编者按：正文开始前，请允许小编再次介绍下作者，这位充电宝都在公司充的北大学霸，不但会过日子，还讲究情怀，是个锤粉。因此他对精益求精都有一种偏执的追求，比如，不但要让 Docker 使开发，测试，生产环境的统一变得容易，还要达到如丝般润滑的持续集成。

应学霸要求插播广告一张

作者简介：刘梦馨，灵雀云高级软件工程师，兼相声大师，目前在灵雀云从事CaaS平台的研发工作。从事过开发、测试、运维相关职位，专注于云计算和虚拟化技术。个人博客 http://oilbeater.com，微信号@oilbeater。学霸的其它文章：《Docker工程师必读论文：Google Borg》，《使用云计算的正确姿势》。

下面进入正文：

Docker 的出现使开发，测试，生产环境的统一变得容易，但是在搭建好基于 Docker 的一整套流水线之后，却发现它运行得不像丝般润滑，甚至比在本地开发测试的效率还低。为什么呢？让我们来看一下这个过程中 Docker 的使用以及 Docker 自身存在着哪些问题，我们又该如何克服这些问题，从而达到如丝般润滑的持续集成。

我们首先来分解一下现在常见的一种利用 Docker 做持续部署的流程：

1. 开发者提交代码
2. 触发镜像构建

3. 构建镜像上传至私有仓库
4. 镜像下载至执行机器
5. 镜像运行

在这五步中，1 和 5 的耗时都比较短，主要耗时集中在中间 3 步，也就是 `docker build`，`push`，`pull` 的时间消耗，我们就来分别看一下如何加速这三个步骤。

docker build

1 选择国外构建

由于 dockerhub 的官方镜像再国外，而这些基础镜像的软件源都在国外，国内构建的时候网络会是很大的瓶颈，有能力在国外机器进行构建，并且可以通过专线和国内进行传输的话，还是优先将构建节点放在国外，会省很多无谓的在网络上的纠缠，并且很多软件源国外的也要更稳定写，更新也更及时。

如果只能在国内进行构建的话，建议使用国内的镜像，或者自己在私有仓库存一份官方镜像，并且对镜像进行改造，做一份软件源都在国内的基础镜像，把构建过程中的网络传输都控制在国内或者内网，这样就不用和网络进行纠缠了。

2 善用 .dockerignore

.dockerignore 可以减少构建时的文件传输，一般通过 git 进行持续构建的时候不做设置都会把 .git 文件夹进行传输造成很多无用的传输，一些与构建无关的代码也尽量写在.dockerigonre 文件中。

3 缓存优化的dockerfile

dockerfile 的优化也是一个比较直接的优化方式，优化的核心就是能充分利用 build cache，把每次变化的部分放在最后，一般把加入代码放在最后一步，这样每次构建只有最后一层是新的，其他部分都是可以用 cache 的。对于 node、python、go 之类要在构建过程中安装依赖的服务，可以把安装依赖和加入代码分两步完成，这样在依赖不变的情况下这部分的缓存也是可以利用的。以 node 为例：

```
COPY package.json /usr/src/app/
RUN npm install
COPY . /usr/src/app
```

其他关于 dockerfile 优化的建议可以再单独开一篇了，基本上每个命令都需要特殊对待才能不掉

坑里，可以参考一个在线 dockerfile 语法优化器（http://dockerfile-linter.com/），里面会提供一些相关的 dockerfile 优化建议和一些资源,作者一定是个大好人。

4 smart cache

在单机模式下充分利用 build cache 是个不错的注意，但是在多个构建机器的情况下就会有问题了。出于磁盘空间考量不可能所有机器都存着所有的镜像，这样缓存优化的 dockerfile 就没有用武之地了。为了让 cache 重新发挥作用我们可以在构建开始时将旧的镜像 pull 下来，这样一来就可以再次利用 cache 了。但是：

1. pull 镜像也是需要很多时间的，并且 pull 下来的镜像并不会全部有用，会浪费一定的时间；
2. 而来如果 dockerfile 变化比较大有可能没有一层能用 pull 下来反而会浪费更多的时间；
3. 三来仓库内可能会有其他的镜像更适合做当前构建的缓存所以我们需要实现一个精准的镜像拉取，不能出错也不能浪费。

举个栗子，如下图所示：

想要构建 node:wheezy 的话，那么 node:0-wheezy 是一个比较合适的镜像来做 cache 而想要构建 node:5 的话那么 node:wheezy 和 node:0-wheezy 都不太合适，反而是 python:latest 会更合适。如果我们把仓库中所有的镜像都做成这样一个森林，利用 tire 树可以很精准的知道，哪个镜像的哪几层是 cache 的最好选择，这样精确制导不会有一点浪费。

docker push

docker registry 在升级到 v2 后加入了很多安全相关检查，使得原来很多在 v1 already exist 的层依然要 push 到 registry,并且由于 v2 中的存储格式变成了 gzip，在镜像压缩过程中占用的时间很有可能比网络传输还要多。我们简单分解一下 `docker push` 的流程。

1. buffer to disk 将该层文件系统压缩成本地的一个临时文件
2. 上传文件至 registry
3. 本地计算压缩包 digest，删除临时文件，digest 传给 registry
4. registry 计算上传压缩包 digest 并进行校验
5. registry 将压缩包传输至后端存储文件系统
6. 重复 1-5 直至所有层传输完毕
7. 计算镜像的 manifest 并上传至 registry 重复 3-5

此外判断 `already exist skip pushing` 的条件变严格了，必须是本地计算过digest 且 该 digest 对应的文件属在对应 repo 存在才可以。

换句话说就是如果这个镜像层是 pull 下来的，那么是没有digest的还是要把整个压缩包传输并计算 digest，如果这个镜像你之前并没有比如 ubuntu 的 base image 你的 repo 第一次创建之前没传输过，那么第一次也要你传输一次确认你真的有 ubuntu。

这里面的改进点就是在太多了，先列举 Docker 官方已经做得和正在做的。
1. 1.9.1 后 push 是 streaming 式的，也就是把 1 和 2 合并去掉临时文件，直接一边压缩一边传输；
2. pull 镜像后 digest 保存，大概是 1.8.3 之后添加的省去了重复计算
3. registry 可以直接 mount 别人 repo 中的一层到自己的 repo,只要有pull权限即可，这个工作还在进行中。

但是这只解决了一小部分问题，push 依然会很慢，docker 和 registry 的设计更多的考虑了公有云的环境设置了过多的安全防范为了防止镜像的伪造和越权获取，但是在一个可信的环境内如果 `build` 和 `push` 过程都是自己掌控的，那么很多措施都是多余的，我们可以设计一个自己的 smart pusher 挖掘性能的最大潜力：
- 压缩传输 streaming 化和 docker 1.9.1 实现的类似
- 越过 registry 直接和存储系统通信，直接拿掉上面 5 的传输时间
- 如果 digest 在 存储系统中存在则不再重复传输，在 manifest 中写好层次关系就好
- 将多层的传输并行化，多个层一块传，这样才能充分发挥多核的优势，docker 自带的串行 push效率实在是太低了
- 另外针对 build 结果进行 push 的 smart pusher 可以将流水线发挥到极致，build 每构建出一层就进行传输，将 build 和 push 的时间重叠利用

有了 smart pusher，push 时间的绝大多数都被隐藏到了 build 的时间中，我们把并发和流水线的技术都用上，充分发挥了多核的优势。

docker pull

`docker pull` 镜像的速度对服务的启动速度至关重要，好在 registry v2 后可以并行 pull 了，速度有了很大的改善。但是依然有一些小的问题影响了启动的速度：

- 下载镜像和解压镜像是串行的
- 串行解压，由于 v2 都是 gzip 要解压，尽管并行下载了还是串行解压，内网的话解压时间比网络传输都要长
- 和 registry 通信，registry 在 pull 的过程中并不提供下载内容只是提供下载 url 和鉴权，这一部分加长了网络传输而且一些 metadata 还是要去后端存储获取，延时还是有一些的
- `docker pull` 某些情况会卡死，不 `docker restart` 很难解决，而 restart 又会停止所有服务，严重影响服务稳定性。

为此我们还需要一个独特设计的 smart puller 帮助我们解决最后的问题：

1. streaming downloading and extracting 和在 smarter pusher 做的类似将这两步合为一步；
2. 并行解压，也和 smarter pusher 并行压缩类似；
3. 越过 registry 直接和后端存储通信；
4. redis 缓存 metadata

有了 smart puller 我们自然的将 docker pull 的工作和 docker daemon 解耦了，这样再不会发生 pull 导致的 docker hang,服务稳定性也得到了增强，解绑后其实 Docker 只是做一个 runtime 这一部分也可考虑改成 runc 去除掉 daemon 这个单点，不过这个工作量就比较大了。此外 smart puller 也可以帮助我们实现在 smart cache 中的精确 pull 以及 pull cache 的加速，可谓一举多得。

总结

将 push 和 pull 的工作和 daemon 解绑，把 smart cache，smart puller 和 smart pusher 用上后，持续集成如丝般润滑。smart **系列已在灵雀云小部分上线测试，还请大家持续关注。**

至于我把全系列工具都以 smart 命名，主要是为了给 Smartisan T2 开卖造势（逃）
