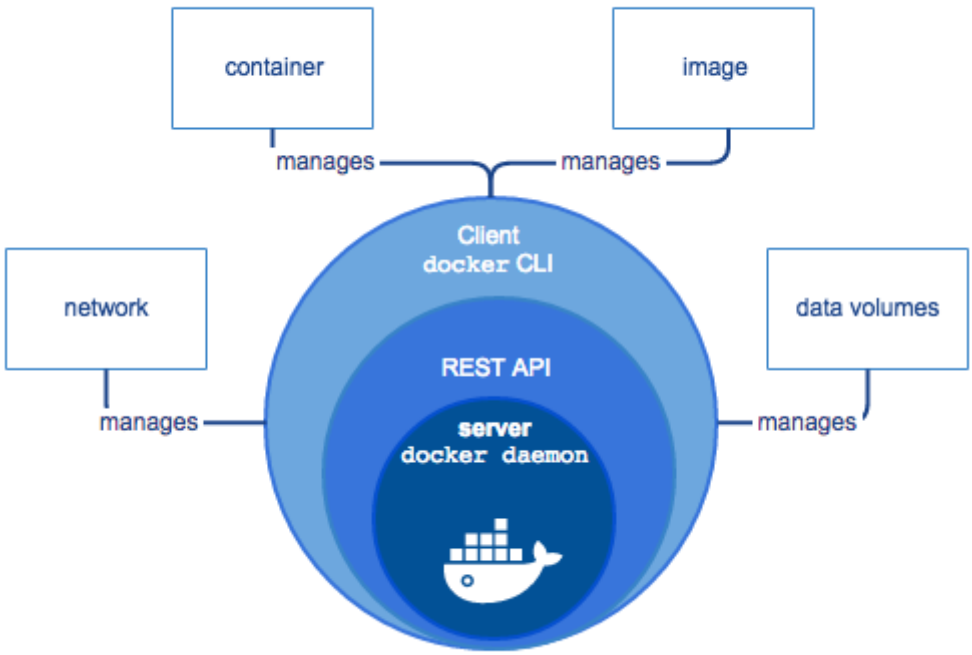
镜像构建原理

这是本专栏的第三部分：镜像篇，共 8 篇。前五篇我分别为你介绍了如何对 Docker 镜像进行生命周期的管理，如何使用 Dockerfile 进行镜像的构建和分发，Docker 的构建系统和下一代构建系统——BuildKit 以及 Dockerfile 的优化和实践。下面我们一起进入本篇镜像构建原理的学习。

Docker 架构概览

在前两篇《Docker 下一代构建系统》中，我为你介绍了 Docker 整体而言是 C/S 架构的，我们通常使用的 Docker 命令，其实是它的 CLI 工具，而它的服务端程序是 Dockerd，整体如下图所示：



图片来源：Docker overview

Docker CLI 与 dockerd 的交互都是通过 Rest API 完成的，我们可以通过 `docker version` 查看到当前所使用的 API 版本。

复制

```
/ # docker version | grep API
 API version:       1.40
  API version:      1.40 (minimum version 1.12)
```

这里有个值得注意的点是 `minimum version 1.12`。这句说明，它表示最小可兼容的 API 版本是 1.12，这是因为 Docker 的 API 有很良好的兼容性，所以即使 Docker CLI 与 Dockerd 的版本不完全一致，也是可以工作的。

C/S 架构一定是 API 先行，由于之前那篇的重点在于对 BuildKit 的介绍，所以并没有过于深入，本篇我们一起来完整地探究下。

镜像构建 API

对于我们所用的 Docker CE v19.03.5 而言，其 API 的在线文档地址是：

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

如果你想要自己在本地随时翻阅 API 文档，那你也可以在本地自行构建此文档。

复制

```
(MoeLove) ➜  ~ git clone -q https://github.com/docker/docker-ce.git
(MoeLove) ➜  ~ cd docker-ce
(MoeLove) ➜  ~ git checkout -b v19.03.5 v19.03.5
(MoeLove) ➜  docker-ce git:(v19.03.5) cd components/engine/
(MoeLove) ➜  engine git:(v19.03.5) make swagger-docs
API docs preview will be running at http://localhost:9000
```

现在你在浏览器打开 http://localhost:9000 便可在本地查看 API 文档了。

接口地址和方法

- 接口地址是：/v1.40/build
- 请求方法是：POST

我们可以使用一个较新版本的 curl 工具来验证下此接口（需要使用 `--unix-socket`，连接 Docker 监听的 UNIX Domain Socket），/var/run/docker.sock 这是默认情况下 Dockerd 所监听的地址，当然你也可以给 Dockerd 传递 `--host` 参数用于监听 HTTP 端口，或者其他路径的 Unix Socket。

复制

```
(MoeLove) ➜  engine git:(v19.03.5) curl -X POST --unix-socket /var/run/docker.soc
{"message":"Cannot locate specified Dockerfile: Dockerfile"}
```

从上面的输出我们可以看到，我们确实访问到了该接口，同时该接口的响应是提示需要 Dockerfile。

请求体

> A tar archive compressed with one of the following algorithms: identity (no compression), gzip, bzip2, xz.
>
> string <binary>

请求体是一个 tar 归档文件，可选择无压缩、gzip、bzip2、xz 压缩等形式。关于这几种压缩格式就不再展开介绍了，但值得注意的是 **如果使用了压缩，则传输体积会变小，即网络消耗会相应减少。但压缩/解压缩需要耗费 CPU 等计算资源** 这在我们进行大规模镜像构建时是个值得权衡的点。

请求头

因为要发送的是个 tar 归档文件，Content-type 默认是 application/x-tar。

另一个会发送的头是 X-Registry-Config，这是一个由 Base64 编码后的 Docker Registry 的配置信息，内容与 $HOME/.docker/config.json 中的 auths 内的信息一致。

这些认证相关的配置信息，在你执行 `docker login` 后会默认会自动写入到 $HOME/.docker/config.json。这些信息被传输到 Dockerd 在构建过程中作为拉取镜像的认证信息使用。

请求参数

最后就是请求参数了，参数有很多，通过 `docker build --help` 基本都可以看到对应含义的，这里不再一一展开了，后面会有一些关键参数的介绍。

小结

上面我们介绍了 Docker 构建镜像相关的 API，我们可以通过在线地址访问：

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

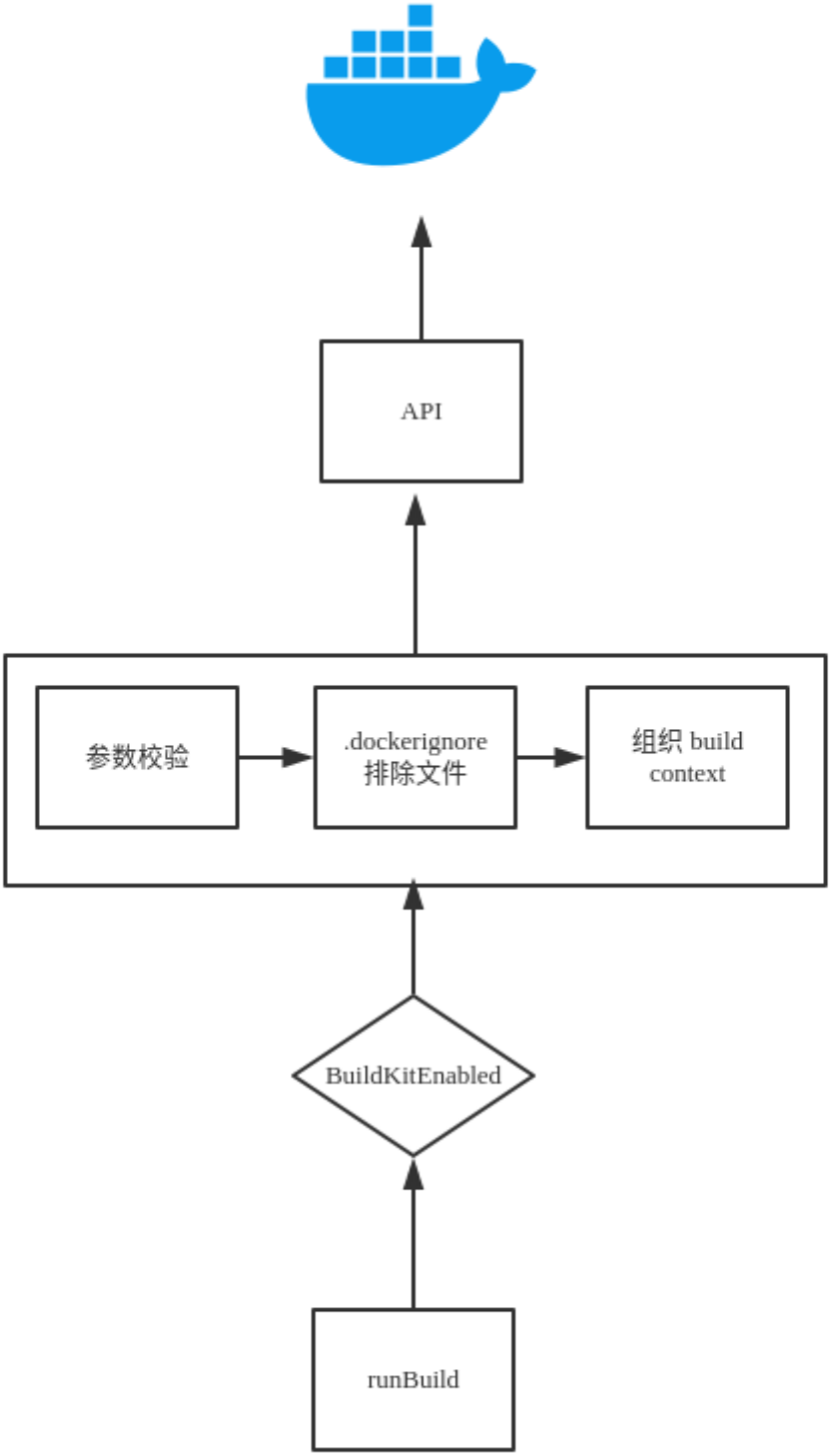
或者通过源码仓库，自己来构建一个本地的 API 文档服务，使用浏览器进行访问。

通过 API 我们也知道了该接口所需的请求体是一个 tar 归档文件（可选择压缩算法进行压缩），同时它的请求头中会携带用户在镜像仓库中的认证信息。

Docker CLI 构建镜像

在前两篇《Docker 下一代构建系统》 中，我为你深入源码中介绍了 Docker CLI 使用 builder v1 构建镜像的大体流程。

但无论是使用 builder v1 还是开启 BuildKit 的支持，整体的处理逻辑上是大体相同的。如下图所示：



仅有的一些区别在于如果开启了 BuildKit，在构建过程（或者说在 Dockerfile 的特性支持）上，会更加丰富，且 BuildKit 提供了多种输出形式。

此处，就不再另外对 Docker CLI 进行介绍了。我们将重点放在 Dockerd 上。

构建的核心：Dockerd

在正式深入到 Dockerd 构建镜像的行为前，我们不妨再看看 Docker CLI 最终是如何调用 Dockerd API 的。

复制

```
// components/engine/client/image_build.go#L20
func (cli *Client) ImageBuild(ctx context.Context, buildContext io.Reader, options ty
    query, err := cli.imageBuildOptionsToQuery(options)
    if err != nil {
        return types.ImageBuildResponse{}, err
    }

    headers := http.Header(make(map[string][]string))
    buf, err := json.Marshal(options.AuthConfigs)
    if err != nil {
        return types.ImageBuildResponse{}, err
    }
    headers.Add("X-Registry-Config", base64.URLEncoding.EncodeToString(buf))

    headers.Set("Content-Type", "application/x-tar")

    serverResp, err := cli.postRaw(ctx, "/build", query, buildContext, headers)
    if err != nil {
        return types.ImageBuildResponse{}, err
    }

    osType := getDockerOS(serverResp.header.Get("Server"))

    return types.ImageBuildResponse{
        Body:   serverResp.body,
        OSType: osType,
    }, nil
}
```

这是 Docker CLI 调用 Dockerd 构建镜像 API 时的 client 函数。我们来具体看看。

imageBuildOptionsToQuery 实际就是将我们通过 `docker build` 传递进去的参数，都转换成调用 Dockerd API 时 URL 的请求参数。

然后添加两个请求头，之后调用 /build API 将所有构建所需信息传递给 Dockerd，最终等待结果并返回。

可以看到构建镜像真正的核心还是在于 Dockerd 的 /build API。我们继续对它进行分解。

先知道这个 /build API 的入口：

复制

```
// components/engine/api/server/router/build/build.go#L32
func (r *buildRouter) initRoutes() {
    r.routes = []router.Route{
        router.NewPostRoute("/build", r.postBuild),
        router.NewPostRoute("/build/prune", r.postPrune),
        router.NewPostRoute("/build/cancel", r.postCancel),
    }
}
```

Dockerd 提供了一套类 RESTful 的后端接口服务，处理逻辑的入口便是上面的 postBuild 函数。

该函数位于 build_routes.go 文件中，内容较多，不到 100 行，篇幅原因这里就不全贴出来了，我们来分解下它的主要步骤。

复制

```
    buildOptions, err := newImageBuildOptions(ctx, r)
    if err != nil {
        return errf(err)
    }
```

newImageBuildOptions 函数就是构造构建参数的，将通过 API 提交过来的参数转换为构建动作实际需要的参数形式。

复制

```
    buildOptions.AuthConfigs = getAuthConfigs(r.Header)
```

getAuthConfigs 函数用于从请求头拿到认证信息：

复制

```
    imgID, err := br.backend.Build(ctx, backend.BuildConfig{
        Source:         body,
        Options:        buildOptions,
        ProgressWriter: buildProgressWriter(out, wantAux, createProgressReader),
    })
    if err != nil {
        return errf(err)
    }
```

这里就需要注意了，真正的构建过程要开始了。使用 backend 的 **Build 函数**来完成真正的构建过程：

复制

```
// components/engine/api/server/backend/build/backend.go#L52
func (b *Backend) Build(ctx context.Context, config backend.BuildConfig) (string, err
    options := config.Options
    useBuildKit := options.Version == types.BuilderBuildKit

    tagger, err := NewTagger(b.imageComponent, config.ProgressWriter.StdoutFormatter,
    if err != nil {
        return "", err
    }

    var build *builder.Result
    if useBuildKit {
        build, err = b.buildkit.Build(ctx, config)
        if err != nil {
            return "", err
        }
    } else {
        build, err = b.builder.Build(ctx, config)
        if err != nil {
            return "", err
        }
    }

    if build == nil {
        return "", nil
    }

    var imageID = build.ImageID
    if options.Squash {
        if imageID, err = squashBuild(build, b.imageComponent); err != nil {
            return "", err
        }
        if config.ProgressWriter.AuxFormatter != nil {
            if err = config.ProgressWriter.AuxFormatter.Emit("moby.image.id", types.Bu
                return "", err
            }
        }
    }

    if !useBuildKit {
        stdout := config.ProgressWriter.StdoutFormatter
        fmt.Fprintf(stdout, "Successfully built %s\n", stringid.TruncateID(imageID))
    }
    if imageID != "" {
        err = tagger.TagImages(image.ID(imageID))
    }
```

```
    return imageID, err
}
```

这个函数看着比较长，但主要功能就以下三点：

- NewTagger 是用于给镜像打标签，也就是我们的 `-t` 参数相关的内容，这里不做展开。
- 通过判断是否使用了 BuildKit 来调用不同的构建后端。

复制

```
    useBuildKit := options.Version == types.BuilderBuildKit

    var build *builder.Result
    if useBuildKit {
        build, err = b.buildkit.Build(ctx, config)
        if err != nil {
            return "", err
        }
    } else {
        build, err = b.builder.Build(ctx, config)
        if err != nil {
            return "", err
        }
    }
```

- 处理构建完成后的动作。

到这个函数之后，就分别是 builder v1 与 BuildKit 对 Dockerfile 的解析，以及对 `build context` 的相关操作了。

这里不再进行展开，但是要注意的是，为什么在之前内容中，我们提到 builder v1 比 BuildKit 效率低呢？这是因为在对 Dockerfile 解析完后，builder v1 对这个结果列表直接用了一个 for 循环顺序处理了，而 BuildKit 则对依赖进行了分析，可跳过无关内容，可并行处理同级内容等。

总结

本篇，我为你介绍了 Docker 镜像构建的原理，深入到源码来具体分析其行为。整体而言，Docker 的镜像构建是由 Docker CLI 通过接收用户提供的内容和参数并进行预处理之后，调用 Dockerd 的 API 发送给服务端，再由 Dockerd 解析内容后进行构建，构建结束后返回相应结果，并展示给用户。

了解 Docker 镜像构建原理的意义是什么呢？为何我在这部分内容上写了这么多内容？

这是因为无论你在使用 Docker 或者是之后用 Kubernetes 做容器编排，亦或者是使用其他的容器化相关技术，容器镜像始终是其基础。能深入地掌握镜像构建的方法，及其原理，自然就可以更好地应对后续的需求或对它进行优化。

在企业的生产环境中，效率及成本都是非常重要的点，能够通过理解构建原理而对其进行一些优化和效率提升，那在生产环境中产生的益处将会放大很多倍，希望你能学有所用。

通过本系列的学习，想必你已经能构建出高质量的 Docker 镜像了，为了能将你的镜像给更多人或者给你的其他环境使用，那镜像的分发是必不可少的。