7.创建自己的镜像

1.创建一个简单的应用程序

2.创建镜像的Dockerfile

3.构建镜像

镜像是如何构建的

什么是镜像层

4.运行镜像

1.创建一个简单的应用程序

在安装好Docker后，现在让我们来创建一个简单的应用程序。

我们先创建一个简单的Node.js Web应用，然后将它打包到镜像中。该应用可以接受HTTP请求并返回主机名。虽然应用进程和其他进程一样都是运行在宿主机上，但是在容器中运行的应用获取到的是它所在容器的主机名，而不是宿主机名。

这个Node.js应用只有app.js一个文件。

在~目录下执行如下命令：

```
mkdir test01
cd test01
vim app.js
```

复制如下代码到app.js文件中：

```
const http = require('http');
const os = require('os');
console.log("开始运行...");
var handler = function(request, response) {
 console.log("收到来自 " + request.connection.remoteAddress + "的消息");
 response.writeHead(200);
 response.end("已发送消息至：  " + os.hostname() + "\n");
};
var www = http.createServer(handler);
www.listen(8080);
```

```
[root@dhr-demo test01]# vim app.js

const http = require('http');
const os = require('os');
console.log("开始运行...");
var handler = function(request, response) {
 console.log("收到来自 " + request.connection.remoteAddress + "的消息");
 response.writeHead(200);
 response.end("已发送消息至:  " + os.hostname() + "\n");
};
var www = http.createServer(handler);
www.listen(8080);
~
```

该程序会启动一个监听8080端口的HTTP服务。当收到来自外界的消息后，请求处理器会记录并输出请求方的IP地址，然后给每一个请求返回状态码200以及一段包含主机名的文本。
到这一步后，我们似乎可以下载并安装Node.js环境来直接测试这个应用程序了，但这就背离了本文的初衷了。因为我们要学习的是使用Docker来将这个应用打包到容器镜像中，然后让它不需要下载和安装就可以在任何主机上执行，当然如果想运行这个镜像，主机还是需要准备好最基本的Docker环境。

2.创建镜像的Dockerfile

要想将应用程序打包的镜像里，首先得创建一个叫做Dockerfile的文件。
在~/test01目录下执行如下命令：
touch Dockerfile
这个文件就是一个指令清单，指示Docker在构建镜像时需要执行的命令。Dockerfile需要和app.js文件在同一个目录下，而且应该包含如下命令：
FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]

```
[root@dhr-demo test01]# vim Dockerfile

FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]
~
```

FROM关键字所在的行定义了镜像构建过程所使用的基础镜像。此处我们使用node镜像的tag7版本。
第二行的作用是将app.js文件从本地目录添加到镜像的根目录下，并保持相同的文件名。最后一行的作用是指定运行镜像的时候应该执行的命令。本例中，这个命令是*node app.js*。
或许你会好奇为什么要选择这个node镜像作为基础镜像。因为这个应用是一个Node.js应用，因此需要一个包含node二进制可执行文件的镜像来运行该应用。你也可以使用任何包含这个二进制文件的镜像，甚至可以使用Linux发行版的基础镜像，如fedora或ubuntu，然后在构建镜像之前指定安装Node.js的命令，从而确保运行容器的时候Node.js会被安装到容器中。

但是，因为node镜像是专门用来运行Node.js应用的，而且包含运行应用所需的一切，因此我们使用它作为基础镜像。

3.构建镜像

在Dockerfile和app.js都准备好了之后，我们就可以开始构建镜像了。在Dockerfile所在目录下，通过如下命令构建镜像：

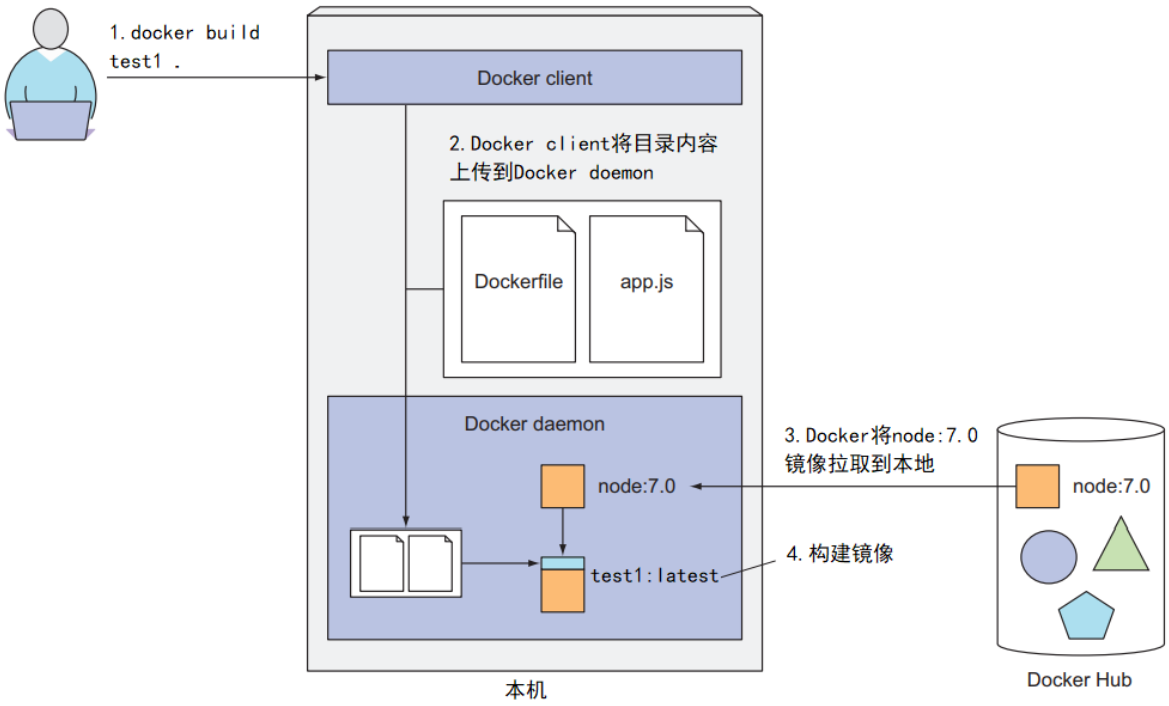
docker build –t test1 .

注意末尾还有一个点号。

```
[root@dhr-demo test01]# docker build -t test1 .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM node:7
 ---> d9aed20b68a4
Step 2/3 : ADD app.js /app.js
 ---> b7be547273cc
Step 3/3 : ENTRYPOINT ["node", "app.js"]
 ---> Running in 69286812ff2e
Removing intermediate container 69286812ff2e
 ---> 773641d7cb99
Successfully built 773641d7cb99
Successfully tagged test1:latest
```

这个命令会告诉Docker基于当前目录下的Dockerfile文件来构建一个叫做test1的镜像。Docker会查找目录中的Dockerfile文件，然后基于文件中的指令构建镜像。

下图展示了镜像的构建过程：



镜像是如何构建的

从上面可以看出，镜像的构建过程不是由Docker client执行的。而是Docker client将整个目录的的文件上传到Docker daemon（守护进程）并由它进行构建。Docker client和daemon不需要在同一台机器上。

如果你在非Linux的操作系统上使用Docker，客户端可以安装在宿主机操作系统，但是daemon需要运行在VM中。因为构建目录下的所有文件都会被上传到daemon中，如果包含了很多大文件而且daemon不在本地运行的话，上传过程就会比较耗时。

需要注意的是，不要在构建目录下存放任何不需要的文件，因为这样会减慢镜像的构建速度，特别是当Docker daemon进程位于远程机器上的时候。

在构建的过程中，Docker会从公共镜像仓库（Docker Hub）拉取基础镜像（node:7），除非这个镜像已经被拉取且存到本机上了。

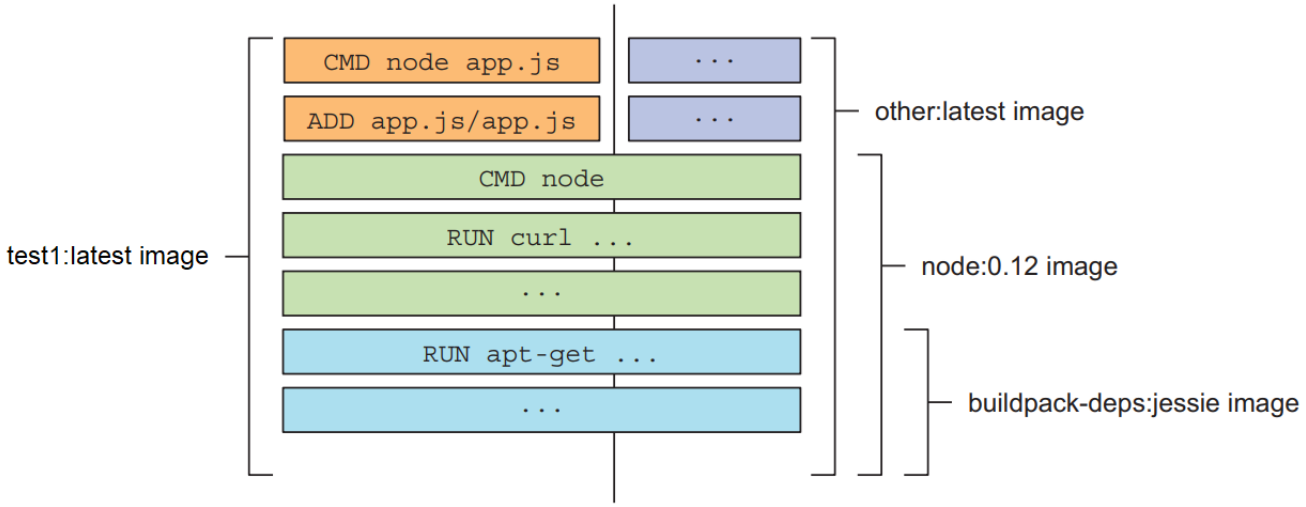
什么是镜像层

一个镜像并不是一个大的二进制块，而是由很多层组成的。不同的镜像之间可能会共享某些层，这使得存储和传输镜像变得更加高效。

例如，如果你基于相同的基础镜像（比如本例中的node:7镜像）创建了几个镜像，构成这个基础镜像的所有层都只会被存储一次。而且，当拉取一个镜像的时候，Docker会单独的下载每一层。某些层可能已经存储在你的机器上了，因此Docker只会下载那些还未下载过的层。

你可能会认为每个Dockerfile只会创建一个新的层，但是事实不是这样。当构建镜像的时候，Dockerfile中的每一条单独的命令都会创建一个新的层。在镜像构建的过程中，在拉取了基础镜像的所有层之后，Docker会在这些层之上创建一个新的层并将app.js文件添加到这个层里，然后会创建另外一个新的层来指定镜像被运行的时候应该执行的命令。

最后一层会被标记为test1:latest。如下图所示：



other:latest的镜像是与我们自己构建的镜像test1:latest共享Node.js的所有层。

当完成镜像构建后，一个新的镜像就被存储到本地了。可以通过docker images命令列出所有本地的镜像：

```
[root@dhr-demo test01]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test1               latest              773641d7cb99        3 minutes ago       660MB
hello-world         latest              bf756fb1ae65        10 months ago       13.3kB
node                7                   d9aed20b68a4        3 years ago         660MB
```

4.运行镜像

现在我们就可以通过如下命令运行我们自己创建的镜像了：

docker run --name test1-container -p 8080:8080 -d test1

该命令会告诉Docker通过test1镜像启动一个叫做test1-container的容器。-d标志表示容器与终端脱离，也是就容器会在后台运行。本机的8080端口与容器的8080端口映射。

```
[root@dhr-demo test01]# docker run --name test1-container -p 8080:8080 -d test1
75e75bb75e166689e1ca533fd068d0543f8932a546acdf666109835585 4d4092
[root@dhr-demo test01]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
75e75bb75e16        test1               "node app.js"       5 seconds ago       Up 3 seconds        0.0.0.0:8080->8080/tcp   test1-container
[root@dhr-demo test01]# curl http://192.168.16.103:8080
已发送消息至： 75e75bb75e16
[root@dhr-demo test01]#
```

可以看到返回的16进制数字就是容器的ID。