Docker的典型应用场景

2015年03月10日

相对于VM，docker在其轻量、配置复杂度以及资源利用率方面有着明显的优势。 随着docker技术的不断成熟，越来越多的企业开始考虑通过docker来改进自己的IT系统。

本文列举一些docker的实际应用场景，以期能够起到抛砖引玉的作用， 来帮助大家更加方便的使用docker.

应用打包

制作过RPM、GEM等软件包的同学可能很清楚，每一个软件包依赖于哪个库的哪个版本， 往往需要明确的写在依赖列表里。而依赖又往往分为编译时依赖和运行时依赖。

在传统的基础设施环境下，为了保证所生成的软件包在其它机器上可正常安装且运行， 一般需要在打包之前创建个干净的虚拟机，或者手工创建个chroot环境， 然后在这个干净的环境下安全各种依赖包，然后执行打包脚本。 生成软件包以后，需要再创建一个干净的环境安装、运行这个软件包，来验证是否符合预期。 这样虽然也能完成打包工作，但至少有以下缺点：

1. 耗时耗力
2. 依赖关系容易漏掉，比如：在干净的环境中经过多次调试，把缺少的依赖包一个一个的装上了，
   但最后写spec文件时却忘记添加某个依赖，导致下次打包时需要重新调试或者打包后软件包无法使用等问题。

【解决方法】

通过docker可以很好的解决打包问题。具体作法如下：

1. “干净的打包环境”很容易准备，docker官方提供的ubuntu、centos等系统镜像天生就能作为纯净无污染的打包环境使用
2. Dockerfile本身能起到文档固化的作用，只要写好Dockerfile，创建好打包镜像，以后就能无限次重复使用这个镜像进行打包

示例：

我们要为某个PHP扩展模块（如：php-redis）制作个RPM包。

首先，需要写个用于创建打包镜像的Dockerfile， 内容如下：

```
FROM centos:centos6
RUN yum update -y
RUN yum install -y php-devel rpm-build tar gcc make
RUN mkdir -p /rpmbuild/{BUILD,RPMS,SOURCES,SPECS,SRPMS} && \
  echo '%_topdir /rpmbuild' > ~/.rpmmacros
ADD http://pecl.php.net/get/redis-2.2.7.tgz /rpmbuild/SOURCES/redis-2.2.7.tgz
ADD https://gist.githubusercontent.com/mountkin/5175c213585d485db31e
/raw/02f6dce79e12b692bf39d6337f0cfa72813ce9fb/php-redis.spec /redis.spec
```

```
RUN rpmbuild -bb /redis.spec
```

然后执行`docker build -t php-redis-builder .`，执行成功后，就会生成我们需要的RPM包。

接下来，执行以下命令把生成的软件包从docker镜像中复制出来：

```
[ -d /rpms ] || mkdir /rpms
docker run --rm -v /rpms:/rpms:rw php-redis-builder cp /rpmbuild/RPMS/x86_64/php-redis-2.2.7-1.el6.x86_64.rpm
/rpms/
```

然后/rpms目录下就会有我们刚刚制作的RPM包。

最后，软件包的验证过各也非常简单，只需要新创建一个docker镜像，把新生成的软件包添加进去并安装即可。

Dockerfile如下（为了ADD RPM文件，需要保存在/rpms目录下）：

```
FROM centos:centos6
ADD php-redis-2.2.7-1.el6.x86_64.rpm /php-redis-2.2.7-1.el6.x86_64.rpm
RUN yum localinstall -y /php-redis-2.2.7-1.el6.x86_64.rpm
RUN php -d "extension=redis.so" -m |grep redis
```

在/rpms目录下执行`docker build -t php-redis-validator .`，如果执行成功,则表明RPM包可正常工作。

多版本混合部署

随着产品的不断更新换代，一台服务器上部署多个应用或者同一个应用的多个版本在企业内部非常常见。

【问题】

但一台服务器上部署同一个软件的多个版本，文件路径、端口等资源往往会发生冲突，造成多个版本无法共存的问题。

【解决方法】

如果用docker，这个问题将非常简单。由于每个容器都有自己独立的文件系统，所以根本不存在文件路径冲突的问题；对于端口冲突问题，只需要在启动容器时指定不同的端口映射即可解决问题。

升级回滚

一次升级，往往不仅仅是应用软件本身的升级，通过还会包含依赖项的升级。 但新旧软件的依赖项很可能是不同的，甚至是有冲突的，所以在传统的环境下做回滚一般比较困难。

【解决方法】

如果使用docker，我们只需要每次应用软件升级时制作一个新的docker镜像，升级时先停掉旧的容器，然后把新的容器启动。 需要回滚时，把新的容器停掉，旧的启动即可完成回滚，整个过程各在秒级完成，非常方便。

【优势】

多租户资源隔离

【VM劣势】

资源隔离对于提供共享hosting服务的公司是个强需求。 如果使用VM，虽然隔离性非常彻底，但部署密度相对较低，会造成成本增加。

docker容器充分利用linux内核的namespaces提供资源隔离功能。
结合cgroup，可以方便的设置某个容器的资源配额。 既能满足资源隔离的需求，又能方便的为不同级别的用户设置不同级别的配额限制。

【Docker优势】

但在这种应用场景下，由于容器中运行的程序对于hosting服务提供方来说是不可信的， 所以需要特殊的手段来保证用户无法从容器中操作到宿主机的资源（即：越狱，尽管这种问题发生的概率很小，但安全无小事，多一层防护肯定让人更加放心）。

安全及隔离性加固方面，可考虑以下措施：

1. 通过iptables阻断从容器到所有内网IP的通信（当然如果需要也可以针对特定的IP/端口开放权限）
2. 通过selinux或者apparmor限制某个容器所能访问的资源
3. 对某些sysfs或者procfs目录，采用只读方式挂载
4. 通过grsec来加固系统内核
5. 通过cgroup对内存、CPU、磁盘读写等资源进行配额控制
6. 通过tc对每个容器的带宽进行控制

另外我们在实际测试中发现系统的随机数生成器很容易因熵源耗尽而发生阻塞。 在多租户共享环境下需要在宿主机上启用rng-tools来补充熵源。

这个应用场景下有很多工作是docker本身所不能提供的，并且实施起来需要关注的细节比较多。 为此我们提供了安全加强版docker管理平台，可完美解决以上问题。 需要的朋友可以通过csphere官网了解更多细节。

内部开发环境

在容器技术出现之前，公司往往是通过为每个开发人员提供一台或者多台虚拟机来充当开发测试环境。

【问题】

开发测试环境一般负载较低，大量的系统资源都被浪费在虚拟机本身的进程上了。

docker容器没有任何CPU和内存上的额外开销，很适合用来提供公司内部的开发测试环境。
而且由于docker镜像可以很方便的在公司内部分享，这对开发环境的规范性也有极大的帮助。

【优势】

如果要把容器作为开发机使用，需要解决的是远程登录容器和容器内进程管理问题。 虽然docker的初衷是为“微服务”架构设计的，但根据我们的实际使用经验， 在docker内运行多个程序，甚至sshd或者upstart也是可行的。

这方面csphere也有成熟的产品及解决方案，欢迎感兴趣的朋友试用反馈。

后记

以上总结了我们在实际开发和生产环境中使用docker的一些场景， 以及在每种情况下遇到的问题和相应的解决方法，

希望对有意使用docker的朋友有所启发。 同时我们也欢迎更多的朋友分享关于docker的使用经验。