speedy 架构介绍

引言

在开始之前我们首先思考一个问题，docker 本质上都涉及了哪些方面的技术。
这个话题我之前也分享过几次，我个人认为主要包含几个方面：

1. Linux 内核系统技术，如：Namespace, Cgroup 等。
2. 存储技术，如：镜像的存储，镜像的 CoW 所需的文件系统，如：overlayfs, aufs, dm 等。
3. 网络技术，如：libnetwork，flannel 等开源项目主要来解决容器的网络互通及 SDN 等问题。

好，我们今天的话题主要集中在存储技术方面，也就是和镜像相关，docker 依赖的存储技术方面也是分为两个方面：

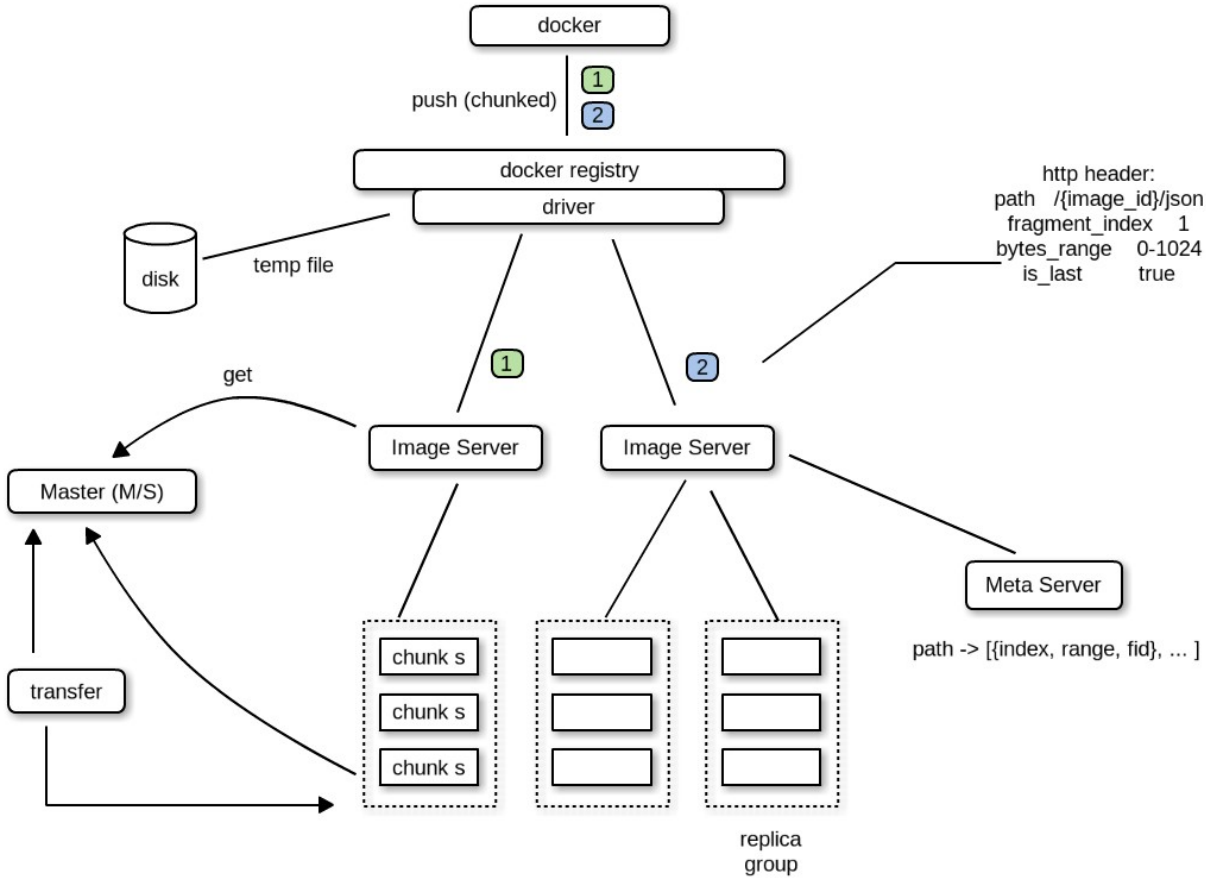
1. 单机内核层存储技术，也就是 overlayfs,aufs 等主要用于提供镜像的 CoW 机制，解决的是一台物理资源跑多个容器，多个容器之间共享同一个 rootfs，然后各自修改是通过 CoW 完成的，这样解决存储空间等问题。这个话题我已经公开分享过很多次，不再多说了。

2. 另一层面，当我们使用 Docker 过程中，可能会使用到很多不同的镜像，这些镜像本身也是需要存储的，目前 Docker 通过 Docker Registry 提供的镜像存储服务选择空间很小，默认只支持本地存储及 S3 存储，社区有 Swift 的驱动可以支持 Swift 存储。个人认为都不是太好的选择。本地存储扩展性和安全性都有问题，Swift 本身架构及性能及使用都存在不少问题。我们也不太倾向于把自己的所有镜像存储到远程的云存储服务上。所以我们开发了自己的镜像存储系统 - speedy。

Speedy 概述

speedy 是我们完全自主研发的一个开源的分布式镜像存储方案，目前支持 Docker Registry 1.0 协议，2.0 协议正在开发中，预计很快会推出。speedy 作为 Docker Registry 的后端存储引擎提供高性能，高可用的，可弹性伸缩的镜像存储服务，用户可以很方便的通过简单加机器的方式来水平扩展存储和服务能力。

Speedy 架构



speedy 本身主要涉及模块：

1. Docker Registry Driver
2. ChunkMaster
3. ChunkServer
4. ImageServer

Docker Registry Driver 是一个遵照 Docker Registry 1.0 协议实现的驱动，完成 Docker Registry 与后端存储系统的对接工作。

ChunkServer 与 ChunkMaster 组成了一个通用的对象存储服务，ChunkMaster 是中心节点，缓存了所有 ChunkServer 的信息，ChunkServer 本身是最终镜像数据落地的存储节点，多个 ChunkServer 会构成一个组，拥有唯一的组 ID，上传这个组内的所有 ChunkServer 都成功才算成功，下载可以随机选择其中一个节点下载。

ImageServer 本身是一个无状态的 Proxy 服务，它相当于是后面通用对象存储服务的一个接入层，Driver 发起的镜像上传/下载操作会直接发给 ImageServer, ImageServer 里面缓存了 ChunkMaster 中的存储节点信息，通过这些信息，ImageServer 会进行 ChunkServer 节点的选择操作，找到一组合适的 ChunkServer 机器完成镜像的上传或下载操作。

上传流程

首先我们通过 docker push 命令发起上传镜像的操作，docker 本身会进行多次与后端存储系统的交互（这里我要简单吐个槽，合理的情况是这个结构化数据和非结构化数据分开存储，docker 本身用 json 表

示结构化的描述信息，也是上传到后端存储系统的，个人觉得 docker 的元数据管理方面很混乱），最后一次交互是上传 image 的 layer 数据到 Docker Registry。

如果使用默认的本地存储，Docker Registry 就直接把数据写到了磁盘上，我们这里通过自己实现的 Driver 完成与后端对象存储系统的上传工作。

我们的 Driver 首先会对源源不断上传过来的字节流进行切割，按照配置的固定大小并发上传到 ImageServer 中，并在上传的 http 请求中携带了该分片的索引及位置信息。

ImageServer 在收到该分片上传请求后，根据自己从 ChunkMaster 中同步过来的 chunk 信息来动态选择一组 ChunkServer，并将分片上传到该组 ChunkServer 中的所有实例上，都成功才返回成功。并将分片索引位置信息及上传成功返回的文件 ID 提交给 MetaServer 保存

Driver 在收到所有分片的上传成功返回后，再返回给前端 Docker，整个上传流程结束。

下载流程

首先 docker 通过 docker pull 请求下载镜像，同样在真正下载数据开始前，docker 同 Docker Registry 以及后端的存储系统间也会产生多次的数据交互，这里省略，最后一步是下载对应的 Image Layer 数据。

Docker Registry 在收到下载请求后首先通过 ImageServer 从 MetaServer 里获取到该文件 path 对应的分片信息，主要是分片的个数，及每一片的索引，然后将这些分片下载请求并发的发送给 ImageServer 服务器

ImageServer 收到分片下载请求后，查询 MetaServer 获得对应的文件 ID，该文件 ID 中包含有 ChunkServer 的位置信息，随后请求相应 ChunkServer 下载数据并返回给 Driver

Driver 收到分片下载的数据后，会根据分片的位置索引进行排序，按文件分片顺序返回给 Docker