镜像分发原理

这是本专栏的第三部分：镜像篇，共 8 篇。前六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，Dockerfile 的优化和实践，以及深入源码介绍了 Docker 镜像构建的原理。下面我们一起进入本篇镜像分发原理的学习。

通过前面内容的学习，我们知道 Docker 镜像可以存储在 dockerd（Docker Daemon）本地，也可以通过 `docker save` 等操作将其保存为一个 tar 归档文件。

然而为了能更方便的达到 Docker 最初的目标“Build Once, Run Anywhere”，使用上述的两种方式对于将镜像分发给其他（大量）用户都不是很方便，所以 Docker 中提供了 `docker push/docker pull` 这两个命令，可以方便地将 Docker 镜像存储在远端 registry 中。例如，Docker 最一开始就推出的 Docker Hub，其目前已托管了海量的 Docker 镜像，这也是 Docker 能快速流行起来的原因之一。

本篇，我们就来重点看看 `docker push` 和 `docker pull` 这两个最为常用的用于镜像分发的命令吧。

注：本篇使用 Docker CE 19.03.5 版本的代码。

docker push

复制

```
(MoeLove) ➜  ~ docker push --help

Usage:  docker push [OPTIONS] NAME[:TAG]

Push an image or a repository to a registry

Options:
      --disable-content-trust   Skip image signing (default true)
```

该命令的使用方法如上，命令后直接跟一个 Docker 镜像的名称和 Tag 即可。

入口

复制

```
// components/cli/cli/command/image/push.go#L41
func RunPush(dockerCli command.Cli, opts pushOptions) error {
    ref, err := reference.ParseNormalizedNamed(opts.remote)
    if err != nil {
        return err
    }

    repoInfo, err := registry.ParseRepositoryInfo(ref)
    if err != nil {
        return err
    }

    ctx := context.Background()

    authConfig := command.ResolveAuthConfig(ctx, dockerCli, repoInfo.Index)
    requestPrivilege := command.RegistryAuthenticationPrivilegedFunc(dockerCli, repoI

    if !opts.untrusted {
        return TrustedPush(ctx, dockerCli, repoInfo, ref, authConfig, requestPrivileg
    }

    responseBody, err := imagePushPrivileged(ctx, dockerCli, authConfig, ref, requestP
    if err != nil {
        return err
    }

    defer responseBody.Close()
    return jsonmessage.DisplayJSONMessagesToStream(responseBody, dockerCli.Out(), nil)
}
```

整个 push 动作的入口就是 RunPush 这个函数，从这个函数来看，其做的操作如下：

- 根据传递进来的镜像名称和 Tag 解析；
- 获取认证信息；
- 携带认证信息，调用 dockerd push 镜像的 API。

push image API

直接参考在线文档：

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

可以看到 push image 的 API 其实很简单，请求地址是 /images/{name}/push 必须携带 X-Registry-Auth 头，其值是 Base64 加密后的认证信息（下一篇会重点介绍）。

其中的 {name} 可以是镜像名称或镜像 ID。

后端逻辑

入口在 components/engine/api/server/router/image/image_routes.go#L100 的 postImagesPush 函数，但这个函数也没什么太多内容，此处略过，有兴趣的读者可自行查看该函数。

我们来看看最后真正完成 push 镜像动作的逻辑：

复制

```
// components/engine/distribution/xfer/upload.go#L67
func (lum *LayerUploadManager) Upload(ctx context.Context, layers []UploadDescriptor,
    var (
        uploads          []*uploadTransfer
        dedupDescriptors = make(map[string]*uploadTransfer)
    )

    for _, descriptor := range layers {
        progress.Update(progressOutput, descriptor.ID(), "Preparing")

        key := descriptor.Key()
        if _, present := dedupDescriptors[key]; present {
            continue
        }

        xferFunc := lum.makeUploadFunc(descriptor)
        upload, watcher := lum.tm.Transfer(descriptor.Key(), xferFunc, progressOutput)
        defer upload.Release(watcher)
        uploads = append(uploads, upload.(*uploadTransfer))
        dedupDescriptors[key] = upload.(*uploadTransfer)
    }

    for _, upload := range uploads {
        select {
        case <-ctx.Done():
            return ctx.Err()
        case <-upload.Transfer.Done():
            if upload.err != nil {
                return upload.err
            }
        }
    }
    for _, l := range layers {
        l.SetRemoteDescriptor(dedupDescriptors[l.Key()].remoteDescriptor)
    }

    return nil
}
```

可以看到这中间有个 `key := descriptor.Key()`，通过这里的逻辑可用于避免重复的上传，并且整体来看，整个上传过程是阻塞的，要确保每个镜像层都已经上传到了远端 registry 中。

当然，在这里还有几个值得注意的点：

- 可以在启动 dockerd 的时候，通过 `--max-concurrent-uploads` 来指定最大的并发上传数，默认是 5；

- 源码中通过 maxUploadAttempts 来控制重试次数，当前值为 5.

docker pull

```
(MoeLove) ➜  ~ docker pull --help

Usage:  docker pull [OPTIONS] NAME[:TAG|@DIGEST]

Pull an image or a repository from a registry

Options:
  -a, --all-tags                Download all tagged images in the repository
      --disable-content-trust   Skip image verification (default true)
      --platform string         Set platform if server is multi-platform capable
  -q, --quiet                   Suppress verbose output
```

复制

该命令的使用方法如上，命令后直接跟一个 Docker 镜像的名称和 Tag 即可。有几个可选参数，上方都有说明，此处略过。

入口

复制

```
// components/cli/cli/command/image/pull.go#L51
func RunPull(cli command.Cli, opts PullOptions) error {
    distributionRef, err := reference.ParseNormalizedNamed(opts.remote)
    switch {
    case err != nil:
        return err
    case opts.all && !reference.IsNameOnly(distributionRef):
        return errors.New("tag can't be used with --all-tags/-a")
    case !opts.all && reference.IsNameOnly(distributionRef):
        distributionRef = reference.TagNameOnly(distributionRef)
        if tagged, ok := distributionRef.(reference.Tagged); ok && !opts.quiet {
            fmt.Fprintf(cli.Out(), "Using default tag: %s\n", tagged.Tag())
        }
    }

    ctx := context.Background()
    imgRefAndAuth, err := trust.GetImageReferencesAndAuth(ctx, nil, AuthResolver(cli),
    if err != nil {
        return err
    }

    _, isCanonical := distributionRef.(reference.Canonical)
    if !opts.untrusted && !isCanonical {
        err = trustedPull(ctx, cli, imgRefAndAuth, opts)
    } else {
        err = imagePullPrivileged(ctx, cli, imgRefAndAuth, opts)
    }
    if err != nil {
        if strings.Contains(err.Error(), "when fetching 'plugin'") {
            return errors.New(err.Error() + " - Use `docker plugin install`")
        }
        return err
    }
    fmt.Fprintln(cli.Out(), imgRefAndAuth.Reference().String())
    return nil
}
```

`docker pull` 由于比 `docker push` 多支持了几个参数，所以入口处的逻辑也就稍微多了一点。

- 如果指定了具体的镜像 tag 那你就不能同时使用 `-a` 参数了；
- 如果不指定任何镜像 tag 则使用默认的 latest.

其基本逻辑也是：

- 解析传递进来的镜像名称和 Tag
- 解析额外的参数

- 获取认证信息
- 调用 dockerd 的 API pull 镜像

pull image API

直接参考在线文档：

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

可以看到 pull image 使用的 API 是 ImageCreate，请求地址是 /images/create 它同样可携带 X-Registry-Auth 头，其值是 Base64 加密后的认证信息（下一篇会重点介绍）。

后端逻辑

入口在 components/engine/api/server/router/image/image_routes.go#L26 的 postImagesCreate 函数，它被同时用于 `docker pull` 和 `docker import`。篇幅原因，这里就不再贴出具体代码了。

我们来看看最后执行 Pull 镜像的真正逻辑。

复制

```
// components/engine/daemon/images/image_pull.go#L23
func (i *ImageService) PullImage(ctx context.Context, image, tag string, platform *spe
    start := time.Now()
    image = strings.TrimSuffix(image, ":")

    ref, err := reference.ParseNormalizedNamed(image)
    if err != nil {
        return errdefs.InvalidParameter(err)
    }

    if tag != "" {
        var dgst digest.Digest
        dgst, err = digest.Parse(tag)
        if err == nil {
            ref, err = reference.WithDigest(reference.TrimNamed(ref), dgst)
        } else {
            ref, err = reference.WithTag(ref, tag)
        }
        if err != nil {
            return errdefs.InvalidParameter(err)
        }
    }

    err = i.pullImageWithReference(ctx, ref, platform, metaHeaders, authConfig, outSt
    imageActions.WithValues("pull").UpdateSince(start)
    return err
}
```

相比 `docker push` 这里的逻辑就简单的多了。只需要注意以下几点：

- 可通过参数 `--max-concurrent-downloads` 设置最大并发下载数；
- 如果 registry 是私有的，则需要保证先登录，并且有权限才可以操作。

总结

本篇，我为你深入源码介绍了 Docker 镜像分发的原理。整体而言，无论是镜像的 pull/push 都相对比较简单，只需要由 Docker CLI 做好预先的参数校验，并且携带用户认证信息即可。

通过 `--max-concurrent-downloads` 和 `--max-concurrent-uploads` 可以配置最大并发数。

本篇的意义在于，无论任何人要使用镜像，分发给他人或是部署到生产环境，一般都是需要使用 `docker push` 和 `docker pull` 等操作的。当然，现在也有很多其他的工具可以进行镜像的 push 和 pull，但相比而言，Docker 的原生命令更直观，也更易用。