Docker 核心架构...

本篇是专栏第四部分"架构篇"的第一个主题"Docker 核心架构及拆解"的下篇。在前两篇中，我分别为你介绍了 docker、containerd、runc 相关的组件，以及 docker-proxy 和 docker-init。本篇，我们来将 Docker 的这些组件组织起来，看看这些组件是如何构建 Docker 核心架构的。

docker 与 containerd

为了更好地理解 Docker 各个组件是如何构建 Docker 核心架构的，本篇我们以一个全新的 Docker 来进行介绍。为了避免我本地环境的差异造成的影响，我们通过以下方式启动一个 Docker In Docker 的 Docker Daemon。

复制

```
(MoeLove) ➜  ~ docker run --name dind --rm -d --privileged docker:dind
293a1a9b565657f713a4dd92306131ef8b209163658f766e61b8db2f5f79e1f2
(MoeLove) ➜  ~ docker ps -l
CONTAINER ID        IMAGE               COMMAND                  CREATED
293a1a9b5656        docker:dind         "dockerd-entrypoint.…"   6 seconds ago
```

上面这种启动 Docker In Docker 容器的方式，在之前"将 Docker 用于 CI/CD pipeline"相关的内容中有做过介绍，这里再次提一下，重点是需要给它加 `--privileged` 参数，否则会启动失败。

我们来看看，在启动 Docker Daemon 时候，容器内的进程状态：

复制

```
(MoeLove) ➜  ~ docker exec dind ps -ef
PID   USER     TIME  COMMAND
    1 root      0:00 dockerd --host=unix:///var/run/docker.sock --host=tcp://0.0.0
   53 root      0:01 containerd --config /var/run/docker/containerd/containerd.tom
  168 root      0:00 ps -ef
```

我们能注意到，containerd 这个时候也已经启动了，并且 containerd 是 dockerd 的子进程。我们来尝试将 containerd 的进程杀掉。

```
(MoeLove) ➜  ~ docker exec dind kill -9 53
(MoeLove) ➜  ~ docker exec dind ps -ef
PID   USER     TIME  COMMAND
    1 root      0:07 dockerd --host=unix:///var/run/docker.sock --host=tcp://0.0.0
  345 root      0:00 ps -ef
```

稍微过一小会儿，再次查看容器内进程的情况：

```
(MoeLove) ➜  ~ docker exec dind ps -ef
PID   USER     TIME  COMMAND
    1 root      0:10 dockerd --host=unix:///var/run/docker.sock --host=tcp://0.0.0
  436 root      0:02 containerd --config /var/run/docker/containerd/containerd.tom
  465 root      0:00 ps -ef
```

可以看到 containerd 进程再次运行了。

你可能会好奇，containerd 的进程为何会再次运行呢？我们可以直接通过 Docker 的源码得到答案：

复制

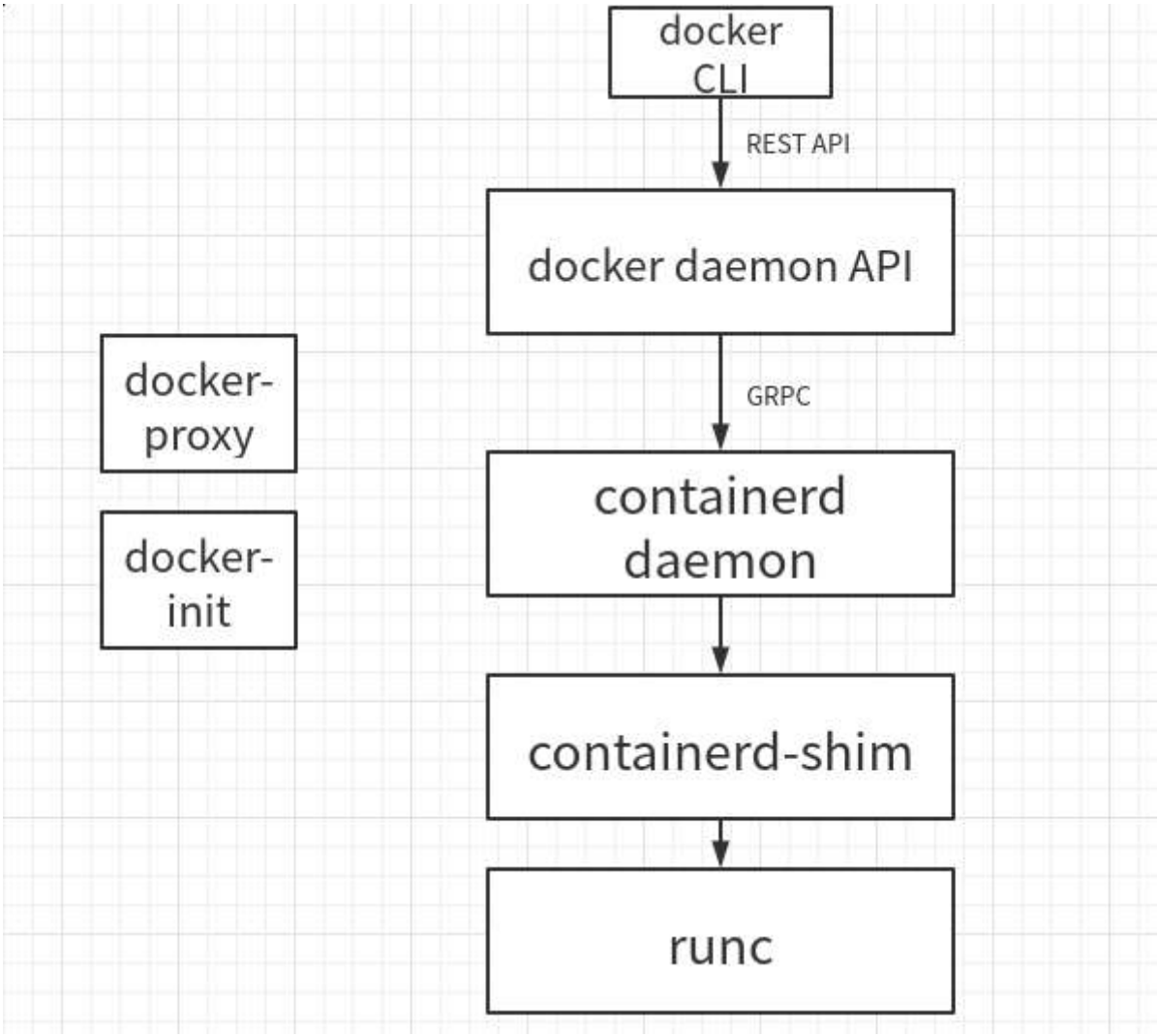
```
// libcontainerd/supervisor/remote_daemon.go#L234
func (r *remote) monitorDaemon(ctx context.Context) {
    var (
        transientFailureCount = 0
        client                *containerd.Client
        err                   error
        delay                 time.Duration
        timer                 = time.NewTimer(0)
        started               bool
    )
    //...
    for {
        //***
        if r.daemonPid == -1 {

            os.RemoveAll(r.GRPC.Address)
            if err := r.startContainerd(); err != nil {
                if !started {
                    r.daemonStartCh <- err
                    return
                }
                r.logger.WithError(err).Error("failed restarting containerd")
                delay = 50 * time.Millisecond
                continue
            }
        }
        //***
    }
}
```

monitorDaemon 函数比较长，这里只截取了其中重点的部分。这个函数名也很清晰，就是在监控 containerd daemon 是否正常运行，如果 containerd daemon 没有正常运行，就拉起 containerd。

如果 containerd daemon 无法正常启动的话，Docker 也无法正常启动容器。关于具体 Docker 调用 containerd 的整个实现，我会在下篇为你介绍。

整体结构

我们来对 Docker 整体做个概览：



其中 docker-proxy 和 docker-init 可通过配置禁用，但 containerd、containerd-shim 和 runc 如果不存在的话，都将影响容器的正常创建。

总结

本篇，我为你介绍了 docker 与 containerd 的关系，Docker Daemon 会持续的监控并拉起 containerd 的进程，但这也仅仅是它们关系的一部分。

下一篇，我会为你从容器创建的角度，详细地分析 docker 这些组件之间具体的调用过程，也便于后续内容中对内部原理进行介绍。