目录

使用 **Docker** 搭建 **Honkit** 环境

Docker 快速指南

关于 **Honkit**

honkit 是一个基于 Node.js、使用 Markdown 构建静态页面的命令行工具，通常被用于制作静态博客、软件/程序说明文档等。Honkit 是 Gitbook 的分支之一，兼容 Gitbook 的插件。我们推荐使用 Docker 搭建 Honkit 环境，使用 Typora 进行文档编写。

Honkit/Gitbook 指南: https://snowdreams1006.github.io/markdown/

Docker 指南: https://tsejx.github.io/devops-guidebook/deploy/docker/overview

Docker 官方文档 (强烈建议快速阅读): https://docs.docker.com/get-started/overview/ 和 https://docs.docker.com/desktop/

项目示例

Command Parser

文档预览: http://pmglab.top/commandParser/

代码仓库: https://github.com/Zhangliubin/commandParser-1.1

Genotype Blocking Compressor (GBC)

文档预览: http://pmglab.top/gbc/

代码仓库: https://github.com/Zhangliubin/GBC

配置环境

下载及使用 **Typora**

类型	地址
MacOs (免费版)	https://download.typora.io/mac/Typora-0.11.18.dmg
Windows x64 (免费版)	https://download.typora.io/windows/typora-update-x64-1117.exe
Windows x86 (免费版)	https://download.typora.io/windows/typora-update-ia32-1117.exe
Linux (免费版)	https://download.typora.io/linux/typora_0.11.18_amd64.deb
最新版	https://typoraio.cn

破解教程：https://github.com/taozhiyu/TyProAction/blob/main/README.zh.md

Typora 语法教程: https://support.typoraio.cn/zh/Markdown-Reference/

> [!WARNING|label:Typora 支持广泛的 html 语法和 markdown 语法]

> Honkit 不支持 Markdown 中的 `==内容==` 高亮语法。

安装并启动 **Docker**

Windows

Step1: 在 `控制面板-->所有控制面板项-->程序和功能-->启用或关闭 Windows 功能` 中开启 Hyper-V；

Step2: 前往 https://www.docker.com/get-started 下载、安装桌面版 Docker Desktop；

Step3: 前往 https://docs.microsoft.com/zh-cn/windows/wsl/install-manual#step-4---download-the-linux-kernel-update-package 下载内核更新包。

> [!NOTE|label:安装完成后需要重启设备]
>
> 官方指南: https://docs.docker.com/desktop/windows/install/

MacOS

Macos 系统前往 https://www.docker.com/get-started/ 下载安装，即可直接使用。

还可以使用 homebrew 安装：

```
# 安装 homebrew
/bin/zsh -c "$(curl -fsSL https://gitee.com/cunkai/HomebrewCN/raw/master/Homebrew.sh)"

# 使用 homebrew 安装 docker
brew install --cask docker
```

通过 **Dockerfile** 构建 **Honkit** 服务镜像

从 github 上下载资源：

```
# 下载 honkit-docker
git clone https://github.com/Zhangliubin/honkit-docker.git honkit-docker

# 进入下载的资源的文件夹
cd honkit-docker
```

> [!NOTE|label:下载 git]
>
> 下载 Git 工具，请参考：安装 Git

honkit-docker 的目录结构如下:

```
+- LICENSE          开源许可证
+- README.md        本文档 Web 的主页内容
+- SUMMARY.md       本文档 Web 的导航信息
+- book.json        本文档 Web 的配置信息
+- book.pdf         本文档的 pdf 版, 使用 honkit pdf 导出
+- docker           docker 镜像
```

```
+- multi-lang        多语言环境模版
+- single-lang       单语言环境模版
```

接着，我们通过 dockerfile 安装 honkit 服务：

```
# 构建名为 honkit 的镜像 (需要挂上梯子进行下载)
docker build -t honkit -f docker/Dockerfile docker/
```

> [!NOTE|label:如果无法构建该镜像 (通常是网络连接问题)，也可以选择下载打包好的镜像]
>
> - 实验室内网: http://192.168.30.2/download/honkit.tar.gz
> - 中山大学校内网: http://hpc.snplife.com/share/honkit.tar.gz
> - 校外使用百度云: https://pan.baidu.com/s/1MTPJEa67g1YkZ_6S88gf6w 提取码: 5b83
>
> 下载完成后，建议将该文件放在 honkit-docker/docker 目录下，使用以下语句加载镜像：
>
> ```
> # 加载打包好的镜像，自动命名为 honkit:latest
> docker load -i docker/honkit.tar.gz
> ```

使用 **Honkit**

以下操作请在项目文档文件夹 (例如: /commandParser/docs) 中进行，路径中的 `.` 代表当前文件夹路径。

> [!TIP|label:honkit-docker 实际上包含 3 个项目的资源]
>
> - 一个完整项目的示例: honkit-docker
> - 单语言环境模版: honkit-docker/single-lang
> - 多语言环境模版: honkit-docker/multi-lang
>
> 因此，这三个文件夹都支持下面的操作 2~4 (搭建服务、构建网页文件夹、创建 PDF)

Windows

1. 初始化环境

在终端 (cmd) 中输入以下指令：

```
# 语法: honkit init [workspace]
# 默认: honkit init .
docker run -v %cd%:/.honkit/ -w /.honkit/ --rm -it honkit init .
```

随后 `[workspace]` 文件夹中出现 `SUMMARY.md` 和 `README.md` 文件。这两个文件是启动 **Honkit** 必备的文件。

> [!TIP|label:从模版中构建初始环境]
>
> Honkit 依赖大量插件进行配置、美化。我们建议从已经配置好的模版中进行修改：
>
> - 单语言环境：honkit-docker/single-lang
> - 多语言环境：honkit-docker/multi-lang

> 将对应语言环境内的文件复制到当前 docs 目录下。

2. 搭建服务进行本地测试

在终端中输入以下指令：

```
# 语法: honkit serve [workspace] [output_dir]
# 默认: honkit serve . ./_book
docker run -it --init -p 4000:4000 -v %cd%:/.honkit/ -w /.honkit/ --rm honkit serve . ./_book
```

等待服务构建完成后 (即出现 `Serving book on http://localhost:4000`)，在浏览器中输入 `http://localhost:4000` 或 `http://127.0.0.1:4000/` 访问网站。

> [!DANGER|label:请勿将 `[workspace]` 和 `[output_dir]` 设置为同一个值]

> [!TIP|label:这个网页能否被其他人浏览?]
>
> 搭建好测试服务后，本机通过 `http://127.0.0.1:4000/` 可以访问该页面，同个局域网下的用户也可以通过 `http://局域网IP地址:4000` 访问该页面。对于外网环境，可以使用 花生壳域名代理服务 映射到公网上供其他用户访问。
>
> 当然，我们更建议使用专门的 Web 服务器进行部署访问，见 搭建 Web 服务器 一节。

3. 构建网页文件

在终端中输入以下指令：

```
# 语法: honkit build [workspace] [output_dir]
# 默认: honkit build . ./_book
docker run -v %cd%:/.honkit/ -w /.honkit/ --rm -it honkit build . ./_book
```

该指令会在 `[workspace]` 目录下生成 `[output_dir]` 文件夹，该文件夹内的文件即为网页资源。

> [!DANGER|label:请勿将 `[workspace]` 和 `[output_dir]` 设置为同一个值]

4. 创建 **PDF** 文件

在终端中输入以下指令：

```
# 语法: honkit pdf [workspace] [output]
# 默认: honkit pdf . ./book
docker run -v %cd%:/.honkit/ -w /.honkit/ --rm -it honkit pdf . ./assets/book.pdf
```

在单语言环境下，该指令会导出文件 `[output]` ；

在多语言环境下，该指令会导出文件 `[output]_[lang].pdf` (如果 `[output]` 包含了 `.pdf` 扩展名，则会去掉扩展名后重新生成)。

> [!TIP|label:导出 PDF 时隐藏网页页脚信息]

> 在导出 PDF 时建议将 book.json 文件中的 `pluginsConfig - pagefooter-freedom - hide` 设置为 `true`。

Macos 或 Linux

1. 初始化环境

在终端中输入以下指令：

```
# 语法: honkit init [workspace]
# 默认: honkit init .
docker run -v `pwd`:`pwd` -w `pwd` --rm -it honkit init .
```

随后 `[workspace]` 文件夹中出现 `SUMMARY.md` 和 `README.md` 文件。这两个文件是启动 **Honkit** 必备的文件。

> [!TIP|label:从模版中构建初始环境]
>
> Honkit 依赖大量插件进行配置、美化。我们建议从已经配置好的模版中进行修改：
>
> - 单语言环境：honkit-docker/single-lang
> - 多语言环境：honkit-docker/multi-lang
>
> 将对应语言环境内的文件复制到当前 docs 目录下。

2. 搭建服务进行本地测试

在终端中输入以下指令：

```
# 语法: honkit serve [workspace] [output_dir]
# 默认: honkit serve . ./_book
docker run -it --init -p 4000:4000  -v `pwd`:`pwd` -w `pwd` --rm honkit serve . ./_book
```

等待服务构建完成后 (即出现 `Serving book on http://localhost:4000`)，在浏览器中输入 `http://localhost:4000` 或 `http://127.0.0.1:4000/` 访问网站。

> [!DANGER|label:请勿将 `[workspace]` 和 `[output_dir]` 设置为同一个值]

> [!NOTE|label:端口占用时如何处理?]
>
> 本机端口被占用时，请尝试将 `4000:4000` 更换为 `5000:4000` 等 (`本机端口:服务端口`)。
>
> ```
> docker: Error response from daemon: Ports are not available: exposing port TCP 0.0.0.0:40
> 00 -> 0.0.0.0:0: listen tcp 0.0.0.0:4000: bind: address already in use.
> ERRO[0000] error waiting for container: context canceled
> ```

> [!TIP|label:这个网页能否被其他人浏览?]
>
> 搭建好测试服务后，本机通过 `http://127.0.0.1:4000/` 可以访问该页面，同个局域网下的用户也可以通过 `http://局域网IP地址:4000` 访问该页面。对于外网环境，可以使用 花生壳域名代理服务 映射到公网上供其他用户访问。

3. 构建网页文件

在终端中输入以下指令：

```
# 语法: honkit build [workspace] [output_dir]
# 默认: honkit build . ./_book
docker run -v `pwd`:`pwd` -w `pwd` --rm -it honkit build . ./_book
```

该指令会在 `[workspace]` 目录下生成 `[output_dir]` 文件夹，该文件夹内的文件即为网页资源。

> [!DANGER|label:请勿将 `[workspace]` 和 `[output_dir]` 设置为同一个值]

4. 创建 PDF 文件

在终端中输入以下指令：

```
# 语法: honkit pdf [workspace] [output]
# 默认: honkit pdf . ./book
docker run -v `pwd`:`pwd` -w `pwd` --rm -it honkit pdf . ./assets/book.pdf
```

在单语言环境下，该指令会导出文件 `[output]`；

在多语言环境下，该指令会导出文件 `[output]_[lang].pdf` (如果 `[output]` 包含了 `.pdf` 扩展名，则会去掉扩展名后重新生成)。

> [!TIP|label:导出 PDF 时隐藏网页页脚信息]
>
> 在导出 PDF 时建议将 book.json 文件中的 `pluginsConfig - pagefooter-freedom - hide` 设置为 `true`。

编写文档

1. 配置资源

1.1 修改网页基本信息

文档配置文件为文档根目录 `book.json`，将字段修改为本站点信息 (修改 `title`，`description`，`author`，`language`)。

其中，若网页主体语言为英语，可将 `language` 修改为 `en`。

1.2 修改插件信息

在配置文件 `plugins` 处填写导入的插件信息，在 `pluginsConfig` 处填写插件的配置信息。默认导入的插件列表如下：

插件名	描述
favicon-absolute	自定义网站 logo

edit-link	主页面左上角 “编辑本页” 按钮，可以跳转到 Github 仓库链接
search-plus	加强版搜索引擎，支持多语言搜索
expandable-chapters-smal	导航栏可折叠
back-to-top-button	页面内 “回到顶部” 按钮
page-toc-button	页面内导航条
katex	数学公式支持
hide-element	隐藏页面内的特定元素
prism	代码块配色
prism-themes	代码块配色主题
github-buttons	主页面右上角添加 “Github” 链接及图标
summary-logo	导航栏上面添加 Logo 及可跳转链接
code	代码可复制、显示行号
todo	使得 “任务列表” 样式不可编辑
flexible-alerts	扩展高亮语法
theme-comscore	页面内标题彩色
language-picker	主页面左上角多语言切换，"grid-columns" 用于控制每行显示的语言个数
pagefooter-freedom	页脚自定义
get-pdf	主页面左上角显示下载 PDF
password-pro	为页面添加密码

根据自己的需求增删插件。插件配置信息需要修改：

- 代码仓库地址：`edit-link` 的 `base`，`github-buttons` 的 `buttons-user` 和 `buttons-repo`
- 版权信息：`pagefooter-freedom` 的 `copyright`
  - 在多语言环境下，还需要修改子语言文件夹中的 `book.json` 文件
- 下载 PDF：`get-pdf` 的 `base`
- 网页导航 logo：`summary-logo` 的 `url` 和 `link`
- 网页密码：`password-pro`

请注意，涉及路径信息的参数尽量保持格式一致 (相对路径、绝对路径、网页路径)。为了节省带宽资源，本页面的所有下载链接 (PDF、honkit-docker 等) 都托管在 Github 或公共平台。对于较小的服务资源，也可以直接存放在 Web 服务器中。

1.3 多语言使用不同的插件配置

`multi-lang` 文件夹下，除了根路径的 `book.json` 外，在每个语言的文件夹内也可以配置 `book.json`，用于规定在该版本下的插件配置。

> [!TIP|label:什么情况下会用到多语言-多插件配置?]
>
> - 多语言环境下在每个语言环境中使用统一的版权声明 (例如模版文件的多语言环境)；
> - 不同的版本中为子页面添加密码、独立版权信息等。

1.4 设置页面密码 (password-pro 插件)

使用 `password-pro` 插件可以设置简易的网页密码。本插件有 4 个全局参数:

```
"password-pro": {
    "password": "123456",
    "tip": "请输入该页面的访问密码:",
    "errorTip": "密码错误, 请重新输入:",
    "reject": "当前页面拒绝访问: 身份验证失败."
},
```

`"password"` 表示为所有页面都添加密码。当传入的字符串为空字符串或不传入时表示不加密。

为指定页面添加密码:

```
"password-pro": {
    "README.md": "",
    "password": "20220611",
    "download.md": {
        "password": "12306",
        "tip": "下载资源需要验证密码:"
    },
    "command-line-interface.md": "complex12306",
    "tip": "请输入当前页面的访问密码:",
    "errorTip": "密码错误, 请重新输入:"
}
```

该语句表示 `README.md` 不加密，全局密码为 `20220611`， `download.md` 密码为 `12306`， `command-line-interface.md` 密码为 `complex12306` 。

即格式有两种:

```
"<page>": "<password>"
```

和

```
"<page>": {
    "password": "<password>",
    "tip": "<tip>",
    "errorTip": "<errorTip>",
    "reject": "<reject>"
},
```

在第二种格式中，缺少的字段将使用全局参数替代。在同一次访问中，只要单个页面正确输入了密码。则下次访问该页面也不需要密码。

1.5 扩展高亮语法使用说明

导入该插件时，支持使用彩色框提示。目前有 6 种主要样式:

- NOTE 样式

> [!NOTE] 包括 NOTE、COMMENT、TIP、WARNING、DANGER

- COMMENT 样式

> [!COMMENT]
>
> COMMENT 基于 NOTE 设置，但图标不一致

- TIP 样式

> [!TIP|label:请联系我] 修改标签内容，使用 `[!默认标签名|label:新标签名]`
>
> 例: `[!TIP|label:请联系我]`

- UPDATE 样式

> [!UPDATE|label:2022/06/10] UPDATE 基于 TIP 设置，但图标不一致

- WARNING 样式

> [!WARNING|style:callout] 修改提示框为精简提示框，使用 `[!默认标签名|style:callout]`
>
> 例: `[!WARNING|style:callout]`

- DANGER 样式

> [!DANGER|label:修改标签内容|style:callout]
>
> 例: `[!DANGER|label:修改标签内容|style:callout]`

```
# 上述样式对应的 markdown 代码
- NOTE 样式

> [!NOTE]
> 包括 NOTE、COMMENT、TIP、WARNING、DANGER

- COMMENT 样式

> [!COMMENT]
>
> COMMENT 基于 NOTE 设置，但图标不一致

- TIP 样式

> [!TIP|label:请联系我]
> 修改标签内容，使用 `[!默认标签名|label:新标签名]`
>
> 例: `[!TIP|label:请联系我]`

- UPDATE 样式

> [!UPDATE|label:2022/06/10]
> UPDATE 基于 TIP 设置，但图标不一致

- WARNING 样式

> [!WARNING|style:callout]
> 修改提示框为精简提示框，使用 `[!默认标签名|style:callout]`
>
> 例: `[!WARNING|style:callout]`

- DANGER 样式
```

```
> [!DANGER|label:修改标签内容|style:callout]
>
> 例: `[!DANGER|label:修改标签内容|style:callout]`
```

2. 编写语言列表 (多语言)

在多语言环境下，文档根目录 `LANGS.md` 声明了包含的语言：

```
# Languages

- [▫▫ 简体中文](zh/)
- [▫▫ English](en/)
```

对于更多语言的支持, 按照如上格式进行续写。 `_layouts/languages.html` 定义了在本文件中的第一个语言将作为站点的默认页面。

多语言实际上也可以用于制作面向不同用户群体的文档界面（如面向用户的文档和面向开发者的文档；面向普通用户的文档和面向管理员的文档）；也可以用于制作文档历史页面。例如：

```
# Languages

- [2022 年度工作日志](2022/)
- [2021 年度工作日志](2021/)
- [2020 年度工作日志](2020/)
```

> [!TIP|label:修改文档历史页面的样式及页面自动跳转]
>
> `_layouts/languages.html` 定义了该页面的样式。我们在第 23~24 行处添加了以下信息，以支持自动跳转:
>
> ```
> <!-- 设置自动跳转到第一个 url 处 -->
> <script type="text/javascript">if ( >= 1) {window.location.href="";}</script>
> ```

3. 制作导航文件 (SUMMARY.md)

在单语言环境下， `SUMMARY.md` 文件位于文档根目录下；在多语言环境下， `SUMMARY.md` 文件位于各个语言子文件夹的根目录下。

```
# Summary

### Part I

- [Section I](part1/README.md)
  - [Sub Section I](part1/section1.md)
  - [Sub Section II](part1/section2.md)

### Part II

- [Section I](part2/README.md)
  - [Sub Section I](part2/section1.md)
  - [Sub Section II](part2/section2.md)
```

```
### Part III

- [Section I](part3/README.md)
  - [Sub Section I](part3/section1.md)
  - [Sub Section II](part3/section2.md)
```

此文件用于制作网页左侧导航，一级标题不设地址；根据语法建议，二级标题应该以单独文件夹的 README.md 文件作为入口 (此建议不是强制性的，实际上任意的 markdown 文件都可以作为入口文件)。

由于页面内部也会自动按照文档级别生成导航条，因此建议左侧导航条至多到 三级/四级。

标题可以通过以下方式设定链接：

- 跳转到某个 md 页面: `[标题名](文件相对路径地址)` ；
- 跳转到某个 md 页面的指定锚点: `[标题名](文件相对路径地址#锚点)` ，锚点需要在页面内使用 `{#锚点}` 进行标记。请注意，锚点字符中不能用空格、点。

4. 制作子页面文件

子页面文件采用常规的 Markdown 格式书写。通过导航目录或者页面内跳转的方式进行连接。

请注意，在单语言环境下，文档根目录的 README.md 文件必不可少，它是网页的入口文件；在多语言环境下，每个语言子文件夹的 README.md 文件必不可少 (例如: `zh/README.md`)。

> [!NOTE|label:为什么 README.md 必不可少?]
>
> 文档根目录或语言子文件夹根目录中的 README.md 是该网页的入口 (例如，当网页资源挂载在 pmglab.top/honkit-docker 目录下时，输入: pmglab.top/honkit-docker 或 pmglab.top/honkit-docker.html 将跳转到该 README.md 文件对应的网页资源文件)。

5. 制作 PDF 文件

要提供页面左上角 “下载 PDF 文件” 功能，需要在本地预先导出。当使用：

```
# Windows
docker run -v %cd%:/.honkit/ -w /.honkit/ --rm -it honkit pdf . ./assets/book.pdf

# Macos 或 Linux
docker run -v `pwd`:`pwd` -w `pwd` --rm -it honkit pdf . ./assets/book.pdf
```

导出 PDF 文件时，单语言环境会生成 `./assets/book.pdf` 文件；多语言环境则会生成 `./assets/book_en.pdf` 和 `./assets/book_zh.pdf` 等。

此时，插件 `get-pdf` 只需要将 `base` 定位到 Web 服务器中的 `assets` 路径 (例如，下面使用的是 Github 仓库)，便能够正确连接。

```
"get-pdf": {
    "base": "https://github.com/Zhangliubin/honkit-docker/tree/main/multi-lang/assets/",
    "prefix": "book",
```

```
    "label": ""
}
```

这里的 `prefix` 就是导出指令中输出文件的文件名。当修改了输出文件名时，请同步修改 `book.json` 中的 `get-pdf` 的 `prefix` 属性字段。

> [!TIP|label:制作 PDF 封面]
>
> 在 Word 中使用 A4 纸样式设计好封面，并导出为 `cover.jpg` 到文档根目录下。

搭建 **Web** 服务器

构建网页数据后，需要部署 Web 服务器以响应用户请求。Web 服务器一般也称为 http 服务器，如 Apache、IIS、Nginx等。此外，还有存放和运行系统程序的应用服务器，负责处理程序中的业务逻辑，如 Tomcat、Weblogic、Jboss（现在大多数应用服务器也包含了web服务器的功能）。

本文介绍使用 apache 服务器搭建 Web 服务器的方式。

在 **Mac** 中使用 **apache** 服务器

启动服务

MacOS 中自带了 apache 服务器，只需要输入以下指令即可启动：

```
sudo apachectl start
```

此时，在浏览器中输入：`http://127.0.0.1` 或 `http://127.0.0.1:80` ，若出现 `It works!` ，说明服务已经启动。默认情况下，该服务器资源存放在：`/Library/WebServer/Documents` 中，将 _book 的内容移动到该文件夹下即可实现 Web 访问。

> [!TIP|label:添加所有者权限，以解决频繁要求输入密码的问题]
>
> `/Library/WebServer/Documents` 文件夹的所有者是 `root` ，这导致我们增删文件时都需要输入密码。一种解决方案是在文件夹鼠标右键 > 显示简介 > 共享与权限 中添加当前用户权限。此外，可以为该文件夹制作快捷方式、重命名，并放置到其他路径。

打开文件访问视图

许多网站除了可以显示 HTTP 网页 (站点服务器路径下的 index.html 文件) 外，还可以使用 “下载模式” 在该路径下显示文件的名称。打开文件访问视图需要修改配置文件: `/etc/apache2/httpd.conf` (通过 访达菜单栏 > 前往 > 前往文件夹打开 该文件)。搜索以下行字段：

```
Options FollowSymLinks Multiviews
```

将其修改为：

```
Options Indexes FollowSymLinks Multiviews
```

重启 apache 服务：

```
sudo apachectl restart
```

此时，目录列表就可以正常访问。

停止服务

停止 apache 服务，请输入：

```
sudo apachectl stop
```

使用 **Docker** 搭建 **apache** 服务器

对于 Linux 或 Windows 设备，我们建议使用 Docker 搭建 Apache 微服务。

从 Docker Hub 上拉取镜像：

```
docker pull httpd:alpine
```

将 honkit-docker/_book 的路径进行挂载：

```
# Windows
docker run -p 8080:80 --rm -v %cd%/_book:/usr/local/apache2/htdocs/ -d --name apache-server htt
pd:alpine

# MacOs 或 Linux
docker run -p 8080:80 --rm -v `pwd`/_book:/usr/local/apache2/htdocs/ -d --name apache-server ht
tpd:alpine
```

此时，在浏览器中输入： `http://127.0.0.1:8080` 访问相应的 Web 资源。修改 `8080` 可以更改映射的端口。

> [!TIP|label:同时挂载多个路径]
>
> 同时挂载多个路径可以通过添加多个挂载点 -v 实现，以 MacOs 为例：
>
> ```
> docker run --name apache-server -p 8080:80 --rm \
> -v `pwd`/_book:/usr/local/apache2/htdocs/honkit-docker \
> -v `pwd`/single-lang/_book:/usr/local/apache2/htdocs/single-lang \
> -v `pwd`/multi-lang/_book:/usr/local/apache2/htdocs/multi-lang \
> -d httpd:alpine
> ```

停止 apache 服务，请输入：

```
docker stop apache-server
```

Docker 简介

Docker 是一种新兴的虚拟化方式，跟传统的虚拟化方式相比具有众多的优势。

- 更高效的利用系统资源：由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销，Docker 对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传统虚拟机技术更高效。因此，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。
- 更快速的启动时间：传统的虚拟机技术启动应用服务往往需要数分钟，而 Docker 容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。
- 一致的运行环境：开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些 bug 并未在开发过程中被发现。而 Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现 「这段代码在我机器上没问题啊」 这类问题。
- 持续交付和部署：对开发和运维人员来说，最希望的就是一次创建或配置，可以在任意地方正常运行。使用 Docker 可以通过定制应用镜像来实现持续集成、持续交付、部署。开发人员可以通过 Dockerfile 来进行镜像构建，并结合持续集成系统进行集成测试，而运维人员则可以直接在生产环境中快速部署该镜像，甚至结合持续部署系统进行自动部署。而且使用 Dockerfile 使镜像构建透明化，不仅仅开发团队可以理解应用运行环境，也方便运维团队理解应用运行所需条件，帮助更好的生产环境中部署该镜像。
- 更轻松的迁移：由于 Docker 确保了执行环境的一致性，使得应用的迁移更加容易。Docker 可以在很多平台上运行，无论是物理机、虚拟机、公有云、私有云，甚至是笔记本，其运行结果是一致的。因此用户可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。
- 更轻松的维护和扩展：Docker 使用的分层存储以及镜像的技术，使得应用重复部分的复用更为容易，也使得应用的维护更新更加简单，基于基础镜像进一步扩展镜像也变得非常简单。此外，Docker 团队同各个开源项目团队一起维护了一大批高质量的 官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大的降低了应用服务的镜像制作成本。

基本概念

镜像

镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

因为镜像包含操作系统完整的 root 文件系统，其体积往往是庞大的，因此在 Docker 设计时，就充分利用 Union FS 的技术，将其设计为分层存储的架构。所以严格来说，镜像并非是像一个 ISO 那样的打包文件，镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，由多层文件系统联合组成。

镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

容器

镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的 类 和 实例 一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立于宿主的系统下操作一样。这种特性使得容器封装的应用比直接在宿主运行更加安全。也因为这种隔离的特性，很多人初学 Docker 时常常会混淆容器和虚拟机。

前面讲过镜像使用的是分层存储，容器也是如此。每一个容器运行时，是以镜像为基础层，在其上创建一个当前容器的存储层，我们可以称这个为容器运行时读写而准备的存储层为 容器存储层。

容器存储层的生存周期和容器一样，容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失。

按照 Docker 最佳实践的要求，容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用==数据卷==或者==绑定宿主目录==，在这些位置的读写会跳过容器存储层，直接对宿主（或网络存储）发生读写，其性能和稳定性更高。

数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。因此，使用数据卷后，容器删除或者重新运行之后，数据却不会丢失。

仓库

镜像构建完成后，可以很容易的在当前宿主机上运行，但是，如果需要在其它服务器上使用这个镜像，我们就需要一个集中的存储、分发镜像的服务，Docker Registry 就是这样的服务。 一个 Docker Registry 中可以包含多个 仓库（Repository）；每个仓库可以包含多个 标签（Tag）；每个标签对应一个镜像。

通常，一个仓库会包含同一个软件不同版本的镜像，而标签就常用于对应该软件的各个版本。我们可以通过 <仓库名>:<标签> 的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 latest 作为默认标签。 以 Ubuntu 镜像 为例，ubuntu 是仓库的名字，其内包含有不同的版本标签，如，16.04, 18.04。我们可以通过 ubuntu:16.04，或者 ubuntu:18.04 来具体指定所需哪个版本的镜像。如果忽略了标签，比如 ubuntu，那将视为 ubuntu:latest。 仓库名经常以两段式路径形式出现，比如 kingfalse/java8，前者往往意味着 Docker Registry 多用户环境下的用户名，后者则往往是对应的软件名。但这并非绝对，取决于所使用的具体 Docker Registry 的软件或服务。

安装 Docker

使用镜像创建一个容器，该镜像必须与 Docker 宿主机系统架构一致，例如 Linux x86_64 架构的系统中只能使用 Linux x86_64 的镜像创建容器。Windows、macOS 除外，其使用了 binfmt_misc 提供了多种架构支持，在 Windows、macOS 系统上 (x86_64) 可以运行 arm 等其他架构的镜像。

安装指南详见: https://docs.docker.com/desktop/

使用镜像

从镜像仓库拉取镜像

```
docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]
```

- Docker 镜像仓库地址：地址的格式一般是 <域名/IP>[:端口号]。默认地址是 Docker Hub(docker.io)。
- 仓库名：仓库名是两段式名称，即 <用户名>/<软件名>。对于 Docker Hub，如果不给出用户名，则默认为 library，也就是官方镜像。

案例：

```
# 拉取官方 ubuntu 镜像
docker pull ubuntu:18.04

# 拉取用户 kingfalse 构建的 oracle-java8 镜像
docker pull kingfalse/java8
```

查看本地镜像

Docker 常用以下四种查看本地镜像：

- 查看本地顶层镜像: `docker image ls` 或 `docker images` 。输出日志由 REPOSITORY、TAG、IMAGE ID、CREATED、SIZE 5个字段组成。一个镜像可以对应多个 REPOSITORY，IMAGE ID 是镜像的唯一标识。
- 查看所有镜像： `docker image ls -a` 。它包含了顶层镜像和中间层镜像。中间层镜像是为了加快镜像构建、重复利用资源而存在的，一般不建议主动删除中间层镜像。
- 根据仓库名列出镜像： `docker image ls 仓库名` 或 `docker image ls 仓库名:标签` 。
- 按构建时间过滤镜像： `docker image ls -f since=仓库名:标签` 列出在仓库名:标签之后构建的镜像。-f 全称是 --filter，将 since 替换为 before 可以列出之前构建的镜像。

> (1) REPOSITORY、TAG 都为 的镜像是虚悬镜像，即失去价值的镜像。可能是由于镜像名被赋予了新的镜像，或镜像构建过程中出现错误导致的。可以使用 `docker image prune` 批量删除虚悬镜像。
>
> (2) 添加 --digests，可以显示镜像的信息摘要 (DIGEST) sha256。
>
> (3) 添加 -q，只显示符合条件的镜像 ID。

删除镜像

```
docker image rm <镜像1> <镜像2> ...
```

- <镜像> 可以是镜像短 ID、镜像长 ID、镜像名或镜像摘要。

此外，它还支持与“查看本地镜像”功能进行结合（使用 -q 表示只列出 ID），实现批量删除：

```
docker image rm $(docker image ls -q [Limit])
```

删除虚悬镜像：`docker image prune`

定制镜像

案例 **1**：构建基于 **ubuntu 18.04** 的 **Oracle-JDK8** 环境

需要将 Dockerfile 与 jdk-8u261-linux-x64.tar 文件放在同一目录下，Dockerfile 文件内容：

```
FROM ubuntu:18.04

MAINTAINER suranyi suranyi.sysu@gmail.com

LABEL create_time=2020.9.5 jdk_version=1.8.0_261-b12

COPY jdk-8u261-linux-x64.tar /home/Java/

RUN tar -xf /home/Java/jdk-8u261-linux-x64.tar -C /home/Java/ \
    && rm /home/Java/jdk-8u261-linux-x64.tar

ENV PATH=/home/Java/jdk1.8.0_261/bin:$PATH

CMD java -version && bash
```

使用命令构建镜像 (如果文件名是标准的 Dockerfile)：

```
docker build -t oracle-jdk8 .
```

案例 **2**：构建基于 **ubuntu 18.04** 的 **Anaconda3** 环境

需要将 Dockerfile 与 Anaconda3-2020.07-Linux-x86_64.sh 放在同一目录下，Dockerfile 文件内容：

```
FROM ubuntu:18.04

MAINTAINER suranyi suranyi.sysu@gmail.com

COPY Anaconda3-2020.07-Linux-x86_64.sh /home/

ENV PATH=/home/anaconda3/bin:$PATH

RUN sh /home/Anaconda3-2020.07-Linux-x86_64.sh -b -p /home/anaconda3 \
    && rm /home/Anaconda3-2020.07-Linux-x86_64.sh \
    && conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/ \
```

```
    && conda config --add channels bioconda \
    && conda config --add channels conda-forge \
    && conda update --all
```

使用命令构建镜像 (如果文件名是标准的 Dockerfile)：

```
docker build -t anaconda3 .
```

Dockerfile 语法

Dockerfile 是一个用于定制每一层所添加配置、文件的文本文件，其内包含了一条条的指令，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

- **FROM** 指定基础镜像（必备指令）：以一个镜像为基础，定制新的镜像。如果不需要基础镜像，可以使用 `scratch`，它表示一个空白镜像。对于 Linux 下静态编译的程序来说，并不需要操作系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 `FROM scratch` 会让镜像体积更加小巧。
- **MAINTAINER** 维护者信息
- **LABEL** 描述信息
- **RUN** 执行命令：用来执行命令行命令，每一次的 RUN 会生成一份中间层镜像（注意，凡是相同指令的镜像，Docker 都不会重复构建，而是采用之前的缓存）。一般使用 \ && 分割不同的命令组，而不是为每一次命令写一次 RUN （会产生非常多臃肿的镜像中间层）。

  > apt-get 是最常见的指令，使用时需要注意：
  >
  > 1. 用户不应该升级使用 apt-get upgrade 批量升级包。它应该交由底层的镜像进行维护。如果需要升级某个特定的包，使用 `apt-get install -y 包名`。
  > 2. `RUN apt-get update` 应该和 `apt-get install 包名 -y` 组合成一条语句。当 `RUN apt-get update` 单独作为一条语句时，其余镜像构建时相同的语句都会链接到最早构建的镜像，导致后续 `apt-get install 包名 -y` 使用的是旧版本的 `apt-get`。
  > 3. 使用完成后，应该使用 `apt-get clean` 清除缓存包。官方的 Debian 和 Ubuntu 镜像会自动运行 apt-get clean，所以不需要显式的调用 `apt-get clean`。
- **COPY** 拷贝文件：`COPY [--chown=<user>:<group>] <源路径1> ... <源路径n> <目标路径>` 。从构建上下文目录中 <源路径> 的文件或目录复制到新的一层的镜像内的 <目标路径> 位置（如果源路径为文件夹，复制的时候不是直接复制该文件夹，而是将文件夹中的内容复制到目标路径）。`--chown=<user>:<group>` 参数用于修改文件所属用户及所属组。
- **ADD** 更高级的复制文件：用法与 COPY 一致，但是 ADD 会自动解压文件，使得镜像构建缓存失效，从而可能会令镜像构建变得比较缓慢。
- **CMD** 容器启动命令：容器启动时，自动执行的指令。有两种格式
  - `CMD <命令>`：使用 shell 执行命令。例如：ubuntu 中默认的 CMD 是 /bin/bash，直接使用 `docker run -it ubuntu` 会进入 bash。
  - `CMD ["可执行文件", "参数1", "参数2"...]`：使用 exec 执行。例如：在包含 anaconda3 的环境的中使用 `CMD ["ipython"]`，则启动该容器后会直接进入 ipython 环境。

> 特别注意，容器不是虚拟机，它的所有程序都应该是前台运行的。诸如 `CMD service nginx start` 是 upstart 以守护进程形式启动 nginx 服务。但是它的主进程是 sh，命令结束后 sh 进程就结束了，容器就会退出。
>
> 正确做法：CMD ["nginx", "-g", "daemon off;"]，以前台形式运行。

- **ENTRYPOINT** 程序入口点：格式与 CMD 一致，Dockerfile 只允许有一个 ENTRYPOINT。将 CMD 改造为 `<ENTRYPOINT> <CMD>`，最佳用处是设置镜像的主命令，允许将镜像当成命令本身来运行（用 `CMD` 提供默认选项）。

```
# example: 如果没有传入参数，则执行 CMD 的 --help。

ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

- **ENV** 设置环境变量： `ENV <key> <value>` 和 `ENV <key1>=<value1> <key2>=<value2>...`
- **VOLUME** 定义匿名卷： `VOLUME <路径>` 和 `VOLUME ["<路径1>", "<路径2>"...]` 。为了防止运行时用户忘记将动态文件所保存目录挂载为卷，可以在 Dockerfile 中事先指定某些目录挂载为匿名卷，这样在运行时如果用户不指定挂载，其应用也可以正常运行，不会向容器存储层写入大量数据。

> VOLUME 指令不能挂载主机中指定的目录，这是为了保证 Dockerfile 的可一致性，因为不能保证所有的宿主机都有对应的目录。
>
> `VOLUME <路径>` 指令并不是持续化挂载，它只能改变一层的挂载状态。实际使用时，一般将 VOLUME 写在最后 (可以在 CMD, ENTRYPOINT，不受入口指令影响)。

- **EXPOSE** 暴露端口： `EXPOSE <端口1> [<端口2>...]` 。EXPOSE 声明容器打算使用什么端口，并不会自动在宿主进行端口映射。映射端口需要使用 `-p <宿主端口>:<容器端口>` 。
- **WORKDIR** 指定工作目录：使用 `RUN cd 路径` 时，只会影响该中间层的内存变化，不会作用到下一层的 `RUN` 。如果需要改变以后各层的工作目录的位置，那么应该使用 WORKDIR 指令。为便于阅读、排错、维护，应使用绝对路径。
- **USER** 指定当前用户：与 WORKDIR 一样，会改变之后各层的执行用户。该用户需要事先建立。

```
# example
RUN groupadd -r redis && useradd -r -g redis redis
USER redis
RUN [ "redis-server" ]
```

- **ONBUILD** 镜像触发器：特殊的指令，它后面跟的是其它指令，如 `RUN` , `COPY` 等。这些指令在当前镜像构建时并不会被执行。只有以当前镜像为基础镜像，去构建下一级镜像的时候才会被执行。

通过 **Dockerfile** 构建镜像

在 Dockerfile 文件所在的目录执行：

```
docker build -t <镜像名> .
```

如果 Dockfile 文件名为其他，则需要添加参数 `-f` 指定文件：

```
docker build -f <Dockerfile文件名> -t <镜像名> .
```

指令最后的 `.` 代表上下文环境，Docker 会发送上下文环境的数据用于构建镜像。如无必要，应当尽量使用只包含 Dockerfile 文件的目录 （也可以将不需要的文件按行写入同级目录下 `.dockerignore` 文件中）。

▶ 如何理解指令中的"上下文环境"?

当我们进行镜像构建的时候，并非所有定制都会通过 RUN 指令完成，经常会需要将一些本地文件复制进镜像，比如通过 COPY 指令、ADD 指令等。而 docker build 命令构建镜像，其实并非在本地构建，而是在服务端，也就是 Docker 引擎中构建的。那么在这种客户端/服务端的架构中，如何才能让服务端获得本地文件呢？

这就引入了上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，docker build 命令得知这个路径后，会将路径下的所有内容打包，然后上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件。

</details>

扩展：多阶段构建

Docker 希望最终的可执行文件放到一个最小的镜像（比如 `alpine` ）中去执行，怎样得到最终的编译好的文件呢？基于 `Docker` 的指导思想，我们需要在一个标准的容器中编译，比如在一个 Ubuntu 镜像中先安装编译的环境，然后编译，最后也在该容器中执行即可。

但是如果我们想把编译后的文件放置到 `alpine` 镜像中执行呢？我们就得通过 Ubuntu 镜像将编译完成的文件通过 `cp容器ID:源路径 目标路径` 挂载主机上，然后我们再将这个文件挂载到 `alpine` 镜像中去。这种解决方案理论上肯定是可行的，但是这样的话在构建镜像的时候我们就得定义两步了：第一步是先用一个通用的镜像编译镜像，第二步是将编译后的文件复制到 `alpine` 镜像中执行，而且通用镜像编译后的文件在 `alpine` 镜像中不一定能执行。

Docker 官方提供了一种多阶段构建的方法，在 COPY 中使用 `--from=` 参数声明之前的镜像，并将指定的数据导入到后续镜像中，从而生成精简的最小镜像。

```
# 先使用 golang 编译文件
FROM golang AS build-env
ADD . /go/src/app
WORKDIR /go/src/app
RUN go get -u -v github.com/kardianos/govendor
RUN govendor sync
RUN GOOS=linux GOARCH=386 go build -v -o /go/src/app/app-server

# 再将编译后的文件拷贝到 alpine 中
FROM alpine
RUN apk add -U tzdata
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai  /etc/localtime
COPY --from=build-env /go/src/app/app-server /usr/local/bin/app-server
EXPOSE 8080
CMD [ "app-server" ]
```

查看镜像构建记录

```
docker history <镜像 ID> [--no-trunc]
```

使用 --no-trunc 显示完成的命令构建过程。

导入与导出本地镜像

Docker 还提供了 docker save 和 docker load 命令，用以将镜像保存为一个文件，然后传输到另一个位置上，再加载进来。这是在没有 Docker Registry 时的做法，现在已经不推荐，镜像迁移应该直接使用 Docker Registry，无论是直接使用 Docker Hub 还是使用内网私有 Registry 都可以。

- 保存镜像：`docker save 镜像ID -o 输出文件名.tar`
- 保存镜像并压缩：`docker save 镜像ID | gzip > 输出文件名.tar.gz`
- 载入镜像：`docker load -i 镜像存档名`
- 通过网络传输镜像：`docker save 镜像名 | gzip | pv | ssh <用户名>@<主机名> 'cat | docker load'`

清除镜像缓存

docker 会缓存镜像构建中每一层的信息，在下一次构建时起到加速作用。但是，当中间层数据发生变动时，缓存机制则会导致构建出来的镜像及容器不是预期的数据文件，通过下列语句清除镜像缓存:

```
docker system prune --volumes
```

操作容器

启动容器有两种方式，一种是基于镜像新建一个容器并启动，另外一个是将在终止状态（`stopped`）的容器重新启动。因为 Docker 的容器实在太轻量级了，很多时候用户都是随时删除和新创建容器。

创建并启动

直接使用 `docker run 镜像名` 可以创建一个容器，该容器结束后不会删除。一般会使用：

```
docker run -it --rm 镜像名
```

- -it 表示进入交互式终端，--rm 表示结束后删除容器。还可以使用 `--name 容器名` 设置容器名。
- 如果需要 Docker 在后台运行而不是直接把执行命令的结果输出在当前宿主机下，可以通过添加 `-d` 参数来实现（守护态运行）。进入守护态的容器可以使用 `docker exec -it 容器名 bash` 进入。
- 使用 -P 标记时，Docker 会随机映射一个端口到内部容器开放的网络端口；使用 `-p hostPort:containerPort` 将主机端口映射到容器端口，类似的还有 `ip:hostPort:containerPort` 和 `ip::containerPort`（指定地址的任意端口映射到容器端口）。
- -m 10g: 设置该容器最多使用的内存量；
- --cpus=2: 设置该容器最多使用的 CPU 核心数；

其他操作

- 查看容器：`docker container ls`，显示当前正在运行的容器。添加 -a 参数显示所有容器。
- 启动已终止的容器：`docker container start`
- 删除容器：`docker container rm <容器ID/容器名> <容器ID/容器名> ...`
- 删除所有已终止的容器：`docker container prune`
- 终止容器：`docker stop ID`
- 导出容器：`docker export ID > 输出文件名.tar`
- 导入容器为镜像：`cat 容器存档名 | docker import - 镜像名`，容器快照文件将丢弃所有的历史记录和元数据信息，从容器快照文件导入时可以重新指定标签等元数据信息。`docker load -i 镜像存档名` 将保存完整记录
- 拷贝容器中的数据到本地：`docker cp 容器ID:源路径 目标路径`

容器互联

创建一个 Docker 网络，实现多个容器之间的相互连接：

```
docker network create -d bridge 网络名
```

-d 参数指定 Docker 网络类型，有 bridge、overlay。在运行容器时，使用 `--network 网络名` 连接到指定的网络。

挂载数据

> [!NOTE|label:--mount 还是 -v]
>
> 如果需要指定卷驱动程序选项，则必须使用 `--mount` 。
>
> - `-v` 或 `--volume`：由三个字段组成，以冒号 (:) 分隔。这些字段必须以正确的顺序排列，并且每个字段的含义并不直观。
>   - 对于命名卷，第一个字段是卷的名称，在给定的主机上是唯一的。对于匿名卷，将省略第一个字段。
>   - 第二个字段是文件或目录在容器中的挂载路径。
>   - 第三个字段是可选的，并且是逗号分隔的选项列表，例如 `ro` 。这些选项将在下面讨论。
> - --mount：由多个键/值对组成，用逗号分隔，每对均由一个 `<key>=<value>` 组成。
>   - 挂载的 `type` ，可以是 `bind` ， `volume` 或 `tmpfs` 。
>   - 挂载的 `source` 。对于命名卷，这是卷的名称。对于匿名卷，将省略此字段。可以指定为 `source` 或 `src` 。
>   - `destination` 表示文件或目录在容器中的挂载路径。可以指定为 `destination` ， `dst` 或 `target` 。
>   - `readonly` 选项（如果存在）使绑定挂载以只读方式挂载到容器中。
>   - `volume-opt` 选项，可以多次指定，由选项名称及其值组成的键值对。

挂载主机路径

挂载主机目录/文件虽然可以实现容器访问主机文件，但其本质上是通过服务器发送文件实现的，对 IO 性能有一定影响。

简单的挂载指令为：`-v [宿主机路径]:[容器路径]`，默认权限为读写。

挂载主机目录

在创建一个容器时，使用 `--mount` 标记可以指定挂载一个本地主机的目录到容器中去：

```
--mount type=bind,source=/src/webapp,target=/usr/share/nginx/html
```

Docker 挂载主机目录的默认权限是读写，用户也可以通过增加 readonly 指定为只读：

```
--mount type=bind,source=/src/webapp,target=/usr/share/nginx/html,readonly
```

注意：这里的 source、target 都必须是 ==绝对路径==，各个键值对中也不能有空格！

挂载主机文件

挂载主机文件和目录的操作一样：

```
--mount type=bind,source=$HOME/.bash_history,target=/root/.bash_history
```

挂载多个分散文件时，可以重复写多个 `--mount` 语句。

挂载数据卷

数据卷是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：数据卷可以在容器之间共享和重用；对数据卷的修改会立马生效；对数据卷的更新，不会影响镜像；数据卷默认会一直存在，即使容器被删除。数据卷的使用，类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会复制到数据卷中（仅数据卷为空时会复制）。

> [!TIP|label:IO密集型任务建议使用挂载数据卷]
>
> 当您的应用程序在Docker Desktop上需要高性能I/O时。卷存储在Linux虚拟机中，而不是主机中，这意味着读写具有更低的延迟和更高的吞吐量

- 创建数据卷：`docker volume create 数据卷名`
- 查看所有数据卷：`docker volume ls`
- 查看指定数据卷的信息：`docker volume inspect 数据卷名`
- 启动一个挂载数据卷的容器：在启动容器的 run 指令中，`--mount type=volume,source=数据卷名,target=/usr/share/`
- 删除数据卷：`docker volume rm 数据卷名`
- 批量删除无效数据卷：`docker volume prune`
- 查看数据卷的具体信息：`docker inspect 容器ID`，数据卷的信息记录在 "Mounts" 下

案例

使用 **Docker** 自动化部署 **JavaGBC** 测试环境

(1) 文件架构

```
project
├─ .history: 历史版本
├─ test: 测试数据及工具
│   ├─ Dockerfile_javagbc_compare_env: 其他工具测试环境的 Dockerfile 文件
│   ├─ data: 测试数据 (约 27.62 GB)
│   └─ .dockerignore: 测试数据不作为上下文的内容，而是直接挂载到主机目录
├─ JavaGBC: JavaGBC 源码
└─ JavaGBC.jar: JavaGBC 可执行 jar 包
```

(2) Dockerfile 文件

Dockerfile_javagbc_compare_env 文件内容如下：

```
FROM ubuntu:18.04

MAINTAINER suranyi suranyi.sysu@gmail.com

RUN apt-get update && apt-get install python3 -y \
    && apt-get --yes install build-essential git cmake wget libcurl4-gnutls-dev zlibc zlib1g zl
ib1g-dev libbz2-dev liblzma-dev \
    && cd /home \
    && git clone https://github.com/refresh-bio/GTShark.git gtshark \
    && git clone https://github.com/refresh-bio/GTC.git gtc \
    && git clone https://github.com/samtools/htslib.git htslib \
    && git clone https://github.com/richarddurbin/pbwt pbwt \
    && git clone https://github.com/lh3/bgt.git bgt \
    && cd /home/gtshark && ./install.sh && make \
    && cd /home/gtc && ./install.sh && make \
    && cd /home/htslib && make \
    && cd /home/pbwt && make \
    && cd /home/bgt && make

ENV PATH=/home/bgt:/home/pbwt:home/gtc:/home/gtshark:/home/htslib:$PATH
```

(3) .dockerignore 文件

```
./data/
```

(4) 构建镜像

```
cd /project/test
docker build -f Dockerfile_javagbc_compare_env -t javagbc_compare_env .
```

(5) 创建容器并挂载数据

```
docker run -it --rm --mount type=bind,source=/project/data,target=/mnt/,readonly javagbc_compar
e_env
```

使用 **Docker** 搭建可以调整 **JVM** 运行参数的运行环境

Dockerfile 文件如下：

```
FROM bellsoft/liberica-openjdk-alpine:17

MAINTAINER suranyi suranyi.sysu@gmail.com

ENV JAVA_OPTS="-Xms4g -Xmx4g"

RUN echo "java \$JAVA_OPTS -jar \$@" > /setup.sh

ENTRYPOINT ["/bin/sh", "/setup.sh"]
```

启动时，通过参数 `-e JAVA_OPTS="-Xms4g -Xmx4g"` 控制 JVM 的启动参数。

```
# Macos 或 Linux
docker run -v `pwd`:`pwd` -w `pwd` --rm -it -e JAVA_OPTS="-Xms4g -Xmx4g" [image] [options]

# Windows
docker run -v %cd%:%cd% -w %cd% --rm -it -e JAVA_OPTS="-Xms4g -Xmx4g" [image] [options]
```

搭建仅运行 **Java** 程序的运行环境

Dockerfile 文件如下：

```
FROM bellsoft/liberica-openjdk-alpine:17

MAINTAINER suranyi suranyi.sysu@gmail.com

LABEL create_time=2022.06.15

RUN wget http://pmglab.top/gbc/download/gbc.jar -O /gbc.jar

ENTRYPOINT ["java", "-XshowSettings:vm", "-XX:InitialRAMPercentage=100", "-XX:MaxRAMPercentage=
100", "-jar", "/gbc.jar"]
```

输入时，通过参数 `-m 4g` 控制堆内存大小，JVM 会自动使用全部的堆内存。