



**91.git常见命令和作用**

* git init  //初始化本地git环境
* git clone XXX  //克隆一份代码到本地仓库
* git pull  //把远程库的代码更新到工作台
* git pull --rebase origin master  //强制把远程库的代码跟新到当前分支上面
* git fetch  //把远程库的代码更新到本地库
* git add .  //把本地的修改加到stage中
* git commit -m 'comments here'  //把stage中的修改提交到本地库
* git push  //把本地库的修改提交到远程库中
* git branch -r/-a  //查看远程分支/全部分支
* git checkout master/branch  //切换到某个分支
* git checkout -b test  //新建test分支
* git checkout -d test  //删除test分支
* git merge master  //假设当前在test分支上面，把master分支上的修改同步到test分支上
* git merge tool  //调用merge工具
* git stash  //把未完成的修改缓存到栈容器中
* git stash list  //查看所有的缓存
* git stash pop  //恢复本地分支到缓存状态
* git blame someFile  //查看某个文件的每一行的修改记录（）谁在什么时候修改的）
* git status  //查看当前分支有哪些修改
* git log  //查看当前分支上面的日志信息
* git diff  //查看当前没有add的内容
* git diff --cache  //查看已经add但是没有commit的内容
* git diff HEAD  //上面两个内容的合并
* git reset --hard HEAD  //撤销本地修改，版本回滚

**92.简述以下git中stash命令作用以及相关其他命令**

'git stash'：将当前工作区所有修改过的内容存储到“某个地方”，将工作区还原到当前版本未修改过的状态；

'git stash list'：查看“某个地方”存储的所有记录；

'git stash clear'：清空“某个地方”；

'git stash pop'：将第一个记录从“某个地方”重新拿到工作区（可能有冲突）；

'git stash apply'：编号, 将指定编号记录从“某个地方”重新拿到工作区（可能有冲突）；

'git stash drop'：编号，删除指定编号的记录

**93.git 中 merge 和 rebase命令 的区别**

merge：

会将不同分支的提交合并成一个新的节点，之前的提交分开显示，注重历史信息、可以看出每个分支信息，基于时间点,遇到冲突,手动解决,再次提交。

rebase：

将两个分支的提交结果融合成线性，不会产生新的节点;

注重开发过程，遇到冲突，手动解决，继续操作

**94.公司如何基于git做的协同开发？**

1、你们公司的代码review分支怎么做？谁来做？

答：组长创建review分支，我们小功能开发完之后，合并到review分支交给老大（小组长）来看。

1.1、你组长不开发代码吗？

他开发代码，但是它只开发核心的东西，任务比较少

或者抽出时间，我们一起做这个事情。

2、你们公司协同开发是怎么协同开发的？

每个人都有自己的分支，阶段性代码完成之后，合并到review，然后交给老大看。

--------------------------------------------------------------------------

# 大致工作流程：

在公司：

下载代码

git clone https://gitee.com/wupeiqi/xianglong.git 或创建目录

cd 目录

git init

git remote add origin https://gitee.com/wupeiqi/xianglong.git

git pull origin maste

创建dev分支  
　　　　 git checkout -b dev

git checkout dev

git pull origin dev

继续写代码

git add .

git commit -m '提交记录'

git push origin dev

回到家中：

拉代码：

git pull origin dev

继续写：

继续写代码

git add .

git commit -m '提交记录'

git push origin dev

**95.如何基于git实现代码review？**

https://blog.csdn.net/june\_y/article/details/50817993

**96.git如何实现v1.0、v2.0等版本的管理？**

在命令行中，使用git tag –a tagname –m 'comment'可以快速创建一个标签。

需要注意，命令行创建的标签只存在本地Git库中，还需要使用Git push –tags指令发布到服务器的Git库中。

**97.什么是gitlab**

gitlab是公司自己搭建的项目代码托管平台。

**98.github和gitlab的区别？**

1、gitHub是一个面向开源及私有软件项目的托管平台(创建私有的话，需要购买，最低级的付费为每月7刀，支持5个私有项目)；

2、gitlab是公司自己搭建的项目托管平台

**99.如何为github上牛逼的开源项目贡献代码？**

1、fork需要协作项目

2、克隆/关联fork的项目到本地

3、新建分支（branch）并检出（checkout）新分支

4、在新分支上完成代码开发

5、开发完成后将你的代码合并到master分支

6、添加原作者的仓库地址作为一个新的仓库地址

7、合并原作者的master分支到你自己的master分支,用于和作者仓库代码同步

8、push你的本地仓库到GitHub

9、在Github上提交 pull requests

10、等待管理员（你需要贡献的开源项目管理员）处理

**100.git中 .gitignore文件的作用**

一般来说每个Git项目中都需要一个“.gitignore”文件，这个文件的作用就是告诉Git哪些文件不需要添加到版本管理中。

实际项目中，很多文件都是不需要版本管理的，比如Python的.pyc文件和一些包含密码的配置文件等等。

**101.什么是敏捷开发？**

'敏捷开发'：是一种以人为核心、迭代、循序渐进的开发方式。

它并不是一门技术，而是一种开发方式，也就是一种软件开发的流程。

它会指导我们用规定的环节去一步一步完成项目的开发。

因为它采用的是迭代式开发，所以这种开发方式的主要驱动核心是人

**102.简述 jenkins 工具的作用?**

'Jenkins'是一个可扩展的持续集成引擎。

主要用于：

持续、自动地构建/测试软件项目。

监控一些定时执行的任务。

**103.公司如何实现代码发布？**

[nginx+uwsgi+django](https://www.cnblogs.com/wupeiqi/articles/8591782.html) 、<https://www.cnblogs.com/wcwnina/p/9906081.html>

**Git冲突解决**

**导语**

* 只要面试官问一些GIt相关问题，是辨别真假程序员的有效手段之一，GIt提交、拉取、分支、冲突解决是在多人协作当中经常遇到的问题，不可能没有遇到！在多人协作开发项目的时候，不可避免的会遇到代码冲突，虽说我们每次再开发以前要拉取代码，但是冲突还是不能彻底避免，这个时候就需要我们的小伙伴熟练掌握解决冲突的技能显得尤为重要。

**产生冲突的场景**

* 一大早同事A来到办公室，打开电脑启动环境执行git pull（拉取代码，修改代码之前一定要先拉取养成良好的习惯，尽可能的减少代码冲突），开始修改项目的首页的Bug，啪啪啪...
* 一会儿同事B气喘吁吁的坐下来，同样的开机启动环境git pull，然后同样去修改首页的Bug，啪啪啪...
* 看似风平浪静，但是掐指一算，一会儿即将可能有大事要发生，等待中...
* 同事A事先修改完了Bug，git push到了远程服务器，然后继续去修改其它Bug，同事B也修改完了，执行了git push，然后高高兴兴去厕所了，回来以后一看小黑窗：

**报.......错......了.......**

* 推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突不要慌，解决办法我给大家介绍两种，酌情选择

**解决方案一（强制覆盖本地代码，你自己修改的代码即将被远程库的代码所覆盖）**

* 把你修改的代码进行备份，然后执行命令：

1. git reset --hard origin/master
2. git pull
3. 从你备份好的文件当中把你写的代码拿过去，修改完成再进行git push

**解决方案二**

* （合并远程库和你本地的代码）执行命令：

1. git fetch 更新远程仓库文件
2. git diff master origin/master 本地文件和远程仓库文件进行对比
3. git merge origin/master 自动合并
4. 根据需求手动删除不必要的代码，修改完成git push到远程仓库

小命令：最后，补充一个我之前在工作当中经常使用的查看Git提交日志的命令，除了git log之外还有一个很好用“gitk”在小黑窗中执行一下，会有神奇的事情发生...

[Git冲突与解决方法](https://www.cnblogs.com/gavincoder/p/9071959.html)

1、git冲突的场景

* 情景一：多个分支代码合并到一个分支时；
* 情景二：多个分支向同一个远端分支推送代码时；

实际上，push操作即是将本地代码merge到远端库分支上。

关于push和pull其实就分别是用本地分支合并到远程分支 和 将远程分支合并到本地分支

所以这两个过程中也可能存在冲突。

git的合并中产生冲突的具体情况：  
　　<1>两个分支中修改了同一个文件（不管什么地方）  
　　<2>两个分支中修改了同一个文件的名称  
两个分支中分别修改了不同文件中的部分，不会产生冲突，可以直接将两部分合并。

2、冲突解决方法

* 情景一：在当前分支上，直接修改冲突代码--->add--->commit。
* 情景二：在本地当前分支上，修改冲突代码--->add--->commit--->push

 注：借用vim或者IDE或者直接找到冲突文件，修改。

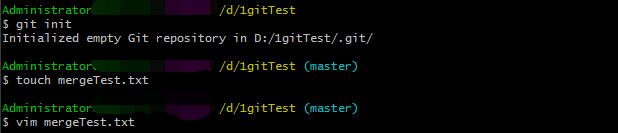
3、实战演示

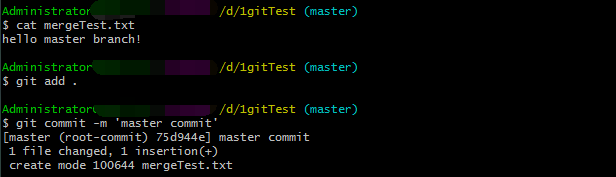
（1）情景

　　本地库中两个不同分支，修改同一个文件同一代码块，两分支先后将修改合并到master分支上，master在合并第二个分支代码时，报错：合并冲突。

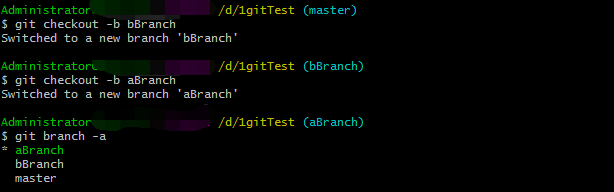
（2）本地库

<1>master分支



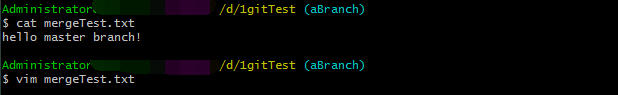


<2>建立两个分支



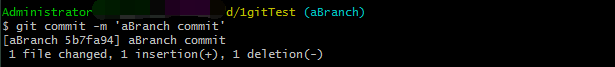
<3>两分支修改提交

aBranch分支：

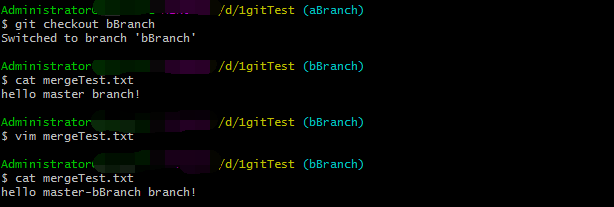




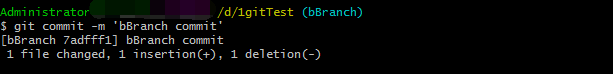




bBranch分支：

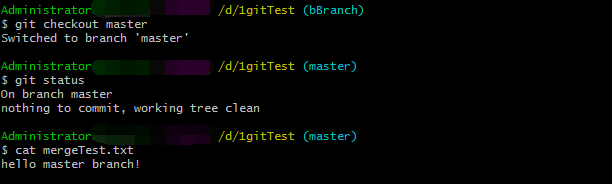


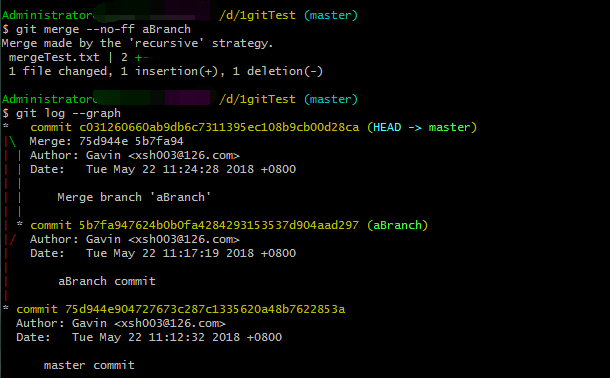




（3）合并分支产生冲突

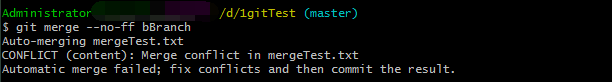
合并aBranch分支（将aBranch分支合并到当前master分支上）：



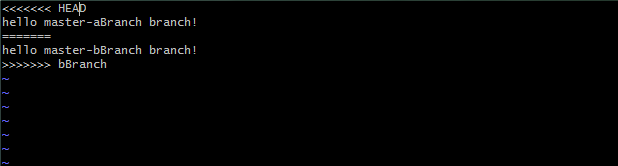


注：  
git merge：默认情况下，Git执行"快进式合并"（fast-farward merge），会直接将Master分支指向Develop分支。  
使用--no-ff参数后，会执行正常合并，在Master分支上生成一个新节点。为了保证版本演进的清晰，建议采用这种方法。

再合并bBranch分支，产生冲突：

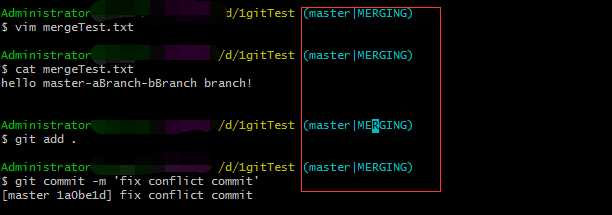


mergeTest.txt 文件内容：

****

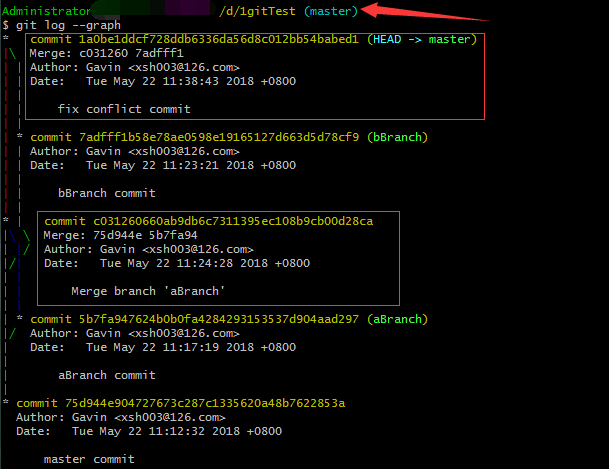
（4）解决冲突

--->在当前分支上（master），找到冲突文件，直接修改冲突代码，add，commit。



 注：简单方法，使用vim修改，cat查看冲突文件。（注意要删除git自动生成的冲突代码分隔符）

（5）完成冲突解决



注：提交或者合并都会生成git节点。每个节点对应一个代码版本。

**git的冲突原因及解决方法（笔记）**

我总结一下在工作中遇到的冲突及避免的办法，

**一、类似与import 引起的冲突**

我们的git 流程管理分为master，release，dev，feature，staging五大类分支，  
staging分支，主要用于测试。但是在feature合并staging分支的时候会出现冲突。

比如staging环境的版本

1. import java.text.SimpleDateFormat;
2. import java.util.Date;
3. import java.util.ArrayList;
4. import java.util.List;

feature\_A的版本

1. import java.text.SimpleDateFormat;
2. import java.util.Date;
3. import java.util.List;

feature\_B的版本

1. import java.text.SimpleDateFormat;
2. import java.util.ArrayList;
3. import java.util.List;

下面展示是如何造成冲突的：

这个时候feature\_B 加了2个import  
feature\_B的版本变为：

1. import java.text.SimpleDateFormat;
2. import org.slf4j.Logger;
3. import org.slf4j.LoggerFactory;
4. import java.util.ArrayList;
5. import java.util.List;

显然这个时候和staging的版本造成冲突：

1. import java.text.SimpleDateFormat;
2. import java.util.Date;
3. import java.util.ArrayList;
4. import java.util.List;

git合并的时候会进行three-way merge，  
这个时候有歧义性：  
1、第一种合并方案  
认为你的版本删除了feature\_A的

import java.util.Date;

又增加了两行

1. import org.slf4j.Logger;
2. import org.slf4j.LoggerFactory;

最终结果为：

1. import java.text.SimpleDateFormat;
2. import org.slf4j.Logger;
3. import org.slf4j.LoggerFactory;
4. import java.util.ArrayList;
5. import java.util.List;

2、第二种合并方案，  
即不删除feature\_A的

import java.util.Date;

最终合并方案为：

1. import java.text.SimpleDateFormat;
2. import java.util.Date;
3. import org.slf4j.Logger;
4. import org.slf4j.LoggerFactory;
5. import java.util.ArrayList;
6. import java.util.List;

避免这种情况的最好方法是import总是放在自己的代码中间，import不要交错放。禁止使用代码格式化工具，尽量手动格式化。

**二，改动他人代码引起冲突：**

另外一种冲突的原因是因为改动了他人的代码，这个是很危险的，非常容易引起冲突和bug。

对于idea，有个annotation功能可以查看每行代码是谁提交的。

image.png(图片略)

右键行号，选择annotate就能打开如上图所示效果。

**3、文件引起的冲突**

因为共同创建了一个同名文件，对方删除了我们正在使用的文件。也会引起冲突，叫做树冲突。

比如，a用户把文件改名为a.c，b用户把同一个文件改名为b.c，那么b将这两个commit合并时，会产生冲突。  
$ git status  
added by us: b.c  
both deleted: origin-name.c  
added by them: a.c  
如果最终确定用b.c，那么解决办法如下：  
git rm a.c  
git rm origin-name.c  
git add b.c  
git commit  
执行前面两个git rm时，会告警“file-name : needs merge”，可以不必理会。

树冲突也可以用git mergetool来解决，但整个解决过程是在交互式问答中完成的，用d 删除不要的文件，用c保留需要的文件。  
最后执行git commit提交即可。

[git 冲突是怎么产生的](https://www.cnblogs.com/xuezizhenchengxuyuan/p/6439782.html)

首先,一个用git 写代码,而且只有一个本地分支的情况下是不会又冲突的.

冲突可以说是两个分支的冲突.具体是两个已经提交的分支的相同文件相同位置的的不同操作进行了合并.

不会冲突的习惯是,修改文件之前先merge 别的分支.

我在master 分支上创建并提交一个文件,切换到新的b分支上是没有这个文件的.这说明分支之间是相互独立的.

通过git merge master 把master上新增的文件给merge 过来.这是不会又冲突的.0+x = x

如果我在新的分支b上把master 对应行的数据给改了,并提交.然后切换到master,我再git merge b -m 'from b' 也不会冲突,结果是被b分支的修改给覆盖了.

冲突的原因:

如果我切换到分支b ,修改了master 原本哪行的代码,提交.然后再切换到master.我不知道b修改了这个文件.我也修改这一行的代码,并提交.好了,"两个分支相同文件相同位置的的不同操作"    我这个时候就在master merge b 或在 b上merge master. 冲突!!! 不解决冲突是没法提交和切换到别的分支的. 反正这个冲突后的文件必须被改动一下.

**自动合并:**

上面冲突的原因大字体的句子改为,我不知道b修改了这个文件.我[我在我自己的代码上做了修改,b并没有动我代码]的代码,并提交,然后去合并b ,不会冲突,即便我没有先merge b.这种情况是自动合并.

**如何解决冲突:**

冲突发生的时候,最好能联系一下开发的人员,一起解决冲突.

一般情况下冲突后的文件会是:

<<<<<<<<HEAD

other code

========

your code

>>>>>>>>your branch name

解决的一般办法是:仔细对比,取综合的并集,就是尽量把所有不同的文件保留,共同的只留一份

个人的一点经验:多人协作开发的时候,如果出现了你没有改过的文件跟你冲突了,一定要去找到当事者,说清楚是如何冲突,然后协商解决,千万不要擅自拉别的分支去试图解决冲突,或找文件覆盖.同时记住,解决了之后,要add 和 commit 最后push.为保证万无一失,最后在冲突都解决之后,重启项目,特别是指服务器项目,保证至少不会有立即奔溃的现象发生.然后才去提交,push.

提交的时候,一定要保持清醒,先搞清楚自己要提交的文件之间的关系,然后再提交,这样才不会有文件缺失的问题,造成奔溃.

如果任务比较多,又建议开多个分支,分别进行开发.还是老话,一定要清楚自己在各个分支上做了什么,自己要提交的是什么.最好是能做个详细的笔记.好记性不如烂笔头.

最笨的办法:没有把握宁愿不要去提交生产机.

提交代码的时候不要走神,因为这是一个神圣的时刻.开发的时候你可以怎么测试都行,一旦上生产机,各种奇葩问题都会出现.

**git文件冲突合并的几种情况？**[**git**](http://www.iteye.com/blogs/tag/git)[**冲突解决**](http://www.iteye.com/blogs/tag/%E5%86%B2%E7%AA%81%E8%A7%A3%E5%86%B3)**？**

本文描述了git冲突的几种常见情况和解决方案，老鸟请直接忽略本文。  
假设冲突文件是 test/TestCase.php  
  下面分5种情况讨论。  
1、本地不变。然后远程别人有更新。git pull，这种最简单，没有冲突，本地工作区直接更新  
2、我本地修改，但是不add。然后远程别人有更新，此时 ：      
git pull，  
git会告诉你：  
error: Your local changes to the following files would be overwritten by merge:  
        tests/TestCase.php  
  此时，我   
   git checkout -- tests/TestCase.php  
  千万注意，上条命令会导致你自己的修改丢失了！！  
  然后git pull.成功。  
  这种情况下，你自己的修改完全丢失。本地接受了远程的修改。  
3、我本地修改，但是已经add了。  
  然后别人远程更新，  
  git pull  
  这种情况，和上面几乎一样。  
  git reset HEAD test/TestCase.php   
  上面这条命令可以理解为add命令的逆命令。即，取消add操作。  
  然后继续  
  git checkout -- tests/TestCase.php  
  千万注意，上条命令会导致你自己的修改丢失了！！  
  然后git pull.成功。  
  这种情况下，你自己的修改完全丢失。 本地接受了远程的修改。  
4、我本地修改，add 且 commit了  
  然后别人远程更新，  
  git pull  
  注意：这种情况和第2，第3种情况不同。git会认为你的commit也很重要。同等重要。  
突然发现，自动进入vi界面。  
Merge branch 'master' of https://github.com/xxxx  
这什么意思呢？就是git会主动再帮你添加一个commit。  
先让你写点注释，并且它已经帮你写了几句话  
那自己写点东西，然后保存退出vi  
请注意，这个commit是在本地的，还没有发到远程。  
这个commit会自动合并 你添加到代码，和 别人远程更新的代码，  
所以，之后请自己检查一下这个代码有无问题。  
  如果都没有问题，则你现在  
  git push  
  这个命令可以把你的本地改动推送到远程。  
  push之后，远程得到你和他人的所有更新。push之前，仅本地得到你和他人的所有更新。  
5、使用分支功能。  
  git的分支功能很强大，应该多使用。  
  上面几种情况都没有使用分支。使用分支会更好。  
  首先，我本地新建并切换分支branch\_1，自己起有意义名字，并修改文件。  
  git checkout -b branch\_1  
  vi test/TestCase.php  
  git add tests/TestCase.php  
  git commit -m "change test"  
  现在，别人已经更新了远程主分支代码（没更新就极其方便了，也不必说了，快速更新）  
  我想把分支合并到主分支。  
  我应该，先回到主分支，并更新。  
  git  checkout master  
  git pull  
  然后，在本地，把我的修改分支合并上去。  
  git merge branch\_1  
  自动进入vi界面，让你写注释。  
  然后会合并好。  
  类似提示：  
  tests/TestCase.php | 3 ++-  
  上面，两个加号表示文件增加了两行，一个减号表示文件减少了一行。  
  此时，注意，凡是进入vi界面后，  
  最好自己再检查一下程序，检查那些被提示修改过的文件，（这是一个好习惯，但我从来不遵守）  
  如果认为正确，最后，  
  git push,推送到远程。  
  push之后，远程得到你和他人的所有更新。push之前，仅本地得到你和他人的所有更新。  
总结，大部分情况下，都推荐使用第5种，使用分支的方法，来管理文件修改冲突。

[git 版本回退](https://www.cnblogs.com/xuezizhenchengxuyuan/p/6439245.html)

有些情况是没法用git 来回退的:

新建了文件,新建了文件并编辑,新增了文件并删除  总之是没有用git add file 命令把文件存到暂存区中.上面的文件只是普通的存放在了硬盘上,所有的改动都没有记录.

git add 后的文件另外存了一份(存的其实是不同的部分) 到暂存区  所以这是可以回退版本的基本条件.

git add 到暂存区之后的文件又被在工作区编辑了,如果要回退到**编辑之前**的样子,或称丢弃这次工作区的编辑: git checkout -- file

如果是git add 到了暂存区,但是又不想提交了,这个时候可以  分两步:git reset HEAD file   先把暂存区的改变回退到上一个版本  然后    git checkout -- file

如果是已经commit 但还没有push 到远程库,可以 git log --pretty=oneline 查看提交的版本号,然后根据  git reset commit\_id 来回退.

**记住以后git fetch 之后 最好有个习惯,那就是用git log git diff git status 看看,然后再决定是否merge.**

**如果已经提交了,那只能是先备份后提交,用push 的方式来进行删除.或者找到提交历史,仔细的找到关联的文件,清楚后再进行整体回退.**

**部署模块**

**Nginx**

**01.什么是Nginx?**

Nginx是一个高性能代理服务，接收客户端发送过来的HTTP请求和websocket请求，响应静态文件请求和转发动态请求。

**02.负载均衡什么意思？**

负载均衡（Server Load Balancer）是将访问流量根据转发策略分发到后端多台服务器的流量分发控制服务。负载均衡扩展了应用的服务能力，增强了应用的可用性。

**03.如何部署Django项目？**

1. Nginx+uWSGI+Django
2. Nginx+Tornado+Django

**04.为什么正式部署时不要开启DEBUG = True配置？**

因为对于一个在线网站, 将路由暴露出来, 是一件非常危险的事情, 所以我们要关掉django的debug模式

了解虚拟化技术吗？容器这些 答：docker比较擅长的。

虚拟化技术与容器技术

作为一种部署应用程序的全新方式，容器技术在短时间内获得了广泛关注，但是这种技术同样存在某些限制，并且和虚拟机存在本质上的不同。

       虚拟化技术已经改变了现代计算方式，它能够提升系统资源使用效率、消除应用程序和底层硬件之间的依赖关系，同时加强负载的可以移植性和安全性，但是hypervisor和虚拟机只是部署虚拟负载的方式之一。作为一种能够替代传统虚拟化技术的解决方案，容器虚拟化技术凭借其高效性和可靠性得到了快速发展，它能够提供新的特性，并且帮助数据中心专家解决新的顾虑。

　　容器和虚拟机之间的主要区别在于虚拟化层的位置和操作系统资源的使用方式。

　　虚拟机依赖于hypervisor，其通常被安装在“裸金属”系统硬件之上，这导致hypervisor在某些方面被认为是一种操作系统。一旦 hypervisor安装完成， 就可以从系统可用计算资源当中分配虚拟机实例了，每台虚拟机都能够获得唯一的操作系统和负载(应用程序)。

　　通常，第一台虚拟机都将会被用来运行系统管理程序，比如Microsoft System Center。而之后的虚拟机可能包含其他企业负载，比如数据库、ERP、CRM、邮件服务器、媒体服务器、web服务器或者其他业务应用。虚拟机之间是完全隔离的——没有任何一台虚拟机知道(或者依赖)相同系统当中的另外一台虚拟机的存在——恶意软件、应用程序崩溃和其他问题只能影响一台虚拟机。虚拟机从一个虚拟系统被移动到另外一个当中，而不用考虑系统硬件和操作系统等因素。

　　而容器环境的工作方式则有所不同。对于容器环境来说，需要首先安装主机操作系统，之后将容器层(比如LXC或libcontainer)安装在主机操作系统(通常是Linux变种)之上。在安装完容器层之后，就可以从系统可用计算资源当中分配容器实例了，并且企业应用可以被部署在容器当中。但是，每个容器化应用都会共享相同的操作系统(单个主机操作系统)。

　　相比于虚拟机，容器拥有更高的资源使用效率，因为它并不需要为每个应用分配单独的操作系统——实例规模更小、创建和迁移速度也更快。这意味相比于虚拟机，单个操作系统能够承载更多的容器。云提供商十分热衷于容器技术，因为在相同的硬件设备当中，可以部署数量更多的容器实例。然而，单个操作系统有可能引起影响所有相关实例的单点事故。比如，恶意软件或者主机操作系统崩溃可能禁用或者影响所有容器。此外，容器易于迁移，但是只能被迁移到具有兼容操作系统内核的其他服务器当中，这样会无形中减少迁移选择。

　　从实际应用的角度来说，容器和虚拟机可以在同一个数据中心当中共存，因此这两种技术被认为是互补的——为现代应用程序架构师和数据中心管理员添加了可用工具集，通过不同的方式为应用负载提供支持。

**今日内容：Docker**

课件地址：<https://www.cnblogs.com/xiaoyuanqujing/articles/11774978.html>

**一 docker介绍**

云服务器厂商：阿里云，腾讯云，新睿云，

阿里云：阿里飞天

小的云服务厂商：openstack，python写的

虚拟机：虚拟化的一种

**Docker**

基于go语言实现的

二次开发：在原有的系统上进行开发，原有系统源码拿不到

docker 小历史：业余项目，对外收费，卖软件，软件卖不出去，没人用，公司濒临倒闭，把docker开源了，出了两个版本，一个是社区版本（免费），一个是企业版（收费）

ERP厂商 ，北用友，南金蝶，浪潮国际 ：c#，python odoo

售前技术工程师 售后技术支持工程师

docker 实现轻量级的操作系统虚拟化，基于linux操作系统上的lxc实现的

没有docker之前，项目上线

开发完成--代码由运维人员从git上来下来---在服务器上装环境（python，mysql，reidis，环境版本问题）---运行项目

如果有了docker

开发完成--代码由运维人员从git上来下来---docker镜像（不再需要重复的安装软件，解决的就是软件版本不一致的问题）---运行

docker 更轻量级

启动速度快，占用体积小，一个centos7的系统，只需要几十个m（没有操作系统）

在操作系统之上通过lxc技术，实现虚拟化出操作系统

docker 是一个软件，通过这个软件虚拟化出操作系统

django、flask 同步框架

tornado，sanic异步框架:一旦异步，以后全都得用异步

asyncio 提前看一下，有兴趣的同学，学习一下

**二 docker架构和使用**

docker 是一个软件，c/s架构的软件（mysql，redis）

但是，mysql客户端和服务端通信时，基于socket，自己定制的协议，pymysql就是个mysql的客户端（跟navicate，go语言的连接mysql的代码和java连接mysql的代码都是一个东西）

docker 客户端跟服务端通信，是通过http协议，resful规范（新的软件，基本上都是走http协议，resful规范），es

**重点 ：镜像和容器**

基于镜像来运行容器：镜像是面向对象中的类，容器是面向对象中的对象

注册中心：docker 是一个软件，需要虚拟化出操作系统，操作系统从哪来？从注册中心拉下来的，从注册中心来下来的东西，叫镜像（比如要拉一个centos7的镜像），镜像运行起来，叫容器（才是真正的跑起来的操作系统）

一个centos7镜像，跑起来两个容器，在一个容器上装了mysql，在另一个上装了python3.6

**2.1docker 安装**

windows（win10及以上，下一个docker软件，下一步安装就可以了）/linux(乌班图，centos)/Mac

乌班图上很简单，因为docker就是在乌班图上开发的

在centos上安装，要在7.x以上，在上面装docker

mac本质就是unix操作系统，在终端连接

ssh root@101.133.225.166

**2.2 docker 安装**

sudo yum update

sudo yum install -y yum-utils device-mapper-persistent-data lvm2

sudo yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

sudo yum install docker-ce

#docker软件安装完成，服务端和客户端都安装完成，

docker -v #查看客户端版本

**2.3 启动docker**

docker 安装完，服务端没有启动，需要启动

systemctl start docker # 7.x用这个命令启动

systemctl stop docker #停止docker服务端

# 了解

启动docker：

systemctl start docker

停止docker：

systemctl stop docker

重启docker：

systemctl restart docker

查看docker状态：

systemctl status docker

开机启动：

systemctl enable docker

查看docker概要信息

docker info

查看docker帮助文档

docker --help

**2.4 镜像操作**

查找镜像：docker search centos，会去<https://hub.docker.com/>查找

直接去该地址查找即可

镜像名称，是否是官方镜像，描述，stars数

拉去centos7的版本镜像

docker pull centos:centos7

docker pull centos:8

docker pull python:3.6 其实下载下什么东西？镜像，其实是一堆文件，

那centos7和python的区别是什么？如果下载python3.6 ：就是一个操作系统上安装了python环境，从官方拉，是乌班图+python

docker pull mysql:5.6  下载了一个镜像，镜像里安装了mysql，就相当于操作系统上安装了mysql

**查看服务端所有的镜像**

docker images

**删除镜像 (容器如果没有删除，镜像删除不了)**

docker rmi id号 id号可以缩写

docker rmi 5e3

**2.5 容器操作**

镜像运行，是启动容器，一个个的容器，就是一个个的操作系统

一个centos7的镜像，可以跑起多个容器，每一个容器就是一个操作系统

**查看容器**

docker ps：查看正在运行的容器

docker ps –a：查看所有容器（包括停止和运行的）

**创建启动容器**

docker run -it --name=mycentos centos:centos7 /bin/bash

exit 退出容器,容器也就停止了

docker run -di --name=mycentos2 centos:centos7 不进入容器 内部

docker exec -it mycentos2 /bin/bash 进入到容器内容

exit退出，容器不停止

docker stop id号或者名字 容器停止

docker start id号或者名字 启动容器

讲一个项目 答：cmdb

怎么起一个容器 答：docker start 容器。

进入Docker容器比较常见的几种做法如下：  
1.使用docker attach  
2.使用SSH  
3.使用nsenter

4.使用exec

怎么新建一个容器 答：。。。。

容器的概念：

独立运行的一个应用 和一组应用

以及这些应用对应的环境。

docker启动有两种方式。

a 基于镜像

docker run centos cal

启动这个centos 打印日历表。

也可以使用

docker run -it centos /bin/bash

会看到一个交互式的客户端

将终止状态下的容器重新启动

docker start centos

在守护状态下运行

docker run -d centos /bin/bash -c “hello wordl;sleep 1;done’

-d 在后台运行一个容器。

docker logs 容器ID

查看某个容器的日志输出。

如何终止一个容器的运行呢？

docker ps -a 查看所有的容器

可以根据容器的ID 进行 stop restart 等操作

终止之后 可以使用 docker ps -a 查看 是否存在呢

**Docker昨日回顾：**

1 介绍docker，一种虚拟化的技术，比虚拟机更轻量级，启动速度更快，基于linux的lxc容器技术，go语言编写，社区版（docker-ce），企业版

2 企业应用比较广泛，方便

3 安装：7.x以上

yum install -y yum-utils device-mapper-persistent-data lvm2

sudo yum-config-manager --add-repo <http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo>

yum install docker-ce

**4 启动停止**

docker -v：客户端的版本

systemctl start docker

systemctl stop docker

**5 镜像操作**

注册中心拉下来

docker search 名字 搜索镜像

docker pull 镜像名字:镜像的tag 拉取镜像，在docker服务端就可以查看到

docker images 查看所有镜像

docker rmi 镜像id 删除镜像

**6 容器操作**

docker ps 查看正在运行的容器

docker ps -a 查看所有容器，包括停止的

docker run -it --name=mycentos -p 80:80 -v 宿主机目录:容器目录 centos:7 /bin/bash 一旦退出容器，容器也就停止了

docker run -id --name=mycentos -p 80:80 -v 宿主机目录:容器目录 centos:7 不进入容器，可以选择进去

docker exec -it mycentos /bin/bash 进入容器内部

exit 出来

docker stop 容器名/容器id 停止容器

docker start 容器名/容器id 启动容器

**1拷贝文件**

1 docker cp 1.txt mycentos2:/home 从宿主机拷贝文件到容器内部

2 docker cp mycentos2:/home/1.txt /home/1.txt 从容器往外拷贝，不需要进入容器内部

**2 目录挂载**

docker run -di -v /home/lqz:/home --name=mycentos3 centos:centos7

**3 删除容器**

docker rm 容器id/名字 删除容器

**4 查看容器的ip地址**

1 查看容器详情：docker inspact 容器名字/id NetworkSettings下的IPAddress是容器的ip地址

2 查看容器ip docker inspect --format='{{.NetworkSettings.IPAddress}}' 容器名称（容器ID）

**4 服务部署**

**部署mysql**

拉取：docker pull mysql:5.6

启动：docker run -di --name=mysql -p 3306:3306 -e MYSQL\_ROOT\_PASSWORD=123456 mysql:5.6

远程连接宿主机的3306端口，即可连接到容器的mysql服务

**部署redis**

docker pull reids 不加tag ，拉最新的版本，也就是lasted版本

docker run -di --name=myredis -p 6379:6379 redis 启动redis

远程连接即可

**5 迁移与备份**

**将容器保存为镜像**

docker commit 容器名字 要打包成的镜像名字

docker commit mycentos2 lqz\_centos7

基于打包好的镜像，再跑起容器来，那么容器内部原来装的软件，都会有

docker run -di --name=lqz\_centos7\_1 lqz\_centos7:latest

启动起容器来，软件都会有

**镜像备份**

docker save -o lqz\_centos7.tar lqz\_centos7

**镜像恢复**

docker load -i lqz\_centos7.tar

**6 私有仓库**

1 docker pull registry 拉下一个镜像

2 docker run -di --name=registry -p 5000:5000 registry 启动镜像形成容器

3 打开浏览器 输入地址<http://101.133.225.166:5000/v2/_catalog> 没有上传镜像，是空的 {"repositories":[]}

**配置私有仓库**

vi /etc/docker/daemon.json

{"insecure-registries":["101.133.225.166:5000"]}

重启docker 服务

systemctl restart docker

重启registry的容器

docker restart registry

**上传到私有仓库**

打标签

docker tag lqz\_centos7 101.133.225.166:5000/lqz\_centos7

上传标记的镜像

docker push 101.133.225.166:5000/lqz\_centos7

本地删除镜像后，可以从私有仓库拉下来

docker pull 101.133.225.166:5000/lqz\_centos7

再跑起容器，内部就有aaa bbb文件夹

**dockerfile，部署一个小项目**

dockerfile：由一系列命令和参数构成的脚本，通过这个文件构建镜像，dockerfile就是一个文件，占得空间非常小。

dockerfile是一个文件+一堆命令

**命令**

FROM 镜像名字:镜像标签 FROM centos:centos7 指定基于哪个镜像

MAINTAINER 作者 作者是谁

ENV key value 环境变量

RUN 命令 要执行的命令

ADD 文件路径 拷贝文件到镜像内（自动解压）

COPY 文件路径 拷贝文件到镜像内（不会自动解压）

WORKDIR 路径 工作目录（工作的目录，一旦启动起容器来，进入，在的路径）

通过dockerfile，构建一个django 1.11.9的镜像，内部装上requests模块,在home目录下创建一个test文件夹,传一个a.txt到镜像中

FROM python:3.6 MAINTAINER lqz

RUN pip install django==1.11.9

RUN mkdir /home/test

RUN pip install requests

ADD a.txt /hom

WORKDIR /home

在当前路径下创建一个a.txt，dockerfile的名字必须是Dockerfile

**构建：docker build -t='镜像名字' .**

docker build -t='django1.11.9' .

**通过docker images 就可以查看构建的镜像**

**通过镜像，跑起容器**

docker run -id --name=django1.11.9\_my django1.11.9:latest

**进入到容器内部**

docker exec -it django1.11.9\_my /bin/bash

**案例一：通过容器部署项目**

通过uwsgi部署到容器中，起一个nginx容器，做转发

**1 先把文件传到服务器解压**

**2 启动一个容器，做目录挂载**

docker run -id -p 8088:8088 -v /root/djangotest:/home --name=my\_django\_test python:3.6

**3 进入容器**

docker exec -it 92c /bin/bash

**4 安装django环境**

pip install django==1.11.9

**5 运行项目**

python manage.py runserver 0.0.0.0:8088 用的是wsgiref

**6 通过uWSGI跑项目**

安装uwsgi：pip install uwsgi

写入：

vim uwsgi.ini #写入 [uwsgi] #配置和nginx连接的socket连接 socket=0.0.0.0:8080

#也可以使用http #http=0.0.0.0:8080

#配置项目路径，项目的所在目录 chdir=/home/untitled3

#配置wsgi接口模块文件路径 wsgi-file=untitled3/wsgi.py

#配置启动的进程数 processes=4

#配置每个进程的线程数 threads=2

#配置启动管理主进程 master=True

#配置存放主进程的进程号文件 pidfile=uwsgi.pid

#配置dump日志记录 daemonize=uwsgi.log

启动

uwsgi --ini uwsgi.ini #启动

**7 用nginx转发**

docker pull nginx

mkdir -p nginx/conf nginx/html nginx/logs

#新建配置文件 在conf目录下新建nginx.conf

worker\_processes 1;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type application/octet-stream;

sendfile on;

keepalive\_timeout 65;

server {

listen 80;

server\_name localhost;

location / {

#方式一

#include uwsgi\_params; # 导入一个Nginx模块他是用来和uWSGI进行通讯的

#uwsgi\_connect\_timeout 30; # 设置连接uWSGI超时时间

#uwsgi\_pass 101.133.225.166:8080;

#方式二

#include uwsgi\_params; # 导入一个Nginx模块他是用来和uWSGI进行通讯的

#uwsgi\_pass unix:///var/www/script/uwsgi.sock; # 指定uwsgi的sock文件所有动态请求

#方式三

proxy\_pass http://101.133.225.166:8088

}

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

}

}

docker run --name mynginx -id -p 80:80 -v /root/nginx/conf/nginx.conf:/etc/nginx/nginx.conf -v /root/nginx/html:/etc/nginx/html -v /root/nginx/logs:/var/log/nginx nginx

在外部浏览器访问宿主机ip地址80端口，就可以访问项目了

**补充1**

K8s:容器编排，夸服务器，多台服务器上的多个docker容器

docker-compos：单机容器编排，8个容器，同时启动，

**补充2**

scp 远程拷贝命令,从mac拷贝到了阿里云服务器的/root路径

scp untitled3.zip root@101.133.225.166:/root

解压文件：yum install zip yum install unzip

**补充3**

**在python3.6的容器中安装软件需要apt-get**

apt-get update

apt-get install vim

**补充4 cgi wsgi**

cgi：通用网关接口

wsgi：python的协议，Web服务网管接口，简单来说它是一种Web服务器和应用程序间的通信规范。

wsgiref，uWSGI（c语言实现的，性能比较高），gunicorn。。。：根据这个协议的实现

java、php都会有一些协议

tomcat就是python中的uwsgi

php服务器

作业：

编写一个dockerfile，实现镜像具有django1.11.9

通过docker部署nginx和项目，实现uwsgi运行项目，外网访问

下午从网上找几份简历截图，发给大家，里面有好的，有不好的，

**1 dockerfile 的编写**

**1.1一个文件，文件名字必须叫Dockerfile，一系列命令的集合**

FROM 镜像名字:镜像标签 FROM centos:centos7 指定基于哪个镜像

MAINTAINER 作者 作者是谁

ENV key value 环境变量

RUN 命令 要执行的命令

ADD 文件路径 拷贝文件到镜像内（自动解压）

COPY 文件路径 拷贝文件到镜像内（不会自动解压）

WORKDIR 路径 工作目录（工作的目录，一旦启动起容器来，进入，在的路径）

EXPOSE 8080 容器对外暴露的端口

CMD 命令 一旦启动容器，会执行该条命令，这条命令，必须是夯住的命令

**1.2构建镜像**

docker build -t='镜像名字' .

**1.3 镜像构建出来后，就可以查看，可以基于镜像运行容器**

**2 通过容器部署项目（nginx+uwsgi+django）**

**2.1 项目传到宿主机上（从git上拉下来），启动了一个python3.6的容器，并且做了目录挂载和端口映射**

docker run -id -p 8088:8088 -v /root/djangotest:/home --name=my\_django\_test python:3.6

如果不做目录映射，需要把项目拷贝到容器内部（docker cp）

**2.3 进入到容器内部，安装模块**

docker exec 进入

pip install django==1.11.9

pip install -r requm.txt

python manage.py runserver 0.0.0.0:8088 能跑起来，再继续

**2.4 使用uwsgi运行项目**

pip install uwsgi （uwsgi是用c语言写的，用python做了封装）

在项目根路径创建一个uwsgi.ini的文件，写入

[uwsgi]  
#配置和nginx连接的socket连接  
socket=0.0.0.0:8080  
#也可以使用http  
#http=0.0.0.0:8080  
#配置项目路径，项目的所在目录，路径是容器的路径，部署宿主机的路径  
chdir=/home/untitled3  
#配置wsgi接口模块文件路径  
wsgi-file=untitled3/wsgi.py  
#配置启动的进程数  
processes=4  
#配置每个进程的线程数  
threads=2  
#配置启动管理主进程  
master=True  
#配置存放主进程的进程号文件  
pidfile=uwsgi.pid  
#配置dump日志记录  
daemonize=uwsgi.log

通过uwsgi启动

uwsgi --ini uwsgi.ini

uwsgi --stop uwsgi.pid #停止

kill -9 进程杀死

查看是否启动

ps aux |grep uwsgi

**2.5 配置nginx转发**

docker pull nginx

mkdir -p nginx/conf nginx/html nginx/logs

#新建配置文件 在conf目录下新建nginx.conf

worker\_processes 1;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type application/octet-stream;

sendfile on;

keepalive\_timeout 65;

server {

listen 80;

server\_name localhost;

location / {

#方式一

#include uwsgi\_params; # 导入一个Nginx模块他是用来和uWSGI进行通讯的

#uwsgi\_connect\_timeout 30; # 设置连接uWSGI超时时间

#uwsgi\_pass 101.133.225.166:8080;

#方式二

proxy\_pass http://101.133.225.166:8088

}

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

}

}

docker run --name mynginx -id -p 80:80 -v /root/nginx/conf/nginx.conf:/etc/nginx/nginx.conf -v /root/nginx/html:/etc/nginx/html -v /root/nginx/logs:/var/log/nginx nginx

在外部浏览器访问宿主机ip地址80端口，就可以访问项目了

**补充**

python manage.py collectstatic 收集静态文件，把所有的静态js，css，图片收集到一个文件夹中，把这个文件夹单独放到一个地址，用nginx转发，做动静分离

192.168.1.11/static/head.png

192.168.1.11/media/head.png

**1 dockerfile构建镜像**

**1.1 创建Dockerfile，在根路径下一定要有项目untitled3.tar**

**tar -cvf untitled3.tar untitled3**

**untitled3项目内部必须有个uwsgi.ini 最后一句的daemonize去掉**

注意：默认linux是解压不了zip格式的，需要安装unzip和zip

现在统一用的tar格式 ，linux就可以解压

FROM python:3.6

MAINTAINER lqz

WORKDIR /home

RUN pip install django==1.11.9

RUN pip install uwsgi

EXPOSE 8081

ADD ./untitled3.tar /home/

CMD ["uwsgi", "--ini", "/home/untitled3/uwsgi.ini"]

**1.2 构建镜像**

docker build -t='mydockerfile\_django' .

**1.3 查看镜像**

docker images 就可以看到mydockerfile\_django镜像

**1.4 通过镜像运行容器(不需要挂载目录，项目已经在镜像内部了)**

docker run -di --name=my\_dockerfile\_django -p 8081:8081 mydockerfile\_django:latest

项目就跑起来了

**数据库主从搭建**

**2.1 主从同步原理**

mysql主从配置的流程大体如图：

1）master会将变动记录到二进制日志里面；

2）master有一个I/O线程将二进制日志发送到slave;

3) slave有一个I/O线程把master发送的二进制写入到relay日志里面；

4）slave有一个SQL线程，按照relay日志处理slave的数据；

**2.2 注意点**

1 咱们用docker模拟了两台服务器，服务器的系统，mysql的版本必须一致

**2.3 具体步骤,启动主库**

docker pull mysql:5.7

#在home目录下创建mysql文件夹，下面创建data和conf.d文件夹

mkdir /home/mysql

mkdir /home/mysql/conf.d

mkdir /home/mysql/data/

# 创建my.cnf配置文件

touch /home/mysql/my.cnf

#写入

[mysqld]

user=mysql

character-set-server=utf8

default\_authentication\_plugin=mysql\_native\_password

secure\_file\_priv=/var/lib/mysql

expire\_logs\_days=7

sql\_mode=STRICT\_TRANS\_TABLES,NO\_ZERO\_IN\_DATE,NO\_ZERO\_DATE,ERROR\_FOR\_DIVISION\_BY\_ZERO,NO\_ENGINE\_SUBSTITUTION

max\_connections=1000

##主库----start--- 同一局域网内注意要唯一

server-id=100

## 开启二进制日志功能，可以随便取（关键）

log-bin=mysql-bin

##主库----end---

[client]

default-character-set=utf8

[mysql]

default-character-set=utf8

#启动主库容器（挂载外部目录，端口映射成33307，密码设置为123456）

docker run -di -v /home/mysql/data/:/var/lib/mysql -v /home/mysql/conf.d:/etc/mysql/conf.d -v /home/mysql/my.cnf:/etc/mysql/my.cnf -p 33307:3306 --name mysql-master -e MYSQL\_ROOT\_PASSWORD=123456 mysql:5.7

**2.4 启动从库**

#在home目录下创建mysql2文件夹，下面创建data和conf.d文件夹

mkdir /home/mysql2

mkdir /home/mysql2/conf.d

mkdir /home/mysql2/data/

#mkdir /home/mysql2 /home/mysql2/conf.d /home/mysql2/data/

# 创建my.cnf配置文件

touch /home/mysql2/my.cnf

[mysqld]

user=mysql

character-set-server=utf8

default\_authentication\_plugin=mysql\_native\_password

secure\_file\_priv=/var/lib/mysql

datadir=/var/lib/mysql

expire\_logs\_days=7

sql\_mode=STRICT\_TRANS\_TABLES,NO\_ZERO\_IN\_DATE,NO\_ZERO\_DATE,ERROR\_FOR\_DIVISION\_BY\_ZERO,NO\_ENGINE\_SUBSTITUTION

max\_connections=1000

server-id=101

log-bin=mysql-slave-bin

relay\_log=edu-mysql-relay-bin

[client]

default-character-set=utf8

[mysql]

default-character-set=utf8

docker run -di -v /home/mysql2/data:/var/lib/mysql -v /home/mysql2/conf.d:/etc/mysql/conf.d -v /home/mysql2/my.cnf:/etc/mysql/my.cnf -p 33306:3306 --name mysql-slave -e MYSQL\_ROOT\_PASSWORD=123456 mysql:5.7

**2.5 远程连接到主库和从库**

#主库

mysql -h 101.133.225.166 -P 33307 -u root -p123456

##创建test用户,设置任意ip地址可以访问，密码为123

create user 'test'@'%' identified by '123';

##授权用户，把所有权限授权给test，他就相当于root了

grant all privileges on \*.\* to 'test'@'%' ;

###刷新权限

flush privileges;

#查看主服务器状态(显示如下图)

show master status;

#从库

mysql -h 101.133.225.166 -P 33306 -u root -p123456

#配置详解

/\*

change master to

master\_host='MySQL主服务器IP地址',

master\_user='之前在MySQL主服务器上面创建的用户名'，

master\_password='之前创建的密码',

master\_log\_file='MySQL主服务器状态中的二进制文件名',

master\_log\_pos='MySQL主服务器状态中的position值';

\*/

#命令如下

change master to master\_host='101.133.225.166',master\_port=33307,master\_user='test',master\_password='123',master\_log\_file='mysql-bin.000003',master\_log\_pos=0;

#启用从库

start slave;

#查看从库状态（如下图）

show slave status\G;

**2.6 测试**

#在主库上创建数据库test1

create database test1;

use test1;

#创建表

create table tom (id int not null,name varchar(100)not null ,age tinyint);

#插入数据

insert tom (id,name,age) values(1,'xxx',20),(2,'yyy',7),(3,'zzz',23);

#在从库上查看是否同步成功

#查看数据库

show database;

use test1;

#查看表

show tables;

#查看数据

select \* from test1;

**补充：查看容器启动日志**

docker logs b8bdd9c57f22

**3 django实现读写分离**

makemigrations

#同步数据库

migrate #表示同步到default数据库

migrate app01 --database=db1

**3.2 手动控制读写分离**

def index(request):

#向数据表中存条数据

#写到哪个库中了？默认情况下写到default

# ret=models.Book.objects.using('default').create(name="xxx")

# print(ret)

#读第一条数据，默认从哪读？default

#指定去从库读？在queryset对象后加个.using('db1')

# ret=models.Book.objects.all().first()

ret=models.Book.objects.all().using('db1').first()

print(ret.name)

return HttpResponse("ok")

**3.3 自动读写分离**

#第一步：在项目根路径下创建一个py文件（database\_router.py）

class DatabaseAppsRouter(object):

def db\_for\_read(self, model, \*\*hints):

#读可能去db1或者db2中读，

return 'db1'

def db\_for\_write(self, model, \*\*hints):

print(model)

return 'default'

def allow\_migrate(self, db, app\_label, model\_name=None, \*\*hints):

print(db)

print(app\_label)

print(model\_name)

if app\_label=='app01' and db=="db1":

return False

return None

# 第二部，在setting中配置

DATABASE\_ROUTERS = ['mydjangotest.database\_router.DatabaseAppsRouter']

**3.4 allow\_migrate的使用**

migrate app01 --database=db1 #不会同步

migrate app01 --database=default #可以同步

#用来控制数据表同步时，不能同步到从库中

**dockerfile中的命令：run, cmd, entrypoint， copy和add**

**总结一下，**

**run 可以有多个，cmd 和entrypoint 只能有一个（常用来跑app）**

**cmd 可以被docker 指令overwrite ，entrypoint不可以**

**此命令会在容器启动且 docker run 没有指定其他命令时运行。**

**如果 docker run 指定了其他命令，CMD 指定的默认命令将被忽略。**

**如果 Dockerfile 中有多个 CMD 指令，只有最后一个 CMD 有效。**

**ENTRYPOINT 的 Exec 格式用于设置要执行的命令及其参数，同时可通过 CMD 提供额外的参数。**

**ENTRYPOINT 中的参数始终会被使用，而 CMD 的额外参数可以在容器启动时动态替换掉。**

**比如下面的 Dockerfile 片段：**

**ENTRYPOINT ["/bin/echo", "Hello"]**

**CMD ["world"]**

**当容器通过 docker run -it [image] 启动时，输出为：**

**Hello world**

**而如果通过 docker run -it [image] CloudMan 启动，则输出为：**

**Hello CloudMan**

**docker 2、优化docker镜像大小方法**

**1 查看镜像id**

**即执行如下命令:**

**[root@node-1 ~]# docker images|grep xxx**

**docker.io/xxx                 1.0              5d1b456495fc        20 hours ago        1.64 GB**

**注意:**

**请将xxx替换为你需要查找的镜像名称**

**2 查看该镜像的分层情况**

**即执行如下命令:**

**docker history <image\_id>**

**解释:**

**docker history是查看镜像的创建历史，最下面是分层的最底层。**

**[root@node-1 ~]# docker history 5d1b456495fc**

**IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT**

**5d1b456495fc        20 hours ago        /bin/sh -c #(nop)  USER yyy                 0 B**

**<missing>           20 hours ago        /bin/sh -c #(nop)  LABEL maintainer=xxx...   0 B**

**<missing>           20 hours ago        /bin/sh -c #(nop)  LABEL last-commit-id=08...   0 B**

**<missing>           20 hours ago        /bin/sh -c chmod 750 /etc/sudoers.d     &&...   535 B**

**<missing>           20 hours ago        /bin/sh -c pip install tenacity===3.1.0   ...   186 MB**

**<missing>           20 hours ago        /bin/sh -c #(nop) COPY file:f483b3f3d6368f...   117 B**

**<missing>           20 hours ago        /bin/sh -c #(nop) COPY file:a82f5a713f9ec5...   418 B**

**<missing>           20 hours ago        /bin/sh -c mkdir /gnocchi-base-source && c...   370 MB**

**<missing>           20 hours ago        /bin/sh -c mkdir -p /var/www/cgi-bin/gnocc...   21.2 kB**

**<missing>           20 hours ago        /bin/sh -c yum -y install httpd mod\_ssl mo...   27 MB**

**<missing>           20 hours ago        /bin/sh -c usermod --append --home /var/li...   23.4 kB**

**<missing>           20 hours ago        /bin/sh -c #(nop)  LABEL maintainer=EasySt...   0 B**

**<missing>           6 months ago        /bin/sh -c pip --no-cache-dir install --up...   94.6 MB**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:2e2bd3f2cc2c54...   1.04 kB**

**<missing>           6 months ago        /bin/sh -c pip --no-cache-dir install --up...   738 kB**

**<missing>           6 months ago        /bin/sh -c yum -y install gcc gcc-c++ libf...   631 MB**

**<missing>           6 months ago        /bin/sh -c yum -y install git iproute open...   75.8 MB**

**<missing>           6 months ago        /bin/sh -c #(nop)  LABEL maintainer=...   0 B**

**<missing>           6 months ago        /bin/sh -c touch /usr/local/bin/kolla\_exte...   15.8 kB**

**<missing>           6 months ago        /bin/sh -c rm /etc/localtime     && ln -s ...   0 B**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:b89240fb6cba42...   97 B**

**<missing>           6 months ago        /bin/sh -c mkdir -p /root/.pip/                 0 B**

**<missing>           6 months ago        /bin/sh -c curl http://..../dum...   10.2 MB**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:f67b9142d3f871...   154 B**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:1c53e6b1b4655f...   831 B**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:945dd7de8a0bc7...   503 B**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:6557b170983c4c...   14.4 kB**

**<missing>           6 months ago        /bin/sh -c yum -y install curl iproute isc...   35.5 MB**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:3a59b0f663154a...   130 B**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:30bbaf3e30e53d...   119 B**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:7a5b14cdd97eb5...   459 B**

**<missing>           6 months ago        /bin/sh -c CURRENT\_DISTRO\_RELEASE=$(awk '{...   3.97 kB**

**<missing>           6 months ago        /bin/sh -c #(nop)  ENV PS1=$(tput bold)$(t...   0 B**

**<missing>           6 months ago        /bin/sh -c cat /tmp/kolla\_bashrc >> /etc/s...   705 B**

**<missing>           6 months ago        /bin/sh -c #(nop) COPY file:1020b4b8ead353...   149 B**

**<missing>           6 months ago        /bin/sh -c #(nop)  ENV KOLLA\_BASE\_DISTRO=e...   0 B**

**<missing>           6 months ago        /bin/sh -c #(nop)  LABEL kolla\_version=5.0.1    0 B**

**<missing>           6 months ago        /bin/sh -c groupadd --force --gid 42491 al...   18.7 MB**

**<missing>           6 months ago        /bin/sh -c #(nop)  LABEL maintainer=EasySt...   0 B**

**<missing>           9 months ago        /bin/sh -c #(nop)  ENV LC\_ALL=en\_US.utf8        0 B**

**<missing>           9 months ago        /bin/sh -c #(nop) ADD file:4dadc4b90f7e5a4...   195 MB**

**分析:**

**下面的部分是gnocchi-base打镜像的各个步骤对应的镜像层级**

**IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT**

**5d1b456495fc        20 hours ago        /bin/sh -c #(nop)  USER gnocchi                 0 B**

**<missing>           20 hours ago        /bin/sh -c #(nop)  LABEL maintainer=EasySt...   0 B**

**<missing>           20 hours ago        /bin/sh -c #(nop)  LABEL last-commit-id=08...   0 B**

**<missing>           20 hours ago        /bin/sh -c chmod 750 /etc/sudoers.d     &&...   535 B**

**<missing>           20 hours ago        /bin/sh -c pip install tenacity===3.1.0   ...   186 MB**

**<missing>           20 hours ago        /bin/sh -c #(nop) COPY file:f483b3f3d6368f...   117 B**

**<missing>           20 hours ago        /bin/sh -c #(nop) COPY file:a82f5a713f9ec5...   418 B**

**<missing>           20 hours ago        /bin/sh -c mkdir /gnocchi-base-source && c...   370 MB**

**<missing>           20 hours ago        /bin/sh -c mkdir -p /var/www/cgi-bin/gnocc...   21.2 kB**

**<missing>           20 hours ago        /bin/sh -c yum -y install httpd mod\_ssl mo...   27 MB**

**<missing>           20 hours ago        /bin/sh -c usermod --append --home /var/li...   23.4 kB**

**<missing>           20 hours ago        /bin/sh -c #(nop)  LABEL maintainer=...   0 B**

**下面分析占用空间占用较大的的层级**

**可以看到:**

**<missing>           20 hours ago        /bin/sh -c pip install tenacity===3.1.0   ...   186 MB**

**这里是gnocchi中需要下载的第三方库对应的镜像层级，对应代码如下:**

**RUN pip install tenacity===3.1.0 \**

**&& pip install oslo.middleware===3.30.1 \**

**RUN chmod 750 /etc/sudoers.d \**

**&& chmod 640 /etc/sudoers.d/kolla\_gnocchi\_sudoers \**

**&& touch /usr/local/bin/kolla\_gnocchi\_extend\_start \**

**&& chmod 755 /usr/local/bin/kolla\_extend\_start /usr/local/bin/kolla\_gnocchi\_extend\_start**

**将多个连续的RUN命令放在一起，即变成如下形式:**

**RUN pip install tenacity===3.1.0 \**

**&& pip install oslo.middleware===3.30.1 \**

**&& chmod 750 /etc/sudoers.d \**

**&& chmod 640 /etc/sudoers.d/kolla\_gnocchi\_sudoers \**

**&& touch /usr/local/bin/kolla\_gnocchi\_extend\_start \**

**&& chmod 755 /usr/local/bin/kolla\_extend\_start /usr/local/bin/kolla\_gnocchi\_extend\_start**

**3 docker镜像大小优化方法**

**优化思路:**

**1) 将连续的RUN命令合并在一起。**

**2) 设置某个组件的基础镜像，这个基础镜像中只存放这个组件中不变的代码，将这个组件中会变化的部分放入该组件的各个服务中。**

**以gnocchi为例，设置gnocchi-base作为gnocchi这个组件的基础镜像，里面放入不会改变的部分，**

**将会修改的内容放入gnocchi的各个服务中，例如: gnocchi-metricd, gnocchi-statsd, gnocchi-api**

**3) 删除镜像中不用的安装库，删除yum的缓存。**

**即在执行执行yum安装命令后，就执行 yum clean all清理缓存和rm -rf /var/lib/yum/yumdb**

**例子如下:**

**RUN yum -y install epel-release \**

**&& yum -y install rsyslog**

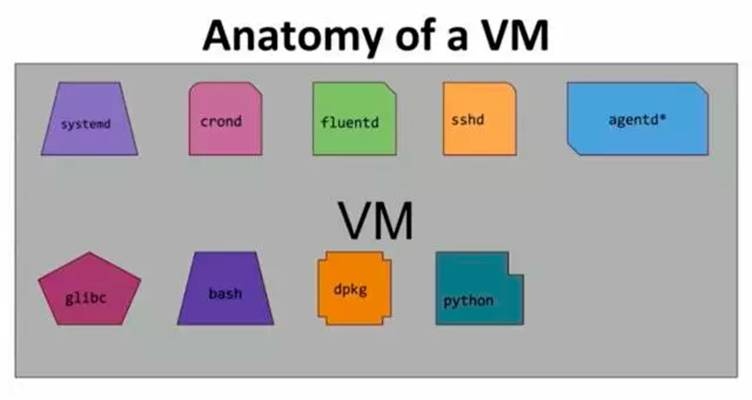
**&& yum clean all**

**&& rm -rf /var/lib/yum/yumdb**

**谷歌技术人员解决Docker镜像体积太大问题的方法**

这篇文章主要介绍了谷歌技术人员解决Docker镜像体积太大问题的方法，涉及虚拟机，谷歌docker镜像构建实践及构建工具bazel的介绍等相关内容，具有一定参考价值，需要的朋友可以了解下。

**虚拟机的问题**



最初，大家都使用虚拟机作为软件的运行环境，对外提供服务。为了在虚拟机上运行你的 Service，你不得不运行一大堆程序：

**系统进程  
定时任务  
SSH  
安装 Agent  
安装 Bash  
安装一大堆 libs**

其实，你仅仅只是想让你的 Service运行起来，但你不得不维护一个 40GB的虚拟机。

然后你开始试用 Docker



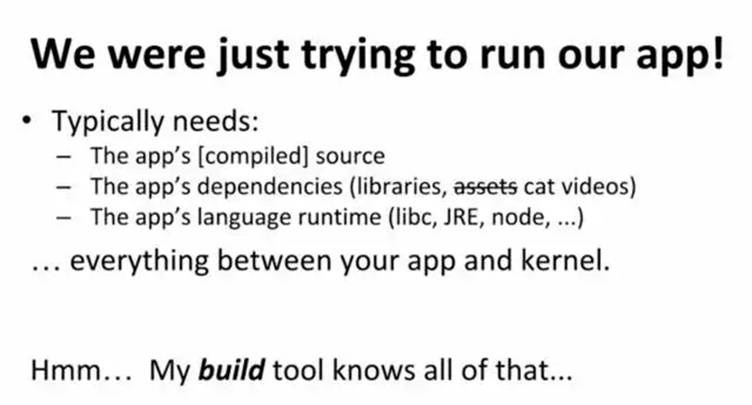
开始试用 Docker，你毫不犹豫选择了和之前虚拟机一样的镜像：Ubuntu 1404，将之前的虚机的内容复制到了 Docker镜像，安装了一堆软件，最后发现你的 Docker镜像有 8GB。

**谷歌的 Docker镜像构建实践**

找到最小的基础镜像

Alpine Linux是基于 musl和 BusyBox的操作系统，目的是为了为用户提供更高效的资源使用效率。它的特性是体积小，最小的 Alpine Linux体积可以只有 5MB。谷歌某些团队使用 Alpine Linux作为 Docker Build的基础镜像。

目的：仅仅为了运行 Service



谷歌认为，为了运行一个 Service，并不需要将那些无关联的包、程序打包到容器里，换句话说，Docker镜像里只留下需要用到的，其他的都删除，从而得到一个最小的镜像。这需要考虑以下几点：

1、程序编译后的二进制文件（从 Artifactory获取）

2、程序的所有依赖（从 Artifactory获取）

3、程序语言的运行时（libc，JRE，node，…）

4、任何程序和 Kernel之间的中间件

其实这一切的信息，构建工具都已经知道。

**谷歌的构建工具 Bazel**

介绍 Bazel之前，先介绍下谷歌的开发模式，对于服务器端代码库，谷歌的开发流程如下：

1、所有的服务器端代码库都在一个巨大的版本控制系统里

2、每个人都用 Bazel构建软件

3、不同的组负责源码树的不同部分，所有的组件都是作为 BUILD目标来用

4、分支主要是用来管理发布，所以每个人都在最新版本上开发软件

Bazel（https://bazel.build/）是 Google内部用来构建自己的服务器端软件的工具。目前变成谷歌公司贡献的一个开源项目，目的是帮助开发者将软件的构建和测试变得更快、更可靠。



从上图可以看到，Bazel有 WORDSPACE的概念，WORDSPACE文件用来准备 Docker镜像构建所依赖的所有材料和来源。BUILD文件用来告诉 Bazel这个镜像应该使用什么命令进行构建，以及如何构建、如何测试。

使用 Bazel的声明式语言: WORKSPACE和 BUILD，开发者可以用文件描述整个构建和部署的环境。谷歌使用 Bazel进行 Docker的构建已经很多年，它为谷歌带来以下收益：

1、支持跨平台构建，分布式缓存，优化依赖解析，并行构建，增量构建。

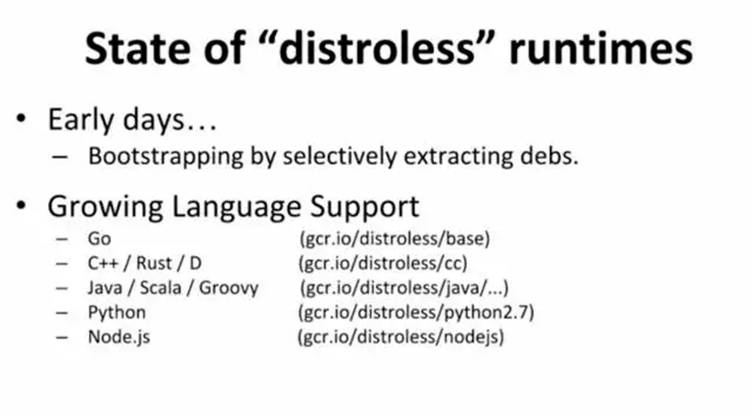
2、支持多语言（Java, C++, Android, iOS, Go等等）。

3、跨平台。

4、水平扩展和自定义扩展。

Bazel是以下理念的奠基石：由于 Bazel需要所有的依赖都被完整地指定，我们可以预测改动影响了哪些程序和测试，并在提交前执行他们。

**谷歌提供的 Distroless镜像构建文件**



Distroless (https://github.com/GoogleCloudPlatform/distroless)是谷歌内部使用的镜像构建文件，包括 Java、Node、Python等镜像构建文件，Distroless仅仅只包含运行服务所需要的最小镜像，不包含包管理工具、shell命令行等其他功能。

为什么你需要这些镜像？这些镜像是谷歌和其他大公司的最佳实践的产物，经过了漏洞扫描，镜像会持续更新，保持安全性。

如何使用？Distroless提供的构建镜像的 BUILD文件，通过 Bazel可以直接进行构建。

谷歌为大家介绍了内部如何解决镜像过大的问题，以及进行大规模并发构建、测试所用到的构建工具 Bazel，并且开源了 Docker镜像构建文件 Distroless项目。如果你认为你的镜像也存在体积太大的问题，可以参考谷歌的实践，体验他们的工具。

**总结**

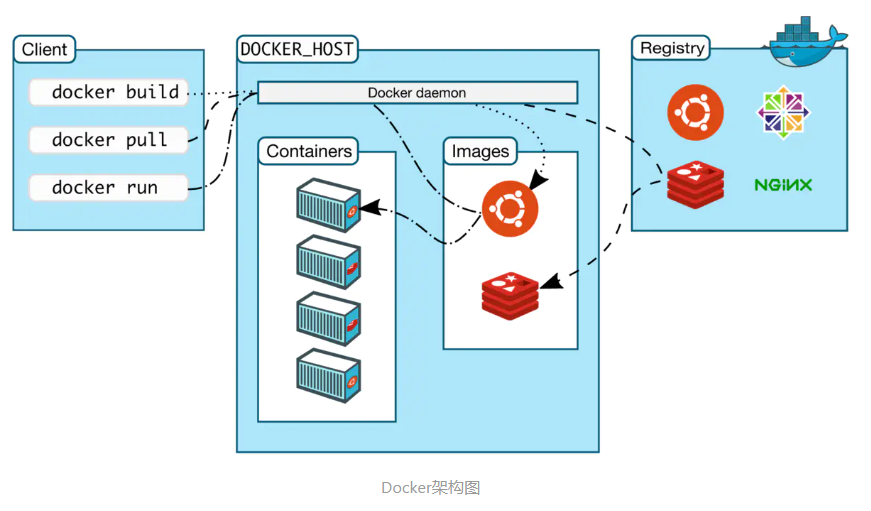
以上就是本文关于谷歌技术人员解决Docker镜像体积太大问题的方法的全部内容，希望对大家有所帮助。感兴趣的朋友可以可以继续参阅本站：

**Docker及Docker-Compose的使用**

[菜鸟Docker](https://links.jianshu.com/go?to=https%3A%2F%2Fwww.runoob.com%2Fdocker%2Fdocker-tutorial.html)  
[阮一峰的Docker教程](https://links.jianshu.com/go?to=http%3A%2F%2Fwww.ruanyifeng.com%2Fblog%2F2018%2F02%2Fdocker-tutorial.html)

Docker是一个开源的容器引擎，它有助于更快地交付应用。方便快捷已经是 Docker的最大优势，过去需要用数天乃至数周的任务，在Docker容器的处理下，只需要数秒就能完成。

**架构**



* Docker daemon（ Docker守护进程）：Docker daemon是一个运行在宿主机（ DOCKER-HOST）的后台进程。可通过 Docker客户端与之通信。
* Client（ Docker客户端）：Docker客户端是 Docker的用户界面，它可以接受用户命令和配置标识，并与 Docker daemon通信。图中， docker build等都是 Docker的相关命令。
* Images（ Docker镜像）：Docker镜像是一个只读模板，它包含创建 Docker容器的说明。它和系统安装光盘有点像，使用系统安装光盘可以安装系统，同理，使用Docker镜像可以运行 Docker镜像中的程序。
* Container（容器）：容器是镜像的可运行实例。镜像和容器的关系有点类似于面向对象中，类和对象的关系。可通过 Docker API或者 CLI命令来启停、移动、删除容器。
* Registry：Docker Registry是一个集中存储与分发镜像的服务。构建完 Docker镜像后，就可在当前宿主机上运行。但如果想要在其他机器上运行这个镜像，就需要手动复制。此时可借助Docker Registry来避免镜像的手动复制。一个 Docker Registry可包含多个 Docker仓库，每个仓库可包含多个镜像标签，每个标签对应一个 Docker镜像。这跟 Maven的仓库有点类似，如果把 Docker Registry比作Maven仓库的话，那么 Docker仓库就可理解为某jar包的路径，而镜像标签则可理解为jar包的版本号。Docker Registry可分为公有Docker Registry和私有Docker Registry。 最常用的DockerRegistry莫过于官网的Docker Hub， 这也是默认的Docker Registry。 Docker Hub上存放着大量优秀的镜像， 我们可使用Docker命令下载并使用。

**安装**

按照菜鸟的步骤，使用yum安装即可。

**常用命令**

**镜像相关**

* docker search java：在Docker Hub（或阿里镜像）仓库中搜索关键字（如java）的镜像
* docker pull java:8：从仓库中下载镜像，若要指定版本，则要在冒号后指定
* docker images：列出已经下载的镜像
* docker rmi java：删除本地镜像
* docker build：构建镜像

**容器相关**

* docker run -d -p 91:80 nginx ：在后台运行nginx，若没有镜像则先下载，并将容器的80端口映射为宿主机的91端口。
  + -d：后台运行
  + -P：随机端口映射
  + -p：指定端口映射
  + -net：网络模式
* docker ps：列出运行中的容器
* docker ps -a ：列出所有的容器
* docker stop 容器id：停止容器
* docker kill 容器id：强制停止容器
* docker start 容器id：启动已停止的容器
* docker inspect 容器id：查看容器的所有信息
* docker container logs 容器id：查看容器日志
* docker top 容器id：查看容器里的进程
* docker exec -it 容器id /bin/bash：进入容器
* exit：退出容器
* docker rm 容器id：删除已停止的容器
* docker rm -f 容器id：删除正在运行的容器

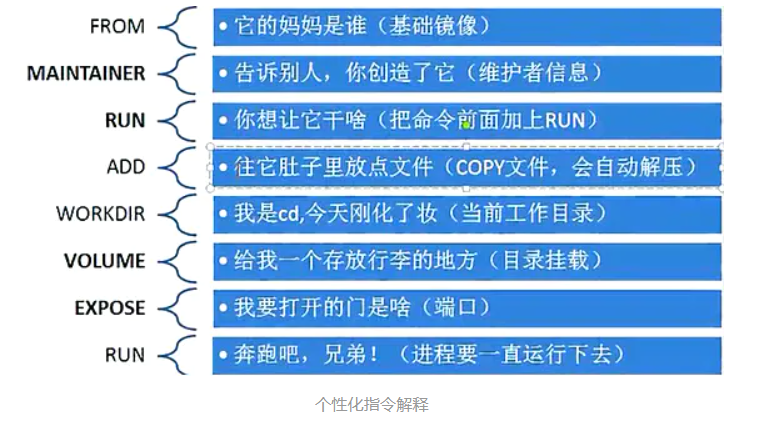
**所有命令**

* docker
* docker COMMAND --help

**构建镜像**

1. 确定镜像模板：如java、nginx
2. 新建Dockerfile文件
3. 使用Dockerfile的指令完善Dockerfile的内容
4. 在Dockerfile文件的所在路径执行docker build -t imageName:tag .，-t指定镜像名称，末尾的点标识Dockerfile文件的路径
5. 执行docker run -d -p 92:80 imageName:tag即可

常用指令如下图，直白用法点[我](https://links.jianshu.com/go?to=https%3A%2F%2Fwww.cnblogs.com%2Fniloay%2Fp%2F6261784.html)，官方介绍点击[我](https://links.jianshu.com/go?to=https%3A%2F%2Fdocs.docker.com%2Fv17.09%2Fengine%2Freference%2Fbuilder%2F)



备注：RUN命令在 image 文件的构建阶段执行，执行结果都会打包进入 image 文件；CMD命令则是在容器启动后执行。另外，一个 Dockerfile 可以包含多个RUN命令，但是只能有一个CMD命令。注意，指定了CMD命令以后，docker container run命令就不能附加命令了，否则它会覆盖CMD命令。

**Docker Compose**

[Docker Compose](https://links.jianshu.com/go?to=https%3A%2F%2Fgithub.com%2Fdocker%2Fcompose)是 docker 提供的一个命令行工具，用来定义和运行由多个容器组成的应用。使用 compose，我们可以通过 YAML 文件声明式的定义应用程序的各个服务，并由单个命令完成应用的创建和启动。

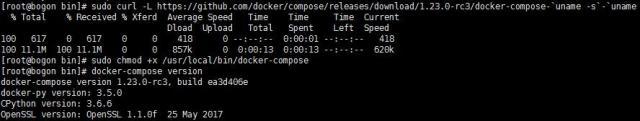
本文主要说Docker Compose，但是在这之前，有必要知道一下，Docker，Docker Compose，Docker Swarm，Kubernetes之间的区别。具体的如图，简单点说，Docker Compose是单机管理Docker的，Kubernetes是多节点管理Docker的。虽然Docker Swarm也是多节点管理，但基本已弃用，了解一下就好了。



安装Docker Compose

接上一篇的例子，安装Docker Compose，并进行赋权和检验。

sudo curl -L https://github.com/docker/compose/releases/download/1.21.2/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-composesudo chmod +x /usr/local/bin/docker-composedocker-compose -v 最后能看到版本信息，就是安装好了，有人可能会遇到一个错误。[53396] Cannot open self /usr/local/bin/docker-compose or archive /usr/local/bin/docker-compose.pkg。删除掉/usr/local/bin/docker-compose，重新下载一下，可能是网络原因下载有中断情况，而这个文件又仍然会生成。



准备Dockerfile和docker-compose.yml

在jar包所在目录，新建Dockerfile文件，touch Dockerfile，然后输入内容vi Dockerfile：

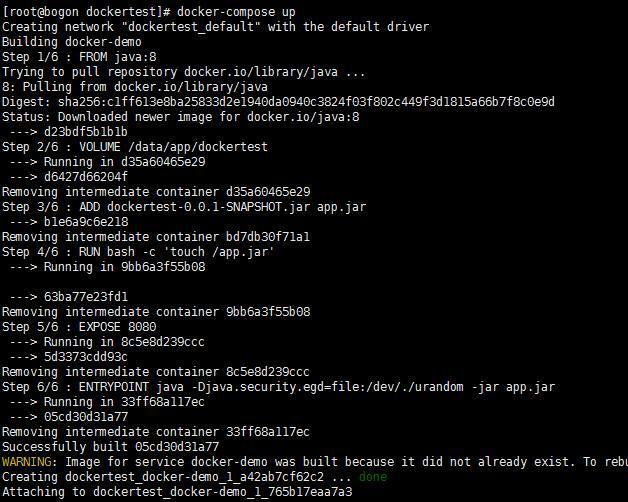
FROM java:8VOLUME /data/app/dockertestADD dockertest-0.0.1-SNAPSHOT.jar app.jarRUN bash -c 'touch /app.jar'EXPOSE 8080ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","app.jar"] 对于这个配置文件的参数含义，可以百度查一下，这里说一下volume就是你jar包的文件夹位置，add后的第一个就是宿主机的待发布应用，后面就是到容器后的（可以回想上一篇最后的启动命令）。

再在这个目录里，新建 docker-compose.yml，参考输入内容：

version: '2' # 表示该 Docker-Compose 文件使用的是 Version 2 fileservices: docker-demo: # 指定服务名称 build: . # 指定 Dockerfile 所在路径 ports: # 指定端口映射 - "8080:8080"启动docker-compose

配置完后，就可以通过Docker Compose启动你配置的镜像了。

先关掉之前使用docker run启动的镜像（docker stop $(docker ps -aq)），再启动Docker Compose（docker-compose up），启动命令后面也有很多参数，比如-d 在后台运行服务容器。这里就简单启动一下看看效果：



看到绿色的done，就知道已经成功了，再等会儿，就打印出应用控制台的内容了，这时候访问一下ip:8080（这里配置的应用端口），发现应用可以正常访问到了。

[**Docker入门之docker-compose**](https://www.cnblogs.com/minseo/p/11548177.html)

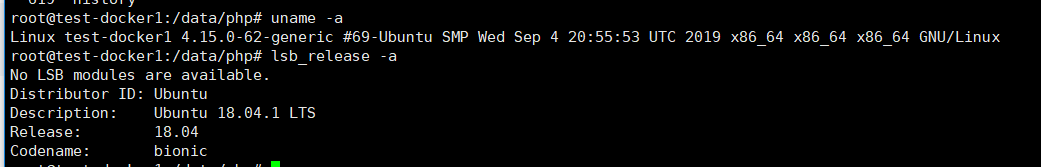
**一，Docker-compose简介**

**1，Docker-compose简介**

　　Docker-Compose项目是Docker官方的开源项目，负责实现对Docker容器集群的快速编排。  
Docker-Compose将所管理的容器分为三层，分别是工程（project），服务（service）以及容器（container）。Docker-Compose运行目录下的所有文件（docker-compose.yml，extends文件或环境变量文件等）组成一个工程，若无特殊指定工程名即为当前目录名。一个工程当中可包含多个服务，每个服务中定义了容器运行的镜像，参数，依赖。一个服务当中可包括多个容器实例，Docker-Compose并没有解决负载均衡的问题，因此需要借助其它工具实现服务发现及负载均衡。  
Docker-Compose的工程配置文件默认为docker-compose.yml，可通过环境变量COMPOSE\_FILE或-f参数自定义配置文件，其定义了多个有依赖关系的服务及每个服务运行的容器。  
使用一个Dockerfile模板文件，可以让用户很方便的定义一个单独的应用容器。在工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个Web项目，除了Web服务容器本身，往往还需要再加上后端的数据库服务容器，甚至还包括负载均衡容器等。  
Compose允许用户通过一个单独的docker-compose.yml模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。  
Docker-Compose项目由Python编写，调用Docker服务提供的API来对容器进行管理。因此，只要所操作的平台支持Docker API，就可以在其上利用Compose来进行编排管理。

**2，Docker-compose的安装**

　　安装环境查看



 　　安装

|  |  |
| --- | --- |
| 1  2 | apt install python-pip  pip install docker-compose |

 　　PS：centos使用命令yum -y install python-pip安装pip

　　查看安装的版本

|  |  |
| --- | --- |
| 1  2 | # docker-compose --version  docker-compose version 1.24.0, build 0aa5906 |

**3，Docker-compose卸载**

|  |  |
| --- | --- |
| 1 | pip uninstall docker-compose |

**二，Docker-compose常用命令**

**1，Docker-compose命令格式**

|  |  |
| --- | --- |
| 1 | docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...] |

 　　命令选项如下

|  |  |
| --- | --- |
| 1  2  3  4  5 | -f --file FILE指定Compose模板文件，默认为docker-compose.yml  -p --project-name NAME 指定项目名称，默认使用当前所在目录为项目名  --verbose  输出更多调试信息  -v，-version 打印版本并退出  --log-level LEVEL 定义日志等级(DEBUG, INFO, WARNING, ERROR, CRITICAL) |

**2，docker-compose up**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | docker-compose up [options] [--scale SERVICE=NUM...] [SERVICE...]  选项包括：  -d 在后台运行服务容器  -no-color 不是有颜色来区分不同的服务的控制输出  -no-deps 不启动服务所链接的容器  --force-recreate 强制重新创建容器，不能与-no-recreate同时使用  –no-recreate 如果容器已经存在，则不重新创建，不能与–force-recreate同时使用  –no-build 不自动构建缺失的服务镜像  –build 在启动容器前构建服务镜像  –abort-on-container-exit 停止所有容器，如果任何一个容器被停止，不能与-d同时使用  -t, –timeout TIMEOUT 停止容器时候的超时（默认为10秒）  –remove-orphans 删除服务中没有在compose文件中定义的容器 |

**3，docker-compose ps**

|  |  |
| --- | --- |
| 1  2 | docker-compose  ps [options] [SERVICE...]  列出项目中所有的容器 |

**4，docker-compose stop**

|  |  |
| --- | --- |
| 1  2  3  4  5 | docker-compose stop [options] [SERVICE...]  选项包括  -t, –timeout TIMEOUT 停止容器时候的超时（默认为10秒）  docker-compose stop  停止正在运行的容器，可以通过docker-compose start 再次启动 |

**5，docker-compose -h**

|  |  |
| --- | --- |
| 1  2 | docker-compose -h  查看帮助 |

**6，docker-compose down**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | docker-compose down [options]  停止和删除容器、网络、卷、镜像。  选项包括：  –rmi type，删除镜像，类型必须是：all，删除compose文件中定义的所有镜像；local，删除镜像名为空的镜像  -v, –volumes，删除已经在compose文件中定义的和匿名的附在容器上的数据卷  –remove-orphans，删除服务中没有在compose中定义的容器  docker-compose down  停用移除所有容器以及网络相关 |

**7，docker-compose logs**

|  |  |
| --- | --- |
| 1  2  3  4  5 | docker-compose logs [options] [SERVICE...]  查看服务容器的输出。默认情况下，docker-compose将对不同的服务输出使用不同的颜色来区分。可以通过–no-color来关闭颜色。  docker-compose logs  查看服务容器的输出  -f 跟踪日志输出 |

**8，docker-compose bulid**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | docker-compose build [options] [--build-arg key=val...] [SERVICE...]  构建（重新构建）项目中的服务容器。  选项包括：  –compress 通过gzip压缩构建上下环境  –force-rm 删除构建过程中的临时容器  –no-cache 构建镜像过程中不使用缓存  –pull 始终尝试通过拉取操作来获取更新版本的镜像  -m, –memory MEM为构建的容器设置内存大小  –build-arg key=val为服务设置build-time变量  服务容器一旦构建后，将会带上一个标记名。可以随时在项目目录下运行docker-compose build来重新构建服务 |

**9，docker-compose pull**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | docker-compose pull [options] [SERVICE...]  拉取服务依赖的镜像。  选项包括：  –ignore-pull-failures，忽略拉取镜像过程中的错误  –parallel，多个镜像同时拉取  –quiet，拉取镜像过程中不打印进度信息  docker-compose pull  拉取服务依赖的镜像 |

**10，docker-compose restart**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | docker-compose restart [options] [SERVICE...]  重启项目中的服务。  选项包括：  -t, –timeout TIMEOUT，指定重启前停止容器的超时（默认为10秒）  docker-compose restart  重启项目中的服务 |

**11，docker-compose rm**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | docker-compose rm [options] [SERVICE...]  删除所有（停止状态的）服务容器。  选项包括：  –f, –force，强制直接删除，包括非停止状态的容器  -v，删除容器所挂载的数据卷  docker-compose rm  删除所有（停止状态的）服务容器。推荐先执行docker-compose stop命令来停止容器。 |

**12，docker-compose start**

|  |  |
| --- | --- |
| 1  2  3 | docker-compose start [SERVICE...]  docker-compose start  启动已经存在的服务容器。 |

**13，docker-compose run**

|  |  |
| --- | --- |
| 1  2  3  4 | docker-compose run [options] [-v VOLUME...] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]  在指定服务上执行一个命令。  docker-compose run ubuntu ping www.baidu.com  在指定容器上执行一个ping命令。 |

**14，docker-compose scale**

|  |  |
| --- | --- |
| 1  2 | docker-compose scale web=3 db=2  设置指定服务运行的容器个数。通过service=num的参数来设置数量 |

**15，docker-compose pause**

|  |  |
| --- | --- |
| 1  2 | docker-compose pause [SERVICE...]  暂停一个服务容器 |

**16，docker-compose kill**

|  |  |
| --- | --- |
| 1  2  3  4 | docker-compose kill [options] [SERVICE...]  通过发送SIGKILL信号来强制停止服务容器。  支持通过-s参数来指定发送的信号，例如通过如下指令发送SIGINT信号：  docker-compose kill -s SIGINT |

**17，docker-compose config**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | docker-compose config [options]  验证并查看compose文件配置。  选项包括：  –resolve-image-digests 将镜像标签标记为摘要  -q, –quiet 只验证配置，不输出。 当配置正确时，不输出任何内容，当文件配置错误，输出错误信息  –services 打印服务名，一行一个  –volumes 打印数据卷名，一行一个 |

**18，docker-compose create**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | docker-compose create [options] [SERVICE...]  为服务创建容器。  选项包括：  –force-recreate：重新创建容器，即使配置和镜像没有改变，不兼容–no-recreate参数  –no-recreate：如果容器已经存在，不需要重新创建，不兼容–force-recreate参数  –no-build：不创建镜像，即使缺失  –build：创建容器前　　，生成镜像 |

**19，docker-compose exec**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | docker-compose exec [options] SERVICE COMMAND [ARGS...]  选项包括：  -d 分离模式，后台运行命令。  –privileged 获取特权。  –user USER 指定运行的用户。  -T 禁用分配TTY，默认docker-compose exec分配TTY。  –index=index，当一个服务拥有多个容器时，可通过该参数登陆到该服务下的任何服务，例如：docker-compose exec –index=1 web /bin/bash ，web服务中包含多个容器 |

**20，docker-compose port**

|  |  |
| --- | --- |
| 1  2  3  4  5 | docker-compose port [options] SERVICE PRIVATE\_PORT  显示某个容器端口所映射的公共端口。  选项包括：  –protocol=proto，指定端口协议，TCP（默认值）或者UDP  –index=index，如果同意服务存在多个容器，指定命令对象容器的序号（默认为1） |

**21，docker-compose push**

|  |  |
| --- | --- |
| 1  2  3  4 | docker-compose push [options] [SERVICE...]  推送服务依的镜像。  选项包括：  –ignore-push-failures 忽略推送镜像过程中的错误 |

**22，docker-compose stop**

|  |  |
| --- | --- |
| 1  2 | docker-compose stop [options] [SERVICE...]  显示各个容器运行的进程情况。 |

**23，docker-compose uppause**

|  |  |
| --- | --- |
| 1  2 | docker-compose unpause [SERVICE...]  恢复处于暂停状态中的服务。 |

**三，Docker-compose模板文件**

**1，Docker-compose模板文件简介**

　　Compose允许用户通过一个docker-compose.yml模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。  
Compose模板文件是一个定义服务、网络和卷的YAML文件。Compose模板文件默认路径是当前目录下的docker-compose.yml，可以使用.yml或.yaml作为文件扩展名。  
Docker-Compose标准模板文件应该包含version、services、networks 三大部分，最关键的是services和networks两个部分。

　　举例

|  |  |
| --- | --- |
| 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 | version: '2'  services:    web:      image: dockercloud/hello-world      ports:        - 8080      networks:        - front-tier        - back-tier      redis:      image: redis      links:        - web      networks:        - back-tier      lb:      image: dockercloud/haproxy      ports:        - 80:80      links:        - web      networks:        - front-tier        - back-tier      volumes:        - /var/run/docker.sock:/var/run/docker.sock    networks:    front-tier:      driver: bridge    back-tier:      driver: bridge |

 　　Compose目前有三个版本分别为Version 1，Version 2，Version 3，Compose区分Version 1和Version 2（Compose 1.6.0+，Docker Engine 1.10.0+）。Version 2支持更多的指令。Version 1将来会被弃用。

**2，image**

　　image是指定服务的镜像名称或镜像ID。如果镜像在本地不存在，Compose将会尝试拉取镜像。

|  |  |
| --- | --- |
| 1  2  3 | services:      web:          image: hello-world |

**3，build**

　　服务除了可以基于指定的镜像，还可以基于一份Dockerfile，在使用up启动时执行构建任务，构建标签是build，可以指定Dockerfile所在文件夹的路径。Compose将会利用Dockerfile自动构建镜像，然后使用镜像启动服务容器。

|  |  |
| --- | --- |
| 1 | build: /path/to/build/dir |

 　　也可以是相对路径，只要上下文确定就可以读取到Dockerfile。

|  |  |
| --- | --- |
| 1 | build: ./dir |

 　　设定上下文根目录，然后以该目录为准指定Dockerfile。

|  |  |
| --- | --- |
| 1  2  3 | build:    context: ../    dockerfile: path/of/Dockerfile |

 　　build都是一个目录，如果要指定Dockerfile文件需要在build标签的子级标签中使用dockerfile标签指定。  
如果同时指定image和build两个标签，那么Compose会构建镜像并且把镜像命名为image值指定的名字。

**4,context**

　　context选项可以是Dockerfile的文件路径，也可以是到链接到git仓库的url，当提供的值是相对路径时，被解析为相对于撰写文件的路径，此目录也是发送到Docker守护进程的context

|  |  |
| --- | --- |
| 1  2 | build:    context: ./dir |

**5,dockerfile**

　　使用dockerfile文件来构建，必须指定构建路径

|  |  |
| --- | --- |
| 1  2  3 | build:    context: .    dockerfile: Dockerfile-alternate |

**6,commond**

　　使用command可以覆盖容器启动后默认执行的命令。

|  |  |
| --- | --- |
| 1 | command: bundle exec thin -p 3000 |

**7,container\_name**

　　Compose的容器名称格式是：<项目名称><服务名称><序号>  
可以自定义项目名称、服务名称，但如果想完全控制容器的命名，可以使用标签指定：

|  |  |
| --- | --- |
| 1 | container\_name: app |

**8,depends\_on**

在使用Compose时，最大的好处就是少打启动命令，但一般项目容器启动的顺序是有要求的，如果直接从上到下启动容器，必然会因为容器依赖问题而启动失败。例如在没启动数据库容器的时候启动应用容器，应用容器会因为找不到数据库而退出。depends\_on标签用于解决容器的依赖、启动先后的问题

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | version: '2'  services:    web:      build: .      depends\_on:        - db        - redis    redis:      image: redis    db:      image: postgres |

 　　上述YAML文件定义的容器会先启动redis和db两个服务，最后才启动web 服务。

**9,PID**

　　pid: "host"  
　　将PID模式设置为主机PID模式，跟主机系统共享进程命名空间。容器使用pid标签将能够访问和操纵其他容器和宿主机的名称空间。

**10,ports**

　　ports用于映射端口的标签。  
　　使用HOST:CONTAINER格式或者只是指定容器的端口，宿主机会随机映射端口。

|  |  |
| --- | --- |
| 1  2  3  4  5 | ports:   - "3000"   - "8000:8000"   - "49100:22"   - "127.0.0.1:8001:8001" |

 　　当使用HOST:CONTAINER格式来映射端口时，如果使用的容器端口小于60可能会得到错误得结果，因为YAML将会解析xx:yy这种数字格式为60进制。所以建议采用字符串格式。

**11,extra\_hosts**

　　添加主机名的标签，会在/etc/hosts文件中添加一些记录。

|  |  |
| --- | --- |
| 1  2  3 | extra\_hosts:   - "somehost:162.242.195.82"   - "otherhost:50.31.209.229" |

 　　启动后查看容器内部hosts：

|  |  |
| --- | --- |
| 1  2 | 162.242.195.82  somehost  50.31.209.229   otherhost |

**12,volumes**

　　挂载一个目录或者一个已存在的数据卷容器，可以直接使用 [HOST:CONTAINER]格式，或者使用[HOST:CONTAINER:ro]格式，后者对于容器来说，数据卷是只读的，可以有效保护宿主机的文件系统。  
Compose的数据卷指定路径可以是相对路径，使用 . 或者 .. 来指定相对目录。  
数据卷的格式可以是下面多种形式

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | volumes:    // 只是指定一个路径，Docker 会自动在创建一个数据卷（这个路径是容器内部的）。    - /var/lib/mysql    // 使用绝对路径挂载数据卷    - /opt/data:/var/lib/mysql    // 以 Compose 配置文件为中心的相对路径作为数据卷挂载到容器。    - ./cache:/tmp/cache    // 使用用户的相对路径（~/ 表示的目录是 /home/<用户目录>/ 或者 /root/）。    - ~/configs:/etc/configs/:ro    // 已经存在的命名的数据卷。    - datavolume:/var/lib/mysql |

 　　如果不使用宿主机的路径，可以指定一个volume\_driver。  
　　volume\_driver: mydriver

**13,volumes\_from**

　　从另一个服务或容器挂载其数据卷：

|  |  |
| --- | --- |
| 1  2  3 | volumes\_from:     - service\_name       - container\_name |

**14,dns**

　　自定义DNS服务器。可以是一个值，也可以是一个列表。

|  |  |
| --- | --- |
| 1  2  3  4 | dns：8.8.8.8  dns：      - 8.8.8.8        - 9.9.9.9 |

**15,expose**

　　暴露端口，但不映射到宿主机，只允许能被连接的服务访问。仅可以指定内部端口为参数，如下所示：

|  |  |
| --- | --- |
| 1  2  3 | expose:      - "3000"      - "8000" |

**16,links**

　　链接到其它服务中的容器。使用服务名称（同时作为别名），或者“服务名称:服务别名”（如 SERVICE:ALIAS），例如：

|  |  |
| --- | --- |
| 1  2  3  4 | links:      - db      - db:database      - redis |

**17,net**

　　设置网络模式。

|  |  |
| --- | --- |
| 1  2  3 | net: "bridge"  net: "none"  net: "host" |

**四，Docker-compose模板文件示例**

**1，Docker-compose模板文件编写**

　　docker-compose.yml

|  |  |
| --- | --- |
| 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 | version: '2'  services:    web1:      image: nginx      ports:        - "6061:80"      container\_name: "web1"      networks:        - dev    web2:      image: nginx      ports:        - "6062:80"      container\_name: "web2"      networks:        - dev        - pro    web3:      image: nginx      ports:        - "6063:80"      container\_name: "web3"      networks:        - pro    networks:    dev:      driver: bridge    pro:      driver: bridge |

 　　docker-compose.yml文件指定了3个web服务

**2，启动应用**

　　创建一个webapp目录，将docker-compose.yaml文件拷贝到webapp目录下，使用docker-compose启动应用。

|  |  |
| --- | --- |
| 1 | docker-compose up -d |

**3，服务访问**

　　通过浏览器访问web1，web2，web3

|  |  |
| --- | --- |
| 1  2  3 | http://127.0.0.1:6061  http://127.0.0.1:6062  http://127.0.0.1:6063 |

**1**

**集群**

有很多项目，客户端发送请求的时候，前面加上nginx，发送请求的时候发送给nginx，主要做反向代理和转发，负载均衡，比如有10个请求，有权重，配置参数，这个可以让接口能够解压服务器的压力。

集群的定义以及类别定义

集群定义

集群类型

scale on:纵向扩展

scale out:横向扩展

LB:Load Balancing:负载均衡集群

HA:High Availability:高可用集群

HP:Hight Performance:高性能

三种集群之间的区别

集群定义

集群（Cluster）技术是一种较新的技术，通过集群技术，可以在付出较低成本的情况下获得在性能、可靠性、灵活性方面的相对较高的收益，其任务调度则是集群系统中的核心技术。集群是一组相互独立的、通过高速网络互联的计算机，它们构成了一个组，并以单一系统的模式加以管理。一个客户与集群相互作用时，集群像是一个独立的服务器。集群配置是用于提高可用性和可缩放性。

集群系统的主要优点：高可扩展性、高可用性、高性能、高性价比。

集群类型

scale on:纵向扩展

将服务器的内存容量调大和 CPU 数量增加些（简单说升级服务器硬件）。

缺点：在一定的范围之内它的性能是上升的趋势，但是超出范围之后就是下降的趋势。因为随着 CPU 的个数增加需要给 CPU 进行仲裁，而且随着 CPU 个数的增加资源竞争性越大。

scale out:横向扩展

一台服务器应付不过来，就再增加一台服务器。

优点：增减服务器很方便，而且没有向上扩展随着增加性能下降。

向外扩张的工作模式：当客户端向服务器端发送请求，服务器端只拿出来一台服务器来相应我们的客户端的请求。

LB:Load Balancing:负载均衡集群

负载均衡集群中有一个分发器或者叫调度器，称之为 Director，它处在多台服务器的上面，分发器根据内部锁定义的规则或调度方式从下面的服务器群中选择一个以此来响应客户端发送的请求。

HA:High Availability:高可用集群

高可用集群是服务的可用性比较高，当我们某台服务器死机后不会造成我们的服务不可用。其工作模式则是将一个具有故障的服务转交给一个正常工作的服务器，从而达到服务不会中断。一般来说我们集群中工作在前端（分发器）的服务器都会对我们的后端服务器做一个健康检查，如果发现我们服务器当机就不会对其在做转发。

衡量标准：

可用性 = 在线时间 / (在线时间 + 故障处理时间)

# 99%、99.9%、99.99%、99.999%

HP:Hight Performance:高性能

高性能的集群是当某一个任务量非常大的时候，我们做一个集群共同来完成这一个任务。这种处理方式我们称为并行处理集群，并行处理集群是将大任务划分为小任务，分别进行处理的机制。一般这样的集群用来科学研究与大数据运算等方面的工作。现在比较火的 Hadoop 就是使用的并行处理集群。

三种集群之间的区别

负载均衡着重在于提供服务并发处理能力的集群

高可用以提升服务在线的能力的集群

高性能着重用于处理一个海量任务

**微服务**

就是把一个大项目分解成为多个小项目。比如用户模块，内容模块，不同模块直接是需要进行调用的，resful接口的方式进行调用，还有一种就是rpc（远程过程调用，基于socket写的，的），大型项目一般使用其他语言。调用远程的函数像调用本地的一样，通过框架（谷歌的grpc,或者阿里开源的dubbo）。

拆成多个项目之后，部署在不同的机器上，或者同一个机器上，如果对于某一个模块访问过多，可以部署在多台服务器上，经过aginx进行转发，这样就可以做到访问。都可以部署到docker 上，这个可以快熟的进行部署。

[**k8s与Docker有啥关系**](https://www.cnblogs.com/haha-9527/p/10463466.html)

**简要介绍：**

    官方定义1：Docker是一个开源的应用容器引擎，开发者可以打包他们的应用及依赖到一个可移植的容器中，发布到流行的Linux机器上，也可实现虚拟化。

    官方定义2：k8s是一个开源的容器集群管理系统，可以实现容器集群的自动化部署、自动扩缩容、维护等功能。

**与传统技术对比：**

    接下来我们看两张经典的图：

一、从虚拟化角度：

图1：

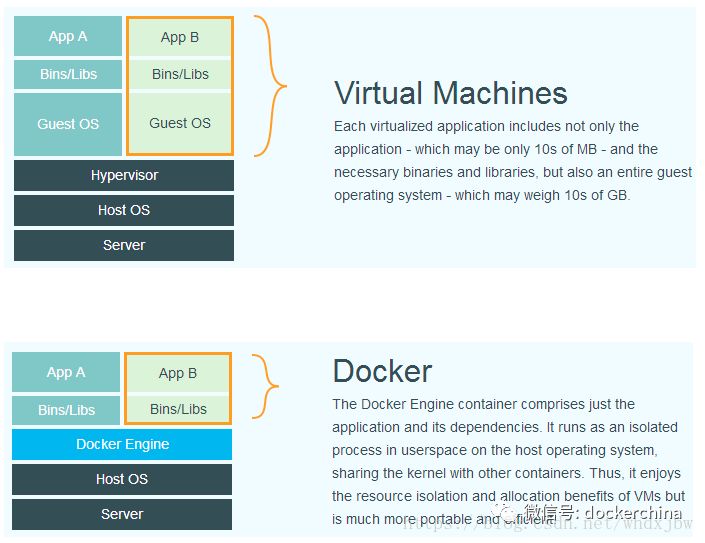
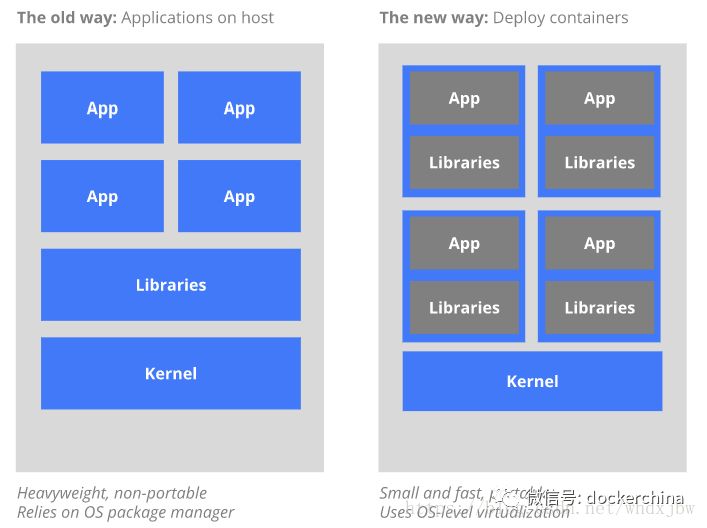


    图1是Docker容器与传统虚拟化方式的不同之处，传统的虚拟技术，在将物理硬件虚拟成多套硬件后，需要再每套硬件上都部署一个操作系统，接着在这些操作系统上运行相应的应用程序。而Docker容器内的应用程序进程直接运行在宿主机(真实物理机)的内核上，Docker引擎将一些各自独立的应用程序和它们各自的依赖打包，相互独立直接运行于未经虚拟化的宿主机硬件上，同时各个容器也没有自己的内核，显然比传统虚拟机更轻便。

    每个集群有多个节点，每个节点可创建多个容器，kuberbete就是管理这些应用程序所在的小运行环境（container）而生。

二、从部署角度

图2：



    注意，大家别把这幅图与上面Docker的那张图混淆了，图1是从虚拟化角度，说明了为应用提供必要的运行环境所需要做的虚拟化操作（即：传统：虚拟出的虚拟机装操作系统、Docker：容器引擎管理下的容器）。

    而图2是在这些具体运行环境上进行真实应用部署时的情况，传统方式是将所有应用直接部署在同一个物理机器节点上，这样每个App的依赖都是完全相同的，无法做到App之间隔离，当然，为了隔离，我们也可以通过创建虚拟机的方式来将App部署到其中（就像图1上半部分那样），但这样太过繁重，故比虚拟机更轻便的Docker技术出现，现在我们通过部署Container容器的技术来部署应用，全部Container运行在容器引擎上即可。既然嫌弃虚拟机繁重，想用Docker，那好，你用吧，怎么用呢？手动一个一个创建？当然不，故kubernetes技术便出现了，以kubernetes为代表的容器集群管理系统，这时候就该上场表演了。

    说白了，我们用kubernetes去管理Docker集群，即可以将Docker看成Kubernetes内部使用的低级别组件。另外，kubernetes不仅仅支持Docker，还支持Rocket，这是另一种容器技术。

还是老问题： 既然有了 Docker，为什么还要 Kubernetes？  
回答这个问题，得从为什么需要引入Docker这个问题先开始。你会说，因为需要容器化部署。那为什么需要容器化部署呢？就这个问题还可以走得更远一点。这里我就偷懒了，先打住。看看容器化部署后出现的新问题，即容器实例越来越多。因为容器实例越来越多，就像鸡蛋太多需要一个篮子来装一样，Kubernetes就是那个篮子。当然这不是一个简单的篮子，此处也不扩展了。  
标准答案就是：因为有了 Docker，所以才要Kubernetes。是不是感觉进入一个loop，我也这感觉。  
从图中也可以看出，Docker容器技术是Kubernetes平台的基础。容器技术主要作用是隔离，通过对系统的关键资源的隔离，实现了主机抽象。Kubernetes平台则是在抽象主机的基础上，实现了集群抽象。  
如果用一句话做个总结，就是：  
容器，提供应用级的主机抽象；Kubernetes，提供应用级的集群抽象。

[**k8s和docker区别**](https://www.cnblogs.com/baby0123/p/12091105.html)

**简要介绍：**

　　docker是一个开源的应用容器引擎，开发者可以打包他们的应用以及依赖到一个容器中，发布到流行的liunx系统上，或者实现虚拟化。

　　k8s是一个开源的容器集群管理系统，可以实现容器集群的自动化部署、自动扩缩容、维护等~

**一、虚拟化角度：**

　传统的虚拟技术，在将物理硬件虚拟成多套硬件后，需要再每套硬件上都部署一个操作系统，接着在这些操作系统上运行相应的应用程序。

Docker容器内的应用程序进程直接运行在宿主机(真实物理机)的内核上，Docker引擎将一些各自独立的应用程序和它们各自的依赖打包，相互独立直接运行于未经虚拟化的宿主机硬件上，同时各个容器也没有自己的内核，显然比传统虚拟机更轻便。每个集群有多个节点，每个节点可，我们的kuberbete就是管理这些应用程序所在的小运行环境（container）而生。

**二、部署角度：**

　　传统方式是将所有应用直接部署在同一个物理机器节点上，这样每个App的依赖都是完全相同的，无法做到App之间隔离，当然，为了隔离，我们也可以通过创建虚拟机的方式来将App部署到其中，但这样太过繁重，故比虚拟机更轻便的Docker技术出现，现在我们通过部署Container容器的技术来部署应用，全部Container运行在容器引擎上即可。以kubernetes为代表的容器集群管理系统，我们用kubernetes去管理Docker集群，即可以将Docker看成Kubernetes内部使用的低级别组件。另外，kubernetes不仅仅支持Docker，还支持Rocket，这是另一种容器技术。

**k8s维护**

**1、 不可调度**

kubectl cordon k8s-node-1

kubectl uncordon k8s-node-1 #取消

**2、驱逐已经运行的业务容器**

kubectl drain --ignore-daemonsets --delete-local-data <node>

kubectl drain {{node-name}} #标记节点为不可调度，并驱逐节点上的pod

**3、停docker、kubelet服务（下线的那台服务器）**

systemctl stop kubelet

systemctl stop docker

**4、删除节点**

kubectl delete node

天天向上，空杯心态。

**​K8s简介**

Kubernetes是Google 2014年创建管理的，是Google 10多年大规模容器管理技术Borg的开源版本。它是容器集群管理系统，是一个开源的平台，可以实现容器集群的自动化部署、自动扩缩容、维护等功能。

通过Kubernetes你可以：

* 快速部署应用
* 快速扩展应用
* 无缝对接新的应用功能
* 节省资源，优化硬件资源的使用

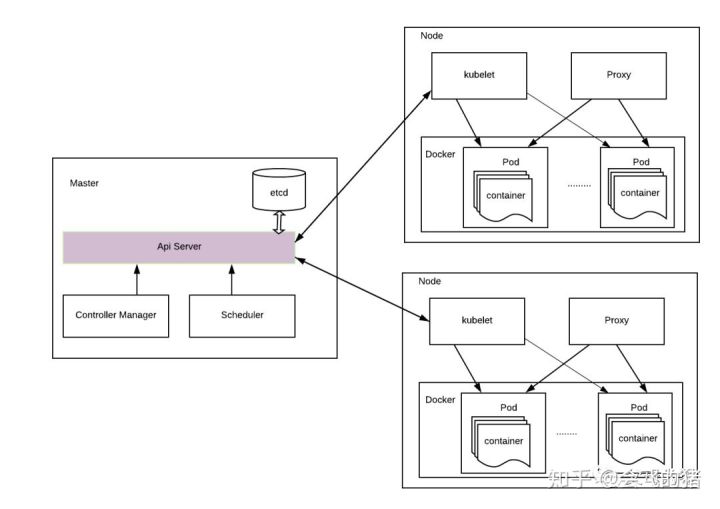
Kubernetes 特点:

* 可移植: 支持公有云，私有云，混合云，多重云（multi-cloud）
* 可扩展: 模块化, 插件化, 可挂载, 可组合
* 自动化: 自动部署，自动重启，自动复制，自动伸缩/扩展

K8s总体架构

K8s集群由两节点组成：Master和Node。在Master上运行etcd,Api Server,Controller Manager和Scheduler四个组件。后三个组件构成了K8s的总控中心，负责对集群中所有资源进行管控和调度.在每个node上运行kubectl,proxy和docker daemon三个组件,负责对节点上的Pod的生命周期进行管理，以及实现服务代理的功能。另外所有节点上都可以运行kubectl命令行工具。

API Server作为集群的核心，负责集群各功能模块之间的通信。集群内的功能模块通过Api Server将信息存入到etcd,其他模块通过Api Server读取这些信息，从而实现模块之间的信息交互。Node节点上的Kubelet每隔一个时间周期，通过Api Server报告自身状态，Api Server接收到这些信息后，将节点信息保存到etcd中。Controller Manager中 的node controller通过Api server定期读取这些节点状态信息，并做响应处理。Scheduler监听到某个Pod创建的信息后，检索所有符合该pod要求的节点列表，并将pod绑定到节点列表中最符合要求的节点上。如果scheduler监听到某个Pod被删除，则调用api server删除该Pod资源对象。kubelet监听pod信息，如果监听到pod对象被删除，则删除本节点上的相应的pod实例，如果监听到修改Pod信息，则会相应地修改本节点的Pod实例。



Kubernetes主要由以下几个核心组件组成：

* etcd保存了整个集群的状态；
* apiserver提供了资源操作的唯一入口，并提供认证、授权、访问控制、API注册和发现等机制；
* controller manager负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
* scheduler负责资源的调度，按照预定的调度策略将Pod调度到相应的机器上；
* kubelet负责本Node节点上的Pod的创建、修改、监控、删除等生命周期管理，同时Kubelet定时“上报”本Node的状态信息到Api Server里；
* Container runtime负责镜像管理以及Pod和容器的真正运行（CRI）；
* kube-proxy负责为Service提供cluster内部的服务发现和负载均衡；

Kubernetes可以做什么？

使用Web服务，用户希望应用程序能够7\*24小时全天运行，开发人员希望每天多次部署新的应用版本。通过应用容器化可以实现这些目标，使应用简单、快捷的方式更新和发布，也能实现热更新、迁移等操作。使用Kubernetes能确保程序在任何时间、任何地方运行，还能扩展更多有需求的工具/资源。Kubernetes积累了Google在容器化应用业务方面的经验，以及社区成员的实践，是能在生产环境使用的开源平台。

**一、核心概念**

1、Node

Node作为集群中的工作节点，运行真正的应用程序，在Node上Kubernetes管理的最小运行单元是Pod。Node上运行着Kubernetes的Kubelet、kube-proxy服务进程，这些服务进程负责Pod的创建、启动、监控、重启、销毁、以及实现软件模式的负载均衡。

Node包含的信息：

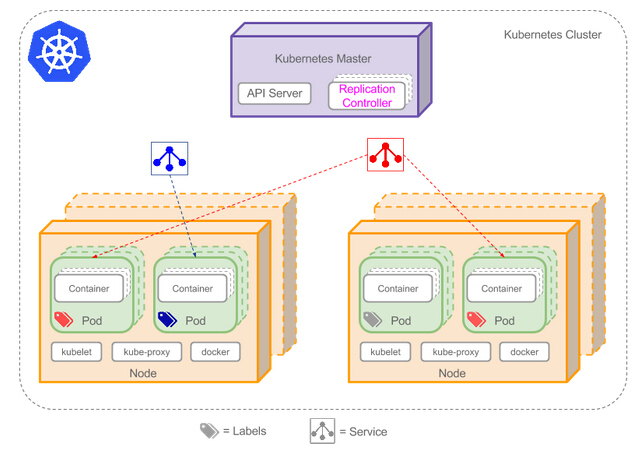
* Node地址：主机的IP地址，或Node ID。
* Node的运行状态：Pending、Running、Terminated三种状态。
* Node Condition：…
* Node系统容量：描述Node可用的系统资源，包括CPU、内存、最大可调度Pod数量等。
* 其他：内核版本号、Kubernetes版本等。

查看Node信息：kubectl describe node

2、Pod

Pod是Kubernetes最基本的操作单元，包含一个或多个紧密相关的容器，一个Pod可以被一个容器化的环境看作应用层的“逻辑宿主机”；一个Pod中的多个容器应用通常是紧密耦合的，Pod在Node上被创建、启动或者销毁；每个Pod里运行着一个特殊的被称之为Pause的容器，其他容器则为业务容器，这些业务容器共享Pause容器的网络栈和Volume挂载卷，因此他们之间通信和数据交换更为高效，在设计时我们可以充分利用这一特性将一组密切相关的服务进程放入同一个Pod中。

同一个Pod里的容器之间仅需通过localhost就能互相通信。



一个Pod中的应用容器共享同一组资源：

* PID命名空间：Pod中的不同应用程序可以看到其他应用程序的进程ID；
* 网络命名空间：Pod中的多个容器能够访问同一个IP和端口范围；
* IPC命名空间：Pod中的多个容器能够使用SystemV IPC或POSIX消息队列进行通信；
* UTS命名空间：Pod中的多个容器共享一个主机名；
* Volumes（共享存储卷）：Pod中的各个容器可以访问在Pod级别定义的Volumes；

Pod的生命周期通过Replication Controller来管理；通过模板进行定义，然后分配到一个Node上运行，在Pod所包含容器运行结束后，Pod结束。

Kubernetes为Pod设计了一套独特的网络配置，包括：为每个Pod分配一个IP地址，使用Pod名作为容器间通信的主机名等。

3、Service

在Kubernetes的世界里，虽然每个Pod都会被分配一个单独的IP地址，但这个IP地址会随着Pod的销毁而消失，这就引出一个问题：如果有一组Pod组成一个集群来提供服务，那么如何来访问它呢？**Service！**

一个Service可以看作一组提供相同服务的Pod的对外访问接口，Service作用于哪些Pod是通过Label Selector来定义的。

* 拥有一个指定的名字（比如my-mysql-server）；
* 拥有一个虚拟IP（Cluster IP、Service IP或VIP）和端口号，销毁之前不会改变，只能内网访问；
* 能够提供某种远程服务能力；
* 被映射到了提供这种服务能力的一组容器应用上；

如果Service要提供外网服务，需指定公共IP和NodePort，或外部负载均衡器；

**NodePort**   
系统会在Kubernetes集群中的每个Node上打开一个主机的真实端口，这样，能够访问Node的客户端就能通过这个端口访问到内部的Service了

4、Volume：Volume是Pod中能够被多个容器访问的共享目录。

5、Label：Label以key/value的形式附加到各种对象上，如Pod、Service、RC、Node等，以识别这些对象，管理关联关系等，如Service和Pod的关联关系。

6、RC（Replication Controller）

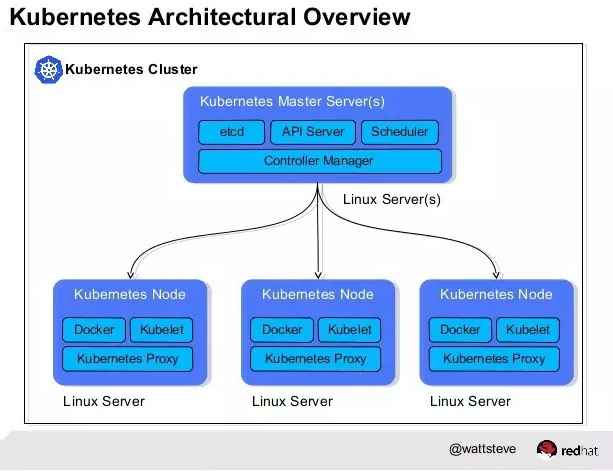
* 目标Pod的定义；
* 目标Pod需要运行的副本数量；
* 要监控的目标Pod标签（Lable）；

Kubernetes通过RC中定义的Lable筛选出对应的Pod实例，并实时监控其状态和数量，如果实例数量少于定义的副本数量（Replicas），则会根据RC中定义的Pod模板来创建一个新的Pod，然后将此Pod调度到合适的Node上启动运行，直到Pod实例数量达到预定目标。

**二、Kubernetes总体架构**

Master和Node

Kubernetes将集群中的机器划分为一个Master节点和一群工作节点（Node）。其中，Master节点上运行着集群管理相关的一组进程**etcd、API Server、Controller Manager、Scheduler**，后三个组件构成了Kubernetes的总控中心，这些进程实现了整个集群的资源管理、Pod调度、弹性伸缩、安全控制、系统监控和纠错等管理功能，并且全都是自动完成。在每个Node上运行**Kubelet、Proxy、**[**Docker**](http://lib.csdn.net/base/docker)**daemon**三个组件，负责对本节点上的Pod的生命周期进行管理，以及实现服务代理的功能。



**流程**   
通过Kubectl提交一个创建RC的请求，该请求通过API Server被写入etcd中，此时Controller Manager通过API Server的监听资源变化的接口监听到这个RC事件，分析之后，发现当前集群中还没有它所对应的Pod实例，于是根据RC里的Pod模板定义生成一个Pod对象，通过API Server写入etcd，接下来，此事件被Scheduler发现，它立即执行一个复杂的调度流程，为这个新Pod选定一个落户的Node，然后通过API Server讲这一结果写入到etcd中，随后，目标Node上运行的Kubelet进程通过API Server监测到这个“新生的”Pod，并按照它的定义，启动该Pod并任劳任怨地负责它的下半生，直到Pod的生命结束。

随后，我们通过Kubectl提交一个新的映射到该Pod的Service的创建请求，Controller Manager会通过Label标签查询到相关联的Pod实例，然后生成Service的Endpoints信息，并通过API Server写入到etcd中，接下来，所有Node上运行的Proxy进程通过API Server查询并监听Service对象与其对应的Endpoints信息，建立一个软件方式的负载均衡器来实现Service访问到后端Pod的流量转发功能。

* etcd   
  用于持久化存储集群中所有的资源对象，如Node、Service、Pod、RC、Namespace等；API Server提供了操作etcd的封装接口API，这些API基本上都是集群中资源对象的增删改查及监听资源变化的接口。
* API Server   
  提供了资源对象的唯一操作入口，其他所有组件都必须通过它提供的API来操作资源数据，通过对相关的资源数据“全量查询”+“变化监听”，这些组件可以很“实时”地完成相关的业务功能。
* Controller Manager   
  集群内部的管理控制中心，其主要目的是实现Kubernetes集群的故障检测和恢复的自动化工作，比如根据RC的定义完成Pod的复制或移除，以确保Pod实例数符合RC副本的定义；根据Service与Pod的管理关系，完成服务的Endpoints对象的创建和更新；其他诸如Node的发现、管理和状态监控、死亡容器所占磁盘空间及本地缓存的镜像文件的清理等工作也是由Controller Manager完成的。
* Scheduler   
  集群中的调度器，负责Pod在集群节点中的调度分配。
* Kubelet   
  负责本Node节点上的Pod的创建、修改、监控、删除等全生命周期管理，同时Kubelet定时“上报”本Node的状态信息到API Server里。
* Proxy   
  实现了Service的代理与软件模式的负载均衡器。

客户端通过Kubectl命令行工具或Kubectl Proxy来访问Kubernetes系统，在Kubernetes集群内部的客户端可以直接使用Kuberctl命令管理集群。Kubectl Proxy是API Server的一个反向代理，在Kubernetes集群外部的客户端可以通过Kubernetes Proxy来访问API Server。

API Server内部有一套完备的安全机制，包括认证、授权和准入控制等相关模块。

**技术干货：Kubernetes面试题汇总及答案**

　　Kubernetes一直是当今业界的流行语，也是最好的编排工具。它吸引了许多想要提升自己职业生涯的经验丰富的专业人士，所以 **陕西优就业**小编今天给大家分享的就是相关面试题。

**1、什么是Kubernetes?**

　　Kubernetes是一个开源容器管理工具，负责容器部署，容器扩缩容以及负载平衡。作为Google的创意之作，它提供了出色的社区，并与所有云提供商合作。因此，我们可以说Kubernetes不是一个容器化平台，而是一个多容器管理解决方案。

**2、Kubernetes与Docker有什么关系?**

　　众所周知，Docker提供容器的生命周期管理，Docker镜像构建运行时容器。但是，由于这些单独的容器必须通信，因此使用Kubernetes。因此，我们说Docker构建容器，这些容器通过Kubernetes相互通信。因此，可以使用Kubernetes手动关联和编排在多个主机上运行的容器。

**3、什么是Container Orchestration?**

　　考虑一个应用程序有5-6个微服务的场景。现在，这些微服务被放在单独的容器中，但如果没有容器编排就无法进行通信。因此，由于编排意味着所有乐器在音乐中和谐共处，所以类似的容器编排意味着各个容器中的所有服务协同工作以满足单个服务器的需求。

**4、Kubernetes如何简化容器化部署?**

　　由于典型应用程序将具有跨多个主机运行的容器集群，因此所有这些容器都需要相互通信。因此，要做到这一点，你需要一些能够负载平衡，扩展和监控容器的东西。由于Kubernetes与云无关并且可以在任何公共/私有提供商上运行，因此必须是您简化容器化部署的选择。

**5、您对Kubernetes的集群了解多少?**

　　Kubernetes背后的基础是我们可以实施所需的状态管理，我的意思是我们可以提供特定配置的集群服务，并且集群服务将在基础架构中运行并运行该配置。

　　因此，正如您所看到的，部署文件将具有提供给集群服务所需的所有配置。现在，部署文件将被提供给API，然后由集群服务决定如何在环境中安排这些pod，并确保正确运行的pod数量。

　　因此，位于服务前面的API，工作节点和节点运行的Kubelet进程，共同构成了Kubernetes集群。

**6、Kubernetes Architecture的不同组件有哪些?**

　　Kubernetes Architecture主要有两个组件 - 主节点和工作节点。如下图所示，master和worker节点中包含许多内置组件。主节点具有kube-controller-manager，kube-apiserver，kube-scheduler等。而工作节点具有在每个节点上运行的kubelet和kube-proxy。

**7、您能否介绍一下Kubernetes中主节点的工作情况?**

　　Kubernetes master控制容器存在的节点和节点内部。现在，这些单独的容器包含在容器内部和每个容器内部，您可以根据配置和要求拥有不同数量的容器。

　　因此，如果必须部署pod，则可以使用用户界面或命令行界面部署它们。然后，在节点上调度这些pod，并根据资源需求，将pod分配给这些节点。kube-apiserver确保在Kubernetes节点和主组件之间建立通信。

**8、kube-apiserver和kube-scheduler的作用是什么?**

　　kube -apiserver遵循横向扩展架构，是主节点控制面板的前端。这将公开Kubernetes主节点组件的所有API，并负责在Kubernetes节点和Kubernetes主组件之间建立通信。

　　kube-scheduler负责工作节点上工作负载的分配和管理。因此，它根据资源需求选择最合适的节点来运行未调度的pod，并跟踪资源利用率。它确保不在已满的节点上调度工作负载。

**9、你对Kubernetes的负载均衡器有什么了解?**

　　负载均衡器是暴露服务的最常见和标准方式之一。根据工作环境使用两种类型的负载均衡器，即内部负载均衡器或外部负载均衡器。内部负载均衡器自动平衡负载并使用所需配置分配容器，而外部负载均衡器将流量从外部负载引导至后端容器。

**10、Replica Set 和 Replication Controller之间有什么区别?**

　　Replica Set 和 Replication Controller几乎完全相同。它们都确保在任何给定时间运行指定数量的pod副本。不同之处在于复制pod使用的选择器。Replica Set使用基于集合的选择器，而Replication Controller使用基于权限的选择器。

　　Equity-Based选择器：这种类型的选择器允许按标签键和值进行过滤。因此，在外行术语中，基于Equity的选择器将仅查找与标签具有完全相同短语的pod。

　　示例：假设您的标签键表示app = nginx，那么，使用此选择器，您只能查找标签应用程序等于nginx的那些pod。

　　Selector-Based选择器：此类型的选择器允许根据一组值过滤键。因此，换句话说，基于Selector的选择器将查找已在集合中提及其标签的pod。

　　示例：假设您的标签键在(nginx，NPS，Apache)中显示应用程序。然后，使用此选择器，如果您的应用程序等于任何nginx，NPS或Apache，则选择器将其视为真实结果。

**50 个你必须掌握的 Kubernetes 面试题**

Kubernetes 一直是当今业界的流行语，也是最好的编排工具。它吸引了许多想要提升自己职业生涯的经验丰富的专业人士。

Huwaei、Pokemon、Box、eBay、Ing、Yahoo Japan、SAP、纽约时报、Open AI、Sound Cloud 等跨国公司也使用 Kubernetes。我相信你已经知道这些事实，这也是促使你打开这个 Kubernetes 面试问题文章原因。

在这篇关于 Kubernetes 面试问题的文章中，我将讨论在面试中提出的与 Kubernetes 相关的最重要问题。因此，为了您的理解，我将此文内容分为以下 4 个部分：

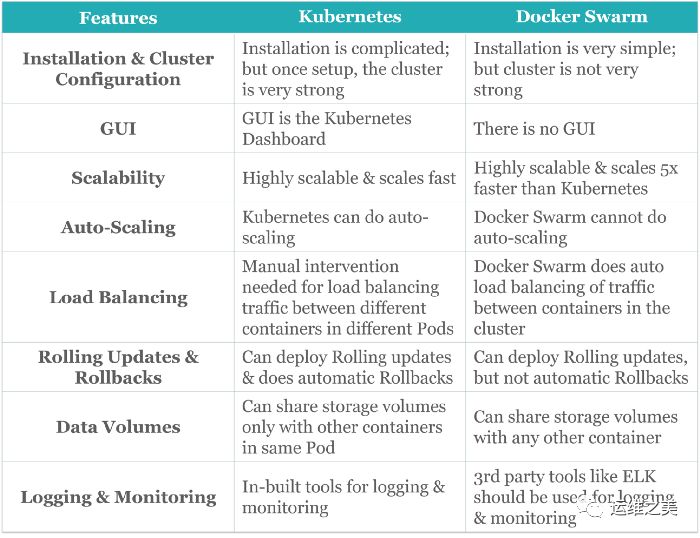
* Kubernetes 基本面试问题
* 基于架构的面试问题
* 基于场景的面试问题
* 多项选择题

现在，让我们开始吧!

**基本的 Kubernetes 面试问题**

这部分问题将包含您需要了解的与 Kubernetes 工作相关的所有基本问题。

**Q1、Kubernetes 与 Docker Swarm 的区别如何？**



**Q2、什么是 Kubernetes？**



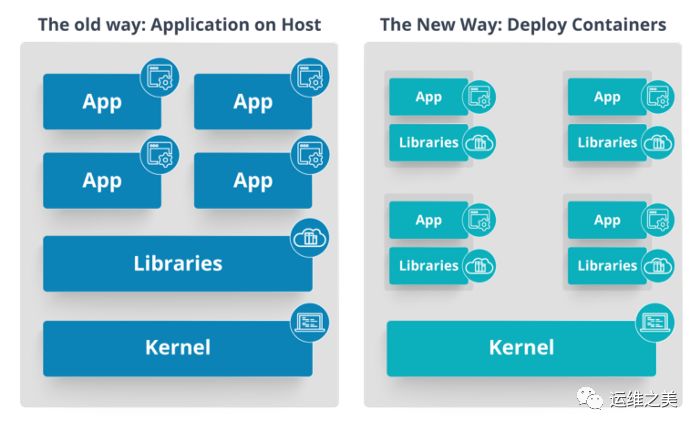
Kubernetes 是一个开源容器管理工具，负责容器部署，容器扩缩容以及负载平衡。作为 Google 的创意之作，它提供了出色的社区，并与所有云提供商合作。因此，我们可以说 Kubernetes 不是一个容器化平台，而是一个多容器管理解决方案。

**Q3、Kubernetes 与 Docker 有什么关系？**

众所周知，Docker 提供容器的生命周期管理和 Docker 镜像构建运行时容器。但是，由于这些单独的容器有时必须跨主机通信，这时我们需要使用 Kubernetes 来解决这个问题。

因此，我们说 Docker 构建容器，但这些容器通过 Kubernetes 来进行跨主机相互通信。我们还可以使用 Kubernetes 手动关联和编排在多个主机上运行的容器。

**Q4、在主机和容器上部署应用程序有什么区别？**



请参考上图。左侧架构表示在主机上部署应用程序。因此，这种架构将具有操作系统，然后操作系统将具有内核，该内核将在应用程序所需的操作系统上安装各种库。因此，在这种框架中，您可以拥有 N 个应用程序，并且所有应用程序将共享该操作系统中存在的库，而在容器中部署应用程序时，体系结构则略有不同。

这种架构将有一个内核，这是唯一一个在所有应用程序之间唯一共同的东西。因此，如果有一个需要 Java 的特定应用程序，那么我们将获得访问 Java 的特定应用程序，如果有另一个需要 Python 的应用程序，则只有该特定应用程序才能访问 Python。

您可以在图表右侧看到的各个块基本上是容器化的，并且这些块与其他应用程序隔离。因此，应用程序具有与系统其余部分隔离的必要库和二进制文件，并且不能被任何其他应用程序侵占。

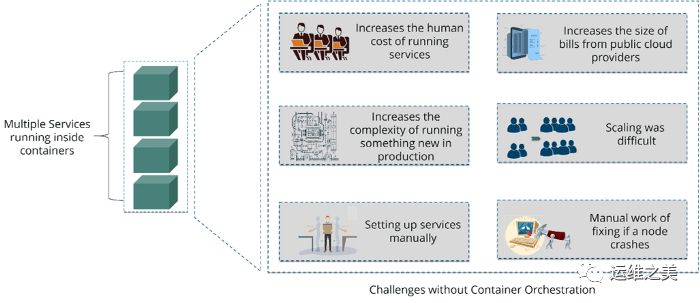
**Q5、什么是 Container Orchestration？**

考虑一个应用程序有 5-6 个微服务的场景。

现在，这些微服务被放在单独的容器中，但如果没有容器编排就无法进行通信。因此，由于编排意味着所有乐器在音乐中和谐共处，所以类似的容器编排意味着各个容器中的所有服务协同工作以满足单个服务器的需求。

**Q6、Container Orchestration 需要什么？**

考虑到你有 5-6 个微服务用于执行各种任务的单个应用程序，所有这些微服务都放在容器中。现在，为了确保这些容器彼此通信，我们需要容器编排。



正如您在上图中所看到的，在没有使用容器编排的情况下，还存在许多挑战。因此，为了克服这些挑战，容器编排就到位了。

**Q7、Kubernetes 有什么特点？**

Kubernetes 的功能如下：



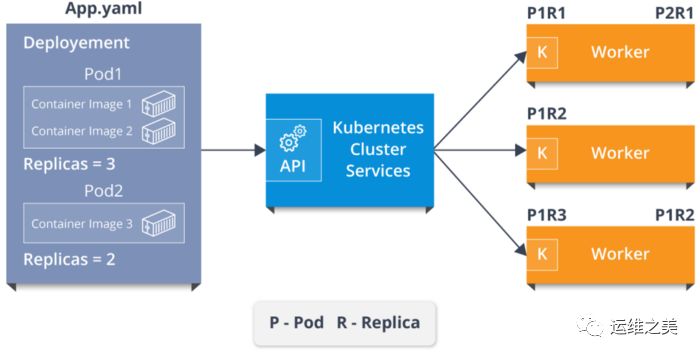
**Q8、Kubernetes 如何简化容器化部署？**

由于典型应用程序将具有跨多个主机运行的容器集群，因此所有这些容器都需要相互通信。因此，要做到这一点，你需要一些能够负载均衡、扩展和监控容器的东西。

由于 Kubernetes 与云无关并且可以在任何公共/私有提供商上运行，因此必须是您简化容器化部署的选择。

**Q9、您对 Kubernetes 的集群了解多少？**

Kubernetes 背后的基础是我们可以实施所需的状态管理，我的意思是我们可以提供特定配置的集群服务，并且集群服务将在基础架构中运行并运行该配置。



因此，正如您在上图中所看到的，部署文件将具有提供给集群服务所需的所有配置。

现在，部署文件将被提供给 API，然后由集群服务决定如何在环境中安排这些 Pod，并确保正确运行的 Pod 的数量。

因此，位于服务前面的 API，工作节点和节点运行的 Kubelet 进程，共同构成了 Kubernetes 集群。

**Q10、什么是 Google 容器引擎？**

Google Container Engine（GKE）是 Docker 容器和集群的开源管理平台。这个基于 Kubernetes 的引擎仅支持在 Google 的公共云服务中运行的群集。

**Q11、什么是 Heapster？**

Heapster 是由每个节点上运行的 Kubelet 提供的集群范围的数据聚合器。

此容器管理工具在 Kubernetes 集群上本机支持，并作为 Pod 运行，就像集群中的任何其他 Pod 一样。因此，它基本上可以发现集群中的所有节点，并通过本机上 Kubernetes 代理查询集群中 Kubernetes 节点的使用信息。

**Q12、什么是 Minikube？**

Minikube 是一种工具，可以在本地轻松运行 Kubernetes 集群。它是在虚拟机中运行一个单节点 Kubernetes 群集。

**Q13、什么是 Kubectl？**

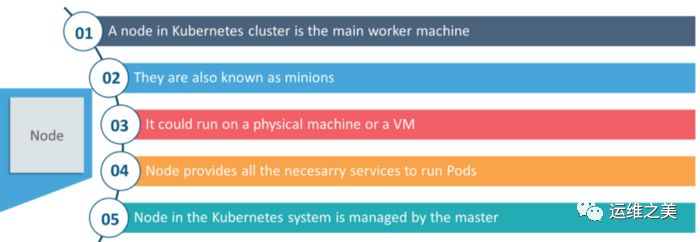
Kubectl 是一个命令行工具，您可以使用该工具将命令传递给集群。

因此，它基本上为 CLI 提供了针对 Kubernetes 集群运行命令的方法，以及创建和管理 Kubernetes 组件的各种方法。

**Q14、什么是 Kubelet？**

这是一个代理服务，它在每个节点上运行，并使从服务器与主服务器通信。因此，Kubelet 处理 PodSpec 中提供给它的容器的描述，并确保 PodSpec 中描述的容器运行正常。

**Q15、你对 Kubernetes 的一个节点有什么了解？**



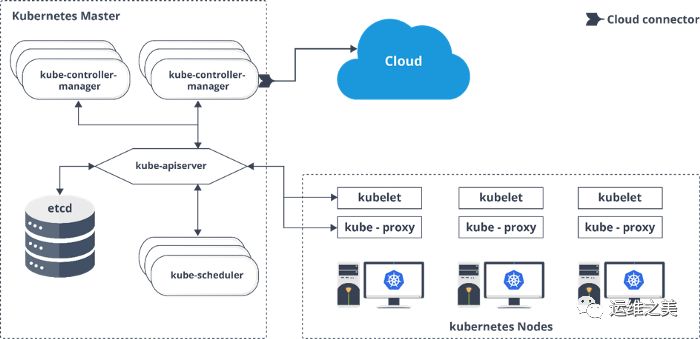
**基于 Kubernetes 架构的问题**

这部分问题将涉及到与 Kubernetes 架构相关的问题。

**Q1、Kubernetes Architecture 的不同组件有哪些？**

Kubernetes Architecture 主要有两个组件：主节点和工作节点。

如下图所示，Master 和 Worker 节点中包含许多内置组件。主节点具有 kube-controller-manager、kube-apiserver、kube-scheduler 等。而工作节点具有在每个节点上运行的 kubelet 和 kube-proxy。



**Q2、你对 Kube-proxy 有什么了解？**

Kube-proxy 可以在每个节点上运行，并且可以跨后端网络服务进行简单的 TCP/UDP 数据包转发。

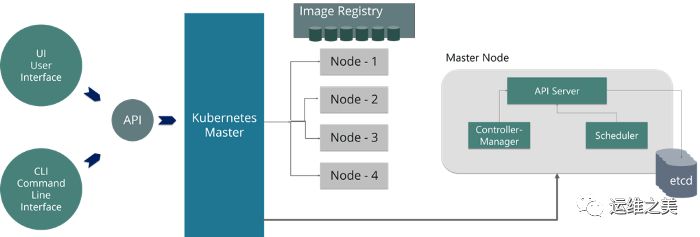
基本上，它是一个网络代理，它反映了每个节点上 Kubernetes API 中配置的服务。因此，Docker 可链接的兼容环境变量由代理打开的群集 IP 和端口提供。

**Q3、您能否介绍一下 Kubernetes 中主节点的工作情况？**

Kubernetes master 控制器存在的节点和节点内部。现在，这些单独的容器包含在容器内部和每个容器内部，您可以根据配置和要求拥有不同数量的容器。

因此，如果必须部署 Pod，则可以使用用户界面或命令行界面部署它们。然后，在节点上调度这些 Pod，并根据资源需求将 Pod 分配给这些节点。

Kube-apiserver 确保在 Kubernetes 节点和主要组件之间建立通信。



**Q4、Kube-apiserver 和 Kube-scheduler 的作用是什么？**

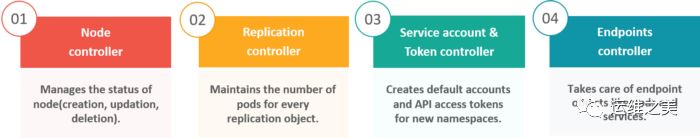
Kube-apiserver 遵循横向扩展架构，是主节点控制面板的前端。这将公开 Kubernetes 主节点组件的所有 API，并负责在 Kubernetes 节点和 Kubernetes 主组件之间建立通信。

kube-scheduler 负责工作节点上工作负载的分配和管理。因此，它根据资源需求选择最合适的节点来运行未调度的 Pod，并跟踪资源利用率。它确保不在已满的节点上调度工作负载。

**Q5、你能简要介绍一下 Kubernetes 控制管理器吗？**

多个控制器进程在主节点上运行，但是一起编译为单个进程运行，即 Kubernetes 控制器管理器。因此，Controller Manager 是一个嵌入控制器并执行命名空间创建和垃圾收集的守护程序。它拥有与 API 服务器通信以管理端点的责任。

因此，主节点上运行的不同类型的控制器管理器是：

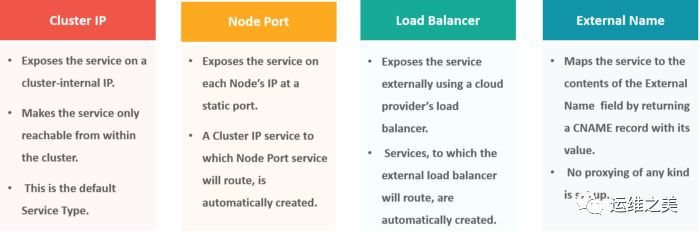


**Q6、什么是 Etcd？**

Etcd 是用 Go 编程语言编写的一个分布式键值存储，用于协调分布式工作的软件。因此，Etcd 用来存储 Kubernetes 集群的配置数据，这些数据代表在任何给定时间点的集群状态。

**Q7、Kubernetes 有哪些不同类型的服务？**

以下是使用的不同类型的服务：



**Q8、你对 Kubernetes 的负载均衡器有什么了解？**

负载均衡器是暴露服务的最常见和标准方式之一。

根据工作环境使用两种类型的负载均衡器，即内部负载均衡器或外部负载均衡器。内部负载均衡器自动平衡负载并使用所需配置分配容器，而外部负载均衡器将流量从外部负载引导至后端容器。

**Q9、什么是 Ingress 网络，它是如何工作的？**

Ingress 网络是一组规则，充当 Kubernetes 集群的入口点。

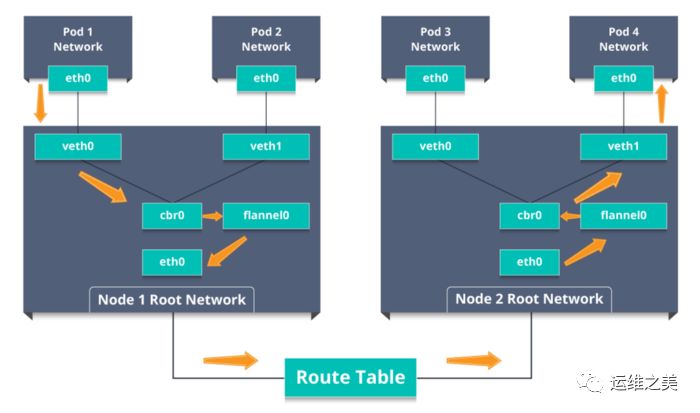
这允许入站连接，可以将其配置为通过可访问的 URL 负载平衡流量或通过提供基于名称的虚拟主机从外部提供服务。因此，Ingress 是一个API对象，通常通过 HTTP 管理集群中服务的外部访问，是暴露服务的最有效方式。

现在，让我以一个例子向您解释 Ingress 网络的工作。

有 2 个节点具有带有 Linux 桥接器的 Pod 和根网络命名空间。除此之外，还有一个名为 flannel0（网络插件）的新虚拟以太网设备被添加到根网络中。

现在，假设我们希望数据包从 Pod1 流向 Pod4。

请参阅下图：



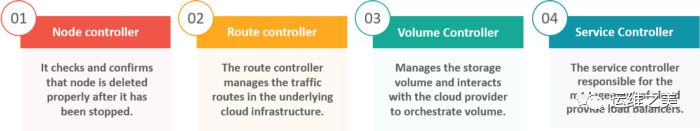
* 因此，数据包将 Pod1 的网络保留在 eth0，并进入 veth0 的根网络。
* 然后它被传递给 cbr0，这使得 ARP 请求找到目的地，并且发现该节点上没有人具有目的地 IP 地址。
* 因此，桥接器将数据包发送到 flannel0，因为节点的路由表配置了 flannel0。
* 现在，flannel 守护程序与 Kubernetes 的 API 服务器通信，以了解所有 Pod IP 及其各自的节点，以创建 Pods IP 到节点 IP 的映射。
* 网络插件将此数据包封装在 UDP 数据包中，其中额外的标头将源和目标 IP 更改为各自的节点，并通过 eth0 发送此数据包。
* 现在，由于路由表已经知道如何在节点之间路由流量，因此它将数据包发送到目标节点2。
* 数据包到达 node2 的 eth0 并返回到 flannel0 以解封装并在根网络命名空间中将其发回。
* 同样，数据包被转发到 Linux 网桥以发出 ARP 请求以找出属于 veth1 的 IP。
* 数据包最终穿过根网络并到达目标 Pod4。

**Q10、您对云控制器管理器有何了解？**

Cloud Controller Manager 负责持久存储、网络路由，从核心 Kubernetes 特定代码中抽象出特定于云的代码，以及管理与底层云服务的通信。

它可能会分成几个不同的容器，具体取决于您运行的是哪个云平台，然后它可以使云供应商和 Kubernetes 代码在没有任何相互依赖的情况下开发。因此，云供应商开发他们的代码并在运行 Kubernetes 时与 Kubernetes 云控制器管理器连接。

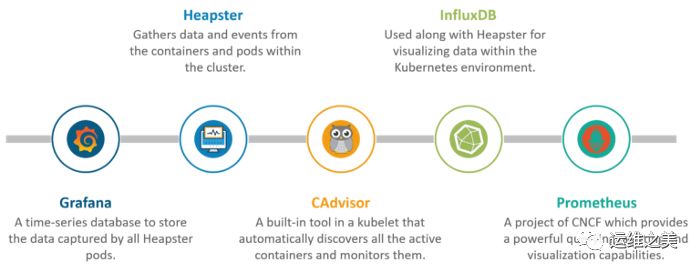
各种类型的云控制器管理器如下：



**Q11、什么是 Container 资源监控？**

对于用户而言，了解应用程序的性能和所有不同抽象层的资源利用率非常重要，Kubernetes 通过在容器、Pod、服务和整个集群等不同级别创建抽象来考虑集群的管理。现在，可以监视每个级别，这只是容器资源监视。

各种容器资源监控工具如下：



**Q12、Replica Set 和 Replication Controller 之间有什么区别？**

Replica Set 和 Replication Controller 几乎完全相同。它们都确保在任何给定时间运行指定数量的 Pod 副本。不同之处在于复制 Pod 使用的选择器。Replica Set 使用基于集合的选择器，而 Replication Controller 使用基于权限的选择器。

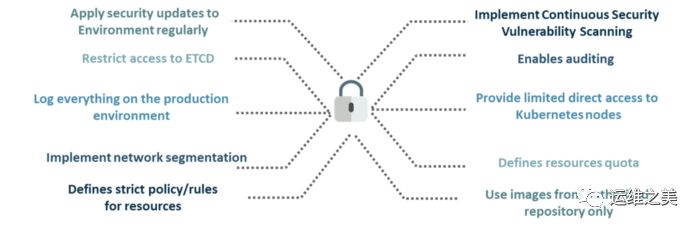
* Equity-Based 选择器：这种类型的选择器允许按标签键和值进行过滤。因此，在外行术语中，基于 Equity 的选择器将仅查找与标签具有完全相同短语的 Pod。示例：假设您的标签键表示 app = nginx，那么使用此选择器，您只能查找标签应用程序等于 nginx 的那些 Pod。
* Selector-Based 选择器：此类型的选择器允许根据一组值过滤键。因此，换句话说，基于 Selector 的选择器将查找已在集合中提及其标签的 Pod。示例：假设您的标签键在（nginx、NPS、Apache）中显示应用程序。然后，使用此选择器，如果您的应用程序等于任何 nginx、NPS或 Apache，则选择器将其视为真实结果。

**Q13、什么是 Headless Service？**

Headless Service 类似于 “普通” 服务，但没有群集 IP。此服务使您可以直接访问 Pod，而无需通过代理访问它。

**Q14、使用 Kubernetes 时可以采取哪些最佳安全措施？**

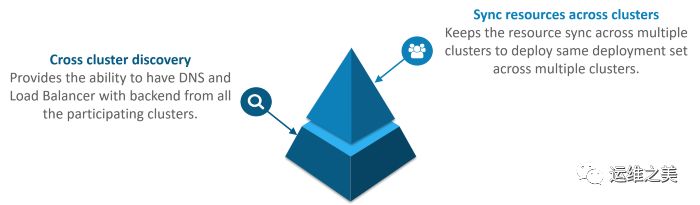
以下是使用 Kubernetes 时可以遵循的最佳安全措施：



**Q15、什么是集群联邦？**

在联邦集群的帮助下，可以将多个 Kubernetes 集群作为单个集群进行管理。因此，您可以在数据中心/云中创建多个 Kubernetes集群，并使用联邦来在一个位置控制/管理它们。

联邦集群可以通过执行以下两项操作来实现此目的。请参考下图。



**基于场景的面试问题**

这部分问题将包含您在面试中可能遇到的各种基于场景的问题。

**场景1**

假设一家基于单一架构的公司处理众多产品。现在，随着公司在当今的扩展行业的扩展，他们的单一架构开始引发问题。

您如何看待公司从单一服务转向微服务并部署其服务容器？

解：由于公司的目标是从单一应用程序转向微服务，它们最终可以逐个构建，并行构建，只需在后台切换配置。然后他们可以将这些内置微服务放在 Kubernetes 平台上。因此，他们可以从一次或两次迁移服务开始，并监控它们以确保一切运行稳定。一旦他们觉得一切顺利，他们就可以将其余的应用程序迁移到他们的 Kubernetes 集群中。

**场景2**

考虑一家拥有分布式系统的跨国公司，拥有大量数据中心，虚拟机和许多从事各种任务的员工。

您认为这样的公司如何以 Kubernetes 一致的方式管理所有任务？

解：正如我们所有人都知道 IT 部门推出了数千个容器，其任务在分布式系统中遍布全球众多节点。在这种情况下，公司可以使用能够为基于云的应用程序提供敏捷性，横向扩展功能和 DevOps 实践的东西。因此，该公司可以使用 Kubernetes 来定制他们的调度架构并支持多种容器格式。这使得容器任务之间的亲和性成为可能，从而提供更高的效率，并为各种容器网络解决方案和容器存储提供广泛支持。

**场景3**

考虑一种情况，即公司希望通过维持最低成本来提高其效率和技术运营速度。您认为公司将如何实现这一目标？

解：公司可以通过构建 CI/CD 管道来实现 DevOps 方法，但是这里可能出现的一个问题是配置可能需要一段时间才能启动并运行。因此，在实施 CI/CD 管道之后，公司的下一步应该是在云环境中工作。一旦他们开始处理云环境，他们就可以在集群上安排容器，并可以在 Kubernetes 的帮助下进行协调。这种方法将有助于公司缩短部署时间，并在各种环境中加快速度。

**场景4**

假设一家公司想要修改它的部署方法，并希望建立一个更具可扩展性和响应性的平台。您如何看待这家公司能够实现这一目标以满足客户需求？

解：为了给数百万客户提供他们期望的数字体验，公司需要一个可扩展且响应迅速的平台，以便他们能够快速地将数据发送到客户网站。现在，要做到这一点，公司应该从他们的私有数据中心（如果他们使用任何）转移到任何云环境，如 AWS。不仅如此，他们还应该实现微服务架构，以便他们可以开始使用 Docker 容器。一旦他们准备好基础框架，他们就可以开始使用最好的编排平台，即 Kubernetes。这将使团队能够自主地构建应用程序并快速交付它们。

**场景5**

考虑一家拥有非常分散的系统的跨国公司，期待解决整体代码库问题。您认为公司如何解决他们的问题？

解：那么，为了解决这个问题，我们可以将他们的单片代码库转移到微服务设计，然后每个微服务都可以被视为一个容器。因此，所有这些容器都可以在 Kubernetes 的帮助下进行部署和协调。

**场景6**

我们所有人都知道，从单片到微服务的转变解决了开发方面的问题，但却增加了部署方面的问题。公司如何解决部署方面的问题？

解：团队可以试验容器编排平台，例如：Kubernetes，并在数据中心运行。因此，通过这种方式，公司可以生成模板化应用程序，在五分钟内部署它，并在此时将实际实例集中在暂存环境中。这种 Kubernetes 项目将有数十个并行运行的微服务，以提高生产率，即使节点出现故障，也可以立即重新安排，而不会影响性能。

**场景7**

假设一家公司希望通过采用新技术来优化其工作负载的分配。公司如何有效地实现这种资源分配？

解：这个问题的解决方案就是 Kubernetes。Kubernetes 确保资源得到有效优化，并且只使用特定应用程序所需的那些资源。因此，通过使用最佳容器编排工具，公司可以有效地实现资源分配。

**场景8**

考虑一家拼车公司希望通过同时扩展其平台来增加服务器数量。您认为公司如何处理服务器及其安装？

解：公司可以采用集装箱化的概念。一旦他们将所有应用程序部署到容器中，他们就可以使用 Kubernetes 进行编排，并使用像 Prometheus 这样的容器监视工具来监视容器中的操作。因此，利用容器的这种使用，在数据中心中为它们提供更好的容量规划，因为它们现在将受到更少的限制，因为服务和它们运行的硬件之间存在抽象。

**场景9**

考虑一种情况，公司希望向具有各种环境的客户提供所有必需的分发。您认为他们如何以动态的方式实现这一关键目标？

解：该公司可以使用 Docker环境，组建一个横截面团队，使用 Kubernetes 构建 Web 应用程序。这种框架将帮助公司实现在最短的时间内将所需产品投入生产的目标。因此，在这样的机器运行的情况下，公司可以向所有具有各种环境的客户发放电子邮件。

**场景10**

假设公司希望在不同的云基础架构上运行各种工作负载，从裸机到公共云。公司将如何在不同界面的存在下实现这一目标？

解：该公司可以将其基础设施分解为微服务，然后采用 Kubernetes。这将使公司在不同的云基础架构上运行各种工作负载。

**多项选择面试问题**

这部分问题将包括多项选择面试问题，这些问题在面试中经常被问到。

**Q1、什么是 Kubernetes 集群中的 minions？**

1. 它们是主节点的组件。
2. 它们是集群的工作节点。[答案]
3. 他们正在监控 Kubernetes 中广泛使用的引擎。
4. 他们是 Docker 容器服务。

**Q2、Kubernetes 集群数据存储在以下哪个位置？**

1. KUBE-API服务器
2. Kubelet
3. ETCD [答案]
4. 以上都不是

**Q3、哪个是 Kubernetes 控制器？**

1. ReplicaSet
2. Deployment
3. Rolling Updates
4. ReplicaSet和Deployment [答案]

**Q4、以下哪个是核心 Kubernetes 对象？**

1. Pods
2. Services
3. Volumes
4. 以上所有[答案]

**Q5、Kubernetes Network 代理在哪个节点上运行？**

1. Master Node
2. Worker Node
3. 所有节点[答案]
4. 以上都不是

**Q6、节点控制器的职责是什么？**

1. 将 CIDR 块分配给节点
2. 维护节点列表
3. 监视节点的运行状况
4. 以上所有[答案]

**Q7、Replication Controller 的职责是什么？**

1. 使用单个命令更新或删除多个 Pod
2. 有助于达到理想状态
3. 如果现有 Pod 崩溃，则创建新 Pod
4. 以上所有[答案]

**Q8、如何在没有选择器的情况下定义服务？**

1. 指定外部名称[答案]
2. 指定具有 IP 地址和端口的端点
3. 只需指定 IP 地址即可
4. 指定标签和 API 版本

**Q9、1.8 版本的 Kubernetes 引入了什么？**

1. Taints and Tolerations [答案]
2. Cluster level Logging
3. Secrets
4. Federated Clusters

**Q10、Kubelet 调用的处理检查容器的 IP 地址是否打开的程序是？**

1. HTTPGetAction
2. ExecAction
3. TCPSocketAction [答案]
4. 以上都不是

**译者注**

这篇文章不仅仅适合相关的面试者，也非常推荐 Kubernetes 的初学者或者想要了解 Kubernetes 技术的产品或管理者阅读。但是这里面还存在几点不足，例如覆盖的内容较浅显，没有非常具体的技术点，缺少大规模的经验和技术点考察等，有机会后面的文章会补充下！

# 一文看懂RPA的技术架构及原理

众所周知，作为一款软件或平台，RPA（机器人流程自动化）是用来替代人类员工实施基于规则的高度重复性工作的程序，而非实体存在的流程处理机器。但由于它的新颖性，许多人可能会对RPA及其构成感到困惑。

典型的RPA平台至少会包含开发、运行、控制等三个组成部分。

**RPA三件套**

• 开发工具

开发工具主要用于建立软件机器人的配置或设计机器人。通过开发工具，开发者可以为机器人执行一系列的指令和决策逻辑进行编程。

就像雇佣新员工一样，新创建的机器人对公司的业务或流程将会一无所知。这就需要我们在业务流程上培训机器人，然后才能发挥出其特有的功能，提高工作效率。

大多数开发工具为了进行商业发展，通常需要开发人员具备相应的编程知识储备，如循环、变量赋值等。不过，好消息是，目前大多数RPA软件代码相对较低，使得一些没有IT背景但训练有素的用户也能快速学习和使用。

开发工具里还包括——

记录仪：也称之为“录屏”，用以配置软件机器人。就像Excel中的宏功能，记录仪可以记录用户界面（UI）里发生的每一次鼠标动作和键盘输入。

插件/扩展：为了让配置的运行软件机器人变得简单，大多数平台都提供许多插件和扩展应用。

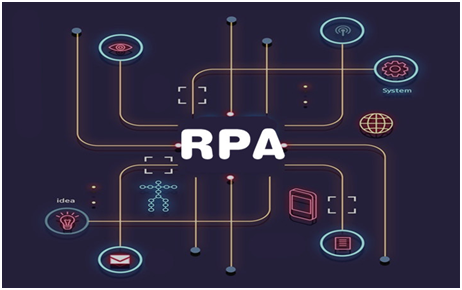
可视化流程图：一些RPA厂商为方便开发者更好地操作RPA开发平台，会推出流程图可视化操作。比如UiBot开发平台就包含三种视图，流程视图、可视化视图、源码视图，分别对应不同用户的需求。

• 运行工具

当开发工作完成后，用户可使用该工具，来运行已有软件机器人，也可以查阅运行结果。

• 控制中心

主要用于软件机器人的部署与管理。包括开始/停止机器人的运行，为机器人制作日程表，维护和发布代码，重新部署机器人的不同任务，管理许可证和凭证等。当需要在多台PC上运行软件机器人的时候，也可以用控制器对这些机器人进行集中控制，比如统一分发流程，统一设定启动条件等。



**RPA机器人的工作流程**

●流程开发及配置：开发人员制定详细的指令并将他们发布到机器上，具体包括应用配置、数据输入、验证客户端文件、创建测试数据、数据加载以及生成报告。

●业务用户能够通过控制中心给机器人分配任务并监视它们的活动，将流程操作实现为独立的自动化任务，交由软件机器人执行。

●机器人位于虚拟化或物理环境中，不需要与系统开放任何接口，仅需通过用户界面与各种各样的应用系统（包括ERP、SAP、CRM、OA等）交互，完全模拟人类操作，自动执行日常的劳动密集且重复的任务。

●业务用户审查并解决任何异常或进行升级。

**UiBot的技术框架**

UiBot是国内机器人流程自动化先驱奥森科技自主研发的RPA工具，能够针对企业和个人提供完整的流程自动化解决方案，帮助组织机构实现降本增效，推动企业数字化升级。

UiBot也由三个部分构成，分别为UiBot Creator，UiBot Worker，UiBot Commander，对应RPA平台的开发工具、运行工具和控制中心。



●UiBot Creator

即机器人的开发工具，负责开发RPA流程自动化机器人。拥有极其便捷的录屏功能，流程一键录制，即可自动生成机器人。300+的预置组件，满足所有日常需要，涵盖了大部分RPA应用场景。具备强大的扩展功能，提供Python、C/C++、Lua、.Net等多种编程语言的扩展接口。另外，为对标不同用户的需求，Creator的表现方式包含三种视图，分别为：

流程视图：针对咨询方，主要用于业务流程的梳理和确认，省略了具体流程细节的实现。

可视化视图：针对不熟悉IT的各领域专家以及各种普通用户，通过简单拖拽、参数配置操作，即可完成流程的连接活动。

源码视图：针对IT专家、编程大神或者是熟悉了本产品的领域专家，能够有效减少鼠标操作，更快捷的生成所需的流程。

其中可视化工作流与源代码两种开发方式可随时切换，无缝衔接，兼顾入门期的简单易用与进阶后的快速开发需要。

●UiBot Worker

即机器人的执行平台，可查看具体的业务机器人，具有完整的机器人添加和运行管理功能。具备人机Robot、无人Robot、双模式。

●UiBot Commander

即机器人的管理中心，对机器人工作站进行综合调度与权限控制。可实现信息统一管理，提供数据可视化图表展示，包括信息汇集、用户管理、机器人管理、系统管理、UiBot Worker管理。

如果只需要少量的PC运行流程，可以由Creator制作出流程后，直接打包分发给Worker使用，Commander不需要参与；如果需要大量的PC运行流程，比较合适的方式是Creator把流程先上传到Commander，再由Commander统一发给各个Worker，并统一指挥各个Worker执行流程。

此外，UiBot Creator社区版，是永久免费的。只需在线登录一下，或离线激活一下，就可以无限制使用。目前，已有2万多名用户都在使用这样一款易学好用的RPA平台。

为了能够让开发者更好地学习和使用RPA工具，UiBot还推出了RPA开发者指南和技术培训课程，而这些目前都是免费的。[返回搜狐，查看更多](https://www.sohu.com/?strategyid=00001&spm=smpc.content.content.2.1590415912037A5gp9rt)

声明：该文观点仅代表作者本人，搜狐号系信息发布平台，搜狐仅提供信息存储空间服务。

RPA 是什么技术

　　一如其名，[rpa](https://support.i-search.com.cn/)真的就是那么简单又强大。机器人流程自动化让您能够使用工具建置自己专属的软体机器人，以便将任何业务流程自动化。您的「机器人」是可设定的软体，用于执行您所指派和控制的任务。

　　您可以将机器人视为您的数位劳动力。告诉您的机器人该做什么，然后让它们执行工作。机器人能够跟您一样以相同的方式与任何系统或应用程式互动。机器人具备学习能力，也可以被复制。了解机器人如何运作，并且依照您的需求进行调整及扩展。不需要使用程式码、不会中断作业、非侵入式，而且使用简易。

　　以零错误率大幅降低营运成本。无须改变任何基础商业系统或应用程式。

　　依照需求扩展，并提升营运灵活性。无须改变任何现行业务流程。

　　企业使用者可在短短数周内完成 RPA 机器人的建置与部署，并在数天内看到正向的投资报酬率。

　　RPA 与传统企业自动化工具的关系

　　RPA 能搭配任何应用程式与系统 (无论新旧) 随插即用。

　　RPA 握有时间、灵活性与智慧等秘密武器。相较于其他自动化方法，机器人流程自动化可在短时间内完成任务，并于各种环境、应用领域与地点执行。RPA 机器人可立即启动、依照需求扩展，并以 100% 的产能持续运作。若与认知自动化功能结合运用，您的数位劳动力就能处理更复杂的非结构化资讯，将各种业务流程从端到端完整自动化。

　　而那些无法胜过 RPA 的业务流程自动化工具，都应尽早打入冷宫

　　成熟的 RPA，可在数周内建置完成并开始运作。直觉易用的 RPA 可让企业使用者自行建置机器人，并在数天内实现投资报酬率。

　　RPA 可让您的劳动力即时自行自动化。灵活且智慧的 RPA 能以 10 倍的速度运作，且不会犯错。

　　依照需求启动、停止与扩展由 RPA 机器人组成的数位劳动力。您可依照需求选择软体机器人数量，并且在必要时立即进行部署。

　　RPA 可带来哪些业务优势？

　　1、加快业务流程速度；机器人会以最快的速度执行，不会放慢速度或休息。

　　2、随选扩充生产力；RPA 具有无限弹性。您可以依照自身需求使用机器人，即时增减生产力。

　　3、提供最大程度的准确性 (100%)；RPA 可毫无错误地执行任务。每次皆可达到 100% 的准确性。

　　4、提供最大程度的可用性 (全天候、全年无休)；机器人可不眠不休地运作。RPA 可不间断地执行，使命必达。

　　5、提高业务流程合规性；RPA 的其中一项特长，就是遵守规则。

　　6、强化控制；只要设定好所需遵守的法规与标准，机器人就不会偏移指定的执行路径。

　　7、提高稽核能力；每个 RPA 机器人都会遭到监控与记录。

　　8、增强安全性；RPA 可透过职责分离、存取控制，以及铜墙铁壁般的加密与架构，实现前所未有的安全性。

　　9、提供商业智慧；RPA 分析可将机器人即时回报的详细资讯转换成效能与营运洞见。

　　10、最佳化人力投资；RPA 能处理重复的业务流程，让您的员工能专心进行具附加价值的工作。

　　11、激励员工士气；RPA 可减轻重复的人工流程所造成的负担，让您的员工有机会实现更宏大的目标。

　　12、加速数位转型；RPA 可扩展您的自动化版图并提供洞见，进而彻底翻转您的业务经营方式。

为了大家近距离了解RPA，推出信用卡账单管理机器人，独立场景，独立使用，欢迎免费下载使用：

RPA是Robotic Process Automation 的缩写，从字面便不难看出其要义，即：机器、流程、自动化，RPA 是以机器人作为虚拟劳动力，依据预先设定的程序与现有用户系统进行交互并完成预期的任务。从目前的技术实践来看，现有的 RPA 还仅适用于高重复性、逻辑确定并且稳定性要求相对较低的流程。

　　用更通俗的解释，RPA 就是借助一些能够自动执行的脚本（这些脚本可能是某些工具生成的，这些工具也可能有着非常有好的用户化图形界面）完成一系列原来需要人工完成的工作，但凡具备一定脚本生成、编辑、执行能力的工具在此处都可以称之为机器人。



　　比如，在游戏领域被广泛为人所熟知的国产软件“按键精灵”，即可以通过它的一些简单功能帮助我们完成一些自动化的工作。推荐阅读：RPA 有哪些优势

　　按键精灵的简要工作原理是通过录制操作者的鼠标和键盘的动作步骤形成操作脚本（用户也可以不用录制的方式，完全手工编写脚本），这里的脚本是可以修改的，用户可以更具需要修改脚本的参数，比如鼠标点击的位置、键盘输入的值，再次运行脚本的时候就会重新执行录制过程中的这些动作，如果脚本的参数有修改，则会执行对应的调整后的动作。

RPA技术原理

目前，大部分的RPA工具都是基于微软.NET Framework的WorkflowFoundation开发，整合OCR、NLP等AI技术，能够以弱耦合，即不改变现有软件系统部署的形式跨系统执行，且实现零出错率。在技术上，RPA早已不是单个企业的定制化功能，事实上已经生长成融合AI技术、系统级的跨软硬件、多账户的服务。

RPA产品形态

当下的市场上存在三种RPA产品形态：

1. 企业定制化的独立RPA软件，打开电脑点击即可使用；
2. 应用于财会、金融等行业场景的平台化RPA产品，客户端包括图形化设计器、执行环境（机器人），云端控制台则是基础的AI服务和结算系统；
3. 支持开发者二次开发的RPA平台，提供第二种所有功能的同时还提供对应的开发接口，以便定制化。

第一种产品形态已经在国内外市场存在了近20年，是一些劳务外包公司常用的基础工具，金融、财会、电信运营商、客服呼叫中心等行业领域一直在使用，属于定制化RPA服务。这种RPA更加精专、服务稳定、价格低廉、合作灵活，适合中小企业、大集团分公司以及外包公司使用。

后两种在垂直行业的通用性更好，产品灵活性高，可实时升级公司内部业务流程，适合大集团部署，产品技术要求高。但缺陷在于，如果产品技术不达标，便容易沦为SI的技术服务商。

RPA企业也在学习SaaS、PaaS模式，将部分功能部署在云端，以出租机器人劳力的方式按时收费——想方设法在交付模式上靠近订阅制。不过，RPA上云不如企服软件直接集成自动化有优势。

未来，RPA企业是否能找到一条比集成自动化更具优势的上云路径，或者直接在集成自动化上形成强势地位，从而引领企服软件发展，直接将集成自动化吞并到RPA的概念之中。

你好，RPA可以分为2种，一种是医疗方面的，另外的则是技术软件类的。医疗方面的楼下已经讲解了，我主要讲解下软件技术的RPA有哪些应用领域

　　1、数据提取：所有企业都需要记录其交易，以便将其用于未来的流程。机器人可以用来收集和合并事务，而不是像手工工作人员一样还得总是在脑海中考虑是否存储正确。

　　2、数据管理：为手动劳动力汇总大量数据将7a686964616fe58685e5aeb931333433626431是一项繁琐的工作。因此，机器人可以为员工完成这项工作，这样员工就可以利用他们的大脑来处理更重要的事情。

　　3、维护客户数据： 在企业中，许多客户可能每天都与业务打交道。现在，每家公司都需要维护每个客户的数据。如果一个人这样做，那么它可能比机器人花费更长的时间。但是，如果机器人执行此任务，则可以定期验证和更新数据。

　　4、执行财务结算和报告：通过 RPA 可以轻松处理将各种来源的数据发布到公司的子分类账户以创建和提交财务报表的繁琐过程。拓展阅读：RPA 在财务领域有哪些应用RPA 可以运用哪些方面5、发票处理：几乎所有公司都有需要处理的发票。为 100 位客户手动操作是一项耗时的工作。如果要做机器人，生产率要高得多。相关阅读：RPA 在税务领域的应用RPA 可以运用哪些方面6、处理门户查询和价格：企业在门户网站上工作，就必须密切关注价格的波动。而不是员工一次又一次地登录和退出，复制粘贴登录信息，并记录价格的波动。机器人可以接管此任务以连接到所有门户网站，收集有关波动价格的信息并报告 / 存储它们。

　　7、获取有关市场的报告： 企业可能需要每天获取关于某些参数的市场情况报告，以便及时掌握竞争对手的情况。机器人可以投入这项工作，按时交付完整的市场报告。

　　8、供应商定价比较： 在准备客户报价时，您需要比较供应商的价格， 这是一个耗时的过程。相反，如果我们让软件为我们收集数据并以我们想要的格式提供，可以节省我们的时间，精力和金钱。

　　9、异常订单处理： 经常观察到销售订单将价格持续几天。在这种情况下，机器人可以对该特定产品进行价格检查，并用所持有的价格验证当前价格。

　　10、交货对帐：当我们在线订购时， 订单需要针对货件进行验证。我们可以让机器人为我们做。因此，机器人将检查并批准所有匹配的订单，并在出现异常时通知我们。

介绍项目过程中，测试完之后是由运维发布，测试让运维发版

有什么能力竞争这个岗位

如何学习一门新技术，学习方式,举一个类子

接触自动化测试

逻辑能力和算法能力

如果一个评估项目，需要一个月的时间，用什么心态和方法来圆满完成。

讲一讲项目组之间如果出现了进度紧张，应该怎么处理

做web项目不懂的，如何让一个不认识的同事来帮你，如何请教他

如何和前端有冲突，比如一个方法的争议，如何处理。

自己的缺点和优点

简单一句话介绍自己的工作上和生活。讲明自己是一个什么样的人

讲一下自己的俩年到五年的计划

应该做哪些努力来提升自己