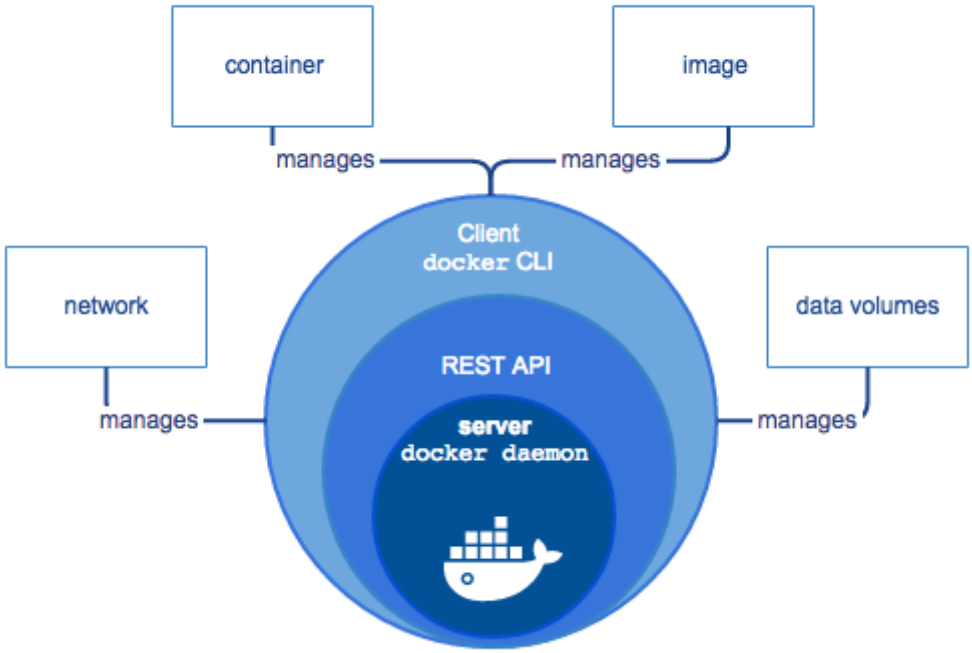
Docker 下一代构...

这是本专栏的第三部分：镜像篇，共 8 篇。前三篇我分别为你介绍了如何对 Docker 镜像进行生命周期的管理，以及如何使用 Dockerfile 进行镜像的构建和分发。本篇，我来为你介绍 Docker 的下一代构建系统——BuildKit，带你了解 Docker 构建系统的发展方向及掌握最新核心特性。下面我们一起进入本篇的学习。

Docker 整体结构介绍



（图片来源：Docker overview）

Docker 的整体结构如上图所示，可以看到它是一个 C/S 的架构，我们平时使用的 docker 命令是它的客户端，通过 API 与它的服务端 `docker daemon` 进行交互。

以我们前面介绍的构建镜像的动作为例：

先写一个 Dockerfile，然后执行 `docker build` 命令，通过一系列的构建过程，最终构建出来了我们预期的镜像。

那么我们来看看在这个过程中到底发生了些什么：

想要深入了解 C/S 架构的系统，一定要先了解其 API，我们来看看 Docker 构建镜像的 API，在线文档地址是：

https://docs.docker.com/engine/api/v1.40/#operation/ImageBuild

构建镜像的 API 是 /build，请求方法是 POST，其请求参数基本是 `docker build` 命令所支持的参数，请求体则是 tar 归档文件，支持不压缩或使用 gzip、bzip2 及 xz 等方式压缩。

由此可以看出，在此过程中，最核心的便是 Docker 的构建系统了。我们来一起了解下。

Docker 构建系统介绍

Docker 现有的构建系统我们称它为 builder v1 好了，它是从 Docker 发布之初就作为 Docker 的一项核心功能存在的。为了更好地理解其行为及更好地了解下一代构建系统，我们不妨深入源码一探究竟。

这里我们使用最新的 Docker CE v19.03.5 版本的代码。

首先我们需要从 https://github.com/docker/docker-ce.git 上 clone Docker CE 的源码，并切换至 v19.03.5 的 tag，接下来正式开始探究源码。

入口

Docker CLI 是我们与 Docker Daemon 交互的重要手段，关于构建镜像，我们所熟知的便是 `docker build` 或 `docker image build`。在 Docker CE v19.03 时，增加了一个 `docker builder build`，但其实它们都是一样的，只是做了 alias 罢了。

复制

```
// components/cli/cmd/docker/docker.go#L231
if v, ok := aliasMap["builder"]; ok {
    aliases = append(aliases,
        [2][]string{{"build"}, {v, "build"}},
        [2][]string{{"image", "build"}, {v, "build"}},
    )
}
```

循着这个路，我们来到它的真正入口，在 cli/cli/command/image/build.go 可以看到真正的逻辑：

复制

```
// components/cli/cli/command/image/build.go#207
func runBuild(dockerCli command.Cli, options buildOptions) error {
    buildkitEnabled, err := command.BuildKitEnabled(dockerCli.ServerInfo())
    if err != nil {
        return err
    }
    if buildkitEnabled {
        return runBuildBuildKit(dockerCli, options)
    }
    // 省略掉其他逻辑
}
```

此处在判断是否支持 BuildKit，Buildkit 便是我们的下一代构建系统。我们根据 BuildKitEnabled 来看看如何启用 BuildKit。

复制

```
// components/cli/cli/command/cli.go#151
func BuildKitEnabled(si ServerInfo) (bool, error) {
    buildkitEnabled := si.BuildkitVersion == types.BuilderBuildKit
    if buildkitEnv := os.Getenv("DOCKER_BUILDKIT"); buildkitEnv != "" {
        var err error
        buildkitEnabled, err = strconv.ParseBool(buildkitEnv)
        if err != nil {
            return false, errors.Wrap(err, "DOCKER_BUILDKIT environment variable expe
        }
    }
    return buildkitEnabled, nil
}
```

由此可以看出：

1. 通过配置 docker daemon 可启用 BuildKit（即 si.BuildkitVersion）。

具体配置方法为：在 /etc/docker/daemon.json 中添加以下内容，并重启 Docker Daemon 即可。

复制

```
```json
{
  "features": {
    "buildkit": true
  }
}
```
```

2. 第二种便是通过增加 DOCKER_BUILDKIT 的环境变量，并且 Docker CLI 端的配置可覆盖服务器端的配置。

通过上面的介绍，我们也就知道了在使用默认的 builder v1 时，入口逻辑就在 runBuild 中，而启用了 BuildKit 后，入口就到了 runBuildBuildKit 中。

虽然本篇的主体是 BuildKit，但为了让内容更连续，也方便你对 Docker 构建系统有更深的了解，我们对 builder v1 的入口 runBuild 做下拆解。

参数处理

由于篇幅原因，这里就不贴出 runBuild 函数的全部代码了，我们可以看到，其在真正执行构建逻辑前首先会做一些参数检查和校验。这里会暴露出很多平时可能不太注意的点，比如：

1. 不可同时使用 stdin 读取 Dockerfile 和 `build context`

在进行镜像构建时，如果我们将 Dockerfile 的名称指定为 `-`，则表示从 stdin 读取 Dockerfile 的内容。

考虑以下场景：某目录下有三个文件，分别是 foo、bar 和 Dockerfile，Dockerfile 中只是简单的 COPY 这两个文件。

复制

```
(MoeLove) ➜  builder echo foo > foo
(MoeLove) ➜  builder echo bar > bar
(MoeLove) ➜  builder cat <<EOF > Dockerfile
FROM alpine
COPY foo /foo
COPY bar /bar
EOF
```

使用管道，将 Dockerfile 的内容通过 stdin 传递给 `docker build`：

复制

```
(MoeLove) ➜  builder cat Dockerfile | DOCKER_BUILDKIT=0 docker build -t local
Sending build context to Docker daemon  4.648kB
Step 1/3 : FROM alpine
 ---> 965ea09ff2eb
Step 2/3 : COPY foo /foo
 ---> b52a3e2637c3
Step 3/3 : COPY bar /bar
 ---> 1df28949fdb9
Successfully built 1df28949fdb9
Successfully tagged local/builder:v1
```

可以看到通过 stdin 传递 Dockerfile 能成功构建镜像。接下来我们尝试通过 stdin 将 build context 传递进去。

复制

```
(MoeLove) ➜  builder cat context.tar | DOCKER_BUILDKIT=0 docker build -t loca
Sending build context to Docker daemon  10.24kB
Step 1/3 : FROM alpine
 ---> 965ea09ff2eb
Step 2/3 : COPY foo /foo
 ---> 401acff47bca
Step 3/3 : COPY bar /bar
 ---> 7b629075234b
Successfully built 7b629075234b
Successfully tagged local/builder:v2
```

同样，通过 stdin 传递 build context 时也可以成功构建镜像。那么当 Dockerfile 与 build context 都从 stdin 读取，即：`docker build -f - -` 会发生什么呢?

复制

```
(MoeLove) ➜  builder DOCKER_BUILDKIT=0 docker build  -f - -
invalid argument: can't use stdin for both build context and dockerfile
```

可以看到会直接报错，所以**不能同时使用 stdin 读取 Dockerfile 及 build context 内容。**

2. build context 支持四种模式

复制

```
switch {
case options.contextFromStdin():
    buildCtx, relDockerfile, err = build.GetContextFromReader(dockerCli.In(), opt
case isLocalDir(specifiedContext):
    contextDir, relDockerfile, err = build.GetContextFromLocalDir(specifiedContex
    if err == nil && strings.HasPrefix(relDockerfile, ".."+string(filepath.Separa
        dockerfileCtx, err = os.Open(options.dockerfileName)
        if err != nil {
            return errors.Errorf("unable to open Dockerfile: %v", err)
        }
        defer dockerfileCtx.Close()
    }
case urlutil.IsGitURL(specifiedContext):
    tempDir, relDockerfile, err = build.GetContextFromGitURL(specifiedContext, op
case urlutil.IsURL(specifiedContext):
    buildCtx, relDockerfile, err = build.GetContextFromURL(progBuff, specifiedCon
default:
    return errors.Errorf("unable to prepare context: path %q not found", specifie
}
```

前面已经演示了通过 stdin 的方式。

- 我们最常用的是第二种，也就是 build context 是一个本地目录；
- 第三种就是给定一个 git 仓库的地址，CLI 则会调用 git 相关命令将仓库 clone 到一个临时目录中，再进行构建；
- 最后一种是给定一个 URL 地址，该地址可以是 Dockerfile 的地址，也可以是一个 tar 归档文件的下载地址，但这里需要注意的是，如果是直接传递 Dockerfile 的 URL，那么 docker build context 就相当于是只有 Dockerfile 自身，所以不能使用 COPY 之类的指令。

3. 可使用 .dockerignore 忽略不需要的文件

我们可以直接看看它的具体逻辑：

复制

```
// components/cli/cli/command/image/build/dockerignore.go#13
func ReadDockerignore(contextDir string) ([]string, error) {
    var excludes []string

    f, err := os.Open(filepath.Join(contextDir, ".dockerignore"))
    switch {
    case os.IsNotExist(err):
        return excludes, nil
    case err != nil:
        return nil, err
    }
    defer f.Close()

    return dockerignore.ReadAll(f)
}
```

.dockerignore 文件名是固定的，需要放在 build context 的根目录下。

.dockerignore 文件可以不存在，但如果读取遇到错误便会抛出异常。

最后 docker CLI 会从 `build context` 中先过滤掉 .dockerignore 忽略的内容，再将剩余内容传递给 Docker Daemon，基于此，也可以看出来使用 .dockerignore 的意义在于，可以减少 Docker CLI 与 Docker Daemon 之间的传输压力等。

最后，将所有需要的内容发送至 Docker Daemon，进行镜像的真正构建。待 Docker Daemon 返回构建结果，再进行展示或者执行其他后续逻辑。

下一代构建系统

前面介绍了 builder v1 的构建系统，它可以比较好地满足我们的大多数需求。但在实际生产环境大规模使用中，就会暴露出来一些问题：

- 构建效率低
- 多阶段构建无法很好的利用缓存
- 功能单一

介绍

基于上述的这些原因，加上 Docker 想要为构建系统赋予更多的活力，所以经过很长时间的设计和考虑之后，开始了 BuildKit 项目的开发，并且从 Docker 18.06 起，便可在 Docker 中使用 BuildKit 的部分功能。

具体打开 BuildKit 的支持的方式前面已经介绍了，通过给 /etc/docker/daemon.json 中添加以下内容：

复制

```
{
  "features": {
    "buildkit": true
  }
}
```

或是设置环境变量 `DOCKER_BUILDKIT=1` 即可启用。普通的用法与 builer v1 基本保持一致，无需刻意适配。

优势

我来写一个示例，让你的直观的感受下 BuildKit 带来的优势：

复制

```
(MoeLove) ➜  multi-stage cat Dockerfile
FROM alpine

RUN apk add --no-cache curl

FROM alpine

RUN apk add --no-cache wget


FROM alpine AS runtime

COPY --from=0 /usr/bin/curl /usr/bin/curl
COPY --from=1 /usr/bin/wget /usr/bin/wget
```

示例很简单，就是多阶段构建（之前的内容中介绍过的），最后的镜像从之前的两个阶段中拷贝了一些内容到自己的目录中。（不用在意这个示例中最终构建出来的镜像是否可用）

我们分别以 builder v1 和 BuildKit 的模式构建下：

复制

```
# 使用builder v1 构建
(MoeLove) ➜  multi-stage time DOCKER_BUILDKIT=0 docker build --no-cache -q -t loc
sha256:5ac17c97c7e2aa38775a20e49d654d5b2604c5b04dbe31a7a2b5f1295788a644
DOCKER_BUILDKIT=0 docker build --no-cache -q -t local/multi-stage:v1 .  0.04s use


# 使用 BuildKit 构建
(MoeLove) ➜  multi-stage time DOCKER_BUILDKIT=1 docker build --no-cache -q -t loc
sha256:3c113ae07fd8af26a633c1626cef697bbb36010111fefd989f7311c90a514ecb
DOCKER_BUILDKIT=1 docker build --no-cache -q -t local/multi-stage:v2 .  0.05s use
```

可以看到，在这样简单的示例中，构建效率已经提升了近一倍。

能这样提升速度，最主要的原因在于，Buildkit 是并发构建，而原本的 builder v1 则是顺序构建。

另外，BuildKit 可进行依赖分析，如果最终的镜像不需要其中某些阶段的产物，则会跳过该阶段。但是 builer v1 则会完全按顺序执行完。

总结

本篇，我为你介绍了 Docker 的构建系统，包括现有的 builder v1 及下一代构建系统 BuildKit。后续内容中，我们还会继续基于 BuildKit 深入研究 Dockerfile 的最佳实践，及镜像构建的原理。

另外需要说明的是：BuildKit 本身是一个独立的项目，你可以在不依赖 Docker 的情况下去使用它。但毕竟我们还是用 Docker 更多一些，且功能完善，无需去学习和掌握 BuildKit 的那一套操作习惯等。

BuildKit 也已经被不少项目作为依赖使用了，如果你还没有使用过 BuildKit，那么现在就执行 `export DOCKER_BUILDKIT=1` 开始使用它吧！

下一篇，我将为你介绍 Dockerfile 的最佳实践，其中一些也是基于 BuildKit 的。