# 今日大纲

1. 了解Docker
2. 安装Docker
3. 学习Docker的三大组件，镜像、容器、仓库
4. 学习Docker的数据管理
5. 学习如何构建镜像
6. 使用Docker部署微服务

# 了解docker

## 思考

我们之前是如何将项目发布到Linux服务器的？

大致步骤如下：

1. 安装jdk
2. 安装tomcat
3. 将项目war包上传到tomcat的webapps下
4. 修改配置文件
5. 启动tomcat

这样看似没问题，其实我们想想，发一台机器还好，这几步就完成了，如果我们要将这个项目发布到N多台机器，那么我们是否需要在每个机器上都进行相同的步骤，并且要进行环境兼容性的测试。

再来看一个例子，我们现在想部署使用一个成熟的产品，这个产品是用go语言开发的，我该如何部署？go语言运行的环境怎么装？这个项目又该如何部署？

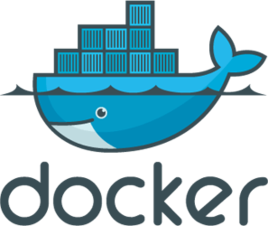
还有，一台linux机器上装了很多软件，部署了很多项目，相互之间有干扰怎么办？

如果有一项技术，可以解决以上问题或者是更多的问题，是不是很爽？ 那就是Docker（容器）技术。

## Docker简介

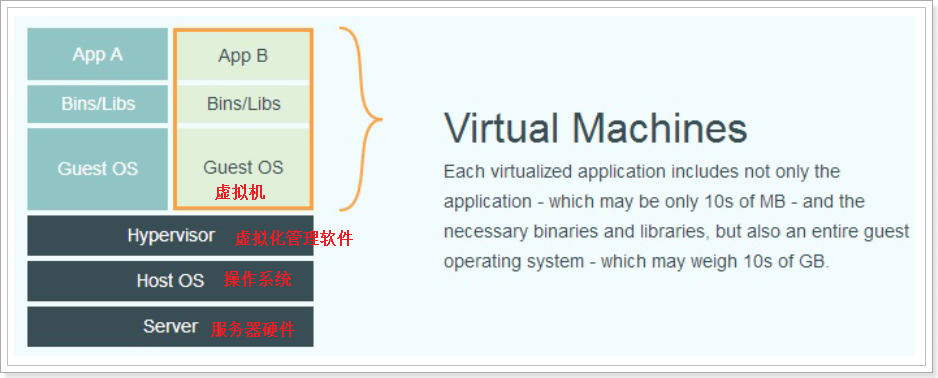


官网：<https://www.docker.com/> （国内打开比较慢）

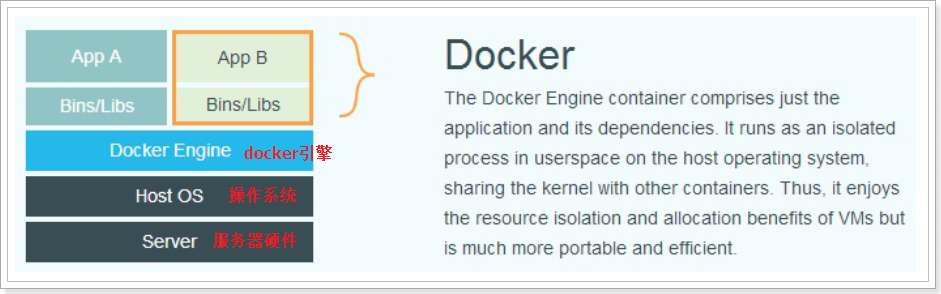


## 虚拟化和docker的对比

虚拟化：

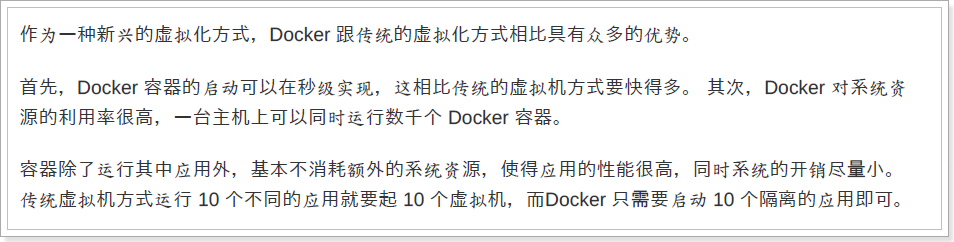


docker：

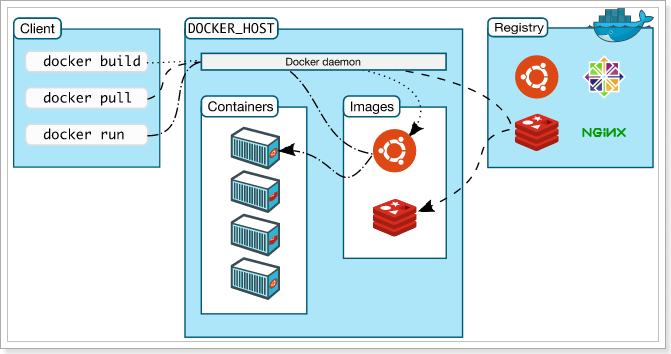


docker的优势在于可以直接使用主机操作系统的资源进行虚拟化。

## 为什么要使用docker？



## 架构



**Docker daemon（Docker进程）：**

Docker进程是部署在linux操作系统上，负责支撑Docker Container的运行以及本地Image的管理。

**Docker client：**

用户不直接操作Docker daemon，用户通过Docker client访问Docker，Docker client提供pull、run等操作命令。

**Docker Image：**

Docker 镜像就是一个只读的模板。

例如：一个镜像可以包含一个完整的 ubuntu 操作系统环境，里面仅安装了 Apache 或用户需要的其它应用程序。

镜像可以用来创建 Docker 容器。

Docker 提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下载一个已经做好的镜像来直接使用。

**Docker Container：**

Docker 利用容器来运行应用。容器是从镜像创建的运行实例。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。可以把容器看做是一个简易版的 Linux 环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

**Docker Registry：**

仓库分为公开仓库（Public）和私有仓库（Private）两种形式

最大的公开仓库是 Docker Hub，存放了数量庞大的镜像供用户下载。

用户也可以在本地网络内创建一个私有仓库。

当用户创建了自己的镜像之后就可以使用 push 命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时候，只需要从仓库上 pull 下来就可以了。

# 安装docker

我们可以在Centos或者Ubuntu下安装docker，要注意的是，centos6对docker支持的不好，使用docker建议升级到centos7。

docker官方建议使用Ubuntu系统，兼容性更好一些。所以，我们使用Ubuntu系统来学习docker。

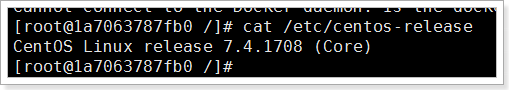
Ubuntu系统的安装参考《VMware Workstation 中安装 Ubuntu16.04 虚拟机.docx》

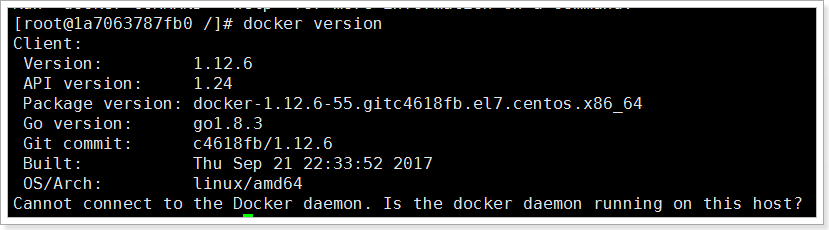
*在课前资料中已经提供了“Ubuntu16-64-镜像”，我们直接导入即可学习，该镜像中包含安装了lrzsz、jdk1.8、docker 1.12并且将所有的应用升级到最新。*

## centos7

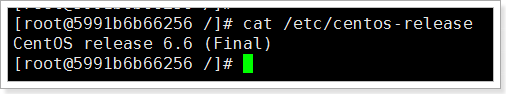
直接通过yum安装即可：

yum install -y docker





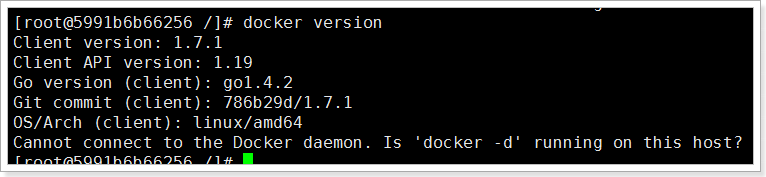
## centos6



rpm -ivh <http://dl.Fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm>

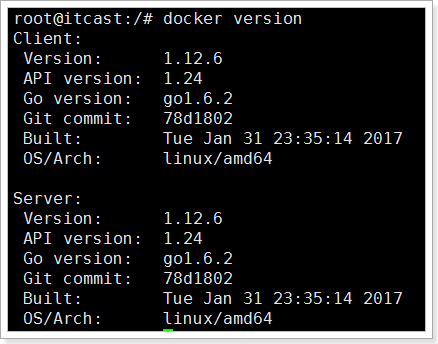
yum install -y docker-io

service docker start



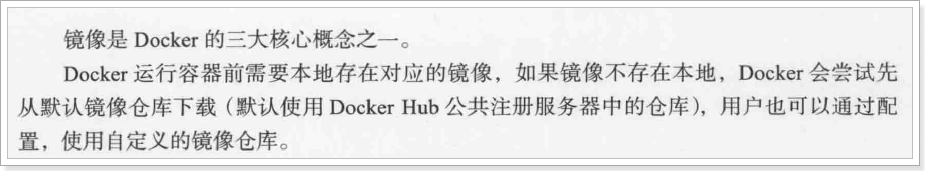
## Ubuntu 16

apt install docker.io -y



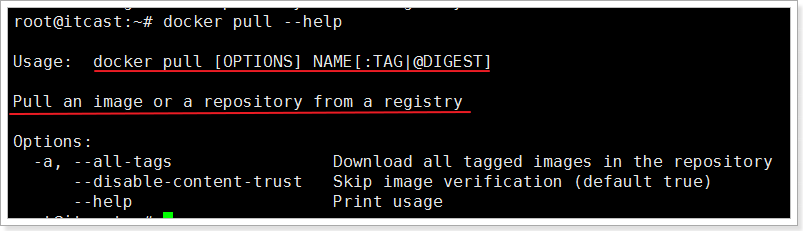
# 镜像

## 介绍



## 获取镜像

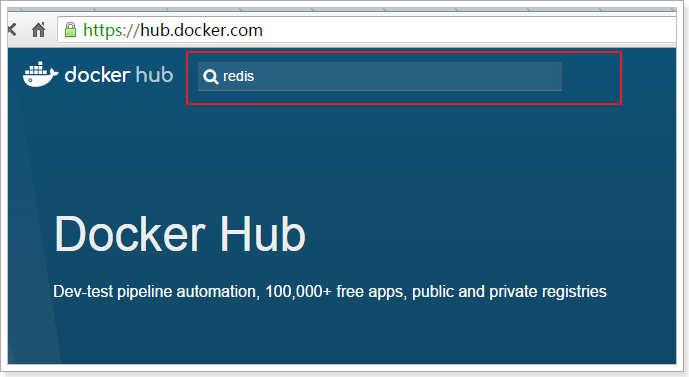
获取镜像通过命令，docker pull 完成：

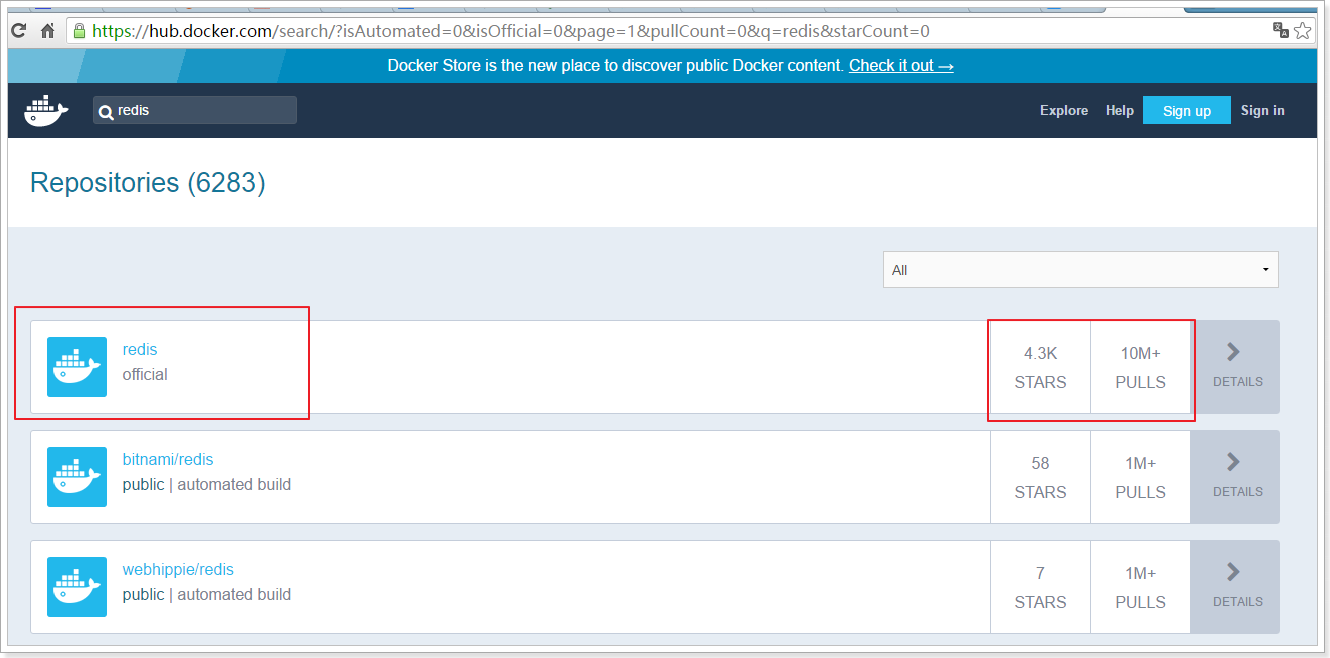


案例：获取Redis 3.0.0的镜像。

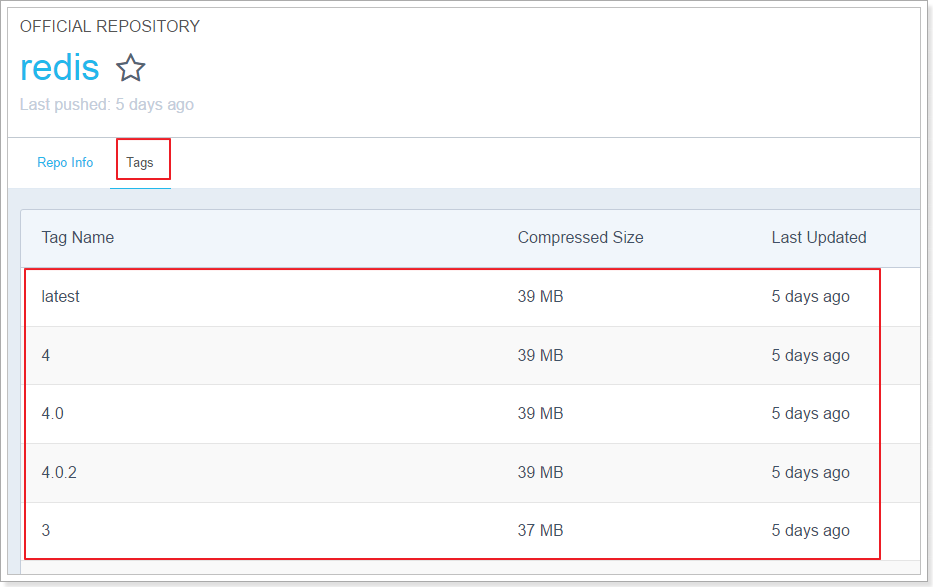
第一步，需要到远程的仓库进行搜索：

<https://hub.docker.com/>

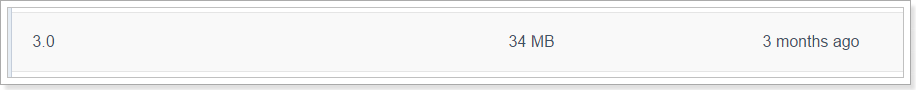




点击查看详情，查看tag：

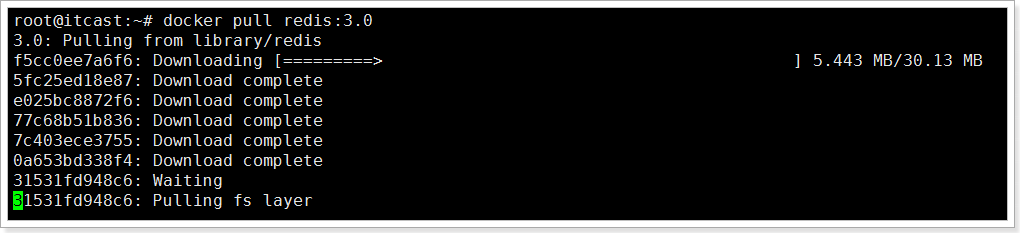


找到我们想要的3.0.0，最新的tag可以用latest标识：

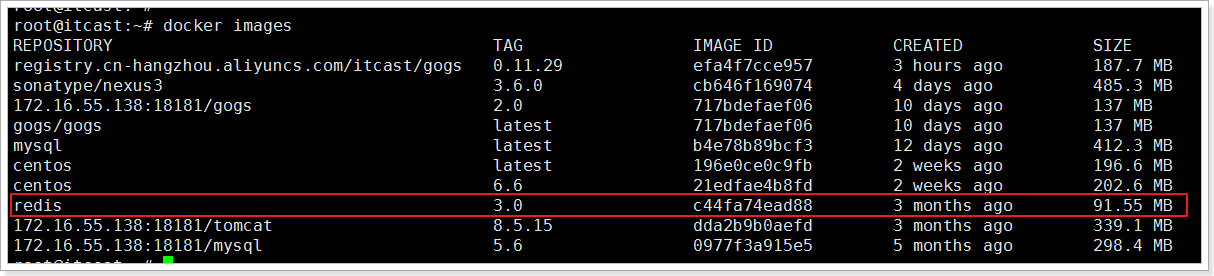


执行命令：

docker pull redis:3.0



通过docker images查看镜像：



## 镜像加速

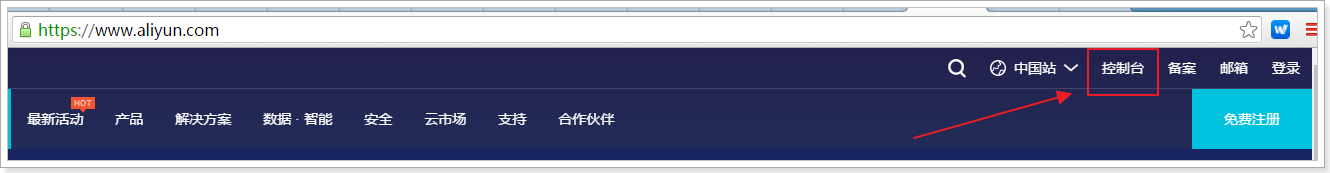
在国内通过docker官方的仓库进行下载镜像是比较慢，这个时候就需要使用加速器了。

在国内，阿里云、163都提供了docker仓库，但是阿里云还提供了加速功能。所以，我们选用阿里云的加速。 163的仓库地址：<https://c.163.com/hub#/m/home/>

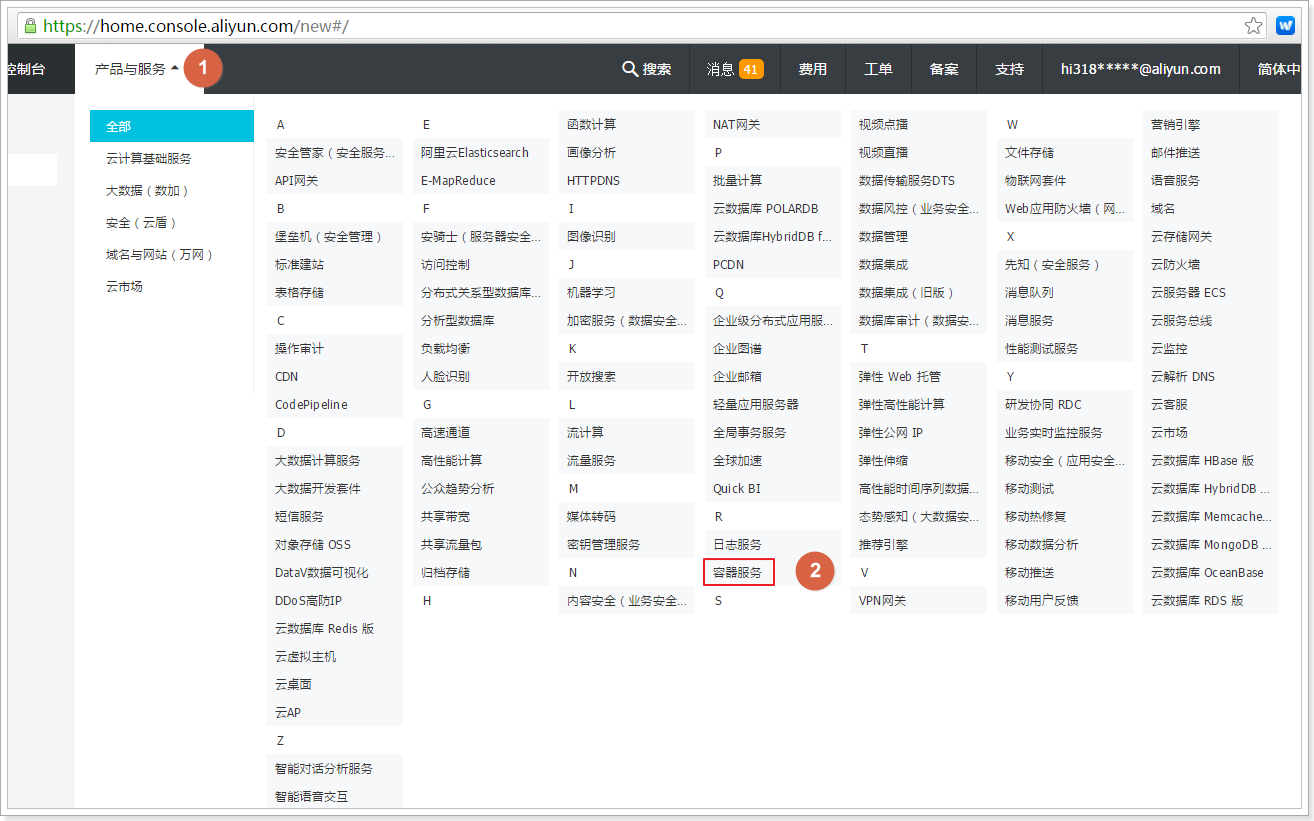
使用阿里云器，需要到阿里云进行注册，获取到自己专属的加速地址，当然也可以使用我的加速地址。（使用加速是无需登录的，所以可以公用）

第一步，打开阿里云网站，点击控制台：

<https://www.aliyun.com/>



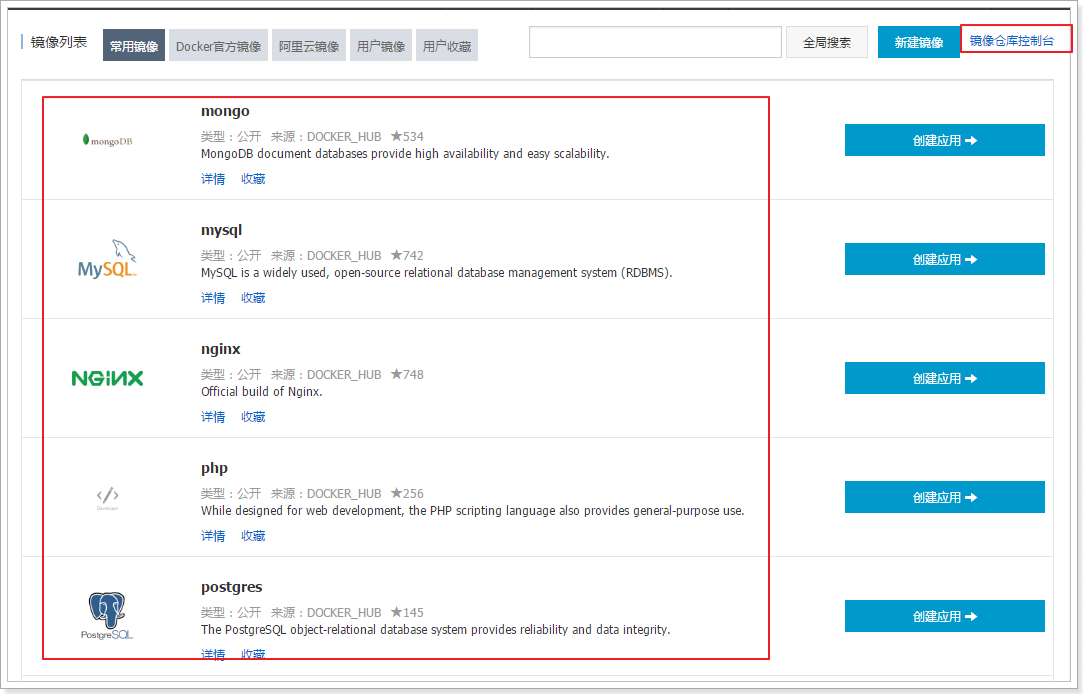
第二步，进行登录，如果没有账号就先注册个账号，登录成功后会跳转到控制台页面，找到容器服务：



第三步，点击“镜像”：



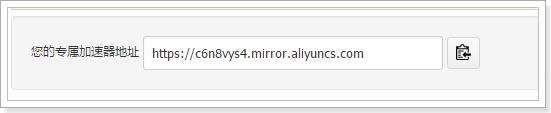
就可以看到有很多的镜像了：



第四步，点击右上角的“镜像仓库控制台”，进入后然后点击“Docker Hub 镜像站点”：



就可以看到你的专属加速地址了：



下面是使用加速地址的方法：



我们采用第2种方法：

sudo mkdir -p /etc/docker

sudo tee /etc/docker/daemon.json <<-'EOF'

{

"registry-mirrors": ["https://c6n8vys4.mirror.aliyuncs.com"]

}

EOF

sudo systemctl daemon-reload

sudo systemctl restart docker

mkdir -p /etc/docker

tee /etc/docker/daemon.json <<-'EOF'

{

"registry-mirrors": ["https://j8bshe1p.mirror.aliyuncs.com"]

}

EOF

systemctl daemon-reload

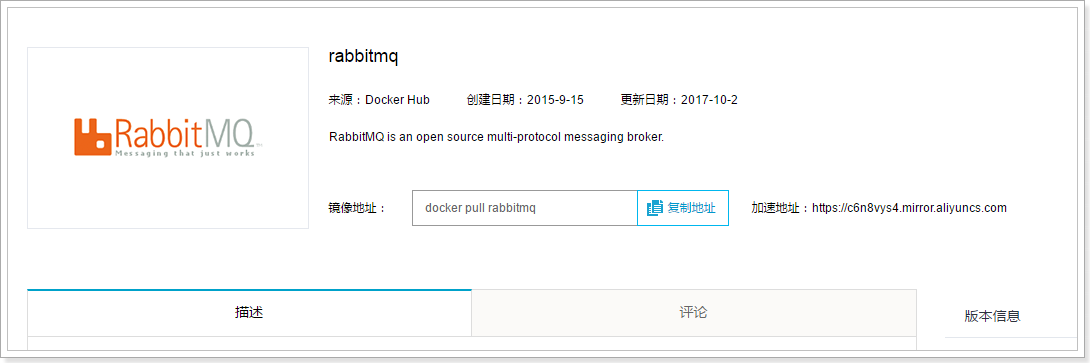
systemctl restart docker

再获取镜像时发现快多了。

搜素镜像：

<https://dev.aliyun.com/search.html>





下载rabbitmq的镜像：

docker pull rabbitmq:3.6.12



下载完成。

## 查询本地镜像

使用docker images 可以看到本地的镜像列表：



## 删除镜像

删除镜像通过命令docker rmi 镜像id完成，需要注意的是，删除镜像前，要删除所有使用到该镜像的容器。



-f参数是强制删除，如果有正在运行的容器使用了该镜像，那么该镜像是不会真正的删除的。

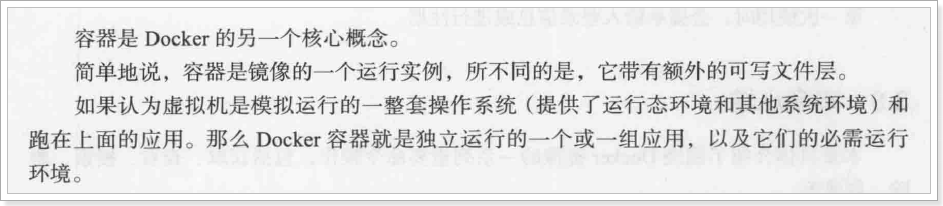
## 搜索镜像

刚刚我们是通过网页进行搜索，其实用过docker的命令也可以搜索，具体如下：



# 容器

## 介绍



## 查看容器列表

通过docker ps命令查看正在运行的容器列表，需要查询所有的容器需要加-a参数：

docker ps

docker ps -a



## 创建容器

通过docker create 命令可以创建一个容器，这仅仅是创建，并没有启动容器。

用法：

root@itcast:~# docker create --help

Usage: docker create [OPTIONS] IMAGE [COMMAND] [ARG...]

Create a new container

Options:

--add-host value Add a custom host-to-IP mapping (host:ip) (default [])

-a, --attach value Attach to STDIN, STDOUT or STDERR (default [])

--blkio-weight value Block IO (relative weight), between 10 and 1000

--blkio-weight-device value Block IO weight (relative device weight) (default [])

--cap-add value Add Linux capabilities (default [])

--cap-drop value Drop Linux capabilities (default [])

--cgroup-parent string Optional parent cgroup for the container

--cidfile string Write the container ID to the file

--cpu-percent int CPU percent (Windows only)

--cpu-period int Limit CPU CFS (Completely Fair Scheduler) period

--cpu-quota int Limit CPU CFS (Completely Fair Scheduler) quota

-c, --cpu-shares int CPU shares (relative weight)

--cpuset-cpus string CPUs in which to allow execution (0-3, 0,1)

--cpuset-mems string MEMs in which to allow execution (0-3, 0,1)

--device value Add a host device to the container (default [])

--device-read-bps value Limit read rate (bytes per second) from a device (default [])

--device-read-iops value Limit read rate (IO per second) from a device (default [])

--device-write-bps value Limit write rate (bytes per second) to a device (default [])

--device-write-iops value Limit write rate (IO per second) to a device (default [])

--disable-content-trust Skip image verification (default true)

--dns value Set custom DNS servers (default [])

--dns-opt value Set DNS options (default [])

--dns-search value Set custom DNS search domains (default [])

--entrypoint string Overwrite the default ENTRYPOINT of the image

-e, --env value Set environment variables (default [])

--env-file value Read in a file of environment variables (default [])

--expose value Expose a port or a range of ports (default [])

--group-add value Add additional groups to join (default [])

--health-cmd string Command to run to check health

--health-interval duration Time between running the check

--health-retries int Consecutive failures needed to report unhealthy

--health-timeout duration Maximum time to allow one check to run

--help Print usage

-h, --hostname string Container host name

-i, --interactive Keep STDIN open even if not attached *打开容器的标准输入*

--io-maxbandwidth string Maximum IO bandwidth limit for the system drive (Windows only)

--io-maxiops uint Maximum IOps limit for the system drive (Windows only)

--ip string Container IPv4 address (e.g. 172.30.100.104)

--ip6 string Container IPv6 address (e.g. 2001:db8::33)

--ipc string IPC namespace to use

--isolation string Container isolation technology

--kernel-memory string Kernel memory limit

-l, --label value Set meta data on a container (default [])

--label-file value Read in a line delimited file of labels (default [])

--link value Add link to another container (default [])

--link-local-ip value Container IPv4/IPv6 link-local addresses (default [])

--log-driver string Logging driver for the container

--log-opt value Log driver options (default [])

--mac-address string Container MAC address (e.g. 92:d0:c6:0a:29:33)

-m, --memory string Memory limit

--memory-reservation string Memory soft limit

--memory-swap string Swap limit equal to memory plus swap: '-1' to enable unlimited swap

--memory-swappiness int Tune container memory swappiness (0 to 100) (default -1)

--name string Assign a name to the container *指定容器的名称*

--network string Connect a container to a network (default "default")

--network-alias value Add network-scoped alias for the container (default [])

--no-healthcheck Disable any container-specified HEALTHCHECK

--oom-kill-disable Disable OOM Killer

--oom-score-adj int Tune host's OOM preferences (-1000 to 1000)

--pid string PID namespace to use

--pids-limit int Tune container pids limit (set -1 for unlimited)

--privileged Give extended privileges to this container

-p, --publish value Publish a container's port(s) to the host (default [])

*指定端口，如：docker create -p 16399:6379 redis:3.0，其中16379是主机端口，6379是容器对外端口，二者对应起来*

-P, --publish-all Publish all exposed ports to random ports *映射一个随机端口*

--read-only Mount the container's root filesystem as read only

--restart string Restart policy to apply when a container exits (default "no")

--runtime string Runtime to use for this container

--security-opt value Security Options (default [])

--shm-size string Size of /dev/shm, default value is 64MB

--stop-signal string Signal to stop a container, SIGTERM by default (default "SIGTERM")

--storage-opt value Storage driver options for the container (default [])

--sysctl value Sysctl options (default map[])

--tmpfs value Mount a tmpfs directory (default [])

-t, --tty Allocate a pseudo-TTY *让Docke分配一个伪端（pseudo-tt）并绑定到容器的标准输入上*

--ulimit value Ulimit options (default [])

-u, --user string Username or UID (format: <name|uid>[:<group|gid>])

--userns string User namespace to use

--uts string UTS namespace to use

-v, --volume value Bind mount a volume (default [])

--volume-driver string Optional volume driver for the container

--volumes-from value Mount volumes from the specified container(s) (default [])

-w, --workdir string Working directory inside the container

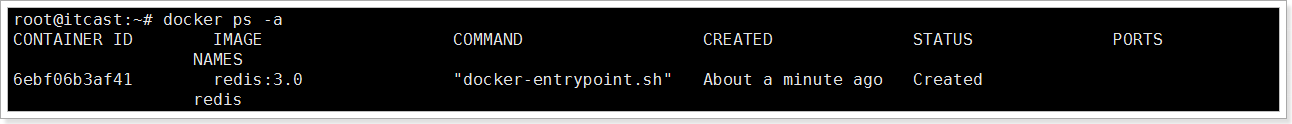
### 案例

案例：创建Redis的容器。

docker create -p 16379:6379 --name redis redis:3.0

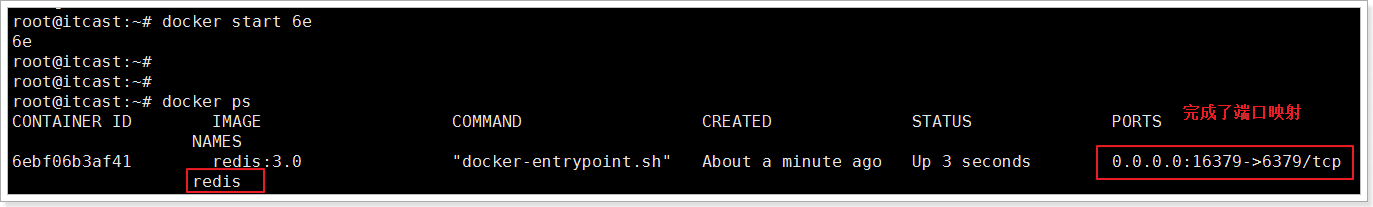


查看容器列表：

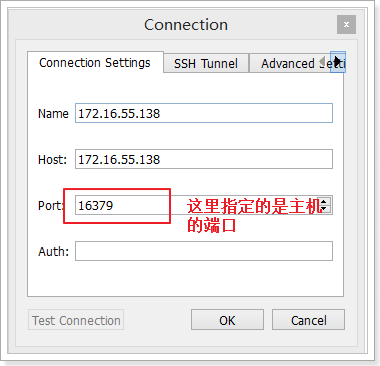


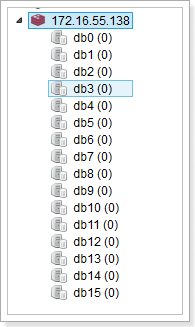
启动容器：

docker start 6e #指定容器的id，只要输入前几位即可



通过客户端进行测试：





至此，第一个docker容器就创建并且已经启动可用了。

## 创建并且运行容器

上面通过docker create创建了容器，然后通过docker start来启动容器，其实这种做法并不常用。

更常用的是docker run命令，这个命令的意思是创建并且启动容器。

用法：

root@itcast:~# docker run --help

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container *在一个新容器中运行一个命令。*

Options:

--add-host value Add a custom host-to-IP mapping (host:ip) (default [])

-a, --attach value Attach to STDIN, STDOUT or STDERR (default [])

--blkio-weight value Block IO (relative weight), between 10 and 1000

--blkio-weight-device value Block IO weight (relative device weight) (default [])

--cap-add value Add Linux capabilities (default [])

--cap-drop value Drop Linux capabilities (default [])

--cgroup-parent string Optional parent cgroup for the container

--cidfile string Write the container ID to the file

--cpu-percent int CPU percent (Windows only)

--cpu-period int Limit CPU CFS (Completely Fair Scheduler) period

--cpu-quota int Limit CPU CFS (Completely Fair Scheduler) quota

-c, --cpu-shares int CPU shares (relative weight)

--cpuset-cpus string CPUs in which to allow execution (0-3, 0,1)

--cpuset-mems string MEMs in which to allow execution (0-3, 0,1)

-d, --detach Run container in background and print container ID *让容器在后台运行*

--detach-keys string Override the key sequence for detaching a container

--device value Add a host device to the container (default [])

--device-read-bps value Limit read rate (bytes per second) from a device (default [])

--device-read-iops value Limit read rate (IO per second) from a device (default [])

--device-write-bps value Limit write rate (bytes per second) to a device (default [])

--device-write-iops value Limit write rate (IO per second) to a device (default [])

--disable-content-trust Skip image verification (default true)

--dns value Set custom DNS servers (default [])

--dns-opt value Set DNS options (default [])

--dns-search value Set custom DNS search domains (default [])

--entrypoint string Overwrite the default ENTRYPOINT of the image

-e, --env value Set environment variables (default [])

--env-file value Read in a file of environment variables (default [])

--expose value Expose a port or a range of ports (default [])

--group-add value Add additional groups to join (default [])

--health-cmd string Command to run to check health

--health-interval duration Time between running the check

--health-retries int Consecutive failures needed to report unhealthy

--health-timeout duration Maximum time to allow one check to run

--help Print usage

-h, --hostname string Container host name

-i, --interactive Keep STDIN open even if not attached

--io-maxbandwidth string Maximum IO bandwidth limit for the system drive (Windows only)

--io-maxiops uint Maximum IOps limit for the system drive (Windows only)

--ip string Container IPv4 address (e.g. 172.30.100.104)

--ip6 string Container IPv6 address (e.g. 2001:db8::33)

--ipc string IPC namespace to use

--isolation string Container isolation technology

--kernel-memory string Kernel memory limit

-l, --label value Set meta data on a container (default [])

--label-file value Read in a line delimited file of labels (default [])

--link value Add link to another container (default [])

--link-local-ip value Container IPv4/IPv6 link-local addresses (default [])

--log-driver string Logging driver for the container

--log-opt value Log driver options (default [])

--mac-address string Container MAC address (e.g. 92:d0:c6:0a:29:33)

-m, --memory string Memory limit

--memory-reservation string Memory soft limit

--memory-swap string Swap limit equal to memory plus swap: '-1' to enable unlimited swap

--memory-swappiness int Tune container memory swappiness (0 to 100) (default -1)

--name string Assign a name to the container

--network string Connect a container to a network (default "default")

--network-alias value Add network-scoped alias for the container (default [])

--no-healthcheck Disable any container-specified HEALTHCHECK

--oom-kill-disable Disable OOM Killer

--oom-score-adj int Tune host's OOM preferences (-1000 to 1000)

--pid string PID namespace to use

--pids-limit int Tune container pids limit (set -1 for unlimited)

--privileged Give extended privileges to this container

-p, --publish value Publish a container's port(s) to the host (default [])

-P, --publish-all Publish all exposed ports to random ports

--read-only Mount the container's root filesystem as read only

--restart string Restart policy to apply when a container exits (default "no")

--rm Automatically remove the container when it exits

--runtime string Runtime to use for this container

--security-opt value Security Options (default [])

--shm-size string Size of /dev/shm, default value is 64MB

--sig-proxy Proxy received signals to the process (default true)

--stop-signal string Signal to stop a container, SIGTERM by default (default "SIGTERM")

--storage-opt value Storage driver options for the container (default [])

--sysctl value Sysctl options (default map[])

--tmpfs value Mount a tmpfs directory (default [])

-t, --tty Allocate a pseudo-TTY

--ulimit value Ulimit options (default [])

-u, --user string Username or UID (format: <name|uid>[:<group|gid>])

--userns string User namespace to use

--uts string UTS namespace to use

-v, --volume value Bind mount a volume (default [])

--volume-driver string Optional volume driver for the container

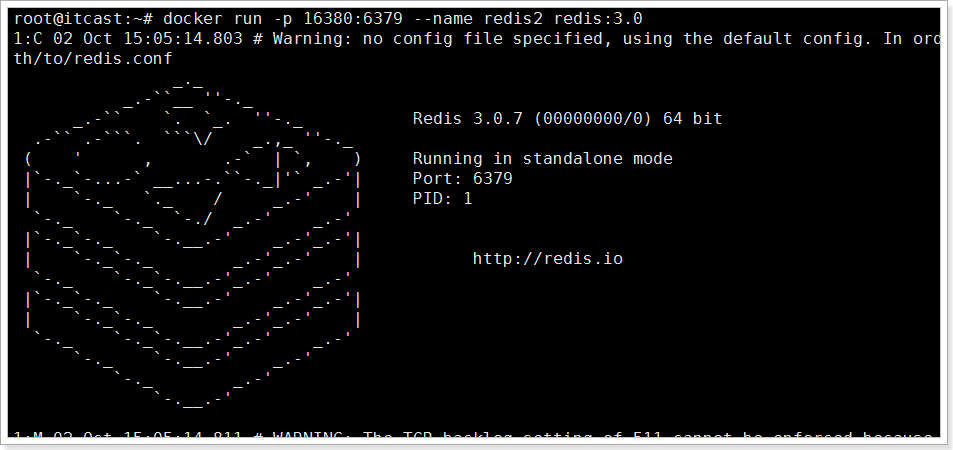
--volumes-from value Mount volumes from the specified container(s) (default [])

-w, --workdir string Working directory inside the container

### 案例

创建并且运行一个redis容器，它的端口是16380。

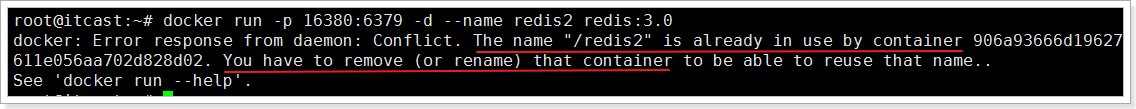
docker run -p 16380:6379 --name redis2 redis:3.0



发现，这个容器已经创建并且运行，但是，它并没有在后台运行，当按下 control + c时，该容器将停止。

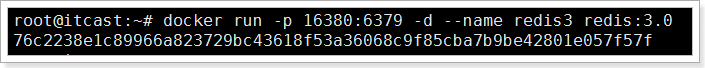
为了让容器在后台运行，需要添加参数 -d。

docker run -p 16380:6379 -d --name redis2 redis:3.0



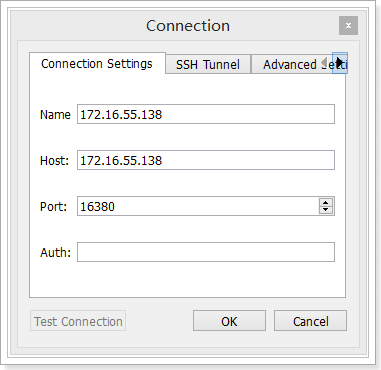
报错，说redis2这个名字的容器已经存在，需要我们删除906的容器或者给新容器重命名。我们重命名试试。

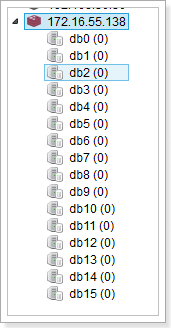
docker run -p 16380:6379 -d --name redis3 redis:3.0



已经创建成功。

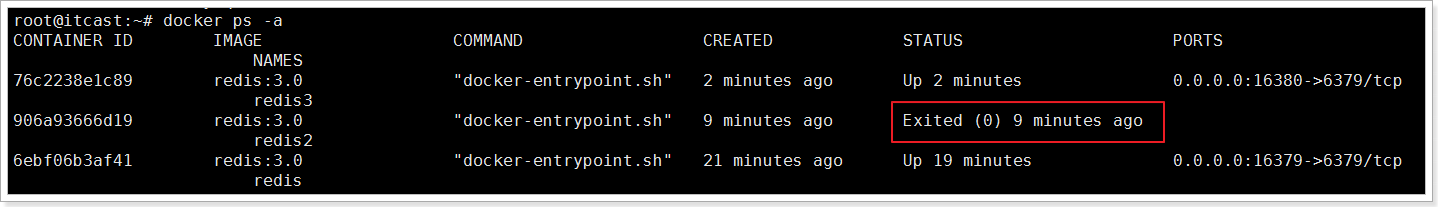
进行测试：





测试结果显示，该容器可用。

接下来，我看下当前运行的容器：



可以看到，有3个redis的容器，并且名为redis2的容器已经退出，并没有运行。

那么问题来了，redis2这个容器我不想要了，怎么删除？

## 启动/停止容器

### 启动容器

root@itcast:~# docker start --help

Usage: docker start [OPTIONS] CONTAINER [CONTAINER...]

Start one or more stopped containers

Options:

-a, --attach Attach STDOUT/STDERR and forward signals

--detach-keys string Override the key sequence for detaching a container

--help Print usage

-i, --interactive Attach container's STDIN

用法：docker start 容器名或容器id

### 停止容器

停止容器有2种方式：

1. docker stop 容器名或容器id
2. docker kill 容器名或容器id

root@itcast:~# docker stop --help

Usage: docker stop [OPTIONS] CONTAINER [CONTAINER...]

Stop one or more running containers

Options:

--help Print usage

-t, --time int Seconds to wait for stop before killing it (default 10)

root@itcast:~# docker kill --help

Usage: docker kill [OPTIONS] CONTAINER [CONTAINER...]

Kill one or more running containers

Options:

--help Print usage

-s, --signal string Signal to send to the container (default "KILL")

## 删除容器

通过命令docker rm 删除容器，删除正在运行的容器添加 -f 参数。

root@itcast:~# docker rm --help

Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers

Options:

-f, --force Force the removal of a running container (uses SIGKILL)

--help Print usage

-l, --link Remove the specified link

-v, --volumes Remove the volumes associated with the container

案例：

docker rm redis2

删除名为redis2的容器。

## 进入容器

有些时候我们需要进入容器内，做一些操作，比如，修改配置文件等。

进入容器，通过命令 docker exec 完成。

root@itcast:~# docker exec --help

Usage: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

Run a command in a running container

-d, --detach Detached mode: run command in the background

--detach-keys Override the key sequence for detaching a container

--help Print usage

-i, --interactive Keep STDIN open even if not attached

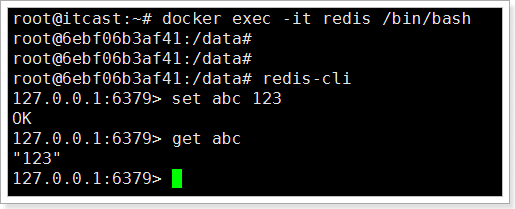
--privileged Give extended privileges to the command

-t, --tty Allocate a pseudo-TTY

-u, --user Username or UID (format: <name|uid>[:<group|gid>])

案例：进入redis容器，。

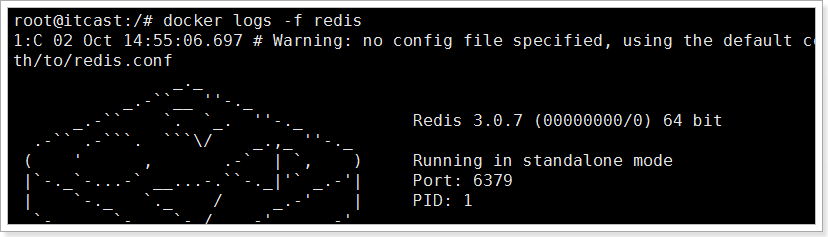
docker exec -it redis /bin/bash



control + d 退出容器。

## 查看日志

通过命令docker logs -f 容器名或id



# 仓库



## 阿里云仓库

阿里云既提供了加速功能，也提供了仓库功能，也就是我们可以将自己的镜像上传到阿里云仓库。

https://cr.console.aliyun.com/#/imageList



创建命名空间：



创建镜像：

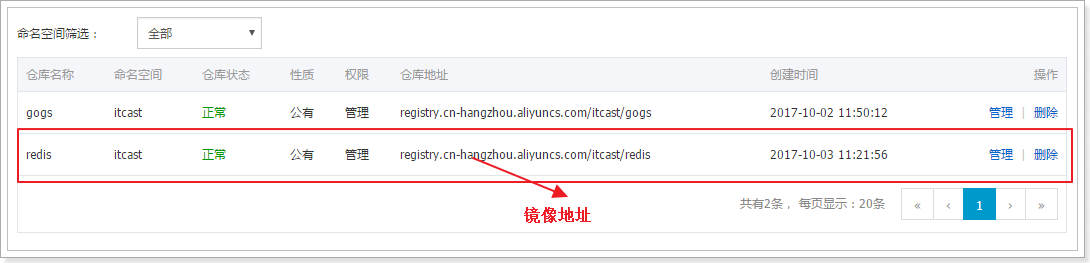




创建完成后就可以上传镜像到该仓库了。

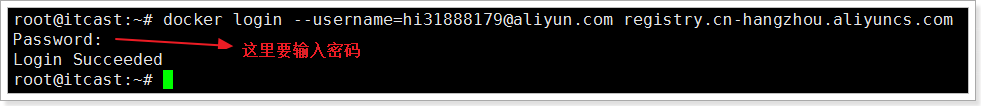
## 案例：推送redis镜像到阿里云仓库

第一步，在阿里云仓库创建redis镜像库：



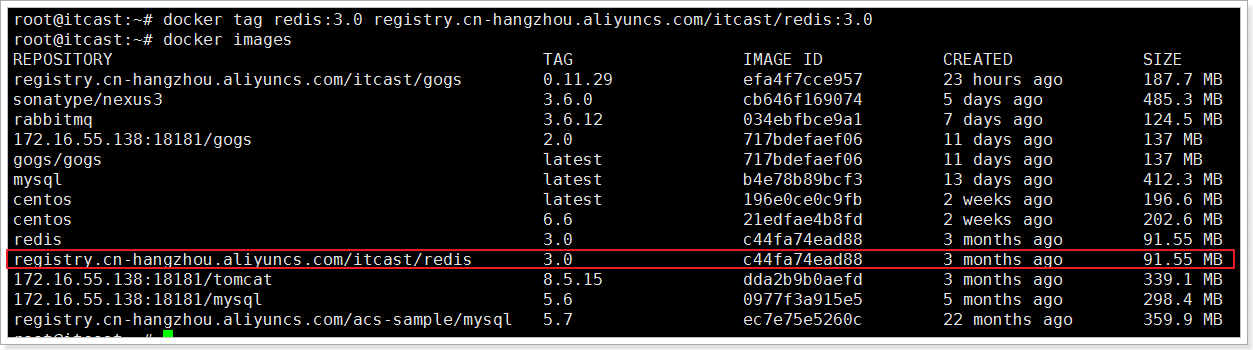
第二步，在docker中进行登录：

docker login --username=hi31888179@aliyun.com registry.cn-hangzhou.aliyuncs.com



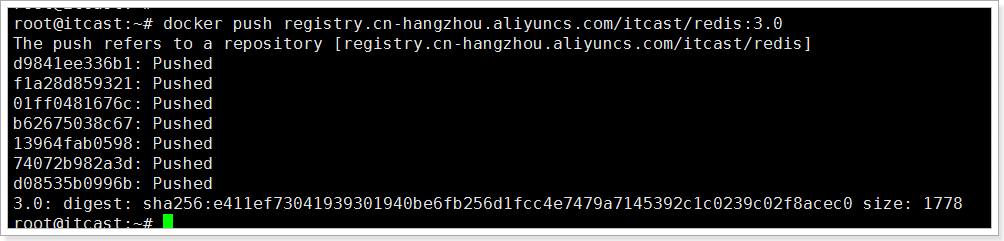
第三步，给镜像打tag：

docker tag redis:3.0 registry.cn-hangzhou.aliyuncs.com/itcast/redis:3.0

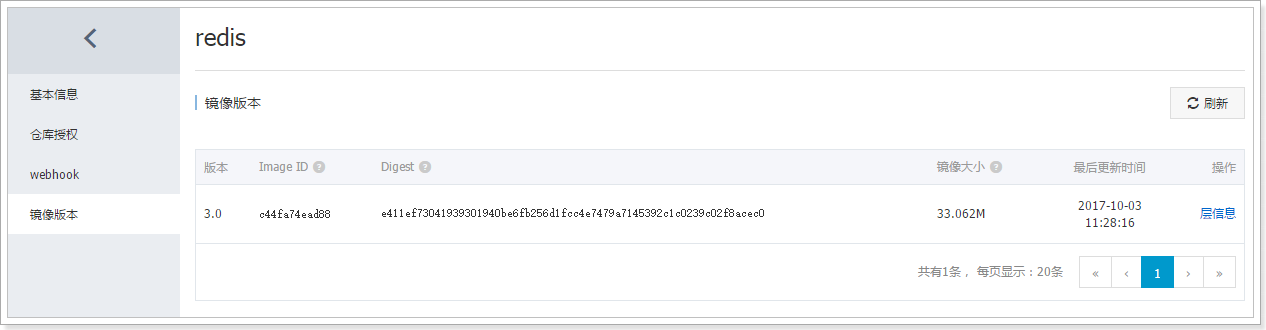


第四步，推送镜像到阿里云：

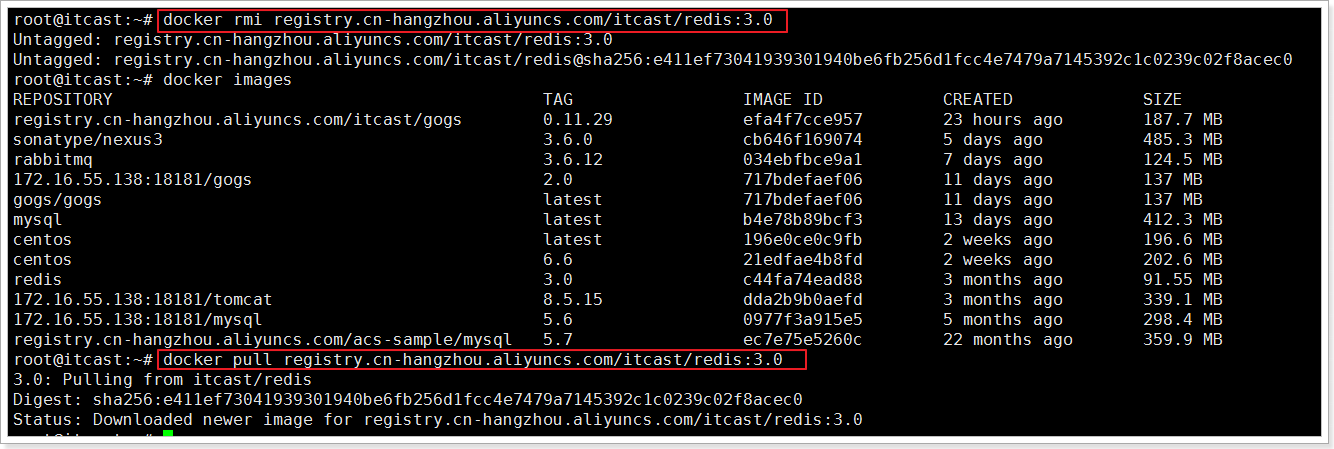
docker push registry.cn-hangzhou.aliyuncs.com/itcast/redis:3.0



第五步，在阿里云查看：



接下来，将本地仓库中的redis镜像删除，从阿里云拉取镜像到本地。



## 案例：推送镜像到内网私服

内网私服地址：<http://192.168.50.33:8081/> 或 <https://192.168.50.33:8443/>

账户：itcast / 123456

仓库地址：<https://192.168.50.33:8443/repository/itcast/> 需要注意的是，仓库的端口是18443，而不是8443，在创建仓库时指定。

docker进行推送镜像时只能使用https协议，所以，我们需要将证书导入到系统。

上传证书到/tmp下面

cp /tmp/nexus.crt /usr/local/share/ca-certificates

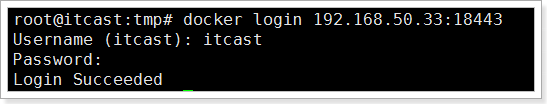
sudo update-ca-certificates

service docker restart

接下来，进行测试：

#使用docker登录到私服

docker login 192.168.50.33:18443

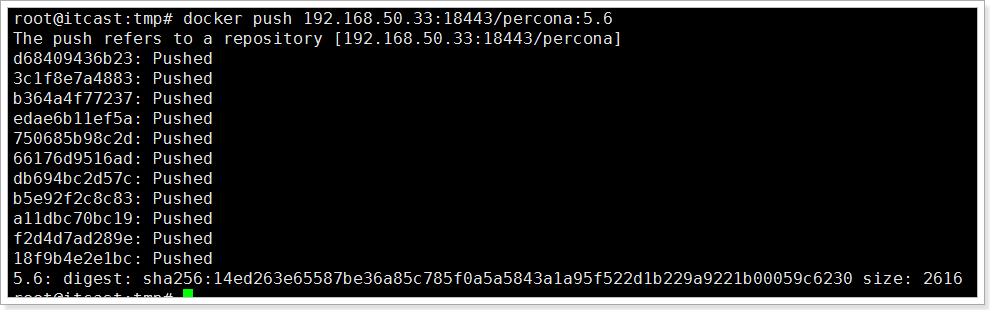


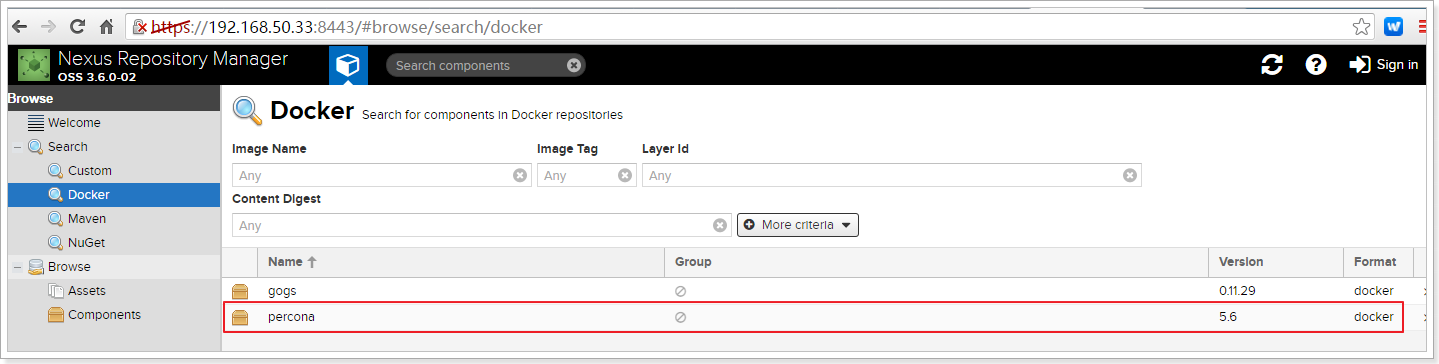
#打tag

docker tag percona:5.6 192.168.50.33:18443/percona:5.6

#推送镜像到私服

docker push 192.168.50.33:18443/percona:5.6





# 数据管理

容器在运行项目时会产生数据，比如运行的mysql容器，那么一定会有数据的产生，那么问题来了，数据是保存在容器内部还是保存在外部？

如果将数据保存在内部，那么也就意味着我们改变了原有镜像，这种做法是不可取的，因为在后期的镜像升级将变得不可能了。

*也就是说，运行的镜像，最好不要改变，如果必须改变的（比如说，修改配置文件等），在改变后记得commit提交打成一个新的镜像。*

显然，数据是应该保存在容器的外部，也就是说保存在主机上。那么问题又来了，数据保存在主机上，那么容器该如何读取主机中的数据呢？

## 数据卷

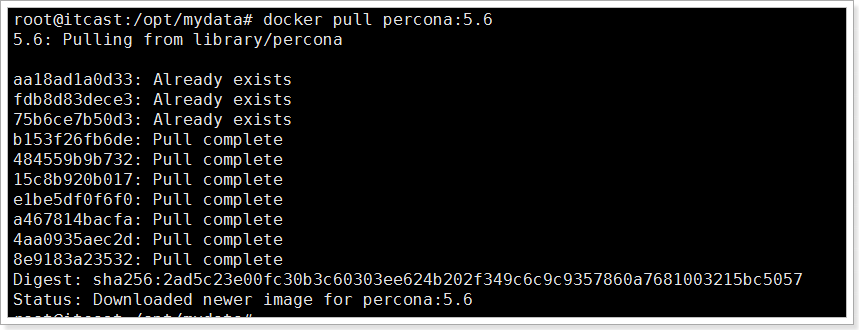
在create或者run容器时，可以通过-v参数指定主机的目录，挂在到容器中的某一个目录上，这样，容器就在这个目录读写数据了。从而实现了容器和数据的分离。

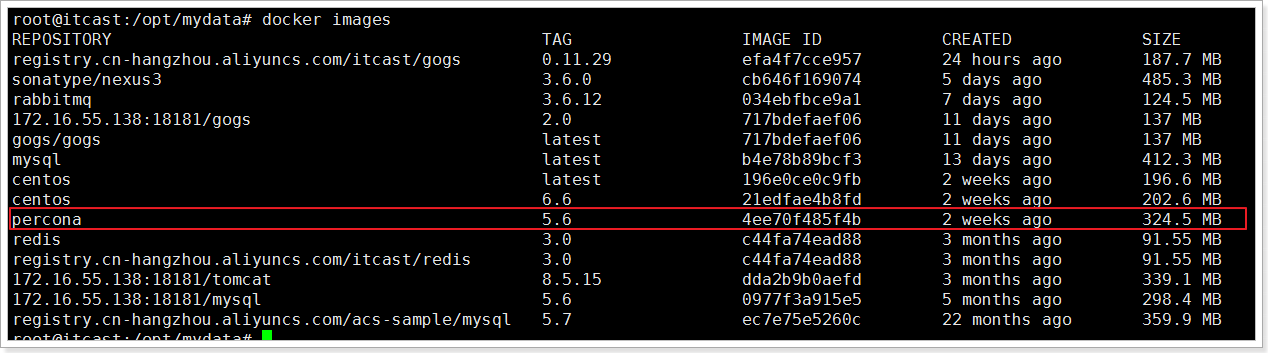
## 案例：运行mysql（percona）容器，将mysql的数据放到主机的/data/mysql-data中。

第一步，下载mysql的镜像：

地址：<https://hub.docker.com/_/percona/>

docker pull percona:5.6





第二步，创建容器：

docker create --name percona -v /data/mysql-data:/var/lib/mysql -p 3306:3306 -e MYSQL\_ROOT\_PASSWORD=root percona:5.6

解释：

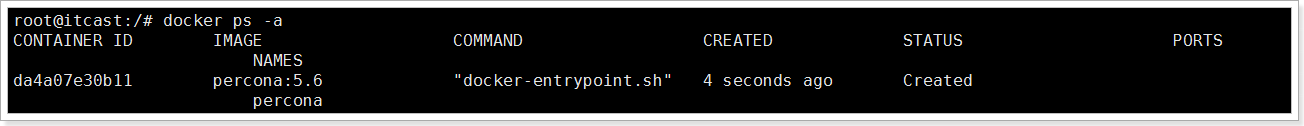
--name percona 指定是容器的名称

-v /data/mysql-data:/var/lib/mysql 将主机目录/data/mysql-data挂载到容器的/var/lib/mysql上

-p 33306:3306 设置端口映射，主机端口是33306，容器内部端口3306

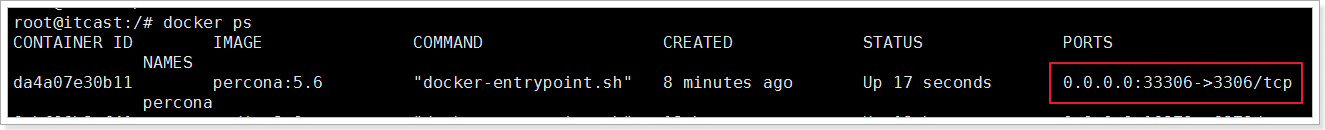
-e MYSQL\_ROOT\_PASSWORD=root 设置容器参数，设置root用户的密码为root

percona:5.6 镜像名:版本

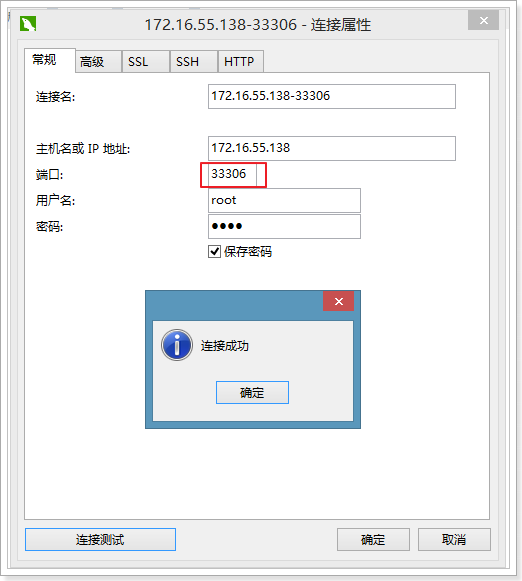


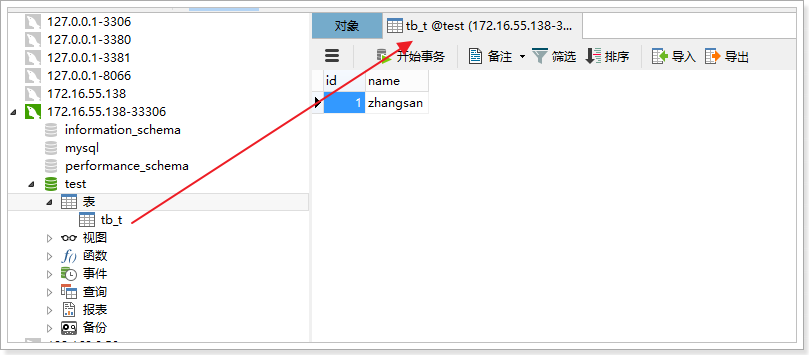
第三步，启动容器：

docker start percona



第四步，进行测试：





查看主机的/data/mysql-data:



# 构建镜像

前面我们的学习都是直接从仓库中拉取镜像，然后创建容器，最后启动容器来使用的。

在实际开发过程中，仓库中的容器可能不能完全满足我们的需求，比如说，我们项目的部署到docker容器，就不能从仓库中直接拉取镜像，就需要自己构建镜像了。

构建镜像通过编写Dockerfile配置文件完成。

## Dockerfile文件

Dockerfile是一个文本文件，里面编写多条命令，这些命令描述了一个镜像构建的细节。

先来看个示例：

#第一行必须指令基于的基础镜像

FROM ubutu

#维护者信息

MAINTAINER docker\_user zhangzhijun@itcast.cn

#镜像的操作指令

RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >> /etc/apt/sources.list

RUN apt-get update && apt-get install -y ngnix

RUN echo "\ndaemon off;">>/etc/ngnix/nignix.conf

#容器启动时执行指令

CMD /usr/sbin/ngnix

DockerFile分为四部分组成：基础镜像信、维护者信息、镜像操作指令和容器启动时执行的指令。

## 命令详解（13个命令）

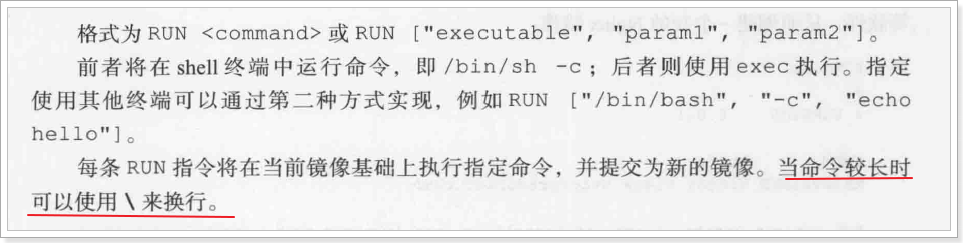
### FROM



### MAINTAINER



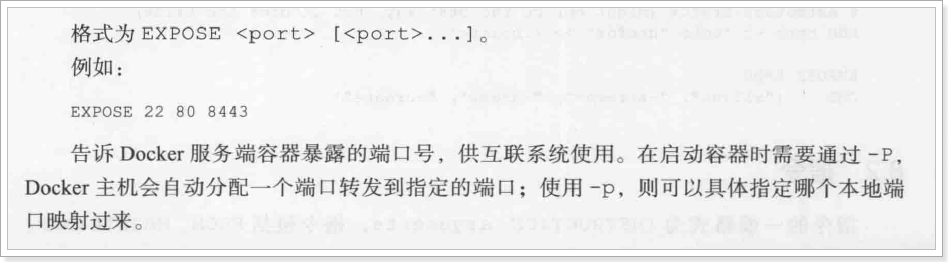
### RUN



### CMD



### EXPOSE



### ENV



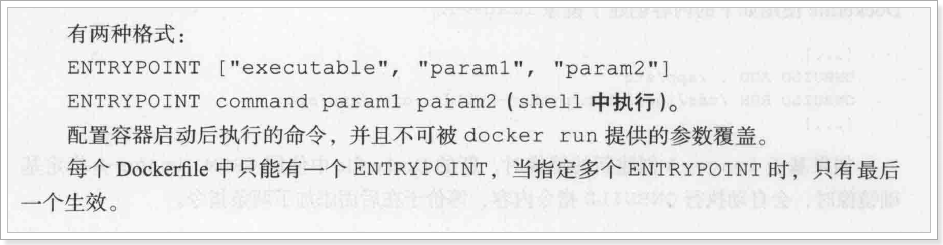
### ADD



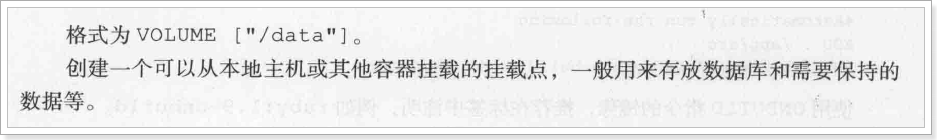
### COPY



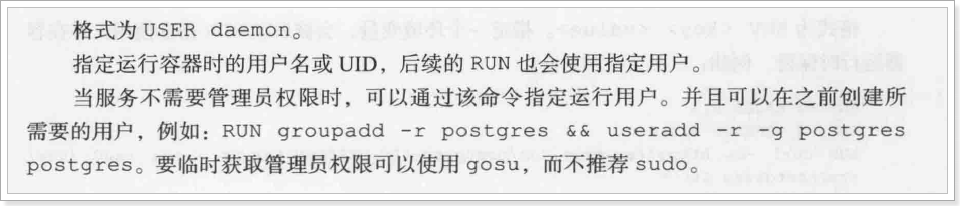
### ENTRYPOINT（入口）



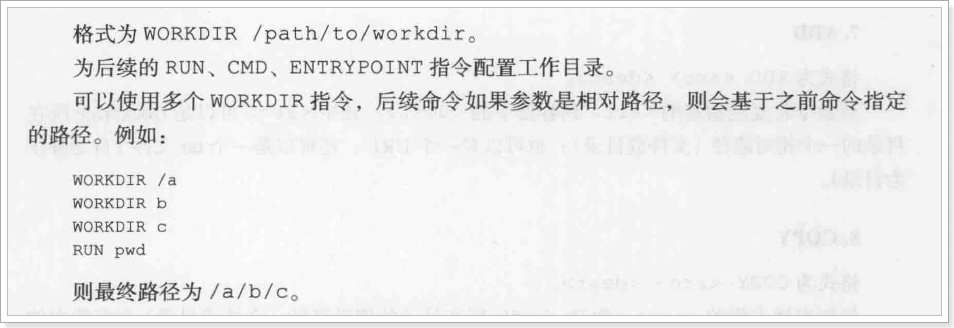
### VOLUME（挂载）



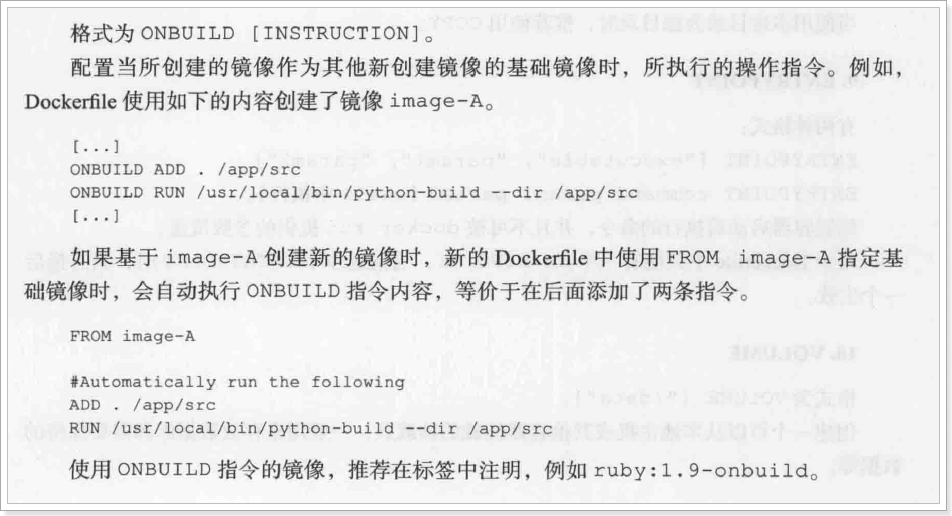
### USER



### WORKDIR



### ONBUILD



## 实战：自己构建redis镜像

编写Dockerfile如下：

# 构建Redis镜像

# itcast

#基于Centos6.6构建

FROM centos:6.6

#安装依赖

RUN yum -y install tar cpp binutils glibc glibc-kernheaders glibc-common glibc-devel gcc make gcc-c++ libstdc++-devel tcl

#创建安装目录

RUN mkdir -p /redis/data && cd /redis

#拷贝redis的安装包

COPY ./redis-3.0.0.tar.gz /redis

#解压

RUN cd /redis && tar -xvf redis-3.0.0.tar.gz && rm -rf redis-3.0.0.tar.gz && cd redis-3.0.0

#编译、安装

RUN cd /redis/redis-3.0.0 && make && make install

#复制配置文件到/redis中，并且修改redis为后台运行

RUN cp /redis/redis-3.0.0/redis.conf /redis/ && echo "daemonize yes" >> redis.conf

#设置数据挂载目录以及工作目录

VOLUME /redis/data

WORKDIR /redis/data

#容器启动后执行该命令

ENTRYPOINT ["/usr/local/bin/redis-server", "/redis/redis.conf"]

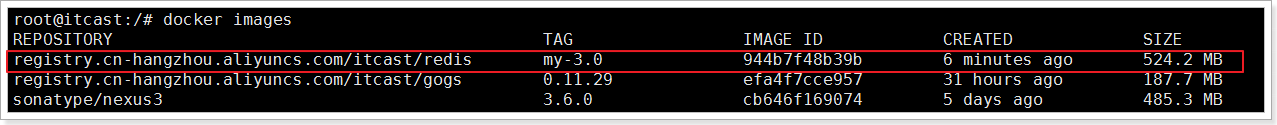
#设置对外的端口号

EXPOSE 6379

构建命令：

docker build -t registry.cn-hangzhou.aliyuncs.com/itcast/redis:my-3.0 /tmp/build-redis-docker-image/

构建完成：



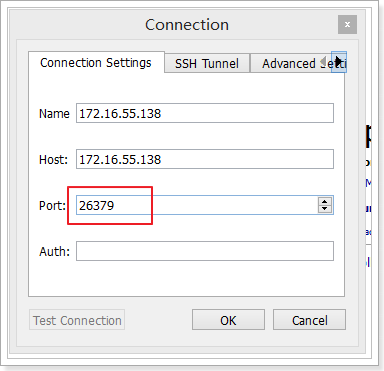
创建容器：

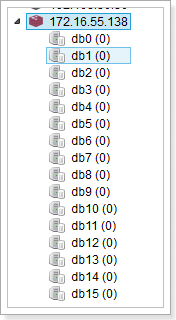
docker create -t --name myRedis -p 26379:6379 registry.cn-hangzhou.aliyuncs.com/itcast/redis:my-3.0

启动容器：



测试：

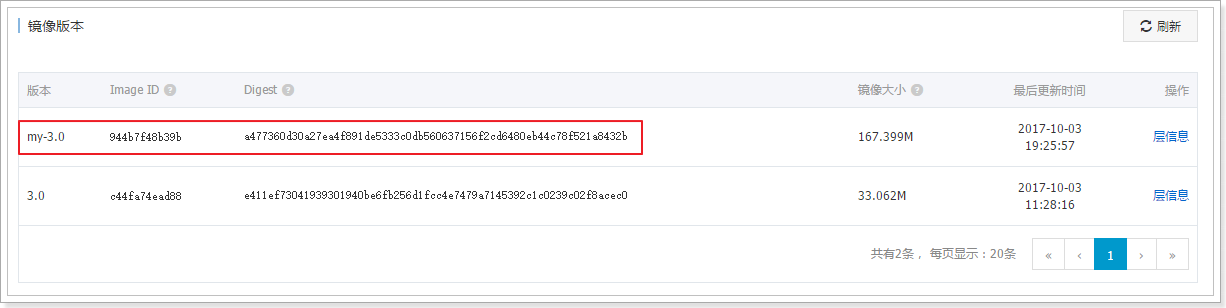




将该镜像上传到阿里云：

docker push registry.cn-hangzhou.aliyuncs.com/itcast/redis:my-3.0





# 部署Spring Cloud微服务

## Spring Boot项目的配置文件

由于项目会经历不同的环境，有开发环境、测试环境、生产环境，每个环境的地址是不同的，所以，需要在每个环境中编写不同的配置文件，比如：端口、Eureka服务地址等。

Spring Boot项目的配置文件提供的多种方式指定不同环境配置项的方法，这里介绍一种通过运行参数或者是JVM参数来指定。

在配置文件中这样配置：

server:

port: ${port:6778} #服务端口

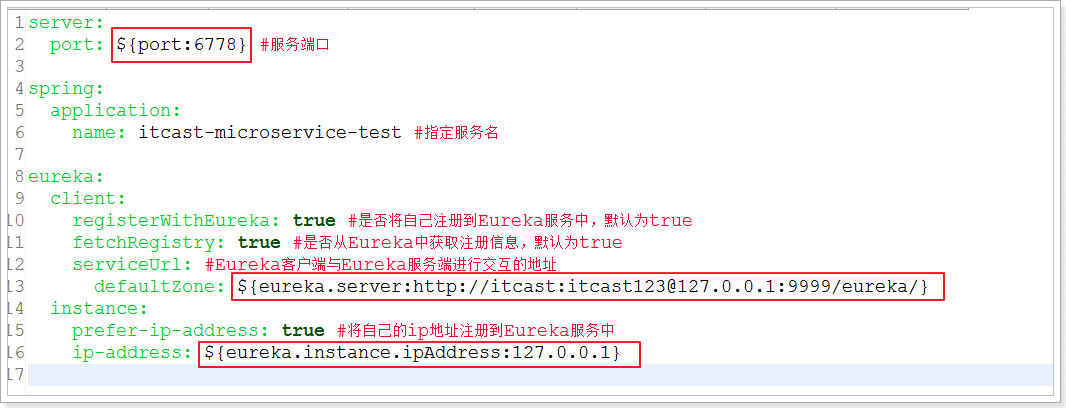
说明：

${} -- 指定这是一个动态的配置文件项，不是固定的内容

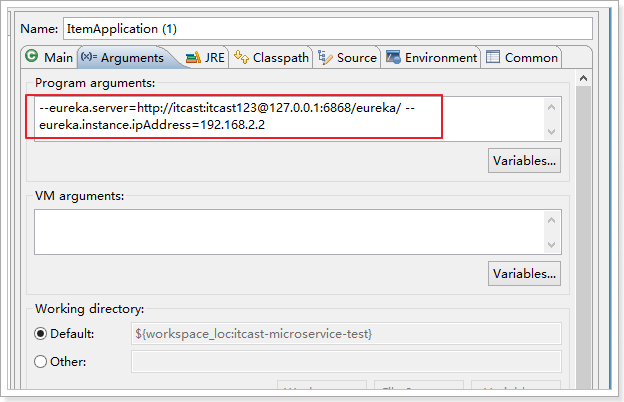
port --指运行参数或者JVM参数来替换的变量值

:6778 -- 指如果运行环境中没有改配置项，就采用该内容

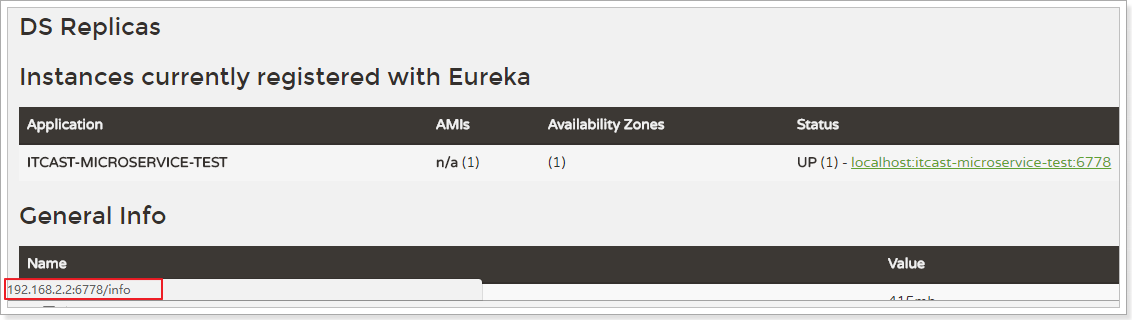
类似的配置如下：



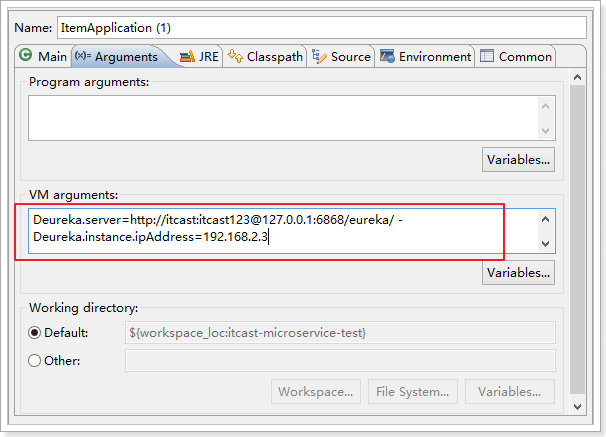
### 方式一：通过运行参数指定



设置方式：--参数名=参数值 --参数名=参数值 ……



### 方式二：通过JVM参数指定



设置方式：-D参数名=参数值 --D参数名=参数值 ……



## 部署的规划

由于我们是测试阶段，所以先只发布到一台机器，在多个容器中运行，一个项目运行一个容器。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 项目 | 容器名称 | IP | 容器内部端口 | 主机端口 | 备注 |
| itcast-microservice-eureka | eureka-server | 172.16.55.138 | 6868 | 6868 | 注册中心 |
| itcast-microservice-config-server | config-server | 172.16.55.138 | 6869 | 6869 | 配置服务 |
| RabbitMQ | RabbitMQ | 172.16.55.138 | 5672/15672 | 5672/15672 | RabbitMQ服务 |
| itcast-microservice-item-1 | item-server-1 | 172.16.55.138 | 6870 | 6870 | 商品服务1 |
| itcast-microservice-item-2 | item-server-2 | 172.16.55.138 | 6871 | 6871 | 商品服务2 |
| itcast-microservice-order-1 | order-server-1 | 172.16.55.138 | 6872 | 6872 | 订单服务1 |
| itcast-microservice-api-gateway-1 | api-gateway-1 | 172.16.55.138 | 6873 | 6873 | 网关服务1 |
|  |  |  |  |  |  |

## 部署Eureka注册中心

### 修改配置文件

server:

port: 6868 #服务端口

spring:

application:

name: itcast-microservice-eureka #指定服务名

eureka:

client:

registerWithEureka: **false** #是否将自己注册到Eureka服务中，本身就是所有无需注册

fetchRegistry: **false** #是否从Eureka中获取注册信息

server:

enable-self-preservation: **true** #禁用自我保护模式

security:

basic:

enable: **true** #开启基于HTTP basic的认证

user: #配置用户的账号信息

name: itcast

password: itcast123

### 导入spring boot打包插件

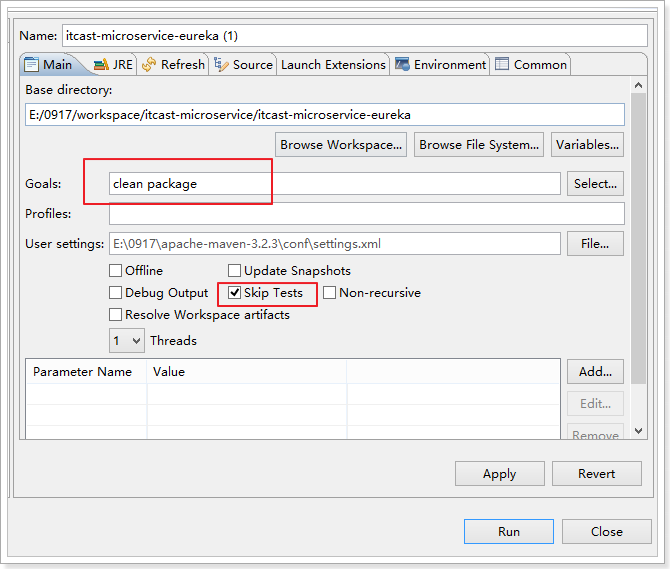
<plugin>

<groupId>org.springframework.boot</groupId>

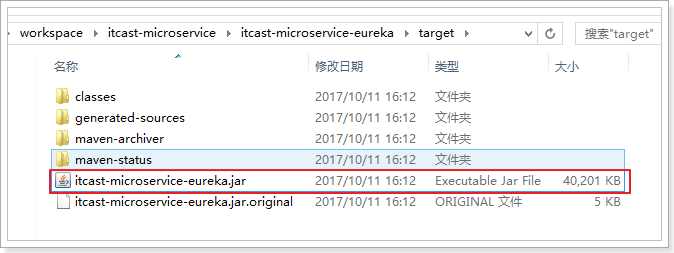
<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

### 进行打包



打包完成：



### 拉取java镜像

docker pull java:8

### 制作Docker镜像

首先将itcast-microservice-eureka.jar文件上传到linux系统中的/tmp/itcast-microservice-eureka目录。

编写Dockerfile文件：

FROM java:8

COPY ./itcast-microservice-eureka.jar /itcast-microservice-eureka/itcast-microservice-eureka.jar

COPY ./app-entrypoint.sh /

RUN chmod +x /app-entrypoint.sh

EXPOSE 6868

ENTRYPOINT ["/app-entrypoint.sh"]

app-entrypoint.sh文件：

#!/bin/bash

java -jar /itcast-microservice-eureka/itcast-microservice-eureka.jar

构建：

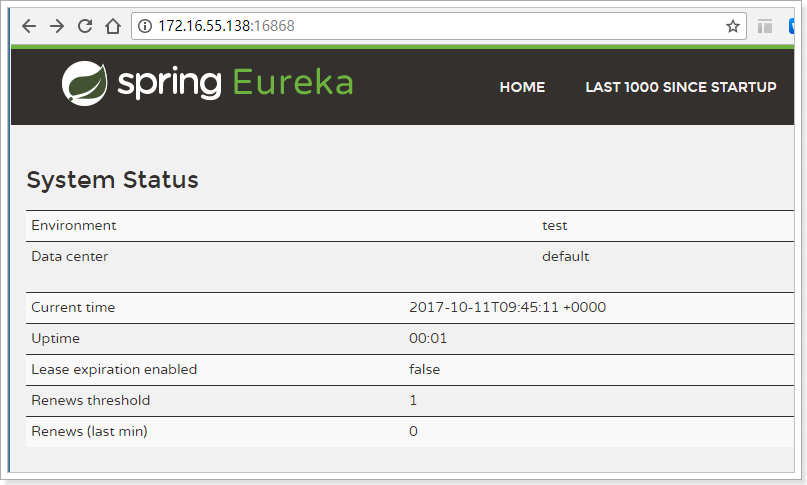
docker build -t itcast-microservice-eureka:1.0.0 .

### 运行容器

不设置变量：

docker create --name eureka-server -t -p 6868:6868 itcast-microservice-eureka:1.0.0

docker start eureka-server && docker logs -f eureka-server



## 安装RabbitMQ服务

同样也是使用容器来运行RabbitMQ。

docker pull rabbitmq:3.6.12-management

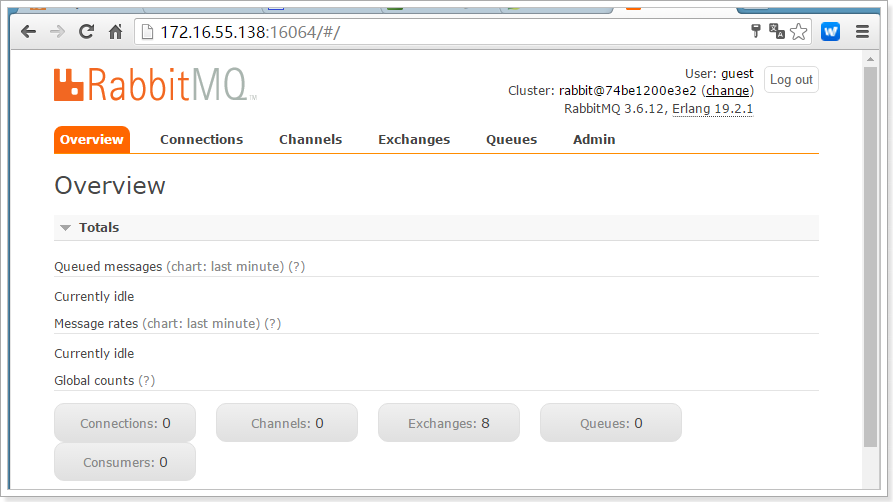
docker create --name RabbitMQ -t -p 5672:5672 -p 15672:15672 rabbitmq:3.6.12-management

docker start RabbitMQ && docker logs -f RabbitMQ

测试：



使用guest/guest进行登录：



RabbitMQ安装完成。

## 部署itcast-microservice-config-server

### 修改配置文件

server:

port: ${port:6869} #服务端口

spring:

application:

name: itcasst-microservice-config-server #指定服务名

cloud:

config:

server:

git: #配置git仓库地址

uri: ${gitUrl:http://172.16.55.138:10080/zhangzhijun/itcast-config-server.git}

#username: zhangzhijun

#password: 123456

rabbitmq: #RabbitMQ相关的配置

host: ${rabbitMQHost:127.0.0.1}

port: ${rabbitMQPort:5672}

username: ${rabbitMQUsername:guest}

password: ${rabbitMQPassword:guest}

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: ${eurekaServerUrl:http://itcast:itcast123@127.0.0.1:6868/eureka/}

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

ipAddress: ${ipAddress:127.0.0.1}

management:

security:

enabled: **false** #是否开启actuator安全认证

### 导入spring boot打包插件

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

### 制作Docker镜像

首先将itcast-microservice-config.jar文件上传到linux系统中的/tmp/itcast-microservice-config目录。

编写Dockerfile文件：

FROM java:8

COPY ./itcast-microservice-config.jar /itcast-microservice-config/itcast-microservice-config.jar

COPY ./app-entrypoint.sh /

RUN chmod +x /app-entrypoint.sh

ENTRYPOINT ["/app-entrypoint.sh"]

app-entrypoint.sh文件：

#!/bin/bash

java -jar -Dport=$PORT -DgitUrl=$GIT\_URL -DrabbitMQHost=$RABBITMQ\_HOST -DrabbitMQPort=$RABBITMQ\_PORT -DrabbitMQUsername=$RABBITMQ\_USERNAME -DrabbitMQPassword=$RABBITMQ\_PASSWORD -DeurekaServerUrl=$EUREKA\_SERVER\_URL -DipAddress=$IP\_ADDRESS /itcast-microservice-config/itcast-microservice-config.jar

构建：

docker build -t itcast-microservice-config:1.0.0 .

格式化下看的清楚些：

java

-jar

-Dport=$PORT

-DgitUrl=$GIT\_URL

-DrabbitMQHost=$RABBITMQ\_HOST

-DrabbitMQPort=$RABBITMQ\_PORT

-DrabbitMQUsername=$RABBITMQ\_USERNAME

-DrabbitMQPassword=$RABBITMQ\_PASSWORD

-DeurekaServerUrl=$EUREKA\_SERVER\_URL

-DipAddress=$IP\_ADDRESS

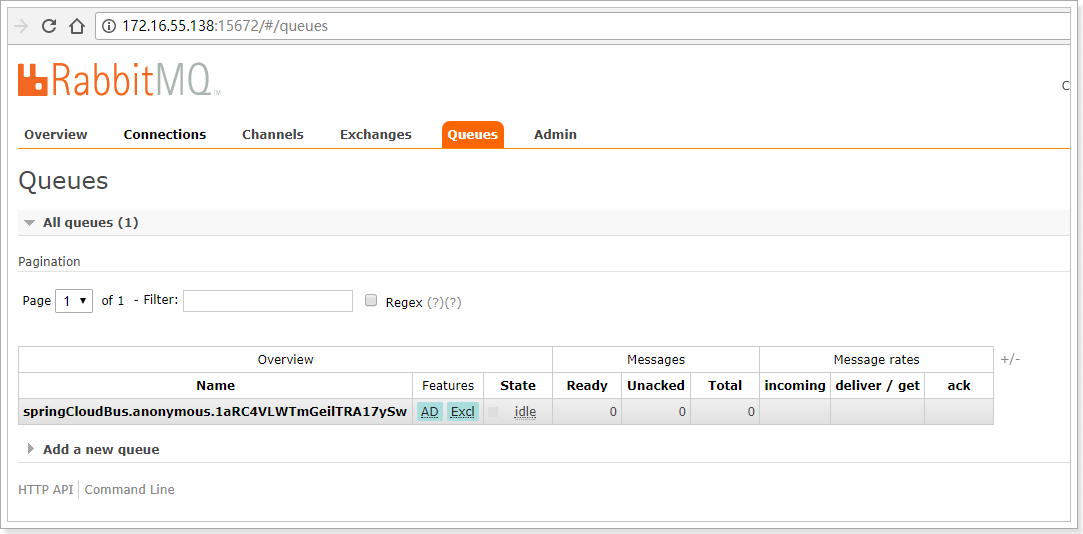
/itcast-microservice-config/itcast-microservice-config.jar

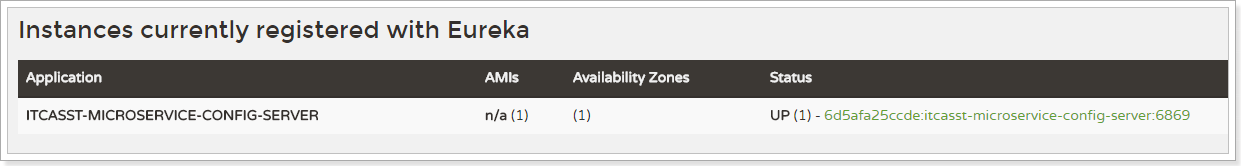
### 运行容器

docker create --name config-server -t -p 6869:6869 -e PORT=6869 -e GIT\_URL=https://gitee.com/zhijun.zhang/itcast-config-server.git -e RABBITMQ\_HOST=172.16.55.138 -e RABBITMQ\_PORT=5672 -e RABBITMQ\_USERNAME=guest -e RABBITMQ\_PASSWORD=guest -e EUREKA\_SERVER\_URL=http://itcast:itcast123@172.16.55.138:6868/eureka/ -e IP\_ADDRESS=172.16.55.138 itcast-microservice-config:1.0.0

docker start config-server && docker logs -f config-server







## 部署itcast-microservice-item-1

### 修改配置文件

bootstrap.yml：

eureka:

client:

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: ${eurekaServerUrl:http://itcast:itcast123@127.0.0.1:6868/eureka/}

spring:

cloud:

config:

name: microservice #对应的配置服务中的应用名称

#uri: http://127.0.0.1:6869/

profile: dev #对应配置服务中的{profile}

label: master #对应的分支

discovery:

enabled: **true** #启用发现服务功能

service-id: itcasst-microservice-config-server #指定服务名称

rabbitmq: #RabbitMQ相关的配置

host: ${rabbitMQHost:127.0.0.1}

port: ${rabbitMQPort:5672}

username: ${rabbitMQUsername:guest}

password: ${rabbitMQPassword:guest}

application.yml：

server:

port: ${port:6870} #服务端口

spring:

application:

name: itcasst-microservice-item #指定服务名

logging:

level:

org.springframework: INFO

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: ${eurekaServerUrl:http://itcast:itcast123@127.0.0.1:6868/eureka/}

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

ip-address: ${ipAddress:127.0.0.1}

management:

security:

enabled: **false** #是否开启actuator安全认证

### 导入spring boot打包插件

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

### 制作Docker镜像

首先将itcast-microservice-item.jar文件上传到linux系统中的/tmp/itcast-microservice-item目录。

编写Dockerfile文件：

FROM java:8

COPY ./itcast-microservice-item.jar /itcast-microservice-item/itcast-microservice-item.jar

COPY ./app-entrypoint.sh /

RUN chmod +x /app-entrypoint.sh

ENTRYPOINT ["/app-entrypoint.sh"]

app-entrypoint.sh文件：

#!/bin/bash

java -jar -Dport=$PORT -DrabbitMQHost=$RABBITMQ\_HOST -DrabbitMQPort=$RABBITMQ\_PORT -DrabbitMQUsername=$RABBITMQ\_USERNAME -DrabbitMQPassword=$RABBITMQ\_PASSWORD -DeurekaServerUrl=$EUREKA\_SERVER\_URL -DipAddress=$IP\_ADDRESS /itcast-microservice-item/itcast-microservice-item.jar

构建：

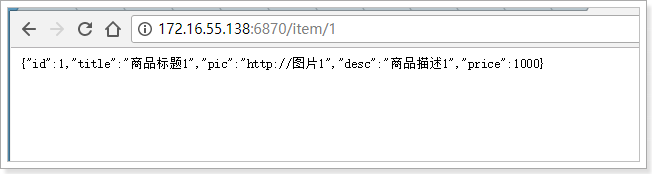
docker build -t itcast-microservice-item:1.0.0 .

### 运行容器

docker create --name item-server-1 -t -p 6870:6870 -e PORT=6870 -e RABBITMQ\_HOST=172.16.55.138 -e RABBITMQ\_PORT=5672 -e RABBITMQ\_USERNAME=guest -e RABBITMQ\_PASSWORD=guest -e EUREKA\_SERVER\_URL=http://itcast:itcast123@172.16.55.138:6868/eureka/ -e IP\_ADDRESS=172.16.55.138 itcast-microservice-item:1.0.0

docker start item-server-1 && docker logs -f item-server-1

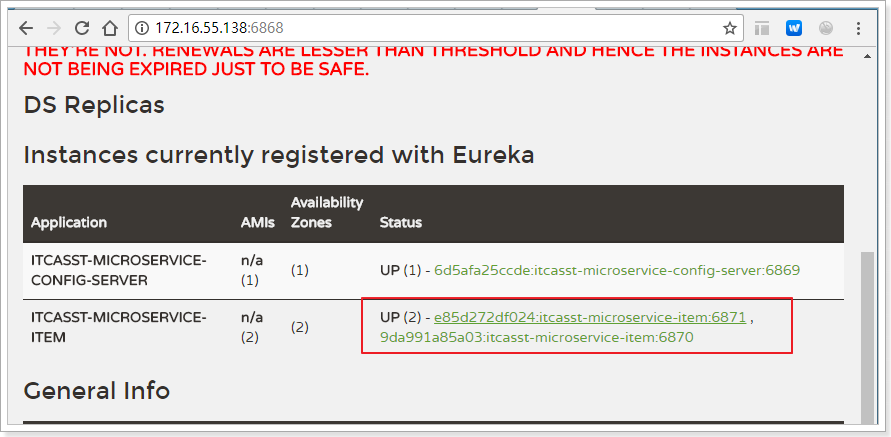


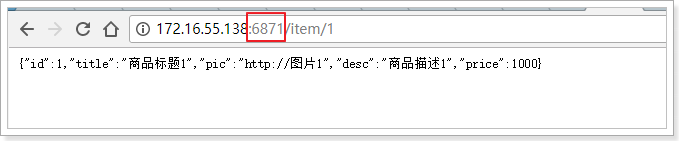


## 部署itcast-microservice-item-2

docker create --name item-server-2 -t -p 6871:6871 -e PORT=6871 -e RABBITMQ\_HOST=172.16.55.138 -e RABBITMQ\_PORT=5672 -e RABBITMQ\_USERNAME=guest -e RABBITMQ\_PASSWORD=guest -e EUREKA\_SERVER\_URL=http://itcast:itcast123@172.16.55.138:6868/eureka/ -e IP\_ADDRESS=172.16.55.138 itcast-microservice-item:1.0.0

docker start item-server-2 && docker logs -f item-server-2





## 部署itcast-microservice-order-1

### 修改配置文件

server:

port: ${port:8082} #服务端口

itcast:

item:

url: http://127.0.0.1:8081/item/

spring:

application:

name: itcasst-microservice-order #指定服务名

eureka:

client:

registerWithEureka: **false** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: ${eurekaServerUrl:http://itcast:itcast123@127.0.0.1:6868/eureka/}

itcasst-microservice-item:

ribbon:

NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule

### 导入spring boot打包插件

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

### 制作Docker镜像

首先将itcast-microservice-order.jar文件上传到linux系统中的/tmp/itcast-microservice-order目录。

编写Dockerfile文件：

FROM java:8

COPY ./itcast-microservice-order.jar /itcast-microservice-order/itcast-microservice-order.jar

COPY ./app-entrypoint.sh /

RUN chmod +x /app-entrypoint.sh

ENTRYPOINT ["/app-entrypoint.sh"]

app-entrypoint.sh文件：

#!/bin/bash

java -jar -Dport=$PORT -DeurekaServerUrl=$EUREKA\_SERVER\_URL /itcast-microservice-order/itcast-microservice-order.jar

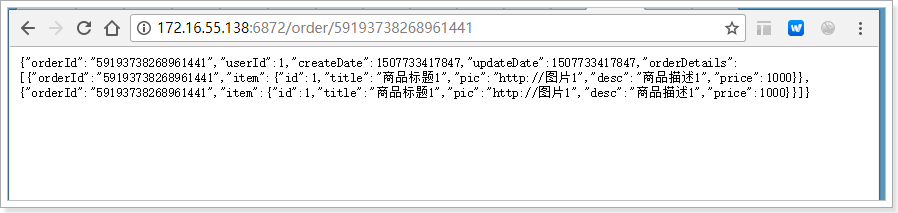
构建：

docker build -t itcast-microservice-order:1.0.0 .

### 运行容器

docker create --name order-server-1 -t -p 6872:6872 -e PORT=6872 -e EUREKA\_SERVER\_URL=http://itcast:itcast123@172.16.55.138:6868/eureka/ itcast-microservice-order:1.0.0

docker start order-server-1 && docker logs -f order-server-1



## 部署itcast-microservice-api-gateway-1

### 修改配置文件

server:

port: ${port:6873} #服务端口

spring:

application:

name: itcasst-microservice-api-gateway #指定服务名

zuul:

routes:

item-service: #item-service这个名字是任意写的

path: /item-service/\*\* #配置请求URL的请求规则

#url: http://127.0.0.1:8081 #真正的微服务地址

serviceId: itcasst-microservice-item #指定Eureka注册中心中的服务id

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: ${eurekaServerUrl:http://itcast:itcast123@127.0.0.1:6868/eureka/}

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

ip-address: ${ipAddress:127.0.0.1}

### 导入spring boot打包插件

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

### 制作Docker镜像

首先将itcast-microservice-api-gateway.jar文件上传到linux系统中的/tmp/ itcast-microservice-api-gateway目录。

编写Dockerfile文件：

FROM java:8

COPY ./itcast-microservice-api-gateway.jar /itcast-microservice-api-gateway/itcast-microservice-api-gateway.jar

COPY ./app-entrypoint.sh /

RUN chmod +x /app-entrypoint.sh

ENTRYPOINT ["/app-entrypoint.sh"]

app-entrypoint.sh文件：

#!/bin/bash

java -jar -Dport=$PORT -DeurekaServerUrl=$EUREKA\_SERVER\_URL -DipAddress=$IP\_ADDRESS /itcast-microservice-api-gateway/itcast-microservice-api-gateway.jar

构建：

docker build -t itcast-microservice-api-gateway:1.0.0 .

### 运行容器

docker create --name api-gateway-1 -t -p 6873:6873 -e PORT=6873 -e IP\_ADDRESS=172.16.55.138 -e EUREKA\_SERVER\_URL=http://itcast:itcast123@172.16.55.138:6868/eureka/ itcast-microservice-api-gateway:1.0.0

docker start api-gateway-1 && docker logs -f api-gateway-1

