[**利用Docker构建开发环境**](http://tech.uc.cn/?p=2726)

Posted by

[*makewonder*](http://tech.uc.cn/?author=54)

 on 2014 年 4 月 2 日

[Tweet](http://twitter.com/share)

Share[**50**](http://tech.uc.cn/?p=2726)

最近接触PAAS相关的知识，在研发过程中开始使用Docker搭建了自己完整的开发环境，感觉生活在PAAS时代的程序员真是幸福，本文会简要介绍下Docker是什么，如何利用Docker来搭建自己的开发环境（本文主要是面向Mac OS X），以及期间所遇到的一些坑和解决方案。（本文会要求你对PAAS、LXC、CGroup、AUFS有一定的了解基础，请自行Google ）

**大背景–虚拟化技术历史**

计算机虚拟化技术由来已久，从硬件仿真到全虚拟化，再到准虚拟化和操作系统虚拟化，各种技术粉墨登场，种类繁多，说实在的有点眼花缭乱和复杂；但用户的核心诉求一直是比较简单的，降低信息技术（IT）的运营成本，提高资源利用率，提高安全性和可靠性等等；虽说用户的核心诉求比较简单，但每个时代的需求场景却是不同的。在大型机时代，虚拟化技术被用来支持多个用户能够同时使用大型机，在x86架构时代，随着企业服务的大规模部署，虚拟化技术主要是用来提高企业资源的利用率，而现如今，随着云计算时代的到来，人们对应用的安全性、隔离性越来越高，对于部署的标准化以及虚拟机的性能要求越来越高。现如今，一种叫Linux容器的虚拟化技术逐渐得到广泛的应用，它的优点有许多，本文不一一赘述，有太多的文章可以参考。

**什么是Docker？**

docker的英文本意是码头工人，也就是搬运工，这种搬运工搬运的是集装箱（Container），集装箱里面装的可不是商品货物，而是任意类型的App，Docker把App（叫Payload）装在Container内，通过Linux Container技术的包装将App变成一种标准化的、可移植的、自管理的组件，这种组件可以在你的latop上开发、调试、运行，最终非常方便和一致地运行在production环境下。

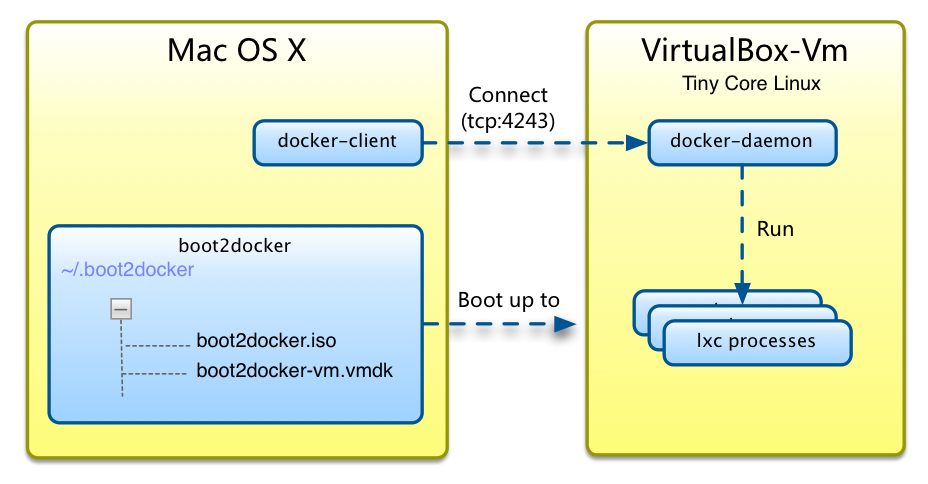
Docker的核心底层技术是LXC（Linux Container），Docker在其上面加了薄薄的一层，添加了许多有用的功能。[这篇stackoverflow](http://stackoverflow.com/questions/17989306/what-does-docker-add-to-just-plain-lxc)上的问题和答案很好地诠释了Docker和LXC的区别，能够让你更好的了解什么是Docker， 简单翻译下就是以下几点：

* Docker提供了一种可移植的配置标准化机制，允许你一致性地在不同的机器上运行同一个Container；而LXC本身可能因为不同机器的不同配置而无法方便地移植运行；
* Docker以App为中心，为应用的部署做了很多优化，而LXC的帮助脚本主要是聚焦于如何机器启动地更快和耗更少的内存；
* Docker为App提供了一种自动化构建机制（Dockerfile），包括打包，基础设施依赖管理和安装等等；
* Docker提供了一种类似git的Container版本化的机制，允许你对你创建过的容器进行版本管理，依靠这种机制，你还可以下载别人创建的Container，甚至像git那样进行合并；
* Docker Container是可重用的，依赖于版本化机制，你很容易重用别人的Container（叫Image），作为基础版本进行扩展；
* Docker Container是可共享的，有点类似github一样，Docker有自己的INDEX，你可以创建自己的Docker用户并上传和下载Docker Image；
* Docker提供了很多的工具链，形成了一个生态系统；这些工具的目标是自动化、个性化和集成化，包括对PAAS平台的支持等；

那么Docker有什么用呢？对于运维来说，Docker提供了一种可移植的标准化部署过程，使得规模化、自动化、异构化的部署成为可能甚至是轻松简单的事情；而对于开发者来说，Docker提供了一种开发环境的管理方法，包括映像、构建、共享等功能，而后者是本文的主题。

**Docker的安装和构成**

Docker官方本身提供了非常具体的安装教程，这里不说具体的安装过程，请参考[Docker安装（Mac系统）](http://docs.docker.io/en/latest/installation/mac/)，重要的是描述下原理和安装完成后的结构，好对Docker更好的了解。 由于LXC本身不支持Mac内核，因此需要跑一个VirtualBox虚拟机（[TinyCoreLinux](http://tinycorelinux.net/)）来安装，幸好Docker社区提供了一个非常方便的工具boot2docker（其实就是一个VBoxManage的包装shell脚本），用于安装Mac下的整个Docker环境。具体的结构如下：

[](http://tech.uc.cn/wp-content/uploads/2014/04/docker-install.png)

如图所示，安装完成后，具体情况如下：

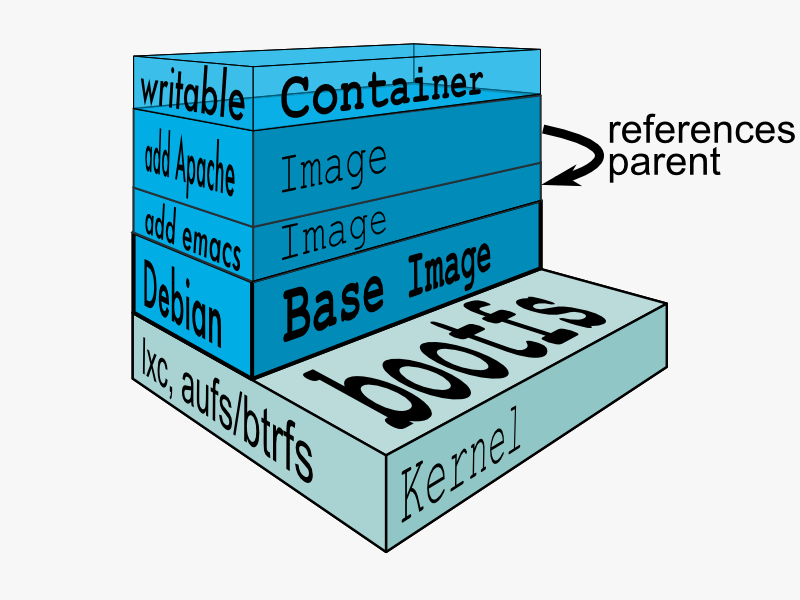
* 在Mac的home目录~/.boot2docker下创建了虚拟机所需要的文件，其中boot2docker.iso是虚拟机映像，这是一个由CD-ROM引导的TinyCoreLinux系统；而boot2docker-vm.vmdk文件则是你的虚拟机磁盘，你所有的持久化数据都存放在这里，包括docker创建的lxc容器等文件。
* 在Mac下，docker被分为客户端docker-client和服务端docker-daemon两部分，如果是在linux（比如ubuntu），实际上则是同一个可执行文件同时充当客户端和服务端。docker-daemon可以监听unix scoket，也可以在tcp socket（默认端口为4234），docker-client会通过一个叫DOCKER\_HOST的环境变量读取服务地址和端口，因此你应该在你的bash\_profile文件里面添加这么一行：



|  |  |
| --- | --- |
| 1  2 | export DOCKER\_HOST=tcp://127.0.0.1:4243 |

docker-daemon跑在虚拟机上，这个程序实际上就是接收docker-client发送过来的消息命令，创建、启动和销毁lxc容器，以及docker本身的版本管理、映像存储等等 运行你的第一个docker容器 安装完成后，就差不多可以开始创建和运行docker容器了，在这之前，你首先得下载一个Image，什么是Image？我们先来了解docker的2个基础概念：**Image**和**Container**。

Container和Image 在Docker的世界里，Image是指一个只读的层（Layer），这里的层是AUFS里的概念，最直观的方式就是看一下docker官方给出的图：

[](http://tech.uc.cn/wp-content/uploads/2014/04/docker-filesystems-multilayer.png)

Docker使用了一种叫AUFS的文件系统，这种文件系统可以让你一层一层地叠加修改你的文件，最底下的文件系统是只读的，如果需要修改文件，AUFS会增加一个可写的层（Layer），这样有很多好处，例如不同的Container可以共享底层的只读文件系统（同一个Kernel），使得你可以跑N多个Container而不至于你的硬盘被挤爆了！这个只读的层就是Image！而如你所看到的，一个可写的层就是Container。

那Image和Container的区别是什么？很简单，他们的区别仅仅是一个是只读的层，一个是可写的层，你可以使用docker commit 命令，将你的Container变成一个Image，也就是提交你所运行的Container的修改内容，变成一个新的只读的Image，这非常类似于git commit命令，感觉真棒！

实际上这就是Docker对Container映像的版本管理基石，AUFS文件系统实在是太美妙了，更多细节可以参考DotCloud的[这篇文章](http://blog.dotcloud.com/kernel-secrets-from-the-paas-garage-part-34-a)。

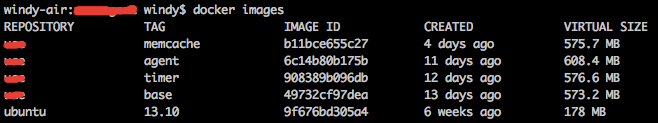
**运行和退出**

在了解了Image和Container的概念后，我们可以开始下载一个Image，Docker的好处就是提供了一个类似github的Image仓库管理，你可以非常方便pull别人的Image下来运行，例如，我们可以下载一个ubuntu Image：

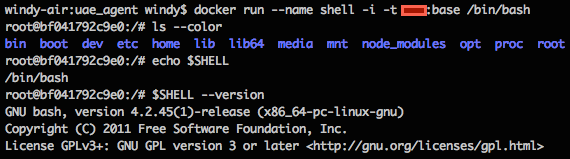


|  |  |
| --- | --- |
| 1  2 | docker pull ubuntu:13.10 |

这里的13.10是一个Tag，类似于git的tag，这里的tag可以为你制定一个ubuntu的版本。下载完成后，执行docker images命令可以列出你已经下载或者自己构建的image：（请允许我使用可爱的马赛克 :) ）

[](http://tech.uc.cn/wp-content/uploads/2014/04/QQ20140322-1.png)

你可以看到ubuntu:13.10的大小为178MB，以及它的IMAGE ID。 现在我们开始运行一个Container，命令很简单，例如我们想运行一个执行Shell终端的Container：

[](http://tech.uc.cn/wp-content/uploads/2014/04/QQ20140322-2.png)

如你看到的，你已经进入到一个Shell里面，可以执行你想执行的任何命令，就和在ubuntu里面一样，进去后默认是在根目录/下，可以看到经典的unix/linux目录结构，以及你所运行的bash版本等信息。你可以给你的Container定一个名字，通过–name选项，例如这里命名了shell，日后你就可以直接用这个名字引用Contanier。

退出一个Container也很简单，你直接exit就好了。 其他更多的命令这里不做赘述，因为官方的文档已经非常全面，这里只是给一个直观的初步印象。下面进入主题。

**利用Docker搭建开发环境**

我们先看看程序员在搭建开发环境时遇到的一些问题：

* **软件安装麻烦**，比如很多公司都使用redhat，一般开发人员又不给root，安装一个nginx或者是mysql都得自己下载编译安装 权限问题，没有root，一些软件无法运行，例如dnsmasq；
* **没有root**，无法修改hosts，无法netstat -nptl，无法tcpdump，无法iptable
* **隔离性差**，例如不同的开发人员如果在同一台主机环境下共享开发，虽然是用户隔离，但端口如果不规范可能会冲突；同一个Mysql如果权限管理不好很有可能误删别人的数据
* **可移植性差**，例如和生产环境不一致，开发人员之间也无法共享；更严重的情况是当有新人入职时，通常需要又折腾一遍开发环境，无法快速搭建

这些问题可以通过在本地搭建虚拟机来解决，但虚拟机是一个很笨重的解决方案，Docker是一个非常轻量级的方案，而且还拥有虚拟机没有的一些功能，例如标准化Image，Image共享等，更重要的是，利用Docker，你可以运行非常多的容器，在你的Mac下搭建一个分布式的开发环境根本不是什么大的问题，而且对内存、磁盘和cpu的消耗相比传统的虚拟机要低许多，这些都要归功于AUFS和LXC这两大神奇的技术。

**构建基础Image**

想要搭建一个节省磁盘空间和扩展性良好的开发环境，最重要的第一步就是构建一个基础性的Image，比如你的主要开发语言是Ruby，那么你肯定需要一个已经安装好以下工具的基础Image：

* ruby
* bundler
* gem

然后在此基础上，你可以扩展这个基础的Image（下面叫base）为不同的开发环境，例如rails，或者是nats。当然，你的这个base也可以从别人的Image扩展而来，还记得我们刚刚pull下来的ubuntu:13.10这个Image吗？你可以从这个Image扩展开始构建你的base，如何做呢？Docker提供了一种标准化的DSL方式，你只需要编写一个Dockerfile，运行docker build指令，就可以构建你自己的Image，这有点像Makefile和make命令一样，只是大家要构建的内容和构建语言不同。

Dockerfile的语法请参考[Dockerfile Reference](http://docs.docker.io/en/latest/reference/builder/)，这里给出上面提到的Ruby开发的base Dockerfile示例：



|  |  |
| --- | --- |
| 1  2  3  4  5 | FROM ubuntu:13.10  RUN apt-get update  RUN apt-get install -y ruby ruby-dev gem  RUN gem install bundler |

这里只用到了很简单的2个指令：FROM和RUN，FROM指定了我们要扩展的Image，RUN指定我们要运行的命令，这里是安装ruby，gem、bundler等软件。写好Dockerfile后，运行以下指令就可以创建你的base image了：



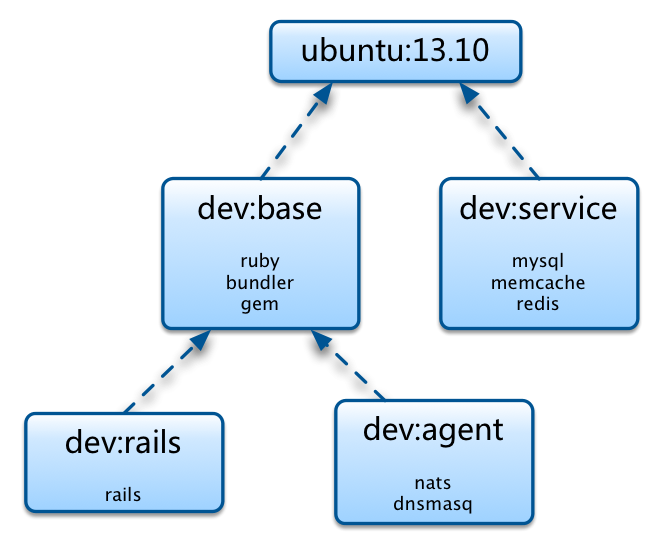
|  |  |
| --- | --- |
| 1  2 | docker build --rm -t dev:base . |

-t 选项是你要构建的base image的tag，就好比ubuntu:13.10一样 –rm 选项是告诉Docker在构建完成后删除临时的Container，Dockerfile的每一行指令都会创建一个临时的Container，一般你是不需要这些临时生成的Container的 如你所想，我们可以像运行ubuntu:13.10那样运行我们的base了：



|  |  |
| --- | --- |
| 1  2 | docker run -i -t --name ruby dev:base irb |

这里我们使用dev:base这个Image运行了一个irb解释器（Ruby的交互式解释器）。 在构建完base之后，你可以依样画葫芦构建你的rails环境，很简单，只需要FROM dev:base，然后RUN安装你的rails组件就可以了，不再赘述。最终你可能构建的开发环境是这样的：

[](http://tech.uc.cn/wp-content/uploads/2014/04/docker-dev.png)

如上图所示，base和service都是从ubutnu:13.10继承而来，他们作为不同的基础开发环境，base是ruby开发环境（也许命名为dev:ruby更为合适？），而service是一些基础数据服务，例如mysql，memcache，我建议将这些第三方组件集中在一个Container中，因为他们的环境不经常修改，可以作为一种底层服务Container运行，除非你需要构建分布式的服务，例如memcache集群，那可以继续拆分。

**指定Image入口**

当你构建完你的base Image和其他应用的Image之后，你就可以启动这些Image了，还记得前面我们给出的运行命令吗？



|  |  |
| --- | --- |
| 1  2 | docker run -i -t --name shell dev:base /bin/bash |

这里我们运行了一个bash，这样你就可以在shell里面执行你所想要执行的任何命令了，但是我们有时候并不想每次都启动一个shell，接着再在shell里面启动我们的程序，比如一个mysql，而是想一启动一个容器，mysql服务就自动运行了，这很简单，Dockerfile提供了CMD和ENTRYPOINT这2个指令，允许你指定一个Image启动时的默认命令。CMD和ENTRYPOINT的区别是CMD的参数可以由docker run指令指定的参数覆盖，而ENTRYPOINT则不可以。例如我们想运行一个memcached服务，可以这么写Dockerfile：



|  |  |
| --- | --- |
| 1  2  3 | FROM ubuntu:13.10  RUN apt-get install -y memcached CMD memcached -u root -p 40000 |

或者可以这么写：



|  |  |
| --- | --- |
| 1  2  3 | FROM ubuntu:13.10  RUN apt-get install -y memcached ENTRYPOINT ["memcached", "-u", "root", "-p", "40000"] |

注意不要把memcached启动为后台进程，即加上-d选项，否则docker启动的container会马上stop掉，这点我也觉得比较意外。 接着我们build这个Image：



|  |  |
| --- | --- |
| 1  2 | docker build -t dev:memcache . |

这样，当你build完你的Image后，你可以直接将该Image运行为一个容器，它会自动启动mysql服务：

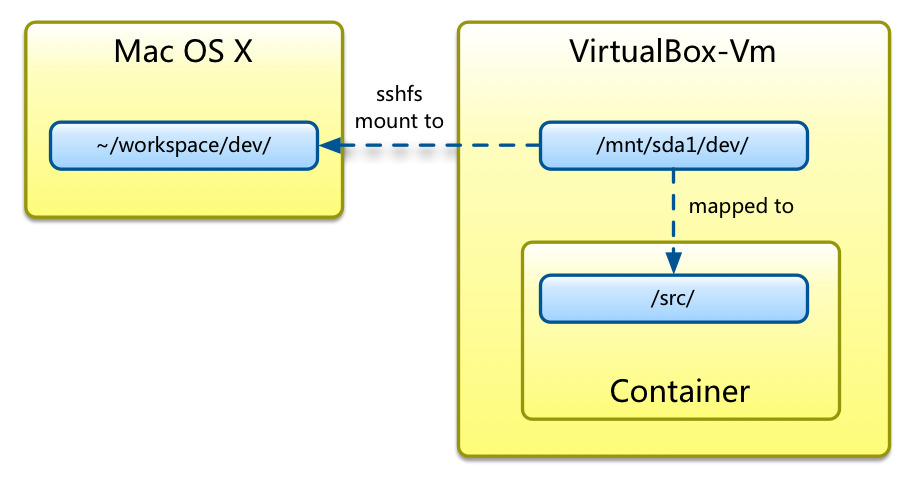


|  |  |
| --- | --- |
| 1  2 | docker run --name memcache\_service -d dev:memcache |

注意使用-d (detach) 选项，这样这个container就会作为后台进程运行了，接着你可以使用docker ps命令查看是否有在运行。

**磁盘映射**

大部分时候你会需要把你host主机（宿主）上的目录映射到Container里面，这样你就非常方便地在host主机上编辑代码，然后直接就可以在Container里面运行它们，而不用手动copy到Container里面再重启Container。按理将host的目录映射到guest（指Container）上应该是一件很容易的事情，就好像VMWare那样，但可惜的是，由于Mac上的Docker多了一层虚拟机，因此多了一层周折，你必须先VM上的目录通过sshfs mount到host（指Mac）上，然后再将你的目录或文件copy到这个mount的目录，再将VM上的这个目录映射到Container里，听起来比较拗口，画个图会清晰很多。

[](http://tech.uc.cn/wp-content/uploads/2014/04/docker-disk-map.png)

如上图所示，VM里面的/mnt/sda1/dev/目录（你需要自己创建）通过sshfs命令mount到了host主机（Mac）的~/workspace/dev/目录 ，而VM里的/mnt/sda1/dev/目录又被映射到了Container的/src/目录下，这样你就可以在Container里面的/src/目录下访问你的host文件了。具体如何做呢？首先你需要安装sshfs命令，然后将VM的password写到一个文件中，例如~/.boot2docker/b2d-passwd，在用sshfs命令mount起VM的/mnt/sda1/dev目录：



|  |  |
| --- | --- |
| 1  2  3  4 | brew install sshfs  cat tcuser > ~/.boot2docker/b2d-passwd  sshfs docker@localhost:/mnt/sda1/dev ~/workspace/dev -p 2022 -o reconnect -o password\_stdin < ~/.boot2docker/b2d-passwd |

接着你在run一个Container的时候需要通过-v选项来将/mnt/sda1/dev/映射到/src目录：



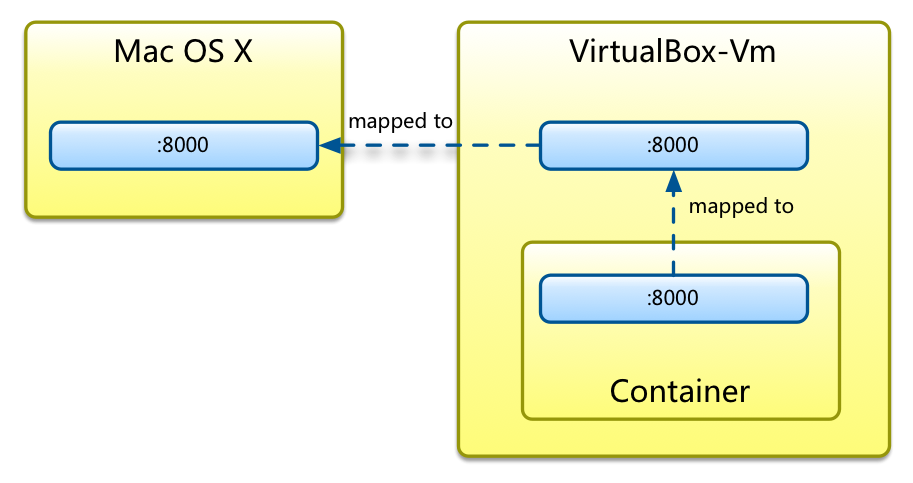
|  |  |
| --- | --- |
| 1  2 | docker run -i -t dev:base -v /mnt/sda1/dev:/src /bin/bash |

这样你就可以在你的Container的/src目录下看到你host里的文件了。 磁盘映射还有2个地方需要注意：

* 你的文件实际上是存储在VM里面的，也就是说你需要将你的目录或者文件copy到VM里面，你sshfs之后，就是copy到~/workspace/dev目录下
* 千万不要sshfs mount非/mnt/sda1下的目录，因为VM里面跑的是TinyCoreLinux，这个OS的rootfs是临时性的（放在内存的，实际上就是boot2docker.iso文件里面的一个rootfs），因此其根目录/下的东西（包括/home）根本不会持久化，只有/mnt/sda1这个目录下的才能持久化。如果你放在/home目录下，只要VM一重启，就会丢失的，/mnt/sda1则不会，实际上就是那个~/.boot2docker-vm.vmdk文件挂载到了/mnt/sda1目录下

**端口映射**

和磁盘映射一样，你有时候会需要将Container的端口映射到host主机上，同样蛋疼的是，由于多了一层VM，端口映射也显得比较麻烦。首先你需要设置VirtualBox的端口映射，然后再将Container的端口映射到你的VM里面：

[](http://tech.uc.cn/wp-content/uploads/2014/04/docker-port-map.png)

具体是这么做的，通过2条命令：



|  |  |
| --- | --- |
| 1  2  3 | boot2docker ssh -L 8000:localhost:8000  docker run -i -t -p 8000:8000 |

也就是说在docker run的时候通过-p选项指定要映射的端口到VM，而boot2docker ssh命令则是将VM的8000端口映射到了host(Mac)的8000端口，这样你就可以通过Mac的localhost:8000访问Container的8000端口了。 其实，有另一种解决方案就是你不用映射到host（Mac），而是直接登录到VM里面进行访问就好了，boot2docker ssh就可以登录到VM，这样就类似于你的host是ubuntu，但这种解决方案的问题是这个ubuntu太弱了（TinyCoreLinux），如果你在这个ubuntu里面开发代码，或者是运行浏览器，是非常蛋疼的事情，关键还是这个ubuntu是每次重启都会复原的！所以我建议还是做多一层映射好了。 最后，实际上在VM里面，你是可以直接访问所有的Container的端口的，因为VM到Container的网络都是桥接的。

**其他的一些坑**

在使用的过程中，还遇到一些不少的坑：

1. /etc/hosts文件无法修改，这样你就不能自己做域名解析
2. VM的系统时间是UTC +0000的，而且貌似无法修改
3. Container的IP无法指定为静态IP，因此每次重启Container时，IP可能会变化

第1个问题的解决方案是通过安装dnsmasq软件来做域名解析：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | # 首先，在你的Container里面安装dnsmasq软件:    apt-get install dnsmasq    # 将以下文本添加到 /etc/dnsmasq.conf文件的最后：    listen-address=127.0.0.1 resolv-file=/etc/resolv.dnsmasq.conf conf-dir=/etc/dnsmasq.d user=root    # 接着在/etc/dnsmasq.d/目录下新建一个文件，随意起个名字    vi /etc/dnsmqsq.d/dns.conf    # 指定你要映射的域名，例如google.com，则将下面贴进dns.conf文件    address="/google.com/172.17.0.4"    # 最后退出容器，重启启动容器时，通过-dns选项指定域名服务器    docker run -i -t -dns 127.0.0.1 -dns 8.8.8.8 dev:base /bin/bash    # 一定要注意上面添加google的域名服务器8.8.8.8，否则你访问不了外网    # 进去Container后，启动dnsmasq，这样你就能够ping google.com了    /etc/init.d/dnsmasq start |

第2个问题的解决方案就稍微麻烦些，起码我没有找到更好的解决方案，我是将boot2docker.iso文件重新制作一次来解决这个问题的：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | # 首先你需要将boot2docker.iso文件mount到一个目录下    hdiutil mount ~/.boot2docker/boot2docker.iso    # 系统会mount到/Volumes/boot2docker目录下，然后你最好将这下面的东西copy出来到一个另外的目录，这样我们好制作一张新的ISO    cp -r /Volumes/boot2docker/\* ~/tmp/    # 接着我们修改以下文件    vi ~/tmp/boot/isolinux/isolinux.cfg    # 将其中的以下这行修改：    append loglevel=3 user=docker console=ttyS0 console=tty0 nomodeset norestore base    # 修改为：（其实就是加了tz的启动参数），然后保存    append tz=CST-8 loglevel=3 user=docker console=ttyS0 console=tty0 nomodeset norestore base    # 接着你必须在ubuntu环境下重新制作ISO文件，你可以利用docker跑一个ubuntu，哈哈，假设你将boot2docker目录copy到了ubuntu的/src/目录下，那么接着这么做    # 安装xorriso命令    apt-get install xorriso    # 构建ISO映射    xorriso -as mkisofs -J -R -V boot2docker -no-emul-boot -boot-load-size 4 -boot-info-table -b boot/isolinux/isolinux.bin -c boot/isolinux/boot.cat -o /boot2docker.iso /src/    # 这样就生成了/boot2docker.iso文件，最后你就可以替换到VM的启动ISO文件了，然后重启VM了    boot2docker restart    # 最后你必须设置你的VM为正确的时间，使用date -s 命令，最后用date命令查看，你就能看到CST时区的正确时间了    Sun Mar 30 00:27:13 CST 2014    # 对于你启动的container，你都必须重新设置TZ环境变量，否则即使VM是CST-8，你的container还是UCT +00:00的时间    export TZ='CST-8' |

第三个问题暂时无法解决（可能需要编辑底层的LXC配置文件）。

**docker的限制以及后续的一些想法**

docker其实还是有一些限制的：

* 要求你的环境是Linux的，而且内核必须很新（>= 2.6.27 (29)），这其实是LXC本身的限制，和docker无关
* docker的Container目前host是不能修改的，当然有解决方案（dnsmasq）
* docker的Container也暂时无法指定静态IP

用docker作为开发环境甚至是生产环境其实还有很多地方值得尝试：

* 在团队内部构建本地的仓库，标准化所有的开发环境，使得团队的新人可以快速上手
* 在生产环境部署docker，这其实是PAAS的虚拟化和自动化的一种方式，利用LXC和Docker能够更便捷地实施PAAS
* 尝试用docker做分布式集群模拟和测试，成本会更加低廉，更加容器维护

**参考文章**

1. [Linux虚拟化技术](https://www.ibm.com/developerworks/cn/linux/theme/virtualization/)
2. [利用Linux容器实现可移植的应用部署](http://www.infoq.com/cn/articles/docker-containers)
3. [如何修改host](https://github.com/dotcloud/docker/issues/1951)
4. [Building a Development With Docker](http://tersesystems.com/2013/11/20/building-a-development-environment-with-docker/)
5. [boot2docker的build](https://github.com/boot2docker/boot2docker/blob/master/rootfs/make_iso.sh)
6. [PAAS Under the Hood](https://www.dotcloud.com/static/paas_under_the_hood_printversion.pdf)

**您可能感兴趣的文章**

* [把Sheepdog装进OpenStack](http://tech.uc.cn/?p=2697)
* [自动增量升级方案的设计及实现](http://tech.uc.cn/?p=1621)
* [FastDFS使用经验分享](http://tech.uc.cn/?p=2579)