认证流程和原理

这是本专栏的第三部分：镜像篇，共 8 篇。前七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 镜像构建和镜像分发的原理原理。下面我们一起进入本篇认证流程和原理的学习。

通过前面内容的学习，想必你对本篇的内容已经有所期待。当你想把 Docker 镜像 push 到某个开启认证的 registry 或者想要从某些私有的 registry 中 pull 镜像时，偶尔会看到 `denied: requested access to the resource is denied` 类似这样的错误信息，如果不是地址有误，那基本就是未登录或者没有权限了。

本篇，我来为你介绍 Docker 的认证流程和原理，我们先来看看 `docker login`。注意：本篇的内容以 Docker CE v19.03.5 为例，系统环境为 Linux。

`docker login` CLI

我们从 `docker login` 作为入口，可以接收用户名/密码相关的选项及服务地址。

复制

```
(MoeLove) ➜  ~ docker login --help

Usage:  docker login [OPTIONS] [SERVER]

Log in to a Docker registry.
If no server is specified, the default is defined by the daemon.

Options:
  -p, --password string   Password
      --password-stdin    Take the password from stdin
  -u, --username string   Username
```

Server

默认情况下，如果不提供任何参数，则服务地址是由 Docker daemon 提供。这个地址可以通过以下两种方式获取。

使用 `docker info` 获取：

复制

```
(MoeLove) ➜  ~ docker info |grep Registry
 Registry: https://index.docker.io/v1/
```

通过 /info API 获取：

```
(MoeLove) ➜  ~ curl -s --unix-socket /var/run/docker.sock localhost/info | jq
"https://index.docker.io/v1/"
```

上述两种方式本质都是使用 /info 接口获取的。我们也可以从源码中直接看到配置：

```
var (
    IndexHostname = "index.docker.io"
    IndexServer = "https://" + IndexHostname + "/v1/"
)
```

用户名密码

```
(MoeLove) ➜  ~ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't ha
Username: test
Password:
```

如果没有使用选项指定的话，默认会使用一个交互环境，允许你输入自己的用户名或者密码。但如果我们想要在非交互环境下（脚本或者 CI 环境下）登录的话，则需要通过 `-u` 和 `-p` 指定用户名和密码，为了安全起见，**建议使用** `echo $PASSWORD | docker login -u test --password-stdin` **类似这样的方式，传递用户密码。**

如果已经登录过，并且在执行 `docker login` 时，没有指定任何用户名或密码，则 Docker CLI 会先查找本地存储过的认证信息，并使用对应的认证信息进行登录。 示例如下：

```
(MoeLove) ➜  ~ docker login myregisty.moelove.info
Authenticating with existing credentials...
Login Succeeded
```

/auth API

前面课程中已经提到过的，我们可以查看 Docker 的在线文档：

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

可以看到此处的接口地址为 /auth，请求参数是 username\password\mail 和 serveraddress，这里的 mail 是可选项。

如果登录成功的话，则会返回登录成功的状态。

复制

```
(MoeLove) ➜  ~ curl --data '{"username": "my-username", "password": "my-password"
{"IdentityToken":"","Status":"Login Succeeded"}
```

从上面可以看到 Docker CLI 调用 Docker Daemon 认证接口的结果。接下来我们来看看 Docker Daemon 是如何处理认证流程的。

Docker Daemon 的认证逻辑

入口函数在 postAuth，可以看到这个函数很简单，接收认证信息并向 registry 进行认证，并返回结果。

复制

```
// components/engine/api/server/router/system/system_routes.go#L226
func (s *systemRouter) postAuth(ctx context.Context, w http.ResponseWriter, r *http.R
    var config *types.AuthConfig
    err := json.NewDecoder(r.Body).Decode(&config)
    r.Body.Close()
    if err != nil {
        return err
    }
    status, token, err := s.backend.AuthenticateToRegistry(ctx, config)
    if err != nil {
        return err
    }
    return httputils.WriteJSON(w, http.StatusOK, &registry.AuthenticateOKBody{
        Status:        status,
        IdentityToken: token,
    })
}
```

在我们继续深入探究之前，需要先明确几个问题。

1. registry 并不一定都需要认证，默认它可以不开启认证。

复制

```
(MoeLove) ➜  ~ docker run -d -p 5000:5000 --restart=always --name registry regist
665025339cea4624326dc499ae5626fa21f3438679f926c1ad224be2d6c3b3d5
(MoeLove) ➜  ~ docker tag alpine:3.10 localhost:5000/alpine:3.10
(MoeLove) ➜  ~ docker push localhost:5000/alpine:3.10
The push refers to repository [localhost:5000/alpine]
77cae8ab23bf: Pushed
3.10: digest: sha256:e4355b66995c96b4b468159fc5c7e3540fcef961189ca13fee877798649f5
```

以上便是在本地启动一个 registry 的情况。

2. 具体的认证逻辑需要看所使用的认证模式，但大多数现有的公共镜像/提供私有部署的镜像服务都是选择开启了 Token 认证的模式。比如：DockerHub。

复制

```
(MoeLove) ➜  ~ curl -I https://index.docker.io/v2/
HTTP/1.1 401 Unauthorized
Content-Type: application/json
Docker-Distribution-Api-Version: registry/2.0
Www-Authenticate: Bearer realm="https://auth.docker.io/token",service="registr
Date: Sun, 08 Dec 2019 16:15:16 GMT
Content-Length: 87
Strict-Transport-Security: max-age=31536000
```

比如：Quay.io。

复制

```
(MoeLove) ➜  ~ curl -I https://quay.io/v2/
HTTP/2 401
server: nginx/1.12.1
date: Sun, 08 Dec 2019 16:15:00 GMT
content-type: text/html; charset=utf-8
content-length: 4
docker-distribution-api-version: registry/2.0
www-authenticate: Bearer realm="https://quay.io/v2/auth",service="quay.io"
```

注意到此返回中 `www-authenticate: Bearer realm="xxx",service="xxx"` 格式的内容了吧，这表明它们都是启动了 Token 验证的模式。当然，包括现很多公司在用的私有镜像仓库 Harbor 其实也是如此。

3. 但这并不是全部，比如说 GitHub 的 Docker 镜像仓库，就是启动了 Basic Auth 的认证模式。

复制

```
(MoeLove) ➜  ~ curl  -v https://docker.pkg.github.com/v2/
...
< HTTP/1.1 401 Unauthorized
< Date: Sun, 08 Dec 2019 16:21:04 GMT
< Content-Type: text/plain; charset=utf-8
< Content-Length: 84
< Content-Security-Policy: default-src 'none';
< Server: GitHub Registry
< Strict-Transport-Security: max-age=31536000;
< Www-Authenticate: Basic realm="GitHub Package Registry"
< X-Content-Type-Options: nosniff
< X-Frame-Options: DENY
< X-Xss-Protection: 1; mode=block
< X-GitHub-Request-Id: 8DBC:22FA:54DC7:79098:5DED22EF
<
{"errors":[{"code":"UNAUTHORIZED","message":"GitHub Docker Registry needs logi
```

我也可以在本地启动一个开启 Basic Auth 认证的 registry：

复制

```
(MoeLove) ➜  ~ docker run -d \
  -p 5000:5000 \
  --restart=always \
  --name registry \
  -v "$(pwd)"/auth:/auth \
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
  registry:2
f5047241eebf2a81628b46a0d5c4dfdb4ff37b39ca892f9294bac84649eb3a35
```

直接访问下 API：

复制

```
(MoeLove) ➜  ~ curl -I localhost:5000/v2/
HTTP/1.1 401 Unauthorized
Content-Type: application/json; charset=utf-8
Docker-Distribution-Api-Version: registry/2.0
Www-Authenticate: Basic realm="Registry Realm"
X-Content-Type-Options: nosniff
Date: Sun, 08 Dec 2019 16:32:12 GMT
Content-Length: 87
```

可以看到和 GitHub 的 registry 基本一致，均是开启了 Basic Auth 的认证。现在携带认证信息访问下：

```
(MoeLove) ➜  ~ curl -I -u testuser:testpassword localhost:5000/v2/
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json; charset=utf-8
Docker-Distribution-Api-Version: registry/2.0
X-Content-Type-Options: nosniff
Date: Sun, 08 Dec 2019 16:34:40 GMT
```

使用 Docker CLI 登录并推送镜像：

```
(MoeLove) ➜  ~ docker login localhost:5000
Username: testuser
Password:
Login Succeeded
(MoeLove) ➜  ~ docker push localhost:5000/alpine:3.10
The push refers to repository [localhost:5000/alpine]
77cae8ab23bf: Pushed
3.10: digest: sha256:e4355b66995c96b4b468159fc5c7e3540fcef961189ca13fee8777986
```

通过 API 验证：

```
(MoeLove) ➜  ~ curl -u testuser:testpassword localhost:5000/v2/_catalog
{"repositories":["alpine"]}
```

可以看到已经推送成功。

回到之前讨论的问题，刚才的 AuthenticateToRegistry 方法，其实经过了一系列的封装。

复制

```
// components/engine/registry/auth.go#L126
func loginV2(authConfig *types.AuthConfig, endpoint APIEndpoint, userAgent string) (s
    logrus.Debugf("attempting v2 login to registry endpoint %s", strings.TrimRight(en

    modifiers := Headers(userAgent, nil)
    authTransport := transport.NewTransport(NewTransport(endpoint.TLSConfig), modifie

    credentialAuthConfig := *authConfig
    creds := loginCredentialStore{
        authConfig: &credentialAuthConfig,
    }

    loginClient, foundV2, err := v2AuthHTTPClient(endpoint.URL, authTransport, modifi
    if err != nil {
        return "", "", err
    }

    endpointStr := strings.TrimRight(endpoint.URL.String(), "/") + "/v2/"
    req, err := http.NewRequest("GET", endpointStr, nil)
    if err != nil {
        if !foundV2 {
            err = fallbackError{err: err}
        }
        return "", "", err
    }

    resp, err := loginClient.Do(req)
    if err != nil {
        err = translateV2AuthError(err)
        if !foundV2 {
            err = fallbackError{err: err}
        }

        return "", "", err
    }
    defer resp.Body.Close()

    if resp.StatusCode == http.StatusOK {
        return "Login Succeeded", credentialAuthConfig.IdentityToken, nil
    }

    err = errors.Errorf("login attempt to %s failed with status: %d %s", endpointStr,
    if !foundV2 {
        err = fallbackError{err: err}
    }
    return "", "", err
}
```

我们可以从这个 loginV2 看到其大致逻辑，v2 是当前在用的接口版本，从上面的代码中也可以看到都硬编码成了 /v2/ 的形式。

至于对于各种不同的 registry 所采用的认证模式，主要就由 v2AuthHTTPClient 处理了，此处不再展开了。感兴趣的读者可以自行查看其逻辑，或自己手动部署启用不同认证模式的 registry 进行实验。

总结

本篇，我为你深入源码介绍了 Docker 的认证流程和原理。如果仅针对使用 Docker CLI 而言，会用 `docker login/logout` 便已够用。

但在实际工作中，你可能不仅仅只是使用现成的 registry 服务，你也可能需要自己去部署一套 registry，或是使用别人搭建的 registry。了解到认证的流程和原理，有利于排查 registry 相关的问题。

另外，掌握了这些原理，有些时候你可以实现自己的 Client 或是通过脚本调用完成一些自动化的工作。

以上，便是“镜像篇”部分的所有内容。从管理镜像到自己构建镜像，优化构建流程再到分发镜像等均已涉及。

下一部分，是特别准备的两篇“Docker 与 CI/CD pipeline”相关的内容，将前面所学内容应用于实际中。