:
    - common-network
```

```

1. **image**:
* 使用官方的 `mysql` 镜像。
2. **volumes**:
* 将宿主机上的 `/www/apps/docker-data/mysql` 目录挂载到容器内的 `/var/lib/mysql` 目录，以持久化 MySQL 数据。
3. **environment**:
* `MYSQL_DATABASE`: 初始化数据库名为 `picals`。
* `MYSQL_ROOT_PASSWORD`: 设置 MySQL root 用户的密码为 `xxxxxx`。
4. **networks**:
* `common-network` 指定该服务加入到 common-network 网络中。

**Redis 容器（redis-container）**

```
redis-container:
  image: redis
  volumes:
    - /www/apps/docker-data/redis:/data
  networks:
    - common-network
```

1. **image**:
* 使用官方的 `redis` 镜像。
2. **volumes**:
* 将宿主机上的 `/www/apps/docker-data/redis` 目录挂载到容器内的 `/data` 目录，以持久化 Redis 数据。
3. **networks**:
* `common-network` 指定该服务加入到 common-network 网络中。

#### 网络（Networks）

```
networks:
  common-network:
    driver: bridge
```
```

1. **common-network**:
* 定义了一个名为 `common-network` 的网络，并指定使用 `bridge` 驱动。这使得所有加入该网络的容器可以相互通信。

总结

* 当前这个 `docker-compose.yml` 文件定义了三个服务：`nest-app`、`mysql-container` 和 `redis-container`。
* 这些服务通过 `common-network` 网络互相连接。
* `nest-app` 服务依赖 `mysql-container` 和 `redis-container`，并将其暴露在端口 `3005` 上。
* `mysql-container` 和 `redis-container` 分别挂载了宿主机上的目录以持久化数据。

更改 .env 的配置项

在上面的 docker-compose-yml 文件中，我们已经指定了 mysql 和 redis 数据库的容器为 mysql-container 和 redis-container，因此我们需要把生产环境中的 .env 文件的相关数据库的 HOST 从 localhost 更改为这两个：

```
# 开发环境配置
NODE_ENV=development

# Redis相关
REDIS_HOST=redis-container
REDIS_PORT=6379
REDIS_DB=0

# nodemailer相关
NODEMAILER_HOST=xxxx
NODEMAILER_PORT=xxxx
NODEMAILER_NAME=xxxx
NODEMAILER_AUTH_USER=xxxx
NODEMAILER_AUTH_PASS=xxxx

# mysql相关
MYSQL_HOST=mysql-container
MYSQL_PORT=xxxx
MYSQL_USER=xxxx
MYSQL_PASS=xxxx
```

```
MYSQL_DB=xxxx

# nest服务配置
NEST_PORT=0721

# jwt配置
JWT_SECRET=xxxx
JWT_ACCESS_TOKEN_EXPIRES_TIME=30m
JWT_REFRESH_TOKEN_EXPIRES_TIME=7d

# 其他测试环境配置
CAPTCHA_SECRET=111111

# 跨域配置
CORS_ORIGIN=xxxx

# 腾讯云 COS 配置
COS_SECRETID=xxxx
COS_SECRETKEY=xxxx
COS_BUCKET=xxxx
COS_DOMAIN=xxxx
```

编写 Dockerfile

Dockerfile 是一个文本文件，包含一系列指令，用于构建一个 Docker 镜像。每一条指令在执行后，都会在镜像上添加一层，最终形成一个可以运行的 Docker 镜像。

文件模板

同样地，我也给出我的 Dockerfile 方案，仅供参考：

```
FROM node:18.0-alpine3.14 as build-stage

WORKDIR /app

COPY package.json .

RUN npm config set registry https://registry.npmmirror.com/
```

```
RUN npm install

COPY . .

RUN npm run build

# production stage
FROM node:18.0-alpine3.14 as production-stage

COPY --from=build-stage /app/dist /app
COPY --from=build-stage /app/package.json /app/package.json

WORKDIR /app

RUN npm config set registry https://registry.npmmirror.com/

RUN npm install --production

EXPOSE 3005

CMD ["node", "/app/main.js"]
```

配置解析

第一阶段：构建阶段（build-stage）

```
FROM node:18.0-alpine3.14 as build-stage
```

* 使用 `node:18.0-alpine3.14` 作为基础镜像。`alpine` 是一个轻量级的 Linux 发行版，有助于减小镜像的体积。
* `as build-stage` 将该阶段命名为 `build-stage`，以便在后续阶段中引用。

```
WORKDIR /app
```

* 设置工作目录为 `/app`。

```
COPY package.json .
```

* 将本地的 `package.json` 文件复制到容器的工作目录中。

```
RUN npm config set registry https://registry.npmmirror.com/
```

* 设置 npm 的镜像源为 `https://registry.npmmirror.com/`，加速依赖包的下载。

```
RUN npm install
```

* 安装项目依赖。

```
COPY . .
```

* 将当前目录下的所有文件复制到容器的工作目录中。

```
RUN npm run build
```

* 运行构建命令，生成生产环境的代码。构建后的文件通常放在 `dist` 目录中。

第二阶段：生产阶段（production-stage）

```
 n=`date +%Y%m%d%H%M`;

# 关闭容器
docker-compose stop || true;
# 删除容器
docker-compose down || true;
# 构建镜像
docker-compose build;
# 启动并后台运行
docker-compose up -d;
# 查看日志
docker logs nest-app;
# 对空间进行自动清理
docker system prune -a -f
```

这个脚本总共执行了下面六步操作：

1. 停止当前运行的容器。
2. 删除所有的容器。
3. 构建镜像。
4. 在后台启动服务。
5. 查看 nest 应用服务的日志。
6. 清理未使用的 Docker 对象。

编写完毕后，我们只需要在每次主分支合并后，在服务器端对最新的代码进行拉取，然后 **直接运行这个脚本命令就可以了**。整个应用就会在这个脚本的控制下，重新的一步步执行，最终成功的将服务跑起来。

> 顺带一提，如果遇到这个脚本受权限制约无法执行的情况，需要手动的去更改一下这个文件的权限：**可被执行**。
>
>
> 至于是哪个用户的权限，需要看你平时是用什么身份去运行终端的。我平时是 root 超级用户。
>

>
> 而且这个修改权限的操作似乎也是算入对于文件的修改操作的，也就是说你改完权限后需要将其反向提交到 github，由开发者进行拉取。

一些 Docker 部署时的常用命令清单

列举一些 Docker 在部署时的一些命令，便于后续查阅：

Docker

构建和管理镜像

* **`docker build -t <tag_name> <path>`**：根据 Dockerfile 构建镜像，并使用 `<tag_name>` 给镜像打标签。
* **`docker images`**：列出所有本地存储的 Docker 镜像。
* **`docker rmi <image_id>`**：删除指定的镜像。

管理容器

* **`docker run -d --name <container_name> <image_name>`**：使用指定镜像在后台启动一个容器，并命名为 `<container_name>`。
* **`docker ps`**：列出当前正在运行的所有容器。
* **`docker ps -a`**：列出所有容器，包括停止的容器。
* **`docker stop <container_id>`**：停止指定的容器。
* **`docker start <container_id>`**：启动已停止的容器。
* **`docker restart <container_id>`**：重启指定的容器。
* **`docker rm <container_id>`**：删除指定的容器。

容器日志和监控

* **`docker logs <container_id>`**：查看指定容器的日志。
* **`docker stats`**：显示所有容器的实时资源使用情况。

网络和卷管理

* **`docker network ls`**：列出所有 Docker 网络。

* **`docker network create <network_name>`**：创建一个新的 Docker 网络。
* **`docker volume ls`**：列出所有 Docker 卷。
* **`docker volume create <volume_name>`**：创建一个新的 Docker 卷。

清理系统

* **`docker system prune -a -f`**：删除所有未使用的容器、网络、镜像（包括悬空镜像）和构建缓存。
* **`docker volume prune -f`**：删除所有未使用的卷。

Docker Compose

启动和管理服务

* **`docker-compose up`**：启动定义在 `docker-compose.yml` 文件中的所有服务。
* **`docker-compose up -d`**：在后台启动所有服务。
* **`docker-compose down`**：停止并删除所有服务和网络。
* **`docker-compose stop`**：停止所有服务。
* **`docker-compose start`**：启动已停止的服务。
* **`docker-compose restart`**：重启所有服务。

构建和管理服务

* **`docker-compose build`**：根据 `docker-compose.yml` 文件构建或重新构建服务。
* **`docker-compose build --no-cache`**：不使用缓存构建服务。
* **`docker-compose logs`**：查看所有服务的日志。
* **`docker-compose logs <service_name>`**：查看指定服务的日志。
* **`docker-compose ps`**：列出所有服务的状态。
* **`docker-compose exec <service_name> <command>`**：在运行中的服务容器内执行命令。
* **`docker-compose run <service_name> <command>`**：在新容器中运行命令。

github workflows CI/CD

如果你成功编写了 `scripts/setup.sh` ，那么你现在的开发部署流程是：

1. 手动上传代码分支；
2. 提交 pr 请求；
3. 手动操作合并 master 分支；
4. 远程连接到你服务器的终端；
5. cd 到你目前项目所在目录；
6. 进行 git 拉取请求；
7. 手动执行 `setup.sh` 脚本命令。

可以看到，4~7 步都是服务器端所要执行的机械操作。我们可以进一步的将部署流程简化，也就是说，**我们可以仅仅执行 1~3 步，由 github 自动检测 master 分支的提交，然后自动在服务器上执行我们编写好的脚本。**

那么怎么来实现这一点呢？自然是借助 github workflow 来实现。

配置工作流的步骤其实很简单。首先在你的 nest 项目根目录下新建目录及文件 `.github/workflows/deploy.yml`，然后往其中编写如下内容：

```
name: Deploy-nest

on:
  push:
    branches:
      - master

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: "20"

      - name: Run setup script
        run: bash ./scripts/setup.sh
```

```
      - name: Deploy to server
        env:
          SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
          SSH_HOST: ${{ secrets.SSH_HOST }}
          SSH_USER: ${{ secrets.SSH_USER }}
        run: |
          mkdir -p ~/.ssh
          echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
          chmod 600 ~/.ssh/id_rsa
          ssh-keyscan $SSH_HOST >> ~/.ssh/known_hosts
          ssh $SSH_USER@$SSH_HOST "cd /path/to/your/project && git pull origin main && npm install && bash ./scripts/setup.sh"
```

这个工作流会在 `main` 分支被合并时触发，自动在服务器上拉取最新代码，执行 `npm install`，然后运行 `scripts/setup.sh` 文件。

此处的 `runs-on` 是 github workflow 预设的一个值，主要是用来指定用户运行的系统环境。你可以改成你自己服务器上的系统版本。

这里是每个步骤的详细说明：

1. **Checkout code**: 使用 `actions/checkout@v2` 来检出仓库代码。
2. **Set up Node.js**: 设置 Node.js 环境，指定版本为 20。
3. **Run setup script**: 运行 `scripts/setup.sh` 脚本进行 Docker 部署。
4. **Deploy to server**: 使用 SSH 部署到服务器。你需要在 GitHub 仓库的 Secrets 中设置 `SSH_PRIVATE_KEY`, `SSH_HOST`, 和 `SSH_USER` 这三个秘密变量。

* `SSH_PRIVATE_KEY`: 你的 SSH 私钥，用于连接服务器。
* `SSH_HOST`: 服务器的地址。
* `SSH_USER`: 用于登录服务器的用户名。

将这些 Secret 添加到 GitHub 仓库的设置中：

1. 打开你的 GitHub 仓库。
2. 点击 `Settings`。
3. 在左侧菜单中找到 `Secrets`，然后点击 `Actions`。
4. 点击 `New repository secret` 按钮，分别添加 `SSH_PRIVATE_KEY`, `SSH_HOST`, 和 `SSH_USER`。

后记

至此，算是把之前没有填好的坑基本上都填上了。希望能给大家带来一点点使用 Docker 部署 Nest.js 应用一点点参考的价值~~!

原文链接: https://juejin.cn/post/7375345026429845515