Docker 内部组件...

本篇是专栏第四部分"架构篇"的第四篇，前面三篇内容，我为你介绍了 Docker 的核心组件及其相关功能。本篇，我们从容器创建的角度来看看 Docker 内部这些组件之间协作的原理。

经过前面内容的学习，想必你已经对 Docker 的核心组件有了一些认识。本篇我们从容器创建的角度，来看看 Docker 的这些组件是如何协作的。本篇使用 Docker CE v19.03.5 的源码进行介绍。

CLI 创建容器

在本课程的第一部分，我们已经知道了创建容器有两种方式：

- 使用 `docker create` 创建 `Created` 状态的容器，之后再通过 `docker start` 来运行容器；
- 使用 `docker run` 创建并运行容器。

我们从 Docker CLI 入手，分别看看这两个命令之间的差别。

docker create

它的入口函数是 runCreate，处理逻辑分以下几步：

- 解析 CLI 的配置文件，默认是 $HOME/.docker/config.json，看其中是否有配置 proxy 相关的参数，如果有则记录下来，如果没有则跳过进入后续流程；
- 解析通过 `docker create` 传递进来的参数，生成创建容器的基本配置；
- 验证配置中是否包含当前 API 版本中不支持的参数；
- 最后调用 createContainer 函数，并返回结果。

复制

```
// cli/command/container/create.go#L64
func runCreate(dockerCli command.Cli, flags *pflag.FlagSet, options *createOptions, c
    proxyConfig := dockerCli.ConfigFile().ParseProxyConfig(dockerCli.Client().DaemonH
    newEnv := []string{}
    for k, v := range proxyConfig {
        if v == nil {
            newEnv = append(newEnv, k)
        } else {
            newEnv = append(newEnv, fmt.Sprintf("%s=%s", k, *v))
        }
    }
    copts.env = *opts.NewListOptsRef(&newEnv, nil)
    containerConfig, err := parse(flags, copts, dockerCli.ServerInfo().OSType)
    if err != nil {
        reportError(dockerCli.Err(), "create", err.Error(), true)
        return cli.StatusError{StatusCode: 125}
    }
    if err = validateAPIVersion(containerConfig, dockerCli.Client().ClientVersion());
        reportError(dockerCli.Err(), "create", err.Error(), true)
        return cli.StatusError{StatusCode: 125}
    }
    response, err := createContainer(context.Background(), dockerCli, containerConfig,
    if err != nil {
        return err
    }
    fmt.Fprintln(dockerCli.Out(), response.ID)
    return nil
}
```

可以看到，最终创建容器的还是 createContainer 函数，我们来看看它的实现。该函数内容较多，以下省略了部分内容。

它的核心逻辑是调用客户端的 ContainerCreate 函数，将所有的参数发送给 Docker Daemon 对应的 API——/containers/create。

如果发现本地不存在创建容器所用到的镜像，则会先拉取镜像到本地，再次调用 API 来创建容器。

以上，就是 Docker CLI 在执行 `docker create` 时的客户端逻辑。我们继续看看 `docker run` 的客户端逻辑。

复制

```
// cli/command/container/create.go#L178
func createContainer(ctx context.Context, dockerCli command.Cli, containerConfig *con
    // ...
    response, err := dockerCli.Client().ContainerCreate(ctx, config, hostConfig, netw

    if err != nil {
        if apiclient.IsErrNotFound(err) && namedRef != nil {
            fmt.Fprintf(stderr, "Unable to find image '%s' locally\n", reference.Fami

            if err := pullImage(ctx, dockerCli, config.Image, opts.platform, stderr);
                return nil, err
            }
            // ...
            var retryErr error
            response, retryErr = dockerCli.Client().ContainerCreate(ctx, config, host
            if retryErr != nil {
                return nil, retryErr
            }
        } else {
            return nil, err
        }
    }
    // ...
    return &response, err
}
```

docker run

`docker run` 的入口函数是 runRun，处理逻辑与前面提到的 `docker create` 基本一致，只不过最后调用的是 runContainer。

我们看看在 runContainer 中做了什么操作吧。

复制

```
func runRun(dockerCli command.Cli, flags *pflag.FlagSet, ropts *runOptions, copts *co
    proxyConfig := dockerCli.ConfigFile().ParseProxyConfig(dockerCli.Client().DaemonH
    newEnv := []string{}
    for k, v := range proxyConfig {
        if v == nil {
            newEnv = append(newEnv, k)
        } else {
            newEnv = append(newEnv, fmt.Sprintf("%s=%s", k, *v))
        }
    }
    copts.env = *opts.NewListOptsRef(&newEnv, nil)
    containerConfig, err := parse(flags, copts, dockerCli.ServerInfo().OSType)
    if err != nil {
        reportError(dockerCli.Err(), "run", err.Error(), true)
        return cli.StatusError{StatusCode: 125}
    }
    if err = validateAPIVersion(containerConfig, dockerCli.Client().ClientVersion());
        reportError(dockerCli.Err(), "run", err.Error(), true)
        return cli.StatusError{StatusCode: 125}
    }
    return runContainer(dockerCli, ropts, copts, containerConfig)
}
```

runContainer 函数内容很多，这里只保留了最核心的内容。它的处理流程是：

- **调用 createContainer 函数，也就是 `docker create` 创建容器所用到的函数**；
- 然后再调用客户端的 ContainerStart 函数，调用 Docker Daemon 的 API ——/containers/{id}/start 启动容器。当然，`docker start` 命令用的也是它，这里就不再展开了。

所以，`docker run` 基本上也算是 `docker create` 和 `docker start` 的组合。

复制
```
func runContainer(dockerCli command.Cli, opts *runOptions, copts *containerOptions, co
    //...
    createResponse, err := createContainer(ctx, dockerCli, containerConfig, &opts.crea
    if err != nil {
        reportError(stderr, "run", err.Error(), true)
        return runStartContainerErr(err)
    }
    //...
    if err := client.ContainerStart(ctx, createResponse.ID, types.ContainerStartOptio
        //...
        return runStartContainerErr(err)
    }
    //...
    return nil
}
```

Docker Daemon 创建容器

看完 CLI 的部分，接下来看看 Docker Daemon 是如何创建容器的。

对应于创建 Docker Daemon 创建容器的 API——/containers/create，它的入口函数是 postContainersCreate，我们只看其重点内容。

整体而言，此函数主要就是处理 Docker CLI 通过 API 传递进来的参数，最后传递给 s.backend.ContainerCreate 真正去创建容器。

复制
```
func (s *containerRouter) postContainersCreate(ctx context.Context, w http.ResponseWr
    //...
    config, hostConfig, networkingConfig, err := s.decoder.DecodeConfig(r.Body)
    //...
    ccr, err := s.backend.ContainerCreate(types.ContainerCreateConfig{
        Name:             name,
        Config:           config,
        HostConfig:       hostConfig,
        NetworkingConfig: networkingConfig,
        AdjustCPUShares:  adjustCPUShares,
    })
    if err != nil {
        return err
    }

    return httputils.WriteJSON(w, http.StatusCreated, ccr)
}
```

我们继续对其进行深入，ContainerCreate 之后调用了 containerCreate 函数，并且在此函数中处理了一些判断系统相关的逻辑。

最后，我们可以看到它实际调用的是 daemon.create(opts) 函数。

而通过分析其实现，我们也可以看到它的调用链是 containerCreate -> create -> newContainer -> newBaseContainer，最终返回一个新创建的容器对象。

复制

```
func (daemon *Daemon) containerCreate(opts createOpts) (containertypes.ContainerCreate
    start := time.Now()
    //...
    container, err := daemon.create(opts)
    if err != nil {
        return containertypes.ContainerCreateCreatedBody{Warnings: warnings}, err
    }
    containerActions.WithValues("create").UpdateSince(start)
    //...
    return containertypes.ContainerCreateCreatedBody{ID: container.ID, Warnings: warn
}
```

复制

```
//engine/daemon/create.go#L107
func (daemon *Daemon) create(opts createOpts) (retC *container.Container, retErr erro
    //...
    if container, err = daemon.newContainer(opts.params.Name, os, opts.params.Config,
        return nil, err
    }
    //...
    if err := daemon.Register(container); err != nil {
        return nil, err
    }
    stateCtr.set(container.ID, "stopped")
    daemon.LogContainerEvent(container, "create")
    return container, nil
}
```

复制

```
func (daemon *Daemon) newContainer(name string, operatingSystem string, config *conta
    //...
    base := daemon.newBaseContainer(id)
    //...
    return base, err
}
```

以上便是 Docker Daemon 创建容器的全部流程，接下来我们看看 Docker Daemon 是如何启动容器的。

Docker Daemon 启动容器

最初的入口函数与 Docker Daemon 创建容器时基本一致，直接来查看其核心的实现。

在启动容器时，会先处理容器需要挂载的卷和网络，以及 Apparmor 之类的配置，这些内容在后续课程中也都会涉及到，此处略过。

最后调用了 daemon.containerd.Create 和 daemon.containerd.Start 函数。

复制

```
func (daemon *Daemon) containerStart(container *container.Container, checkpoint strin
    start := time.Now()
    //...
    if err := daemon.conditionalMountOnStart(container); err != nil {
        return err
    }

    if err := daemon.initializeNetworking(container); err != nil {
        return err
    }
    //...
    if daemon.saveApparmorConfig(container); err != nil {
        return err
    }
    //...
    err = daemon.containerd.Create(ctx, container.ID, spec, createOptions)
    //...
    pid, err := daemon.containerd.Start(context.Background(), container.ID, checkpoin
        container.StreamConfig.Stdin() != nil || container.Config.Tty,
        container.InitializeStdio)
    //...
    return nil
}
```

这里我们直接来看看 daemon.containerd.Start 的函数即可，可以看到其最终是调用的 containerd 来启动容器。

复制

```
func (c *client) Start(ctx context.Context, id, checkpointDir string, withStdin bool,
    ctr, err := c.getContainer(ctx, id)
    spec, err := ctr.Spec(ctx)
    if err != nil {
        return -1, errors.Wrap(err, "failed to retrieve spec")
    }
    labels, err := ctr.Labels(ctx)
    if err != nil {
        return -1, errors.Wrap(err, "failed to retreive labels")
    }
    bundle := labels[DockerContainerBundlePath]
    uid, gid := getSpecUser(spec)
    t, err = ctr.NewTask(ctx,
        func(id string) (cio.IO, error) {
        //...
        },
        func(_ context.Context, _ *containerd.Client, info *containerd.TaskInfo) erro
        //...
        })
    if err := t.Start(ctx); err != nil {
        if _, err := t.Delete(ctx); err != nil {
            c.logger.WithError(err).WithField("container", id).
                Error("failed to delete task after fail start")
        }
        return -1, wrapError(err)
    }

    return int(t.Pid()), nil
}
```

当我们继续深入的话，也就会看到 containerd 启动容器的过程了。在之前的内容中，我们也介绍过，当 Docker 经此调用链创建并启动容器时，containerd-shim 也会被拉起，并通过 runc 来启动容器。

总结

本篇，我通过 Docker 的源码介绍了 Docker 创建和启动容器的过程。我们发现 Docker 经过一系列的处理后，最终将调用 containerd 创建并启动容器。

Docker Daemon 和 containerd 的交互是通过 GRPC 进行的，前面内容中也曾介绍过如果 containerd 没有启动的话，Docker Daemon 也将持续的尝试将其拉起。

下一篇，我将为你介绍 Docker Plugin 系统，通过 Plugin 对 Docker 进行扩展。