1. 创建和启动一个容器

1. docker run --detach --name web nginx:latest

Docker 将从 docker hub 上的 Nginx 仓库下载，安装 Nginx:latest 镜像，然后运行该软件. --detach(或者使用 -d 缩写形式)选项，在后台启用该程序

2. Docker run -d --name mailer dockerinaction/ch2_mailer

2. 运行交互容器

1. Docker run --interactive --tty \

--link web:web \

--name web_test

Busybox:latest /bin/sh

该命令使用 run 命令的两个标志: --interactive(或-i)和--tty(或-t).

(1) --interactive 选项告诉 docker 保持标准输入流(stdin，标准输入)对容器开放， 即使容器没有终端连接。

(2) --tty 选项告诉 docker 为容器分配一个虚拟终端， 这将允许你发信号给容器

2.验证是否运行： wget -O - http://web:80/

3. docker run -it \

--name agent \

--link web:insideweb \

--line mailer:insidemailer \

Dockerinaction/ch2_agent

使用缩略标志在交互式容器中启动监控器

3.列举，停止，重新启动和查看容器输出

1. Docker ps

会显示每个运行的容器中的一下信息：

- 该容器 id
- 使用的镜像
- 容器中执行的命令
- 容器运行的市场
- 容器暴露的网络端口
- 容器名

2.

docker logs web 查看 web 服务器运行是否正常

Docker logs mailer 查看邮件服务器运行是否正常

Docker logs agent 查看监控器运行是否正常

3.

Docker stop web 暂停 web 服务

4.

```
SQL_CID = $(docker create -e MYSQL_ROOT_PASSWORD=ch2demo mysql:5)
docker start $SQL_CID
MAILER_CID = $(docker create dockerinaction/ch2_mailer)
docker start $MAILER_CID
WP_CID = $(docker create --link $SQL_CID:mysql -p 80 \
    -v /run/lock/apache2/ -v /run/apache2/ \
    --read-only wordpress:4)
```

```
docker start $WP_CID
AGENT_CID = $(docker create --link $WP_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_mailer)
docker start $AGENT_CID
```

5. 查看容器中那些进程在运行着

```
docker run -d -p 80:80 --name lamp-test tutum/lamp
docker top lamp-test
```

6. 在容器内杀死进程

1) 查看容器内的进程

```
Docker exec lamp-test ps
```

2) 杀死进程

```
Docker exec lamp-test kill <PID>
```

1. 链接别名

① 创建有效的链接目标

```
docker run -d --name mydb --expose 3306 \
    alpine:latest nc -l 0.0.0.0:3306
```

② 测试布简历链接的情况

```
docker run -it --rm \
    dockerinaction/ch5_ff echo This “shouldn’t” work.
```

③ 测试不正确的链接别名情况

```
docker run -it --rm \
    --link mydb:wrongalias \
    dockerinaction/ch5_ff echo Wrong
```

④ 测试错误别名

```
docker run -it --rm \
    --link mydb:database \
    dockerinaction/ch5_ff echo It worked
```

⑤ 停止并删除链接目标容器

```
Docker stop mydb && docker rm mydb
```

5.7.3

创建一条链接会在新容器中添加链接信息。这些链接信息一方面存储在环境变量中，另一方面通过在 dns 覆盖系统中添加主机名的映射来将链接信息注入新容器中。

6.2.1 跨容器的进程间通信

在容器间共享内存比主机上共享内存来的更加安全

启动生产者进程

```
docker run -d -u nobody --name ch6_ipc_producer \
    dockerinaction/ch6_ipc -producer
```

启动消费者进程

```
docker run -d -u nobody --name ch6_ipc_consumer \
    -ipc container:ch6_ipc_producer \
    dockerinaction/ch6_ipc -consumer
```

6.2.2 开放内存容器

想要和主机运行在同一个命名空间中，可以使用开放内存容器

启动生产者进程

```
docker run -d --name ch6_ipc_producer \
    --ipc host \
    dockerinaction/ch6_ipc -producer
```

启动消费者进程

```
docker run -d --name ch6_ipc-consumer \
    --ipc host \
    dockerination/ch6_ipc -consumer
```

6.3.3

1) 主机上的特定文件被特定用户所使用

① 在主机上创建一个新文件

```
echo “e=mc^2” > garbage
```

② 使用这个文件只能被文件的所有者读取

```
chmod 600 garbage
```

③ 将文件的所有者改为 root（假设你拥有 sudo 权限）

```
sudo chown root:root garbage
```

④ 尝试用 nobody 用户读取文件

```
docker run --rm -v “$(pwd)”/garbage:/test/garbage \
    -u nobody \
    ubuntu:latest cat /test/garbage
```

⑤ 尝试用容器 root 用户读取文件

```
docker run --rm -v “$(pwd)”/garbage:/test/garbage \
    -u root \
    ubuntu:latest cat /test/garbage
```

7.1.4 创建新镜像

① 从一个合适的基础镜像创建一个容器，并设置入口点

```
docker run -it --name image-dev ubuntu:latest /bin/bash
```

② 在容器中安装 Git

```
apt-get -y install git
```

③ 创建并标记一个名为 ubuntu-git 的新镜像

```
docker commit -a “@dockerinaction” -m “added git” image-dev ubuntu-git
```

测试：

```
docker run --name cmd-git ubuntu-git version
```

7.3 导入和导出扁平文件系统

1.创建一个新容器并且使用 export 子命令来获得新容器文件系统的扁平复制

导出文件系统内容

```
docker run --name export-test dockerinaction/ch7_packed:latest ./echo For Export
docker export --output contents.tar export-test
docker rm export-test
```

显示归档内容

```
tar -tf contents.tar
```

2.

创建内容

```
package main
import “fmt”
func main() {
    Fmt.Println(“hello, world!”)
}
```

拉取一个包含有 go 编译器的镜像，编译并且静态链接这个代码

```
docker run --rm -v “$(pwd)”:/usr/src/hello -w /usr/src/hello golang:1.3 go build -v
```

将这个程序放到压缩文件中

```
tar -cf static_hello.tar hello
```

可以使用 docker import 命令将它导入到镜像中（通过 Unix 管道将 tar 文件重定向）

```
docker import -c “ENTRYPOINT [\”hello\”]” - \
dockerinaction/ch7_static <static_hello.tar
```

运行并且查看它

```
docker run dockerinaction/ch7_static
docker history dockerinaction/ch7_static
```

8.1 使用 Dockerfile 打包 Git

1. 先创建了带有四个指令的 Dockerfile

```
# an example Dockerfile for installing Git on Ubuntu
FROM ubuntu:latest
MAINTAINER "dockerinaction@allingeek.com"
RUN apt-get update
RUN apt-get install -y git
ENTRYPOINT ["git"]
```

2. 在包含 Dockerfile 文件的目录中使用 docker build 命令，从 Dockerfile 文件创建一个新镜像

```
docker build --tag ubuntu-git:auto .
```

3. 查看镜像及运行镜像

```
docker images
docker run --rm ubuntu-git:auto version
```

8.2 构建一个基础镜像和另外两个大包有不同版本邮件的镜像。

第一步：构建一个基础镜像

1. 创建一个名为 .dockerignore 的文件， 然后将以下内容复制到文件中：

```
.dockerignore
mailer-base.df
mailer-logging.df
mailer-live.df
```

2. 创建一个名为 mailer-base 的文件， 并将下面的内容复制到文件中：

```
FROM debian:wheezy
MAINTAINER Jeff Nickoloff “dia@allingeek.com”
RUN grouopadd -r -g 2200 example && \
    useradd -rM -g exmaple -u 2200 example
ENV APPROOT=“/app” \
```

```
    APP=“mailer.sh”
    VERSION=“0.6”
LABEL base.name=“Mailer Archetype” \
    base.version=“${VERSION}”
WORKDIR $APPROOT
ADD . $APPROOT
ENTRYPOINT [“/app/mailer.sh”]
EXPOSE 33333
#不要在基础镜像中设置默认用户，否则接下来的实现将不能够更新镜像
# USER example.example
```

3. 开始构建镜像

```
docker build -t dockerinaction/mailer-base:0.6 -f mailer-base.df .
```

第二步：在 33333 端口启动一个邮件程序后台进程

1. 在 mailer-logging.df 文件中写入如下：（log-impl 需在同一路径下）

```
FROM dockerinaction/mailer-base:0.6
COPY log-impl ${APPROOT}
RUN chmod a+x ${APPROOT}/${APP} && \
    chown example:example /var/log
USER example:example
VOLUME [“/var/log”]
CMD [“/var/log/mailer.log”]
```

2. 创建一个名为 mailer.sh 的文件，并且将厦门的脚本复制到文件中

```
#! /bin/sh
printf “Logging Mailer has started. \n”
while true
do
    MESSAGE=${NC -L -P 33333}
    printf “[Message]: %s\n” “$MESSAGE” > $1
    sleep 1
done
```

3. 使用下面的命令从包含 mailer-logging.df 文件的目录中构建 mailer-logging 镜像：

```
docker build -t dockerinaction/mailer-logging -f mailer-logging.df .
```

4. 启动一个命名容器

```
docker run -d —name logging-mailer dockerinaction/mailer-logging
```

第三步：使用 Amazon Web Services 提供的 Email Service

1. 在 mailer-live.df 文件中写入如下：

```
FROM dockerinaction/mailer-base:0.6
ADD live-impl ${APPROOT}
RUN apt-get update && \
    apt-get install -y curl python && \
```

```
          curl      “https://bootstrap.pypa.io/get-pip.py”      -o
“get-pip.py” && \
          python get-pip.py && \
          pip install awscli && \
          rm get-pip.py && \
          chmod a+x “${APPROOT}/${APP}”
       RUN apt-get install -y netcat
       USER example:example
       CMD                              [“mailer@dockerinaction.com”,
“paget@dockerinaction.com”]
```

2. 在包含有 mailer-live.df 文件的目录下创建一个名为 live-impl 的子目录，并在这个子目录下， 将下面的脚本复制到名为 mailer.sh 的文件中:

```
       #! /bin/sh
       printf "Live Mailer has started. \n"
       while true
         do
             MESSAGE=$(nc -l -p 33333)
             aws ses send-email --from $1 \
                 --destination {\"ToAddresses\":{\"$2\"} \
                 \"Body\":
{\"Text\":{\"Data\":\"$MESSAGE}\"}}}"
             sleep 1
             Done
```

3. 构建新镜像，然后启动一个新容器

```
       docker build -t dockerinaction/mailer-live -f mailer-live.df .
       docker run -d name live-mailer dockerinaction/mailer-live
```

8.3 注入下游镜像在构建时发生的操作

1. 创建一个上游 Dockerfile, 它使用了 onbuild 指令。将这个文件命名为 base.df，然后将下面的指令复制到该文件

```
       FROM busybox:latest
       WORKDIR /app
       RUN touch /app/base-evidence
       ONBUILD RUN ls -al /app
```

2. 创建下游的 Dockerfile。创建 downstream.df，并将以下内容复制到文件中

```
       FROM dockerinaction/ch8_onbuild
       RUN touch downstream-evidence
       RUN ls -al .
```

3. 创建上游镜像

```
       docker build -t dockerinaction/ch8_onbuild -f base.df .
```

4. 创建下游镜像

```
          docker  build  -t  dockerinaction/ch8_onbuild_down  -f
   downstream.df
```

8.5.2 用户权限

1. 将下面的内容复制到名为 UserPermissionDenied.df 的文件中

```
FROM busybox:latest
USER 1000:1000
ENTRYPOINT [“nc”]
CMD [“-l”, “-p”，“80” ，“0.0.0.0”]
```

2. 构建这个 Dockerfile 生产新镜像，并且使用这个镜像创建一个容器

```
docker build -t dockerinaction/ch8_perm_denied -f UserPermissionDenied.df
docker run dockerinaction/ch8_perm_denied
```

8.5.3 SUID 和 SGID 权限

1.

```
FROM ubuntu:latest
# 设置 whoami 程序的 SUID 位
RUN chmod u+s /usr/bin/whoami
# 创建一个 exmaple 用户，并且将它设置为默认用户
RUN adduser —system —no-create-home —disabled-password —disabled-login \
    —shell /bin/sh example
USER example
#设置默认命令，比较容器用户和执行 whoami 程序的有效用户
CMD printf “Container running as: %s\n” $(id -u -n) && \
    printf “Effectively running whoami as: %s\n” $(whoami)
```

2. 创建镜像， 并且创建容器

```
docker build -t dockerinaction/ch8_whoami
docker run dockerinaction/ch8_whoami
```

9.2.1 通过共有仓库发布： 你好！ Docker Hub

1. 创建一个 HelloWorld.df 的 Dockerfile

```
FROM busybox:latest
CMD echo Hello World
```

2. 构建新镜像

```
docker build -t <username>/hello-dockerfile -f HelloWorld.df .
```

3.登录用户信息

```
docker login
```

4.推送 docker 到远程仓库

```
docker push <username>/hello-dockerfile
```

5.查找 dockerinaction

```
docker search <username>/hello-dockerfile
```

9.2.2 创建一个 docker hub 自动化

1. 创建一个名为 hello-docker.df 的文件，并且包含如下：

```
FROM busybox:latest
CMD echo Hello World
```

2.

```
git init
git config –global user.emial “you@example.com”
git config –global user.name “You Name”
git remote add origin https://github.com/fkue469577/hello-docker.git
```

3.

```
git add Dockerfile
git commit -m “first commit”
git push -u origin master
docker search <your username>/hello-docker
```

9.4 镜像的手动发布和分发

1. 从 docker hub 拉去一个已知的镜像用来作为分发

```
docker pull registry:2
```

2. 构建镜像分发基础设置

```
docker run -d --name ftp-transport -p 21:12 dockerinaction/ch9_ftpd
```

3. 导出一个为文件格式的镜像

```
docker save -o ./registry.2.tar registry:2
```

4. 创建一个容器，绑定一个卷到你的本地目录。

```
docker run --rm --link ftp-transport:ftp_server \
    -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e 'cd pub/incoming; put registry.2.tar; exit' ftp_server
```

5.通过罗列 FTP 服务器目录下的内容验证你上传的镜像

```
docker run --rm --link ftp-transport:ftp_server \
    -v "$(pwd)":/data \
    dockerinaction/ch9_ftp_client \
    -e "cd pub/incoming; ls; exit" ftp_server
```

10.2.1 创建一个反向代理

1.创建名为 basic-proxy.conf 的新文件，包含如下的配置

```
upstream docker-registry {
    server registry:5000;
}
server {
    listen 80;
    # Use the localhost name for testing purposes server_name localhost;
    # A real deployment would use the real hostname where it is deployed
    # server_name mytotallyawesomeregistry.com

    client_max_body_size 0;
```

```
        chunked_transfer_encoding on;

        # We're going to forward all traffic bound for the registry
        location /v2/ {
            proxy_pass                      http://docker-registry;
            proxy_set_header  Host          host;
            proxy_set_header  X-Real-IP     $remote_addr;
            proxy_set_header  X-Porwarded-For  $proxy_add_x_forwarded_for;
            proxy_set_header  X-Forwarded-Proto $scheme;
            proxy_read_timeout                  900;
        }
    }
```

2.创建一个名为 basic-proxy.df 文件，复制一下内容

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY ./basic-proxy.conf /etc/nginx/conf.d/default.conf
```

3.构建镜像

```
docker build -t dockerinaction/basic_proxy -f basic-proxy.df .
```

4.启动反向代理

```
docker run -d --name basic_proxy -p 80:80 \
    --link personal_registry:registry \
    dockerinaction/basic_proxy
```

5.通过反向代理运行 curl 命令查询你的 registry(作者中没有 curl 的镜像，所以暂时列出来)

```
docker run --rm -u 1000:1000 --net host \
    dockerinaction/curl \
    -s http://localhost:80/v2/distribution/tags/list
```

10.2.2 在反向代理商配置 HTTPS（TLS）

1.生成一个 4096 比特位的 RSA 密钥对，并且在你当前的工作目录中存储私钥文件和自签名证书

```
docker run --rm -e COMMON_NAME=localhost -e KEY_NAME=localhost \
    -v "$(pwd)":/certs centurylink/openssl
```

2.创建 tls-proxy.conf 的文件，包含如下配置

```
upstream docker-registry {
    server registry:5000;
}
server {
    listren 443 ssl;
    serve_name localhost

    client_max_body_size 0;
    chunked_transfer_encoding on;
```

```
        ssl_certificate /etc/nginx/conf.d/localhost.crt;
        ssl_certificate_key /etc/nginx/conf.d/localhost.key;

        location /v2/ {
            proxy_pass
http://docker-registry;
            proxy_set_header    Host                    host;
            proxy_set_header    X-Real-IP               $remote_addr;
            proxy_set_header                            X-Forwarded-For
$proxy_add_x_forwarded_for;
            proxy_set_header    X-Forwarded-Proto       $scheme;
            proxy_read_timeout                          900;
        }
    }
```

3. 创建 tls-proxy.df 的新文件，并把如下复制进去

```
    FROM nginx:latest
    LABEL source=dockerinaction
    LABEL category=infrastructure
    COPY ["./tls-proxy.conf", \
          "./localhost.crt", \
          "./localhost.key", \
          "/etc/nginx/conf.d/"]
```

4. 构建新镜像

```
    docker build -t dockerinaction/tls_proxy -f tls-proxy.df .
```

5. 使用 curl 命令启动你的反向代理并测试

```
    docker run -d --name tls-proxy -p 443:443 \
        --link personal_registry:registry \
        dockerinaction/tls_proxy
    docker run --rm \
        --net host \
        dockerinaction/curl -ks \
        https://localhost:443/v2/distribution/tags/list
```

10.2.3 添加身份认证层

1.创建 htpasswd.df

```
    FROM debian:jessie
    LABEL source=dockerinaction
    LABEL category=utility
    RUN apt-get update && \
        apt-get install -y apache2-utils
    ENTRYPOINT ["htpasswd"]
```

2.构建新镜像

```
    docker build -t htpasswd -f htpasswd.df .
```

3.为密码文件创建一个新条目

```
docker run -it --rm htpasswd -nB <USERNAME>
```

4.创建 tls-authproxy.conf 文件，并包含如下：

```
#filename: tls-auth-proxy.conf
upstream docker-registry {
    server registry:5000;
}
server {
    listen 443 ssl;
    server_name localhost

    client_max_body_size 0;
    chunked_transfer_encoding on;

    #SSL
    ssl_certificate /etc/nginx/conf.d/localhost.crt;
    ssl_certificate_key /etc/nginx/conf.d/localhost.key;

    location /v2/ {
        auth_basic "registry.localhost";
        auth_basic_user_file /etc/nginx/conf.d/registry.password;

        proxy_pass                          http://docker-registry;
        proxy_set_header   Host             $http_host;
        proxy_set_header   X-Real-IP        $remote_addr;
        proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
        proxy_read_timeout                  900;
    }
}
```

5.创建名为 tls-auth-proxy.df 的 dockerfile 文件

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
COPY ["./tls-auth-proxy.conf", \
      "./localhost.crt", \
      "./localhost.key", \
      "./registry.password", \
      "/etc/nginx/confi.d/"]
```

6.新建 tls-auth-registry.yml，并填入下面代码

```
version: 0.1
log:
    level: debug
    fields:
```

```
            service: registry
            environment:development
    storage:
        filesystem:
            rootdirectory: /var/lib/registry
        cache:
            layerinfo: inmemory
        maintenance:
            uploadpurging:
                enabled: false
    http:
        addr: :5000
        secret: asecretforlocaldevelopment
        tls:
            certificate: /localhost.crt
            key: /localhost.key
        debug:
            addr: localhost:5001
    auth:
        htpasswd:
            realm: registry.localhost
            path: /registry.password
```

7.创建一个便捷的 Dockerfile（tle-auth-registry.df）来打包处理

```
# Filename: tls-auth-registry.df
FROM registry:2
LABEL source=dockerinaction
LABEL category=infrastructure
# Set the default argument to specify the config file to use

#Setting it early will enable layer caching if the ths-auth-registry.yml changes.
CMD ["/tls-auth-registry.yml"]
COYP ["./tls-auth-registry.yml", \
        "./localhost.crt", \
        "./localhost.key", \
        "./registry.password", \
        "/"]
```

8. 构建和启动新的安全的 registry

```
docker build -t dockerinaction/secure_registry -f tls-auth-registry.df .
docker run -d --name secure_registry \
    -p 5443:5000 --restart=always \
    dockerinaction/secure_registry
```

10.2.4 客户端兼容性

需要三个步骤：

- 创建一个 Nginx 配置文件（dual-client-proxy.conf）
- 创建一个简洁的 Dockerfile（dual-client-proxy.df）
- 构建一个新的镜像

1.在名为 dual-client-proxy.conf 的文件中找到新的反向代理配置

```
upstream docker-registry-v2 {
    server registry2:5000;
}
upstream docker-registry-v1 {
    server registry1:5000;
}
server {
    listen 80;
    server_name localhost;

    client_max_body_size 0;
    chunked_transfer_encoding on;

    location /v1/ {
        proxy_pass
http://docker-registry-v1;
        proxy_set_header    Host                    host;
        proxy_set_header    X-Real-IP
$remote_addr;
        proxy_set_header    X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header    X-Forwarded-Proto   $scheme;
        proxy_read_timeout                          900;
    }
    location /v1/ {
        proxy_pass
http://docker-registry-v2;
        proxy_set_header    Host                    host;
        proxy_set_header    X-Real-IP
$remote_addr;
        proxy_set_header    X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header    X-Forwarded-Proto   $scheme;
        proxy_read_timeout                          900;
    }
}
```

2. 创建一个 dual-client-proxydf 文件

```
FROM nginx:latest
LABEL source=dockerinaction
LABEL category=infrastructure
```

```
COPY ./dual-client-proxy.conf /etc/nginx/conf.d/default.conf
```

3. 创建镜像

```
docker build -t dual_client_proxy -f dual-client-proxy.df .
```

4.拉取 0.9.1 版本的 registry，并在容器中启动

```
docker run -d —name registry v1 registry:0.9.1
```

5.创建反向代理并连接这两个 Registry

```
docker run -d —name dual_client_proxy \
    -p 80:80 \
    —link personal_registry:registry2 \
    —link registry_v1:registry1 \
    dual_client_proxy
```

10.3.1 微软 Azure 托管远程存储

1.创建 zaure-config.yml 文件

```
# Filename: azure-config.yml
version： 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    azure:
        accountname: <your account name>
        accoutkey: <your base64 encoded account key>
        container: <your container>
        realm: core.windows.net
    cache:
        layerinfo: inmemory
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
```

2. 把新的配置打包到原生 Registry 镜像的上一层，命名为 azure-config.df

```
#Filename: azure-config.df
FROM registry:2
LABEL source=dockerinaction
LABEL category=infrastructure
# Set the default argument to specify the config file to use
#  Setting  it  early  will  enable  layer  caching  if  the
```

```
azure-config.yml changes.
       CMD ["/azure-config.yml"]
       COPY ["./azure-config.yml", "/azure-config.yml"]
```

3. 构建镜像

```
       docker    build    -t    dockerinaction/azure-registry    -f
azure-config.df .
```

10.3.1 AWS S3 托管远程存储

1.创建 s3-config.yml 文件

```
       # Filename: s3-config.yml
       version： 0.1
       log:
          level: debug
          fields:
             service: registry
             environment: development
       storage:
          s3:
             accountname: <your account name>
             accoutkey: <your base64 encoded account key>
             container: <your container>
             bucket: <your bucketname>
             encrypt: true
             secure: true
             v4auth: true
             chunksize: 5242880
             rootdirectory: /s3/object/name/prefix
          maintenance:
             uploadpurging:
                enabled: false
       http:
          addr: :5000
          secret: asecretforlocaldevelopment
          debug:
             addr: localhost:5001
```

2. 把新的配置打包到原生 Registry 镜像的上一层，命名为 s3-config.df

```
       #Filename: s3-config.df
       FROM registry:2
       LABEL source=dockerinaction
       LABEL category=infrastructure
       # Set the default argument to specify the config file to use
       # Setting it early will enable layer caching if the s3-config.yml
changes.
       CMD ["/s3-config.yml"]
```

```
COPY ["./s3-config.yml", "/s3-config.yml"]
```

3. 构建镜像

```
docker build -t dockerinaction/s3-registry -f s3-config.df .
```

10.3.3 RADOS(Ceph)的内部远程存储

1.创建 rados-config.yml 文件

```
# Filename: rados-config.yml
version： 0.1
log:
   level: debug
   fields:
      service: registry
      environment: development
storage:
   ceph:
      layerinfo: inmemory
   rados:
      poolname: radospool
      username: radosuser
      chunksize: 4194304
   maintenance:
      uploadpurging:
         enabled: false
http:
   addr: :5000
   secret: asecretforlocaldevelopment
   debug:
      addr: localhost:5001
```

2. 把新的配置打包到原生 Registry 镜像的上一层，命名为 s3-config.df

```
#Filename: rados-config.df
FROM registry:2
LABEL source=dockerinaction
LABEL category=infrastructure
# Set the default argument to specify the config file to use
# Setting it early will enable layer caching if the rados-config.yml changes.
CMD ["/rados-config.yml"]
COPY ["./rados-config.yml", "/rados-config.yml"]
```

3. 构建镜像

```
docker build -t dockerinaction/rados-registry -f rados-config.df .
```

10.4.1 与元数据缓存集成

1.创建 redis-config.yml 文件

```
# Filename: redis-config.yml
version： 0.1
log:
   level: debug
   fields:
      service: registry
      environment: development
storage:
   cache:
      blobdescriptor: redis
   s3:
      accountname: <your account name>
      accoutkey: <your base64 encoded account key>
      container: <your container>
      bucket: <your bucketname>
      encrypt: true
      secure: true
      v4auth: true
      chunksize: 5242880
      rootdirectory: /s3/object/name/prefix
   maintenance:
      uploadpurging:
         enabled: false
http:
   addr: :5000
   secret: asecretforlocaldevelopment
   debug:
      addr: localhost:5001
redis:
   addr: redis-host:6379
   password: asecret
   dialtimeout: 10ms
   readtimeout: 10ms
   writetimeout: 10ms
   pool:
      maxidle: 16
      maxactive: 64
      idletimeout: 300s
```

2. 构建一个 Registry 并链接到一个 Redis 容器

```
docker run -d --name redis redis
docker build -t dockerinaction/redis-registry -f redis-config.df .
docker run -d --name redis-registry
   --link redis:redis-host -p 5001:5000
```

```
    dockerinaction/redis-registry
```

10.4.2 使用存储中间件简化 BLOB 传输

1.创建 scalabel-config.yml 文件

```
# Filename: scalabel-config.yml
version： 0.1
log:
    level: debug
    fields:
        service: registry
        environment: development
storage:
    cache:
        blobdescriptor: redis
    s3:
        accountname: <your account name>
        accoutkey: <your base64 encoded account key>
        container: <your container>
        bucket: <your bucketname>
        encrypt: true
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /s3/object/name/prefix
    maintenance:
        uploadpurging:
            enabled: false
http:
    addr: :5000
    secret: asecretforlocaldevelopment
    debug:
        addr: localhost:5001
redis:
    addr: redis-host:6379
    password: asecret
    dialtimeout: 10ms
    readtimeout: 10ms
    writetimeout: 10ms
    pool:
        maxidle: 16
        maxactive: 64
        idletimeout: 300s
middleware:
    storage:
```

```
            name: cloudfront
            options:
               baseurl: <https://my.cloudfronted.domain.com/>
               rpivatekey: </path/to/pem>
               keypairid: <cloudfrontkeypairid>
               duration: 3000
```

11.1.1 用一个简单的开发环境入门

1.创建一个名为 wp-example 的新目录,并复制以下的 docker-compose.yml 文件到这个目录

```
# Filename: docker-compose.yml
wordpress:
   image: wordpress:4.2.2
   links:
      - db: mysql
   ports:
      - 8080: 80
db:
   image: mariadb
   environment:
      MYSQL_ROOT_PASSWORD: example
```

2. 来到你创建 docker-compose.yml 文件的目录并用以下的命令启动

```
docker-compose up
```

11.1.2 一个复杂的架构： 分布式系统和 Elasticsearch 的集成

1. 创建 ch11_notifications/registry/Dockerfile 文件，关联文件在 https://github.com/dockerinaction/ch11_notifications 下载

```
FROM registry:2
CMD ["serve", "/hooks-config.yml"]
COPY ["./hooks-config.yml","/hooks-config.yml"]
```

2. 创建 ch11_notifications/pump/Dockerfile 文件

```
FROM node:latest
COPY ./endpoint.js /usr/src/app/endpoint.js
WORKDIR /usr/src/app
EXPOSE 8000
ENTRYPOINT ["node"]
CMD ["endpoint.js"]
```

3. 创建 ch11_notifications/calaca/Dockerfile 文件，

```
FROM node:0.12
MAINTAINER Jeff Nickoloff <jeff@allingeek.com>
RUN adduser --system --group --disabled-password -shell /bin/bash example
COPY ./service /usr/src/app
COPY ./Calaca /usr/src/app/public
```

```
WORKDIR /usr/src/app
RUN npm install

CMD [ "npm", "start" ]
```

4. 创建 ch11_notifications/docker-compose.yml 文件

```
# A Docker Distribution based registry. The service listens on port 5000.
# This registry has been specialized to push notifications to
# "webhookmonitor:3000" which in this environment is filled by the "pump"
# service.
registry:
  build: ./registry
  ports:
    - "5555:5000"
  links:
    - pump:webhookmonitor

# This is a small NodeJS application that listens on port 3000 and pumps
# received JSON messages to an Elasticsearch node. The application itself
# keeps no state.
pump:
  build: ./pump
  expose:
    - "8000"
  links:
    - elasticsearch:esnode

# The elasticsearch image declares a volume at /usr/share/elasticsearch/data
# for that reason we need not declare a volume here unless we want to
# bind-mount that volume to a specific location on the disk.
# Doing so may be useful for integration with volume management tools
elasticsearch:
  image: elasticsearch:1.5
  ports:
    - "9200:9200"
  command: "-Des.http.cors.enabled=true"

# Calaca is stateless and client side only.
```

```
    # NPM is used for a simple web server.
    calaca:
      build: ./calaca
      ports:
        - "3000:3000"
```

4. docker-compose up -d

12.1.1 构建和管理 docker machine

1.利用 docker machine 创建一个 docker 主键

```
docker-machine create --driver virtualbox host1
docker-machine create --driver virtualbox host2
docker-machine create --driver virtualbox host3
```

2.利用 ls 子命令来获得被管理的机器列表

```
docker-machine ls
```

3.使用 inspect 获取特定的及其

```
docker-machine inspect host1
docker-machine inspect --format "{{.Driver.IPAddress}}" host1
```

4.被管理的及其都可以用 upgrade 子命令升级

```
docker-machine upgrade host3
```

5.想要在一台名为 host1 的机器上创建一个文件

```
docker-machine ssh host1
touch dog.file
exit
```

6.写入一只狗的名字到刚刚创建的文件

```
docker-machine ssh host1 "echo spot > dog.file"
```

7.将你刚才创建的文件从 host1 复制到 host2,然后使用 ssh 子命令来查看 host2

```
docker-machine scp host1:dog.file host2:dog.file
docker-machine ssh host2 "cat dog.file"
```

8.容器启动、停止（或杀死）和删除机器的命令

```
docker-machine stop host2
docker-machine kill host3
docker-machine start host2
docker-machine rm host1 host2 host3
```