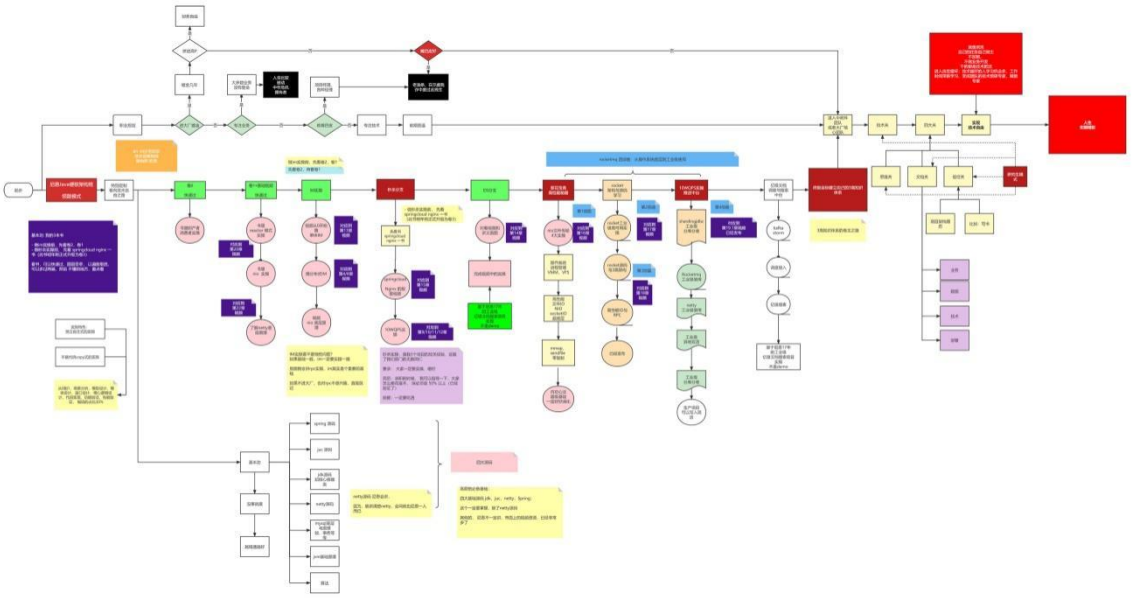
# 牛逼的职业发展之路

40 岁老架构尼恩用一张图揭秘：Java 工程师的高端职业发展路径，走向食物链顶端的之路

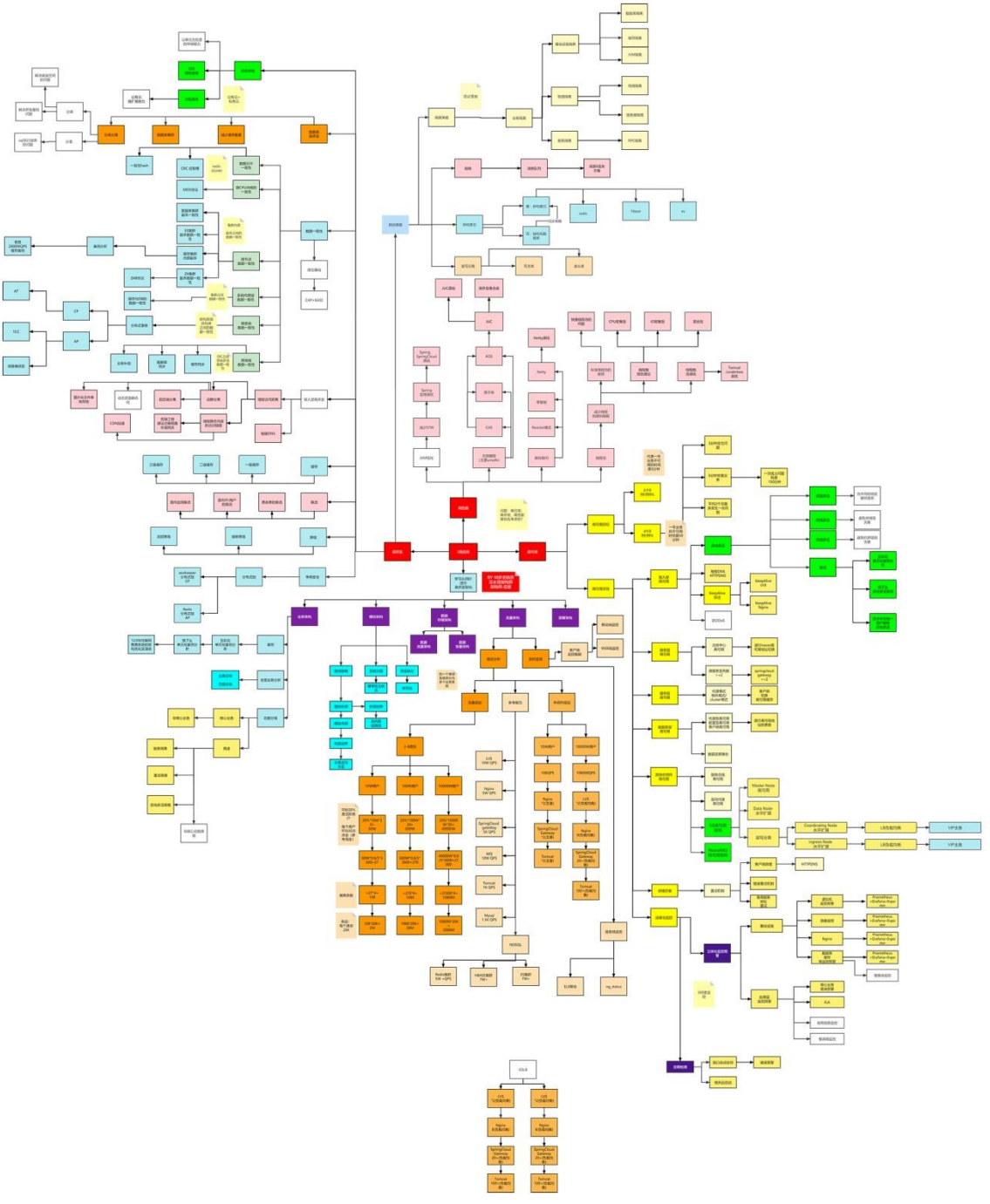
链接：https://www.processon.com/view/link/618a2b62e0b34d73f7eb3cd7



# 史上最全：价值10W的架构师知识图谱

此图梳理于尼恩的多个 3 高生产项目：多个亿级人民币的大型 SAAS 平台和智慧城市项目

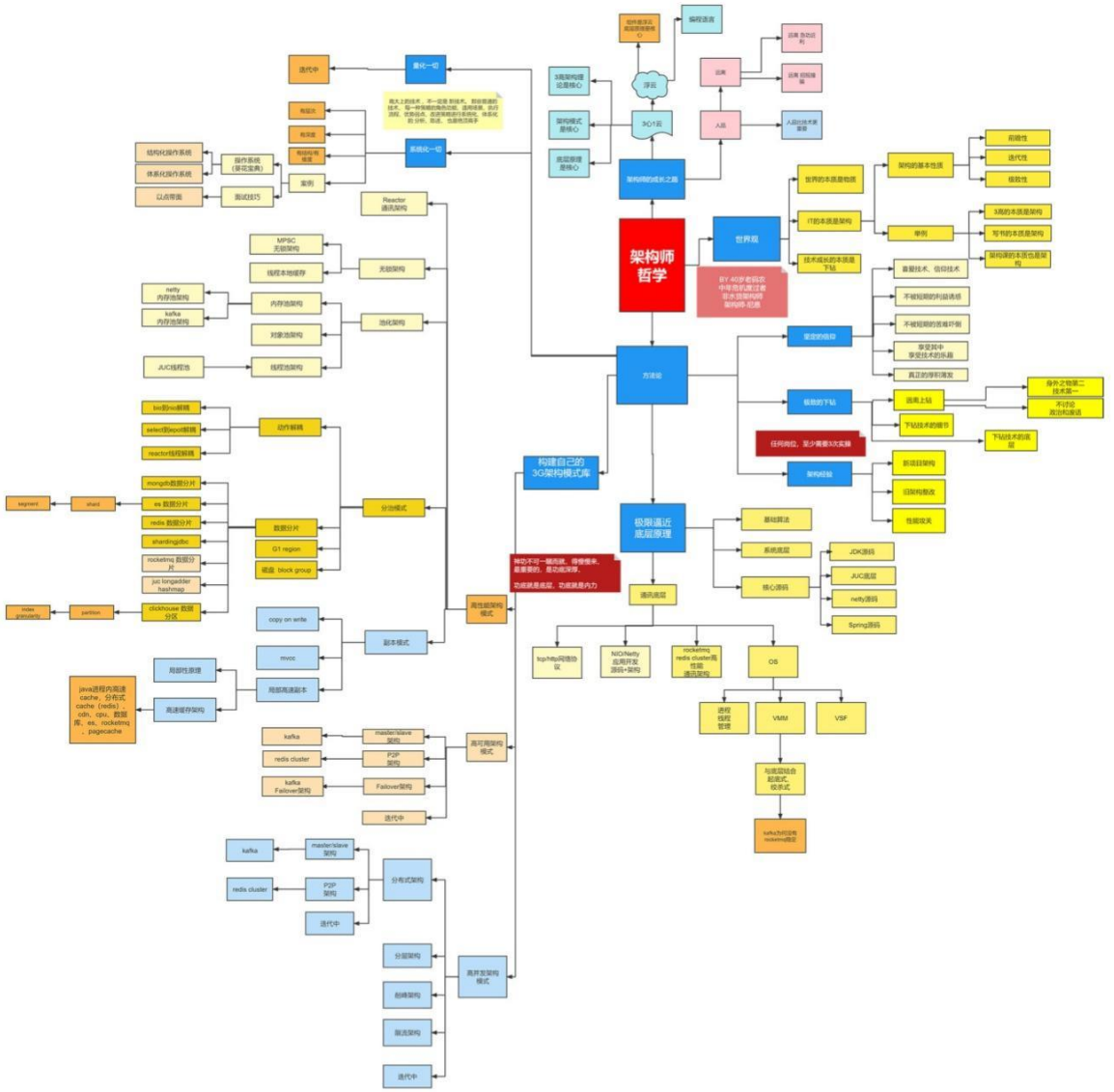
链接：https://www.processon.com/view/link/60fb94216376897l9d246739



# 牛逼的架构师哲学

40 岁老架构师尼恩对自己的 20 年的开发、架构经验总结

链接：https://www.processon.com/view/link/616f801963768961e9d9aec8



# 牛逼的3高架构知识宇宙

尼恩 3 高架构知识宇宙，帮助大家穿透 3 高架构，走向技术自由，远离中年危机

链接：https://www.processon.com/view/link/635097d2e0b34d40be778ab4

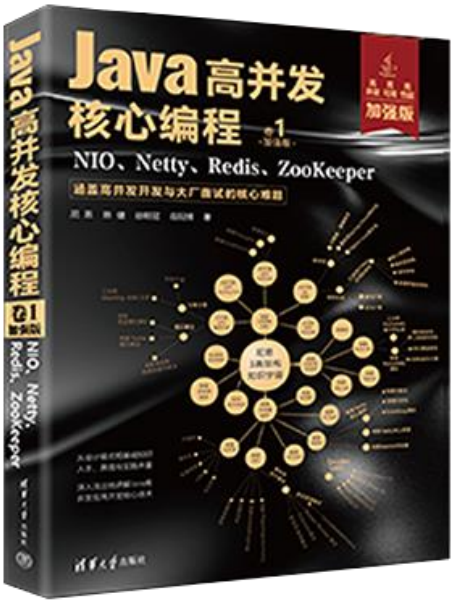
# 尼恩Java高并发三部曲（卷1加强版）

老版本：《Java 高并发核心编程 卷 1：NIO、Netty、Redis、ZooKeeper》（已经过时，不建议购买）

新版本：《Java 高并发核心编程 卷 1 **加强版**：NIO、Netty、Redis、ZooKeeper》

- 由浅入深地剖析了高并发 IO 的底层原理。
- 图文并茂的介绍了 TCP、HTTP、WebSocket 协议的核心原理。
- 细致深入地揭秘了 Reactor 高性能模式。
- 全面介绍了 Netty 框架，并完成单体 IM、分布式 IM 的实战设计。
- 详尽地介绍了 ZooKeeper、Redis 的使用，以帮助提升高并发、可扩展能力

详情：https://www.cnblogs.com/crazymakercircle/p/16868827.html



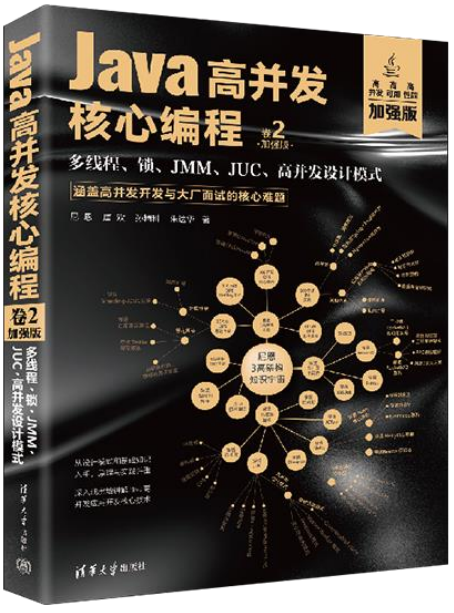
# 尼恩Java高并发三部曲（卷2加强版）

老版本：《Java 高并发核心编程 卷 2：多线程、锁、JMM、JUC、高并发设计模式》（已经过时，不建议购买）

新版本：《Java 高并发核心编程 卷 2 **加强版**：多线程、锁、JMM、JUC、高并发设计模式》

- 由浅入深地剖析了 Java 多线程、线程池的底层原理。
- 总结了 IO 密集型、CPU 密集型线程池的线程数预估算法。
- 图文并茂的介绍了 Java 内置锁、JUC 显式锁的核心原理。
- 细致深入地揭秘了 JMM 内存模型。
- 全面介绍了 JUC 框架的设计模式与核心原理，并完成其高核心组件的实战介绍。
- 详尽地介绍了高并发设计模式的使用，以帮助提升高并发、可扩展能力

详情参阅：https://www.cnblogs.com/crazymakercircle/p/16868827.html



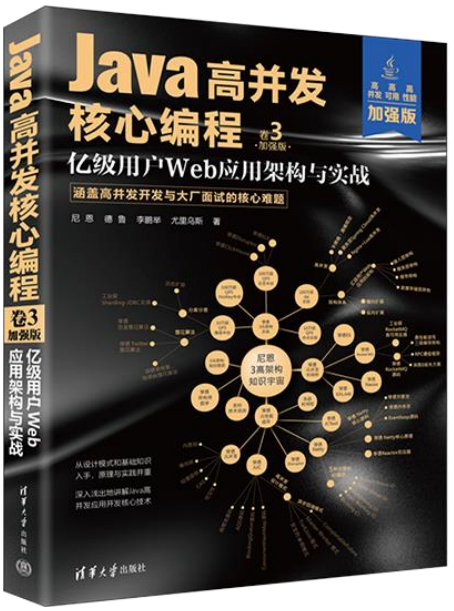
# 尼恩Java高并发三部曲（卷3加强版）

老版本：《SpringCloud Nginx 高并发核心编程》（已经过时，不建议购买）

新版本：《Java 高并发核心编程 卷 3 **加强版**：亿级用户 Web 应用架构与实战》

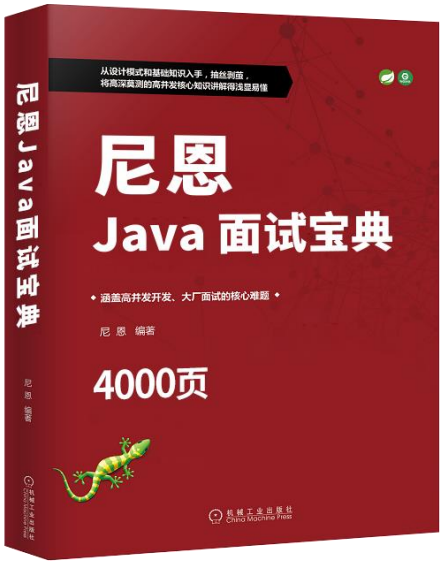
- 在当今的面试场景中，3 高知识是大家面试必备的核心知识，本书基于亿级用户 3 高 Web 应用的架构分析理论，为大家对 3 高架构系统做一个系统化和清晰化的介绍。
- 从 Java 静态代理、动态代理模式入手，抽丝剥茧地解读了 Spring Cloud 全家桶中 RPC 核心原理和执行过程，这是高级 Java 工程师面试必备的基础知识。
- 从Reactor 反应器模式入手，抽丝剥茧地解读了Nginx核心思想和各配置项的底层知识和原理，这是高级 Java 工程师、架构师面试必备的基础知识。
- 从观察者模式入手，抽丝剥茧地解读了 RxJava、Hystrix 的核心思想和使用方法，这也是高级 Java 工程师、架构师面试必备的基础知识。

详情：https://www.cnblogs.com/crazymakercircle/p/16868827.html



# 尼恩Java面试宝典

**40 个专题**（卷王专供+ 史上最全 + 2023 面试必备）
详情：https://www.cnblogs.com/crazymakercircle/p/13917138.html




专题01：JVM面试题（卷王专供 + 史上最全 + 2023面试必备）-V20-from-尼恩Java面试宝典.pdf
专题02：Java算法面试题（卷王专供 + 史上最全 + 2023面试必备）-V2 -from-Java面试红宝书-release.pdf
专题03：Java基础面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书 -release.pdf
专题04：架构设计面试题（卷王专供+ 史上最全 + 2023面试必备）-V24-from-Java面试红宝书-release.pdf
专题05：Spring面试题_专题06：SpringMVC_专题07：Tomcat面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩面试宝典-release.pdf
专题08：SpringBoot面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题09：网络协议面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题10：TCPIP协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题11：JUC并发包与容器类（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题12：设计模式面试题 （卷王专供+ 史上最全 + 2023面试必备）-V14-from-Java面试红宝书.pdf
专题13：死锁面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题14：Redis 面试题 （卷王专供+ 史上最全 + 2023面试必备）-V22-from-Java面试红宝书-release.pdf
专题15：分布式锁 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题16：Zookeeper 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题17：分布式事务面试题（卷王专供 + 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题18：一致性协议 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题19：Zab协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题20：Paxos 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题21：raft 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题22：Linux面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题23：Mysql 面试题（卷王专供+ 史上最全 + 2023面试必备）-V29-from-尼恩Java面试宝典-release.pdf
专题24：SpringCloud 面试题（卷王专供+ 史上最全 + 2023面试必备）-V12-from-Java面试红宝书-release.pdf
专题25：Netty 面试题 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题26：消息队列面试题：RabbitMQ、Kafka、RocketMQ（卷王专供+ 史上最全 + 2023面试必备）-V10-from-Java面试红宝书-release.pdf
专题27：内存泄漏 内存溢出（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题28：JVM 内存溢出 实战（卷王专供+ 史上最全 + 2023面试必备）-V17-from-Java面试红宝书-release.pdf
专题29：多线程面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题30：HR面试题：过五关斩六将后，小心阴沟翻船！（史上最全、避坑宝典）-V2-from-Java面试红宝书-release.pdf
专题31：Hash连环炮面试题（卷王专供+ 史上最全 + 2023面试必备）-V16-from-Java面试红宝书-release.pdf
专题32：大厂面试的基本流程和面试准备（阿里、腾讯、网易、京东、头条.....）-V2-from-Java面试红宝书-release.pdf
专题33：BST、AVL、RBT红黑树、三大核心数据结构（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题34：Elasticsearch面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-Java面试红宝书-release.pdf
专题35：Mybatis面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩Java面试宝典-release.pdf
专题36：Dubbo面试题（卷王专供+ 史上最全 + 2023面试必备）-V21-from-尼恩Java面试宝典-release.pdf
专题37：Docker面试题（卷王专供+ 史上最全 + 2023面试必备）-V26-from-尼恩Java面试宝典-release.pdf
专题38：K8S面试题（卷王专供+ 史上最全 + 2023面试必备）-V26-from-尼恩Java面试宝典-release.pdf
专题39：Nginx面试题（卷王专供+ 史上最全 + 2023面试必备）-V27-from-尼恩Java面试宝典-release.pdf
专题40：操作系统面试题（卷王专供+ 史上最全 + 2023面试必备）-V28-from-尼恩Java面试宝典-release.pdf

# 专题37：Docker面试题（史上最全、定期更新）

## 本文版本说明：V47

> 此文的格式，由markdown 通过程序转成而来，由于很多表格，没有来的及调整，出现一个格式问题，尼恩在此给大家道歉啦。
>
> 由于社群很多小伙伴，在面试，不断的交流最新的面试难题，所以，《尼恩Java面试宝典》， 后面会不断升级，迭代。
>
> 本专题，作为 《尼恩Java面试宝典》专题之一， 《尼恩Java面试宝典》一共**40个面试专题，后续还会增加**

### V47(2023-02-15)升级说明

极兔一面：Dockerfile如何优化？注意：千万不要只说减少层数

### 《尼恩面试宝典》升级的规划为：

后续基本上，**每一个月，都会发布一次**，最新版本，可以扫描扫架构师尼恩微信， 发送 “领取电子书” 获取。

尼恩的微信二维码在哪里呢？ 请参见文末

### 面试问题交流说明：

如果遇到面试难题，或者职业发展问题， 或者中年危机问题，都可以来 疯狂创客圈社群交流，

**入交流群**，加尼恩微信即可， 发送**“入群”**，

尼恩微信请打开语雀扫码 https://www.yuque.com/crazymakercircle/gkkw8s/khigna


# Docker 简介

Docker 是一个开源的应用容器引擎，基于 Go 语言并遵从 Apache2.0 协议开源。

Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）,更重要的是容器性能开销极低。

Docker 从 17.03 版本之后分为 CE（Community Edition: 社区版） 和 EE（Enterprise Edition: 企业版），我们用社区版就可以了

# Docker的应用场景

- Web 应用的自动化打包和发布。
- 自动化测试和持续集成、发布。
- 在服务型环境中部署和调整数据库或其他的后台应用。
- 从头编译或者扩展现有的 OpenShift 或 Cloud Foundry 平台来搭建自己的 PaaS 环境。

# Docker 架构

Docker 包括三个基本概念:

- **镜像（Image）**：Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:16.04 就包含了完整的一套 Ubuntu16.04 最小系统的 root 文件系统。
- **容器（Container）**：镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。
- **仓库（Repository）**：仓库可看成一个代码控制中心，用来保存镜像。

Docker 使用客户端-服务器 (C/S) 架构模式，使用远程API来管理和创建Docker容器。

Docker 容器通过 Docker 镜像来创建。

| 概念 | 说明 |
| --- | --- |
| Docker 镜像(Images) | Docker 镜像是用于创建 Docker 容器的模板，比如 Ubuntu 系统。 |
| Docker 容器(Container) | 容器是独立运行的一个或一组应用，是镜像运行时的实体。 |
| Docker 客户端(Client) | Docker 客户端通过命令行或者其他工具使用 Docker SDK (https://docs.docker.com/develop/sdk/) 与 Docker 的守护进程通信。 |
| Docker 主机(Host) | 一个物理或者虚拟的机器用于执行 Docker 守护进程和容器。 |
| Docker Registry | Docker 仓库用来保存镜像，可以理解为代码控制中的代码仓库。Docker Hub(https://hub.docker.com) 提供了庞大的镜像集合供使用。一个 Docker Registry 中可以包含多个仓库（Repository）；每个仓库可以包含多个标签（Tag）；每个标签对应一个镜像。通常，一个仓库会包含同一个软件不同版本的镜像，而标签就常用于对应该软件的各个版本。我们可以通过 **<仓库名>:<标签>** 的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 **latest** 作为默认标签。 |

# docker的基础命令

### docker的守护进程查看

systemctl status docker

### docker 镜像查看

docker image  ls

### docker 容器查看

docker  ps

### Docker Registry配置和查看

cat   /etc/docker/daemon.json

```
配置私有仓库

cat>/etc/docker/daemon.json<<EOF

{

  "registry-mirrors":
["http://10.24.2.30:5000","https://tnxkcso1.mirrors.aliyuncs.com"],

  "insecure-registries":["10.24.2.30:5000"]

}

EOF

```

## 在线安装docker

# 离线安装docker

一、基础环境
1、操作系统：CentOS 7.3
2、Docker版本：[19.03.9 官方下载地址]
3、官方参考文档：[https://docs.docker.com/install/linux/docker-ce/binaries/#install-static-binaries]

**二、Docker安装**

1、下载

wget https://download.docker.com/linux/static/stable/x86_64/docker-19.03.9.tgz

注意：如果事先下载好了可以忽略这一步

2、解压

把压缩文件存在指定目录下(如root)，并进行解压

tar -zxvf docker-19.03.9.tgz

```
cd root
[root@localhost ~]# tar -zxvf docker-19.03.6.tgz
docker/
docker/containerd
docker/docker
docker/ctr
docker/dockerd
docker/runc
docker/docker-proxy
docker/docker-init
docker/containerd-shim

```

3、将解压出来的docker文件内容移动到 /usr/bin/ 目录下

cp docker/* /usr/bin/

4、将docker注册为service

> cat /etc/systemd/system/docker.service

> vi /etc/systemd/system/docker.service

```
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network-online.target firewalld.service
Wants=network-online.target

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues
still
# exists and systemd currently does not support the cgroup feature set
required
```

```
# for containers run by docker
ExecStart=/usr/bin/dockerd
ExecReload=/bin/kill -s HUP $MAINPID

# Having non-zero Limit*s causes performance problems due to accounting overhead

# in the kernel. We recommend using cgroups to do container-local accounting.

LimitNOFILE=infinity
LimitNPROC=infinity
LimitCORE=infinity

# Uncomment TasksMax if your systemd version supports it.
# Only systemd 226 and above support this version.
#TasksMax=infinity
TimeoutStartSec=0

# set delegate yes so that systemd does not reset the cgroups of docker containers

Delegate=yes

# kill only the docker process, not all processes in the cgroup

KillMode=process

# restart the docker process if it exits prematurely

Restart=on-failure
StartLimitBurst=3
StartLimitInterval=60s


[Install]
WantedBy=multi-user.target
```

5、启动

chmod +x /etc/systemd/system/docker.service  #添加文件权限并启动docker

systemctl daemon-reload  #重载unit配置文件

systemctl start docker  #启动Docker

systemctl enable docker.service  #设置开机自启

```
[root@localhost ~]# vi /etc/systemd/system/docker.service
[root@localhost ~]# chmod +x /etc/systemd/system/docker.service
[root@localhost ~]# systemctl daemon-reload
[root@localhost ~]# systemctl start docker
[root@localhost ~]# systemctl enable docker.service
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service to /etc/systemd/system/docker.service.

```

6、验证

systemctl status docker #查看Docker状态

docker -v #查看Docker版本

docker info

```
[root@localhost ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/etc/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since Sat 2021-10-09 15:25:44 CST; 29s ago
     Docs: https://docs.docker.com
 Main PID: 1916 (dockerd)
   CGroup: /system.slice/docker.service
           ├─1916 /usr/bin/dockerd
           └─1927 containerd --config /var/run/docker/containerd/containerd.toml --log-level info

Oct 09 15:25:43 localhost.localdomain dockerd[1916]: time="2021-10-09T15:25:43.671407996+08:00" level=info msg="scheme \"unix\" not r...e=grpc
Oct 09 15:25:43 localhost.localdomain dockerd[1916]: time="2021-10-09T15:25:43.671440368+08:00" level=info msg="ccResolverWrapper: se...e=grpc
Oct 09 15:25:43 localhost.localdomain dockerd[1916]: time="2021-10-09T15:25:43.671462935+08:00" level=info msg="ClientConn switching ...e=grpc
Oct 09 15:25:43 localhost.localdomain dockerd[1916]: time="2021-10-09T15:25:43.750687781+08:00" level=info msg="Loading containers: start."
Oct 09 15:25:44 localhost.localdomain dockerd[1916]: time="2021-10-09T15:25:44.072960862+08:00" level=info msg="Default bridge (docke...dress"
Oct 09 15:25:44 localhost.localdomain dockerd[1916]: time="2021-10-09T15:25:44.153444071+08:00" level=info msg="Loading containers: done."
Oct 09 15:25:44 localhost.localdomain dockerd[1916]: time="2021-10-09T15:25:44.175249299+08:00" level=info msg="Docker daemon" commit...9.03.6
Oct 09 15:25:44 localhost.localdomain dockerd[1916]: time="2021-10-09T15:25:44.175337834+08:00" level=info msg="Daemon has completed ...ation"
Oct 09 15:25:44 localhost.localdomain systemd[1]: Started Docker Application Container Engine.
Oct 09 15:25:44 localhost.localdomain dockerd[1916]: time="2021-10-09T15:25:44.195084106+08:00" level=info msg="API listen on /var/ru....sock"
Hint: Some lines were ellipsized, use -l to show in full.
[root@localhost ~]# docker -v
Docker version 19.03.6, build 369ce74a3c
[root@localhost ~]# docker info

```

# 调整镜像仓库

修改docker的registry
修改/etc/docker目录下的daemon.json文件

在文件中加入

```
{
  "registry-mirrors": ["https://registry.docker-cn.com"]
}
```

保存退出

重新启动docker

```
chmod +x /etc/systemd/system/docker.service  #添加文件权限并启动docker



systemctl daemon-reload               #重载unit配置文件

systemctl start docker     #启动Docker

systemctl restart docker     #重新启动Docker



systemctl enable docker.service       #设置开机自启
```

```
[root@localhost ~]# vi /etc/systemd/system/docker.service
[root@localhost ~]# chmod +x /etc/systemd/system/docker.service
[root@localhost ~]# systemctl daemon-reload
[root@localhost ~]# systemctl start docker
[root@localhost ~]# systemctl enable docker.service
Created symlink from /etc/systemd/system/multi-
user.target.wants/docker.service to /etc/systemd/system/docker.service.

```

发现内网的环境， 改成了阿里云的，但是没有啥用

```
[root@localhost ~]# cat /etc/docker/daemon.json
{
  "registry-mirrors": ["https://ku39pxyp.mirror.aliyuncs.com"],
  "insecure-registries": ["hub.company.com"]
}


```

确保镜像仓库可以ping通

```
[root@localhost ~]# curl https://ku39pxyp.mirror.aliyuncs.com
curl: (6) Could not resolve host: ku39pxyp.mirror.aliyuncs.com; Unknown error

[root@localhost ~]# ping ku39pxyp.mirror.aliyuncs.com
ping: ku39pxyp.mirror.aliyuncs.com: Name or service not known
```

## 查看docker info 的引擎信息

```
[root@localhost ~]# docker info
Client:
 Debug Mode: false

Server:
 Containers: 14
  Running: 7
  Paused: 0
  Stopped: 7
 Images: 19
 Server Version: 19.03.6
 Storage Driver: overlay2
  Backing Filesystem: xfs
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries
splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: b34a5c8af56e510852c35414db4c1f4fa6172339
 runc version: 3e425f80a8c931f88e6d94a8c831b9d5aa481657
 init version: fec3683
 Security Options:
  seccomp
   Profile: default
 Kernel Version: 3.10.0-1062.el7.x86_64
 Operating System: CentOS Linux 7 (Core)
 OSType: linux
 Architecture: x86_64
 CPUs: 4
 Total Memory: 15.49GiB
 Name: localhost.localdomain
 ID: I5KF:Y5JA:VCWG:DJYG:PGZO:PZVA:FYXQ:F624:RWH6:4S6R:BI6Z:L2MT
 Docker Root Dir: /var/lib/docker
```

```
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  hub.gsafety.com
  127.0.0.0/8
 Registry Mirrors:
  https://ku39pxyp.mirror.aliyuncs.com/
 Live Restore Enabled: false
```

## 查看docker相关的进程

```
[root@localhost ~]# ps -ef | grep docker
root       1460      1  0 Jun23 ?        04:45:43 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
root       2249   1460  0 Jun23 ?        00:00:09 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 5433 -container-ip 172.26.0.2 -container-port 5432
root       2280   1460  0 Jun23 ?        00:00:09 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 5432 -container-ip 172.26.0.3 -container-port 5432
root       2310   1455  0 Jun23 ?        00:09:35 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/78dc6aacc7d9490fa7c7252dd6b4df01af3b68c2adb69767fb0d51974ea0728c -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root       2311   1455  0 Jun23 ?        00:16:19 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/d6ec26035ca0428d5c3bd1cc154a76b356cf3a7d0746b0455d81223c7b9ab7fd -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root       2483   1460  0 Jun23 ?        00:00:32 /usr/bin/docker-proxy -proto tcp -host-ip 127.0.0.1 -host-port 1514 -container-ip 172.21.0.4 -container-port 10514
root       2538   1455  0 Jun23 ?        02:25:41 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/98500167fb283c56fd43f42d3357c52b393481fdcca2bc7a87128ac35e19fa5a -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root       2571   1455  0 Jun23 ?        02:17:17 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/412652067b159ca617625c31594oce6865534e80fa94b93ef3174f653d21b826 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
```

```
root       7077   1460  0 Jun23 ?        00:00:09 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 3306 -container-ip 172.19.0.2 -container-port 3306
root       7085   1455  0 Jun23 ?        00:09:39 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/976bb8cd43729535a74d1583a758be937b6cf8f7a3329a1737fcb722576d1fea -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root       7354   1460  0 Jun23 ?        00:00:09 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 3308 -container-ip 172.19.0.3 -container-port 3306
root       7386   1455  0 Jun23 ?        00:09:45 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/14112371b62521a9b52968a6b0d275700343afeceaac478cfb7a90241dfcdf61 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root       7402   1460  0 Jun23 ?        00:00:09 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 3307 -container-ip 172.19.0.4 -container-port 3306
root       7431   1455  0 Jun23 ?        00:10:30 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.linux/moby/a0c6a5d5f891d293ae19e0bc3413729ac41cf38cc7e58d5a547b0f0df87fd6c4 -address /run/containerd/containerd.sock -containerd-binary /usr/bin/containerd -runtime-root /var/run/docker/runtime-runc
root      28336  21582  0 15:32 pts/0    00:00:00 grep --color=auto docker
```

# Harbor概述(开源的镜像仓库)

```
Habor是由VMWare公司开源的容器镜像仓库。
事实上，Habor是在Docker Registry上进行了相应的
企业级扩展，从而获得了更加广泛的应用，这些新的企业级特性包括：管理用户界面，基于角色的访
问控制 ，AD/LDAP集成以及审计日志等，足以满足基本企业需求。
官方地址：https://vmware.github.io/harbor/cn/

```

## 1、什么是Harbor

• Harbor是VMware公司开源的企业级Docker Registry项目，其目标是帮助用户迅速搭建一个企业级的Docker Registry服务

• Harbor以 Docker 公司开源的Registry 为基础，提供了图形管理UI、基于角色的访问控制(Role Based AccessControl)、AD/LDAP集成、以及审计日志(Auditlogging)等企业用户需求的功能，同时还原生支持中文

• Harbor的每个组件都是以Docker 容器的形式构建的，使用docker-compose 来对它进行部署。用于部署Harbor 的docker- compose模板位于harbor/ docker- compose.yml

## 2、Harbor的特性

1.基于角色控制: 用户和仓库都是基于项目进行组织的，而用户在项目中可以拥有不同的权限

2.基于镜像的复制策略: 镜像可以在多个Harbor实例之间进行复制(同步)

3.支持LDAP/AD: Harbor 可以集成企业内部有的AD/LDAP (类似数据库的一-张表)，用于对已经存在的用户认证和管理

4.镜像删除和垃圾回收: 镜像可以被删除，也可以回收镜像占用的空间

5.图形化用户界面: 用户可以通过浏览器来浏览，搜索镜像仓库以及对项目进行管理

6.审计管理: 所有针对镜像仓库的操作都可以被记录追溯，用于审计管理

7.支持RESTful API: RESTful API提供给管理员对于Harbor 更多的操控，使得与其它管理软件集成变得更容易

8.Harbor 和docker registry的 关系: Harbor实质 上是对docker registry做 了封装，扩展了自己的业务模板

## 3、Harbor的构成

Harbor在架构上主要有Proxy、 Registry、 Core services、 Database (Harbor-db) 、Log collector ( Harbor-log)、Job services六个组件

● Proxy: Harbor 的Registry、 UI、Token 服务等组件，都处在nginx 反向代理后边。该代理将来自浏览器、docker clients的请求转发到后端不同的服务上

● Registry:负责储存Docker 镜像，并处理Docker push/pull命令。由于要对用户进行访问控制，即不同用户对Docker 镜像有不同的读写权限，Registry 会指向一个Token 服务，强制用户的每次Docker pull/push 请求都要携带一个合法的Token,Registry会通过公钥对Token进行解密验证

● Core services: Harbor的核心功能，主要提供以下3个服务:
1.UI (harbor-ui) :提供图形化界面，帮助用户管理Registry. 上的镜像( image)，并对用户进行授权
2.WebHook: 为了及时获取Registry.上image 状态变化的情况，在Registry. 上配置 Webhook，把状态变化传递给UI模块
3.Token 服务:负责根据用户权限给每个Docker push/pull 命令签发Token。 Docker 客户端向Registry 服务发起的请求，
如果不包含Token，会被重定向到Token服务，获得Token后再重新向Registry 进行请求

● Database (harbor-db) :为core services提供数据库服务，负责储存用户权限、审计日志、Docker 镜像分组信息等数据

● Job services: 主要用于镜像复制，本地镜像可以被同步到远程Harbor 实例上

● Log collector (harbor-log) :负责收集其他组件的日志到一个地方

• Harbor的每个组件都是以Docker 容器的形式构建的，因此，使用Docker Compose 来对它进行部署。

• 总共分为7个容器运行，通过在docker-compose.yml所在目录中执行docker-compose ps命令来查看，
名称分别为: nginx、 harbor-jobservice、 harbor-ui、 harbor-db、harbor-adminserver、registry、harbor-log.
其中harbor-adminserver主要是作为一个后端的配置数据管理，并没有太多的其他功能。harbor-ui所要操作的所有数据都通过harbor-adminserver这样一个数据配置管理中心来完成。

| 组件 | 功能 |
| --- | --- |
| harbor-adminserver | 配置管理中心 |
| harbor-db | Mysql数据库 |
| harbor-jobservice | 负责镜像复制 |
| harbor-log | 记录操作日志 |
| harbor-ui | Web管理页面和API |
| nginx | 前端代理，负责前端页面和镜像上传/下载转发 |
| redis | 会话 |
| registry | 镜像存储 |

# Docker本地镜像载入与载出

## 两种办法

- 保存镜像（保存镜像载入后获得跟原镜像id相同的镜像）
- 保存容器（保存容器载入后获得跟原镜像id不同的镜像）

## 拉取镜像

通过命令可以从镜像仓库中拉取镜像，默认从Docker Hub 获取。

命令格式：

docker image pull :

docker image pull  rancher/rke-tools:v0.1.52

[rancher/rke-tools:v0.1.52

## 保存镜像

- docker save 镜像id -o /home/mysql.tar
- docker save 镜像id > /home/mysql.tar

docker save  docker.io/rancher/rancher-agent  -o /home/rancher-agent .tar

docker save  f29ece87a195  -o /home/rancher-agent.tar

docker save  docker.io/rancher/rke-tools  -o /home/rke-tools-v0.1.52.tar

## 载入镜像

- docker load -i mysql.tar

docker load -i /usr/local/rancher-v2.3.5.tar

docker load -i /usr/local/rancher-agent.tar

docker inspect f29ece87a1954772accb8a2332ee8c3fe460697e3f102ffbdc76eb9bc4f4f1d0

docker load -i /usr/local/rke-tools-v0.1.52.tar

```
docker load -i mysql.tar

[root@localhost ~]# docker load -i /usr/local/rancher-v2.3.5.tar
43c67172d1d1: Loading layer
[==================================================>]  65.57MB/65.57MB
21ec61b65b20: Loading layer
[==================================================>]  991.2kB/991.2kB
1d0dfb259f6a: Loading layer
[==================================================>]  15.87kB/15.87kB
f55aa0bd26b8: Loading layer
[==================================================>]  3.072kB/3.072kB
e0af200d6950: Loading layer
[==================================================>]  126.1MB/126.1MB
088ed892f9ad: Loading layer
[==================================================>]  6.656kB/6.656kB
6aa3142b4130: Loading layer
[==================================================>]   34.5MB/34.5MB
f4e84c05ab29: Loading layer
[==================================================>]  70.41MB/70.41MB
11a6e4332b53: Loading layer
[==================================================>]  224.8MB/224.8MB
46d1ac556da7: Loading layer
[==================================================>]  3.072kB/3.072kB
0f8b224a5802: Loading layer
[==================================================>]  57.87MB/57.87MB
519eba7d586a: Loading layer
[==================================================>]  99.58MB/99.58MB
3f8bb7c0c150: Loading layer
[==================================================>]  4.608kB/4.608kB
c22c9a5a8211: Loading layer
[==================================================>]  3.072kB/3.072kB
Loaded image: rancher/rancher:v2.3.5

```

## 打个tag

docker tag  f29ece87a1954772accb8a2332ee8c3fe460697e3f102ffbdc76eb9bc4f4f1d0 rancher/rancher-agent:v2.3.5

docker tag  f29ece87a195   172.18.8.104/rancher/rancher-agent:v2.3.5

docker tag 6e421b8753a2  172.18.8.104/rancher/rke-tools:v0.1.52

83fe4871cf67

docker rmi image_name

docker rmi  -f  172.18.8.104/rancher/coredns-coredns:1.6.5

docker rmi   -f   172.18.8.104/rancher/coredns-coredns:v3.4.3-rancher1

docker rmi hub.doge.net/ubuntu:latest

## 保存镜像

- docker export 镜像id -o /home/mysql-export.tar
- docker save 镜像tag  -o /home/mysql-export.tar

## 载入镜像

- docker import  mysql-export.tar

# Docker本地容器相关的操作

## 创建容器

创建名为"centos6"的容器，并在容器内部和宿主机中查看容器中的进程信息

```
docker run -itd -p 6080:80 -p 6022:22 docker.io/lemonbar/centos6-ssh:latest
```

结果如下

```
[root@VM-4-17-centos ~]#    docker run -itd -p 80:80 -p 6022:22
docker.io/lemonbar/centos6-ssh:latest
Unable to find image 'lemonbar/centos6-ssh:latest' locally
```

```
latest: Pulling from lemonbar/centos6-ssh
a3ed95caeb02: Pull complete
f79eb1f22352: Pull complete
67c1aaa530c8: Pull complete
80447774eee7: Pull complete
6d67b3a80e5a: Pull complete
f1819e4b2f8f: Pull complete
09712b5b9acc: Pull complete
8bc987c5494f: Pull complete
c42b021d0ff2: Pull complete
Digest:
sha256:093c2165b3c6fe05d5658343456f9b59bb7ecc690a7d3a112641c86083227dd1
Status: Downloaded newer image for lemonbar/centos6-ssh:latest
a4f1c9b8abcda78c8764cc285183dfa56cd1aa4ce6d111d4d9e77f3a57f3d5fc


```

## 查看活跃容器

docker ps

```
CONTAINER ID        IMAGE                                  COMMAND
         CREATED              STATUS                             PORTS
                                    NAMES
a4f1c9b8abcd        lemonbar/centos6-ssh:latest               "/bin/sh -c
'/usr/sb…"   32 seconds ago      Up 31 seconds
0.0.0.0:6022->22/tcp, 0.0.0.0:6080->80/tcp         determined_curie

```

## 查看全部容器

docker ps -a

## 停止容器

docker stop  id

## 删除容器

docker rm id

## 查看容器的进程信息

**docker top :**查看容器中运行的进程信息，支持 ps 命令参数。

语法

```
docker top [OPTIONS] CONTAINER [ps OPTIONS]
```

容器运行时不一定有/bin/bash终端来交互执行top命令，而且容器还不一定有top命令，可以使用docker top来实现查看container中正在运行的进程。

从Docker 1.11开始，Docker容器运行已经不是简单的通过Docker daemon来启动，而是集成了containerd、runC等多个组件。Docker服务启动之后，我们也可以看见系统上启动了dockerd、docker-containerd等进程，本文主要介绍新版Docker（1.11以后）每个部分的功能和作用。

## 查找容器名称的命令

```
[root@localhost ~]# docker ps --format "{{.Names}}"
```

结果如下：

```
[root@VM-4-17-centos ~]# docker ps --format "{{.Names}}"
determined_curie
redis
nginx_slave
nginx_master
nginx_empty
loving_agnesi
pxc_proxy
pxc03
pxc02
pxc01
affectionate_austin
nostalgic_blackwell

```

# 在容器内部和宿主机中查看容器中的进程信息

```
docker exec -it determined_curie  ps -ef
```

结果如下：

```
[root@VM-4-17-centos ~]# docker exec -it determined_curie  ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 08:14 pts/0    00:00:00 /usr/sbin/sshd -D
root         5     0  0 08:16 pts/1    00:00:00 ps -ef

```

我们可以使用`docker exec`命令进入容器PID名空间，并执行应用。

通过`ps -ef`命令，可以看到每个determined_curie 容器都包含一个PID为1的进程，

"/usr/sbin/sshd"，它是容器的启动进程，具有特殊意义。

利用`docker top`命令，可以让我们从宿主机操作系统中看到容器的进程信息。

```
[root@VM-4-17-centos ~]# docker top determined_curie
UID                 PID                 PPID                C
  STIME               TTY                 TIME                CMD
root                27880               27866               0
  16:14               pts/0               00:00:00            /usr/sbin/sshd
-D
[root@VM-4-17-centos ~]#

```

## 查看其父进程信息

```
[root@VM-4-17-centos ~]#  ps aux | grep 4948
root     17205  1414  0 16:37 pts/0    00:00:00 grep --color=auto 27866
root     27866  5587  0 16:14 ?        00:00:00 docker-containerd-shim -
namespace moby -workdir
/var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/a4f1c9b
8abcda78c8764cc285183dfa56cd1aa4ce6d111d4d9e77f3a57f3d5fc -address
/var/run/docker/containerd/docker-containerd.sock -containerd-binary
/usr/bin/docker-containerd -runtime-root /var/run/docker/runtime-runc
root     27880 27866  0 16:14 pts/0    00:00:00 /usr/sbin/sshd -D

```

## 查看子进程信息

```
[root@VM-4-17-centos ~]#  ps aux | grep 27880
root     17777  0.0  0.0 115928  1008 pts/0    S+   16:38   0:00 grep --
color=auto 27880
root     27880  0.0  0.0  66664  3072 pts/0    Ss+  16:14   0:00
/usr/sbin/sshd -D

```

### 总计三个命令

```
ps -ef | grep 4948 #查看父进程
ps aux | grep 27880 #查看子进程(容器)
docker ps -a | grep determined_curie   #定位容器id

```

## 再启动一个 centos 容器

```
 docker run -itd  --name centos6-2  -p 6081:80 -p 6021:22
docker.io/lemonbar/centos6-ssh:latest
```

输出如下：

```
[root@VM-4-17-centos ~]# docker run -itd  --name centos6-2  -p 6081:80 -p
6021:22 docker.io/lemonbar/centos6-ssh:latest
460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd

```

### 查看进程信息

利用`docker top`命令，可以让我们从宿主机操作系统中看到容器的进程信息。

```
[root@VM-4-17-centos ~]# docker top centos6-2
UID                 PID                 PPID                C
  STIME               TTY                 TIME                CMD
root                4962                4948                0
  16:24               pts/0               00:00:00            /usr/sbin/sshd
-D


```

### 查看其父进程信息

```
[root@VM-4-17-centos ~]# ps -ef | grep 4948
root      4948  5587  0 16:24 ?        00:00:00 docker-containerd-shim -
namespace moby -workdir
/var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/460d688
239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd -address
/var/run/docker/containerd/docker-containerd.sock -containerd-binary
/usr/bin/docker-containerd -runtime-root /var/run/docker/runtime-runc
root      4962  4948  0 16:24 pts/0    00:00:00 /usr/sbin/sshd -D

```

### 查看子进程信息

```
[root@VM-4-17-centos ~]# ps aux | grep 4962
root      4962  0.0  0.0  66664  3068 pts/0    Ss+  16:24   0:00 /usr/sbin/sshd -D
root     15311  0.0  0.0 115932  1008 pts/0    S+   16:35   0:00 grep --color=auto 4962
```

### 总计三个命令

```
ps -ef | grep 4948 #查看父进程
ps aux | grep 4962 #查看子进程(容器)
docker ps -a | grep centos6-2  #定位容器id

```

## 进程的对应关系

### Linux通过进程ID查看文件路径

子进程的文件路径

```
[root@VM-4-17-centos ~]#  ls -l /proc/27880
total 0
dr-xr-xr-x 2 root root 0 Nov  3 16:41 attr
-rw-r--r-- 1 root root 0 Nov  3 16:41 autogroup
-r-------- 1 root root 0 Nov  3 16:41 auxv
-r--r--r-- 1 root root 0 Nov  3 16:14 cgroup
--w------- 1 root root 0 Nov  3 16:41 clear_refs
-r--r--r-- 1 root root 0 Nov  3 16:15 cmdline
-rw-r--r-- 1 root root 0 Nov  3 16:41 comm
-rw-r--r-- 1 root root 0 Nov  3 16:41 coredump_filter
-r--r--r-- 1 root root 0 Nov  3 16:41 cpuset
lrwxrwxrwx 1 root root 0 Nov  3 16:41 cwd -> /
-r-------- 1 root root 0 Nov  3 16:41 environ
lrwxrwxrwx 1 root root 0 Nov  3 16:14 exe -> /usr/sbin/sshd
dr-x------ 2 root root 0 Nov  3 16:14 fd
dr-x------ 2 root root 0 Nov  3 16:41 fdinfo
-rw-r--r-- 1 root root 0 Nov  3 16:41 gid_map
-r-------- 1 root root 0 Nov  3 16:41 io
-r--r--r-- 1 root root 0 Nov  3 16:41 limits
-rw-r--r-- 1 root root 0 Nov  3 16:41 loginuid
dr-x------ 2 root root 0 Nov  3 16:41 map_files
-r--r--r-- 1 root root 0 Nov  3 16:41 maps
-rw------- 1 root root 0 Nov  3 16:41 mem
-r--r--r-- 1 root root 0 Nov  3 16:14 mountinfo
-r--r--r-- 1 root root 0 Nov  3 16:41 mounts
-r-------- 1 root root 0 Nov  3 16:41 mountstats
dr-xr-xr-x 5 root root 0 Nov  3 16:41 net
dr-x--x--x 2 root root 0 Nov  3 16:14 ns
-r--r--r-- 1 root root 0 Nov  3 16:41 numa_maps
-rw-r--r-- 1 root root 0 Nov  3 16:41 oom_adj
-r--r--r-- 1 root root 0 Nov  3 16:41 oom_score
```

```
-rw-r--r-- 1 root root 0 Nov  3 16:41 oom_score_adj
-r--r--r-- 1 root root 0 Nov  3 16:41 pagemap
-r-------- 1 root root 0 Nov  3 16:41 patch_state
-r--r--r-- 1 root root 0 Nov  3 16:41 personality
-rw-r--r-- 1 root root 0 Nov  3 16:41 projid_map
lrwxrwxrwx 1 root root 0 Nov  3 16:41 root -> /
-rw-r--r-- 1 root root 0 Nov  3 16:41 sched
-r--r--r-- 1 root root 0 Nov  3 16:41 schedstat
-r--r--r-- 1 root root 0 Nov  3 16:41 sessionid
-rw-r--r-- 1 root root 0 Nov  3 16:41 setgroups
-r--r--r-- 1 root root 0 Nov  3 16:41 smaps
-r--r--r-- 1 root root 0 Nov  3 16:41 stack
-r--r--r-- 1 root root 0 Nov  3 16:14 stat
-r--r--r-- 1 root root 0 Nov  3 16:41 statm
-r--r--r-- 1 root root 0 Nov  3 16:14 status
-r--r--r-- 1 root root 0 Nov  3 16:41 syscall
dr-xr-xr-x 3 root root 0 Nov  3 16:41 task
-r--r--r-- 1 root root 0 Nov  3 16:41 timers
-rw-r--r-- 1 root root 0 Nov  3 16:14 uid_map
-r--r--r-- 1 root root 0 Nov  3 16:41 wchan

```

以下是/proc目录中进程27880的信息说明：

```
proc/27880 pid为N的进程信息

/proc/27880/cmdline 进程启动命令

/proc/27880/cwd 链接到进程当前工作目录

/proc/27880/environ 进程环境变量列表

/proc/27880/exe 链接到进程的执行命令文件

/proc/27880/fd 包含进程相关的所有的文件描述符

/proc/27880/maps 与进程相关的内存映射信息

/proc/27880/mem 指代进程持有的内存，不可读

/proc/27880/root 链接到进程的根目录

/proc/27880/stat 进程的状态

/proc/27880/statm 进程使用的内存的状态

/proc/27880/status 进程状态信息，比stat/statm更具可读性
```

## 容器的PID namespace（命名空间）

在Docker中，进程管理的基础就是Linux内核中的PID名空间技术。

在不同PID名空间中，进程ID是独立的；即在两个不同名空间下的进程可以有相同的PID。

在Docker中，每个Container进程缺省都具有不同的PID名空间。通过名空间技术，Docker实现容器间的进程隔离。

docker中运行的容器进程，本质上还是运行在宿主机上的，所以也会拥有相对应的PID

**找出容器ID**

```
# docker ps
```

输出

```
[root@VM-4-17-centos ~]# docker ps
CONTAINER ID        IMAGE                                  COMMAND
         CREATED             STATUS                          PORTS
                                    NAMES
460d68823930        lemonbar/centos6-ssh:latest             "/bin/sh -c
'/usr/sb…"   32 minutes ago      Up 32 minutes
0.0.0.0:6021->22/tp, 0.0.0.0:6081->80/tcp         centos6-2

```

## 查看容器信息

```
docker inspect  id
```

输出

```
[root@VM-4-17-centos ~]# docker inspect  460d68823930
[root@VM-4-17-centos ~]#  docker inspect  460d68823930
[
    {
        "Id":
"460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd",
        "Created": "2021-11-03T08:24:36.934129599Z",
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "/usr/sbin/sshd -D"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 4962,
            "ExitCode": 0,
            "Error": "",
```

```
            "StartedAt": "2021-11-03T08:24:37.223255812Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image":
"sha256:efd998bd6817af509d348b488e3ce4259f9f05632644a7bf574b785bbc8950b8",
        "ResolvConfPath":
"/var/lib/docker/containers/460d688239304172f39bb9586bfc5959e0c3db64e7c3a09
37f1003f94408ebbd/resolv.conf",
        "HostnamePath":
"/var/lib/docker/containers/460d688239304172f39bb9586bfc5959e0c3db64e7c3a09
37f1003f94408ebbd/hostname",
        "HostsPath":
"/var/lib/docker/containers/460d688239304172f39bb9586bfc5959e0c3db64e7c3a09
37f1003f94408ebbd/hosts",
        "LogPath":
"/var/lib/docker/containers/460d688239304172f39bb9586bfc5959e0c3db64e7c3a09
37f1003f94408ebbd/460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94
408ebbd-json.log",
        "Name": "/centos6-2",
        "RestartCount": 0,
        "Driver": "overlay2",
        "Platform": "linux",
        "MountLabel": "",
        "ProcessLabel": "",
        "AppArmorProfile": "",
        "ExecIDs": null,
        "HostConfig": {
            "Binds": null,
            "ContainerIDFile": "",
            "LogConfig": {
                "Type": "json-file",
                "Config": {}
            },
            "NetworkMode": "default",
            "PortBindings": {
                "22/tcp": [
                    {
                        "HostIp": "",
                        "HostPort": "6021"
                    }
                ],
                "80/tcp": [
                    {
                        "HostIp": "",
                        "HostPort": "6081"
                    }
                ]
            },
            "RestartPolicy": {
                "Name": "no",
                "MaximumRetryCount": 0
            },
            "AutoRemove": false,
            "VolumeDriver": "",
            "VolumesFrom": null,
            "CapAdd": null,
            "CapDrop": null,
            "Dns": [],
```

```
            "DnsOptions": [],
            "DnsSearch": [],
            "ExtraHosts": null,
            "GroupAdd": null,
            "IpcMode": "shareable",
            "Cgroup": "",
            "Links": null,
            "OomScoreAdj": 0,
            "PidMode": "",
            "Privileged": false,
            "PublishAllPorts": false,
            "ReadonlyRootfs": false,
            "SecurityOpt": null,
            "UTSMode": "",
            "UsernsMode": "",
            "ShmSize": 67108864,
            "Runtime": "runc",
            "ConsoleSize": [
                0,
                0
            ],
            "Isolation": "",
            "CpuShares": 0,
            "Memory": 0,
            "NanoCpus": 0,
            "CgroupParent": "",
            "BlkioWeight": 0,
            "BlkioWeightDevice": [],
            "BlkioDeviceReadBps": null,
            "BlkioDeviceWriteBps": null,
            "BlkioDeviceReadIOps": null,
            "BlkioDeviceWriteIOps": null,
            "CpuPeriod": 0,
            "CpuQuota": 0,
            "CpuRealtimePeriod": 0,
            "CpuRealtimeRuntime": 0,
            "CpusetCpus": "",
            "CpusetMems": "",
            "Devices": [],
            "DeviceCgroupRules": null,
            "DiskQuota": 0,
            "KernelMemory": 0,
            "MemoryReservation": 0,
            "MemorySwap": 0,
            "MemorySwappiness": null,
            "OomKillDisable": false,
            "PidsLimit": 0,
            "Ulimits": null,
            "CpuCount": 0,
            "CpuPercent": 0,
            "IOMaximumIOps": 0,
            "IOMaximumBandwidth": 0
        },
        "GraphDriver": {
            "Data": {
```

```
                "LowerDir":
"/var/lib/docker/overlay2/6835c1b48237aafe27e2efabeda92a3a6623f254f88d54b5e
6acce454e560dd6-
init/diff:/var/lib/docker/overlay2/7139bf0b716c6e0b6a0c709b7043466f9bbfd702
4f8ae584061c00b0bd97348c/diff:/var/lib/docker/overlay2/66a3e278259cdcf50741
ce30a115baa3bd6247a60c487e4118e85f2f39328f11/diff:/var/lib/docker/overlay2/
20e22c4c28ebadb615eb4c7c290253d3eb91cb49722ee2931b0ee628352a5857/diff:/var/
lib/docker/overlay2/a3fa9dbebc83a853083205b8f7921c632cd67f64531f4a25cab419a
43172e3ae/diff:/var/lib/docker/overlay2/3af7958c9a4e54d24598058a9fa1e85eb35
e3d40f766fa498a674b52724ae73e/diff:/var/lib/docker/overlay2/becb65af4396137
ed41fe6d516e834e6e6e9120f4edfac8e2ca8dd67cce23268/diff:/var/lib/docker/over
lay2/fef055305158cc96906514c447f0eaea05945138896b0b35ac4146b6a2a3e273/diff:
/var/lib/docker/overlay2/79158cdf3ba832493ab0d02d560c784208fe51c74236a5a86f
7fb4fb50ab6e44/diff:/var/lib/docker/overlay2/86258a18e1110582b819719593687f
11f0404d00a41667b3432c3b974fb1ce42/diff:/var/lib/docker/overlay2/8826b2e006
8653fb2c5e8a3dbf839470e2b8eef8cf752b5fe901bea1b210849f/diff:/var/lib/docker
/overlay2/145301e2738a8a7581c2bbd5beb9bf7a49b247e46642b8084efbc026a1826116/
diff:/var/lib/docker/overlay2/f621f37535e0db1fe44902e22dba7ef0844b9a8b562a9
daa39a842a49e9cc9bb/diff:/var/lib/docker/overlay2/7b493e4a97907aaa18b97ad2e
9120b5bf87c0e9908ee390a35ea6ff546d8cec6/diff",
                "MergedDir":
"/var/lib/docker/overlay2/6835c1b48237aafe27e2efabeda92a3a6623f254f88d54b5e
6acce454e560dd6/merged",
                "UpperDir":
"/var/lib/docker/overlay2/6835c1b48237aafe27e2efabeda92a3a6623f254f88d54b5e
6acce454e560dd6/diff",
                "WorkDir":
"/var/lib/docker/overlay2/6835c1b48237aafe27e2efabeda92a3a6623f254f88d54b5e
6acce454e560dd6/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [],
        "Config": {
            "Hostname": "460d68823930",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "22/tcp": {},
                "80/tcp": {}
            },
            "Tty": true,
            "OpenStdin": true,
            "StdinOnce": false,
            "Env": [
                "HOME=/",

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "/usr/sbin/sshd -D"
            ],
            "Image": "docker.io/lemonbar/centos6-ssh:latest",
```

```
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
        },
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID":
"ea66261fb6d8089d5b2d585a2dc32b2003365df7118f5f5e898a152fb5b35773",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {
                "22/tcp": [
                    {
                        "HostIp": "0.0.0.0",
                        "HostPort": "6021"
                    }
                ],
                "80/tcp": [
                    {
                        "HostIp": "0.0.0.0",
                        "HostPort": "6081"
                    }
                ]
            },
            "SandboxKey": "/var/run/docker/netns/ea66261fb6d8",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID":
"09ad719a4e9115ee56c5fb0f5b0d39c50bf5acaf0a1afacedc13969c82a2969f",
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.6",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:11:00:06",
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID":
"2586283d16a08210c955d705f05e0f6999b59523a84b0c163e33f535af809ddd",
                    "EndpointID":
"09ad719a4e9115ee56c5fb0f5b0d39c50bf5acaf0a1afacedc13969c82a2969f",
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.6",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:06",
                    "DriverOpts": null
                }
            }
```

```
        }
    }
]

```

## 进入相应目录

```
# cd /sys/fs/cgroup/memory/docker/460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd/
```

输出

```
 cd /sys/fs/cgroup/memory/docker/460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd/
[root@VM-4-17-centos 460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd]# ll
total 0
-rw-r--r-- 1 root root 0 Nov  3 16:24 cgroup.clone_children
--w--w--w- 1 root root 0 Nov  3 16:24 cgroup.event_control
-rw-r--r-- 1 root root 0 Nov  3 16:24 cgroup.procs
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.failcnt
--w------- 1 root root 0 Nov  3 16:24 memory.force_empty
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.kmem.failcnt
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.kmem.limit_in_bytes
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.kmem.max_usage_in_bytes
-r--r--r-- 1 root root 0 Nov  3 16:24 memory.kmem.slabinfo
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.kmem.tcp.failcnt
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.kmem.tcp.limit_in_bytes
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.kmem.tcp.max_usage_in_bytes
-r--r--r-- 1 root root 0 Nov  3 16:24 memory.kmem.tcp.usage_in_bytes
-r--r--r-- 1 root root 0 Nov  3 16:24 memory.kmem.usage_in_bytes
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.limit_in_bytes
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.max_usage_in_bytes
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.memsw.failcnt
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.memsw.limit_in_bytes
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.memsw.max_usage_in_bytes
-r--r--r-- 1 root root 0 Nov  3 16:24 memory.memsw.usage_in_bytes
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.move_charge_at_immigrate
-r--r--r-- 1 root root 0 Nov  3 16:24 memory.numa_stat
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.oom_control
---------- 1 root root 0 Nov  3 16:24 memory.pressure_level
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.soft_limit_in_bytes
-r--r--r-- 1 root root 0 Nov  3 16:24 memory.stat
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.swappiness
-r--r--r-- 1 root root 0 Nov  3 16:24 memory.usage_in_bytes
-rw-r--r-- 1 root root 0 Nov  3 16:24 memory.use_hierarchy
-rw-r--r-- 1 root root 0 Nov  3 16:24 notify_on_release
-rw-r--r-- 1 root root 0 Nov  3 16:24 tasks

```

```
[root@VM-4-17-centos 460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd]# cat cgroup.procs
4962
11539
[root@VM-4-17-centos 460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd]# cat pids.max
max
[root@VM-4-17-centos 460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd]# cat tasks
4962
11539
[root@VM-4-17-centos 460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd]# cat cgroup.clone_children
0
[root@VM-4-17-centos 460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd]# pwd
/sys/fs/cgroup/pids/docker/460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd

```

## 查看容器目录里的进程号

进程号就存在一个文件里面

```
[root@VM-4-17-centos 460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd]#
cat cgroup.procs
4962

```

与前面利用`docker top`命令，可以让我们从宿主机操作系统中看到容器的进程信息。

```
[root@VM-4-17-centos ~]# docker top centos6-2
UID                 PID                 PPID                C                   STIME               TTY                 TIME                CMD
root                4962                4948                0                   16:24               pts/0               00:00:00            /usr/sbin/sshd -D


```

## 启动一个进程

我们下面会在 centos6-2容器中，利用`docker exec`命令启动一个"sleep"进程

```
[root@VM-4-17-centos ]# docker exec -d  centos6-2  sleep 2000
[root@VM-4-17-centos ]# docker exec  centos6-2  ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 08:24 pts/0    00:00:00 /usr/sbin/sshd -D
root         6     0  0 09:06 ?        00:00:00 sleep 2000
root        10     0  0 09:06 ?        00:00:00 ps -ef


```

查看宿主机的进程号

```
[root@VM-4-17-centos ]#  docker top centos6-2
UID                 PID                 PPID                C
  STIME               TTY                 TIME                CMD
root                4962                4948                0
  16:24               pts/0               00:00:00            /usr/sbin/sshd
-D
root                11539               4948                0
  17:06               ?                   00:00:00            sleep 2000

```

我们可以清楚的看到exec命令创建的sleep进程属 centos6-2 容器的名空间，但是它的父进程是Docker容器的启动进程。

## 查看容器目录里的进程号

进程号就存在一个文件里面

```
[root@VM-4-17-centos
460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd]# cat
cgroup.procs
4962
11539


```

```
 docker exec -d  centos6-2 pstree -p

 docker exec -d  centos6-2 ps -auxf

  docker exec -d  centos6-2 ll /proc
```

输出

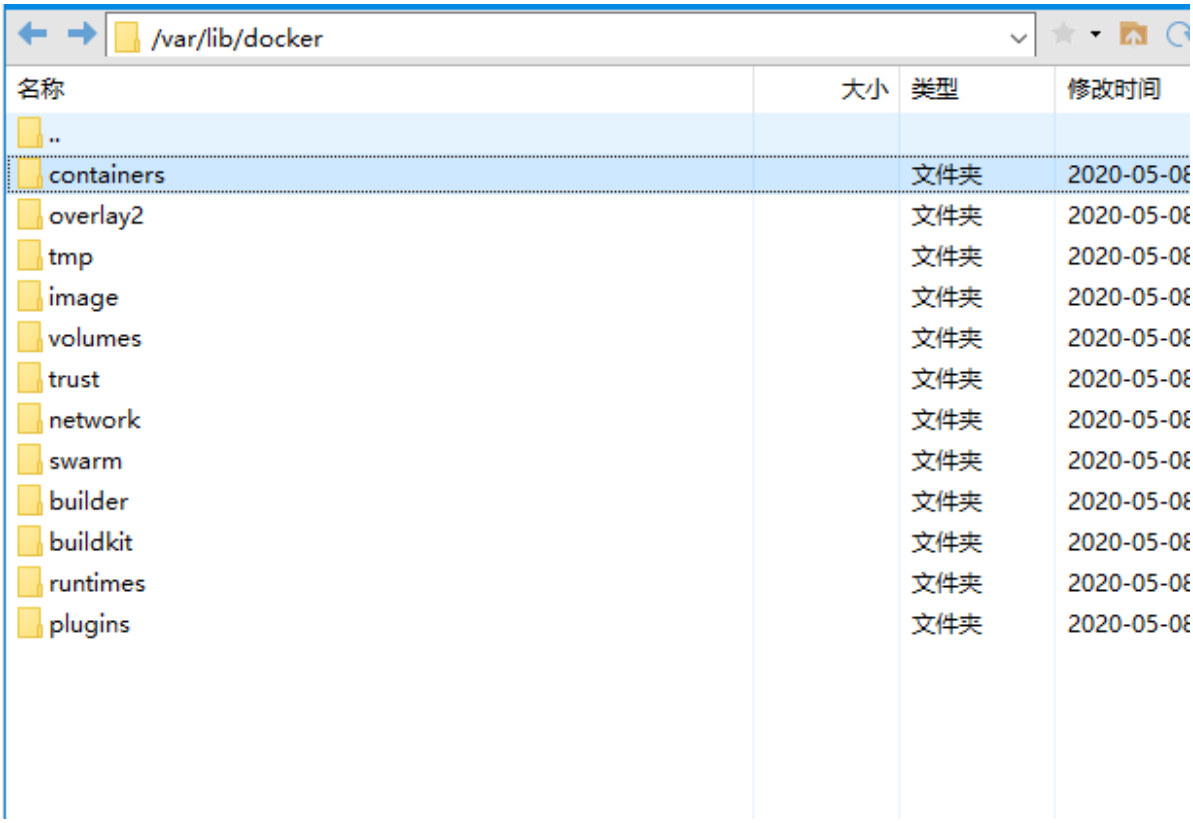
```
[root@VM-4-17-centos docker]#  docker exec  centos6-2  ps -ef
```

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 08:24 pts/0    00:00:00 /usr/sbin/sshd -D
root         6     0  0 09:06 ?        00:00:00 sleep 2000
root        40     0  0 09:26 ?        00:00:00 ps -ef
[root@VM-4-17-centos docker]#  docker exec  centos6-2  ps -auxf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        44  0.0  0.0  13360  1012 ?        Rs   09:26   0:00 ps -auxf
root         6  0.0  0.0   4120   316 ?        Ss   09:06   0:00 sleep 2000
root         1  0.0  0.0  66664  3068 pts/0    Ss+  08:24   0:00
/usr/sbin/sshd -D
Warning: bad syntax, perhaps a bogus '-'? See /usr/share/doc/procps-
3.2.8/FAQ
[root@VM-4-17-centos docker]#  docker exec  centos6-2  pstree -p
sshd(1)

```

# Docker文件目录和容器内部操作

Docker默认的文件目录位于Linux server的/var/lib/docker 下面。目录结构如下




|-----containers：用于存储容器信息

|-----image：用来存储镜像中间件及本身信息，大小，依赖信息

|-----network

|-----swarm

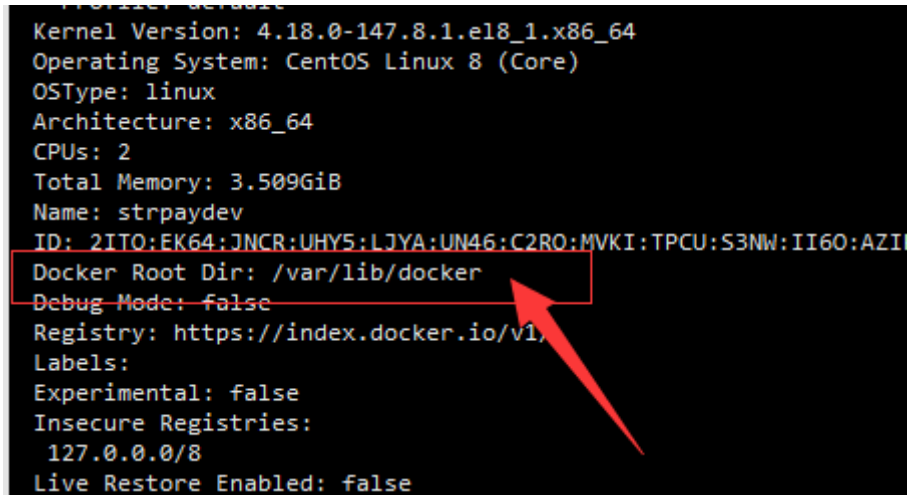
|-----tmp：docker临时目录

|-----trust：docker信任目录

|-----volumes：docker卷目录

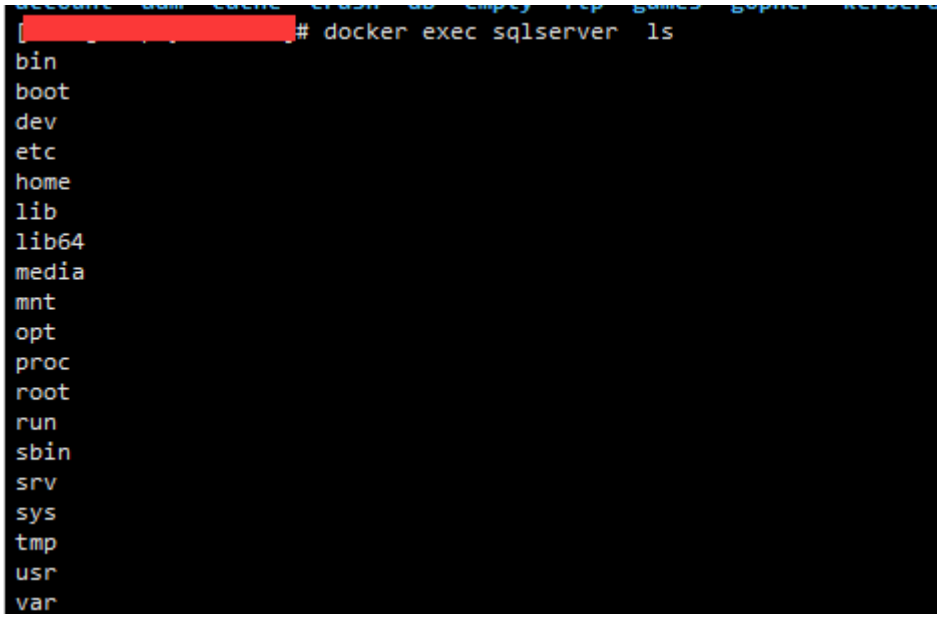
还可以通过docker指令确认文件位置：

```
docker info
```




查看某个容器的文件目录：

```
docker exec 容器name ls
```




```
docker exec centos6-2   ls /proc
```

```
[root@VM-4-17-centos containers]#  docker exec centos6-2  ls /proc
1
103
```

```
acpi
buddyinfo
bus
cgroups
cmdline
consoles
cpuinfo
crypto
devices
diskstats
dma
driver
execdomains
fb
filesystems
fs
interrupts
iomem
ioports
irq
kallsyms
kcore
key-users
keys
kmsg
kpagecount
kpageflags
loadavg
locks
mdstat
meminfo
misc
modules
mounts
mtrr
net
pagetypeinfo
partitions
sched_debug
schedstat
scsi
self
slabinfo
softirqs
stat
swaps
sys
sysrq-trigger
sysvipc
timer_list
timer_stats
tty
uptime
version
vmallocinfo
vmstat
zoneinfo

```

# docker daemon (docker守护进程)

```
pidof dockerd   #查看docker守护进程pid
lsof -p 3197 | wc -l #docker守护进程打开的文件数
```

在两个容器中的"centos "是两个独立的进程，但是他们拥有相同的父进程 Docker Daemon。

所以Docker可以父子进程的方式在Docker Daemon和Redis容器之间进行交互。

另一个值得注意的方面是，`docker exec`命令可以进入指定的容器内部执行命令。由它启动的进程属于容器的namespace和相应的cgroup。

但是这些进程的父进程是Docker Daemon而非容器的PID1进程。

我们下面会在Redis容器中，利用`docker exec`命令启动一个"sleep"进程

```
docker@default:~$ docker exec -d redis sleep 2000
docker@default:~$ docker exec redis ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
redis        1     0  0 02:26 ?        00:00:00 redis-server *:6379
root        11     0  0 02:26 ?        00:00:00 sleep 2000
root        21     0  0 02:29 ?        00:00:00 ps -ef
docker@default:~$ docker top redis
UID                 PID                 PPID                C
    STIME               TTY                 TIME                CMD
999                 9955                1264                0
    02:12               ?                   00:00:00            redis-server
*:6379
root                9984                1264                0
    02:13               ?                   00:00:00            sleep 2000
```

我们可以清楚的看到exec命令创建的sleep进程属Redis容器的名空间，但是它的父进程是Docker Daemon。

如果我们在宿主机操作系统中手动杀掉容器的启动进程（在上文示例中是redis-server），容器会自动结束，而容器名空间中所有进程也会退出。

```
docker@default:~$ PID=$(docker inspect --format="{{.State.Pid}}" redis)
docker@default:~$ sudo kill $PID
docker@default:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
        STATUS                      PORTS               NAMES
356eca186321        redis               "/entrypoint.sh redis"   23 minutes
ago      Up 4 minutes               6379/tcp            redis2
f6bc57cc1b46        redis               "/entrypoint.sh redis"   23 minutes
ago      Exited (0) 4 seconds ago                       redis
```

通过以上示例：

- 每个容器有独立的PID名空间，
- 容器的生命周期和其PID1进程一致
- 利用`docker exec`可以进入到容器的名空间中启动进程

# Docker Daemon 原理

作为Docker容器管理的守护进程，Docker Daemon从最初集成在`docker`命令中（1.11版本前），到后来的独立成单独二进制程序（1.11版本开始），其功能正在逐渐拆分细化，被分配到各个单独的模块中去。

## 演进：Docker守护进程启动

从Docker服务的启动脚本，也能看见守护进程的逐渐剥离：

在Docker 1.8之前，Docker守护进程启动的命令为：

```
docker -d
```

这个阶段，守护进程看上去只是Docker client的一个选项。

Docker 1.8开始，启动命令变成了：

```
docker daemon
```

这个阶段，守护进程看上去是`docker`命令的一个模块。

Docker 1.11开始，守护进程启动命令变成了：

```
dockerd
```

其服务的配置文件为：

```

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd
ExecReload=/bin/kill -s HUP $MAINPID
```

此时已经和Docker client分离，独立成一个二进制程序了。

当然，守护进程模块不停的在重构，其基本功能和定位没有变化。和一般的CS架构系统一样，守护进程负责和Docker client交互，并管理Docker镜像、容器。

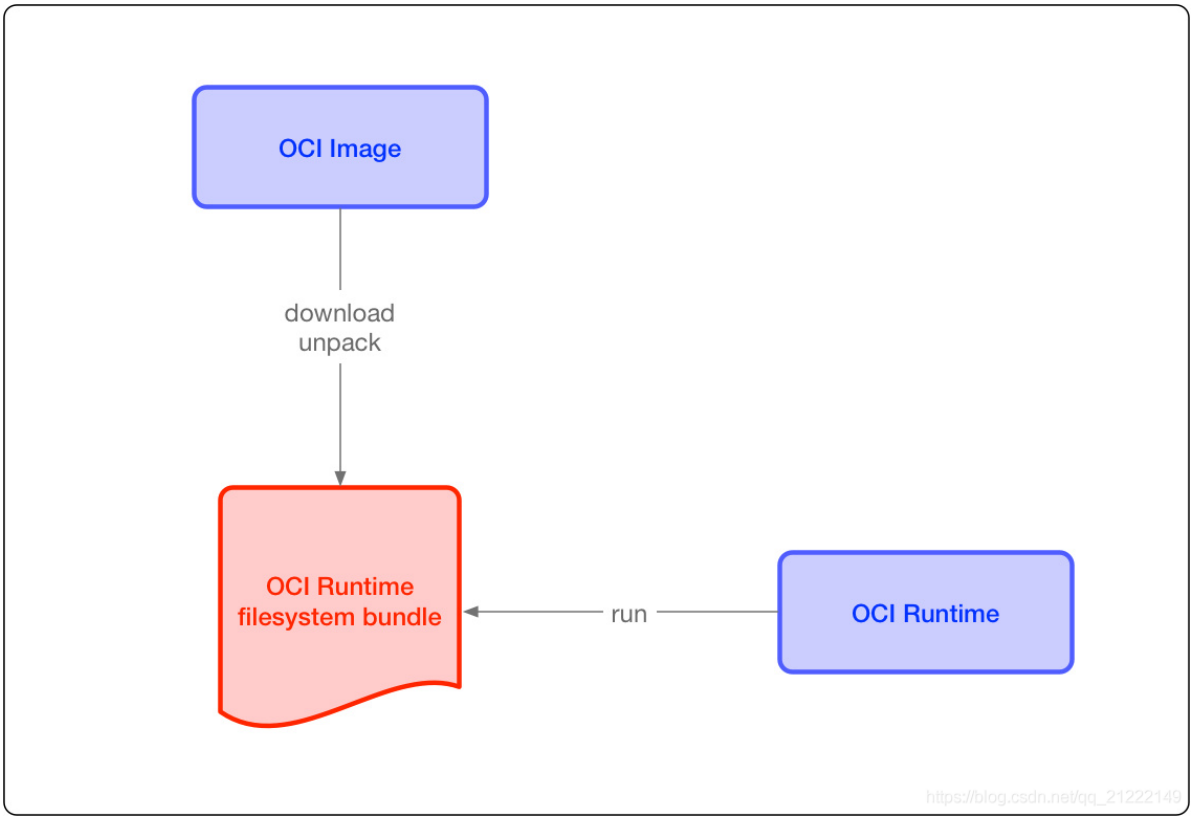
# OCI（Open Container Initiative）

Open Container Initiative，也就是常说的OCI，是由多家公司共同成立的项目，并由linux基金会进行管理，致力于container runtime的标准的制定和runc的开发等工作。
官方的介绍是

> An open governance structure for the express purpose of creating open industry standards around container formats and runtime. – Open Containers Official Site

所谓container runtime，主要负责的是容器的生命周期的管理。oci的runtime spec标准中对于容器的状态描述，以及对于容器的创建、删除、查看等操作进行了定义。

目前主要有两个标准文档：容器运行时标准 （runtime spec）和 容器镜像标准（image spec）。
这两个协议通过 OCI runtime filesytem bundle 的标准格式连接在一起，OCI 镜像可以通过工具转换成 bundle，然后 OCI 容器引擎能够识别这个 bundle 来运行容器。



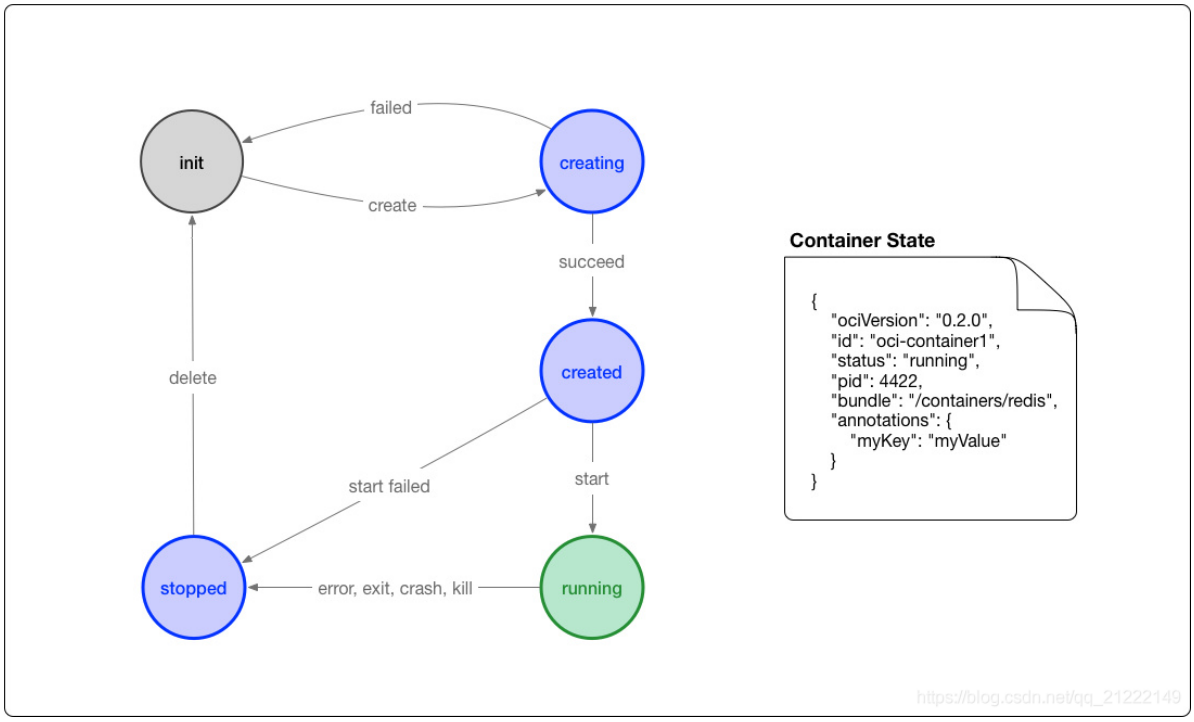

## image spec

OCI 容器镜像主要包括几块内容：

文件系统：以 layer 保存的文件系统，每个 layer 保存了和上层之间变化的部分，layer 应该保存哪些文件，怎么表示增加、修改和删除的文件等
config 文件：保存了文件系统的层级信息（每个层级的 hash 值，以及历史信息），以及容器运行时需要的一些信息（比如环境变量、工作目录、命令参数、mount 列表），指定了镜像在某个特定平台和系统的配置。比较接近我们使用 docker inspect <image_id> 看到的内容
manifest 文件：镜像的 config 文件索引，有哪些 layer，额外的 annotation 信息，manifest 文件中保存了很多和当前平台有关的信息
index 文件：可选的文件，指向不同平台的 manifest 文件，这个文件能保证一个镜像可以跨平台使用，每个平台拥有不同的 manifest 文件，使用 index 作为索引

### runtime spec

OCI 对容器 runtime 的标准主要是指定容器的运行状态，和 runtime 需要提供的命令。下图可以是容器状态转换图：




## Docker CLI

```
/usr/bin/docker
```

Docker 的客户端工具，通过CLI与 dockerd API 交流。 CLI 的例子比如docker build ... docker run ...

## Docker Daemon

```
/usr/bin/dockerd
```

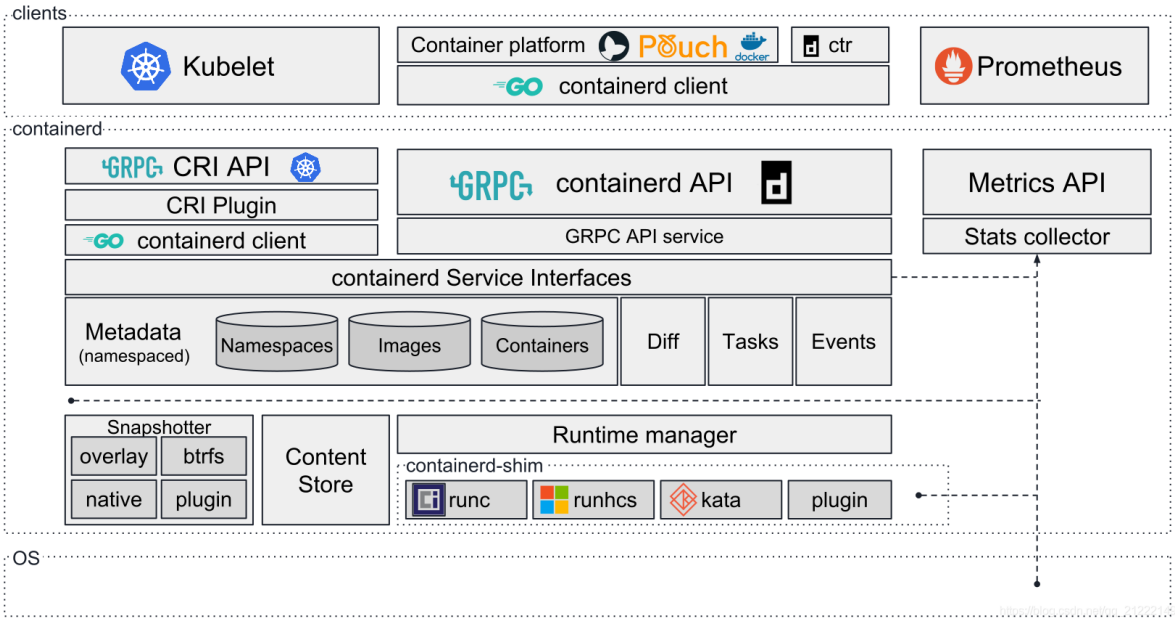
当然，守护进程模块不停的在重构，其基本功能和定位没有变化。和一般的CS架构系统一样，守护进程负责和Docker client交互，并管理Docker镜像、容器。

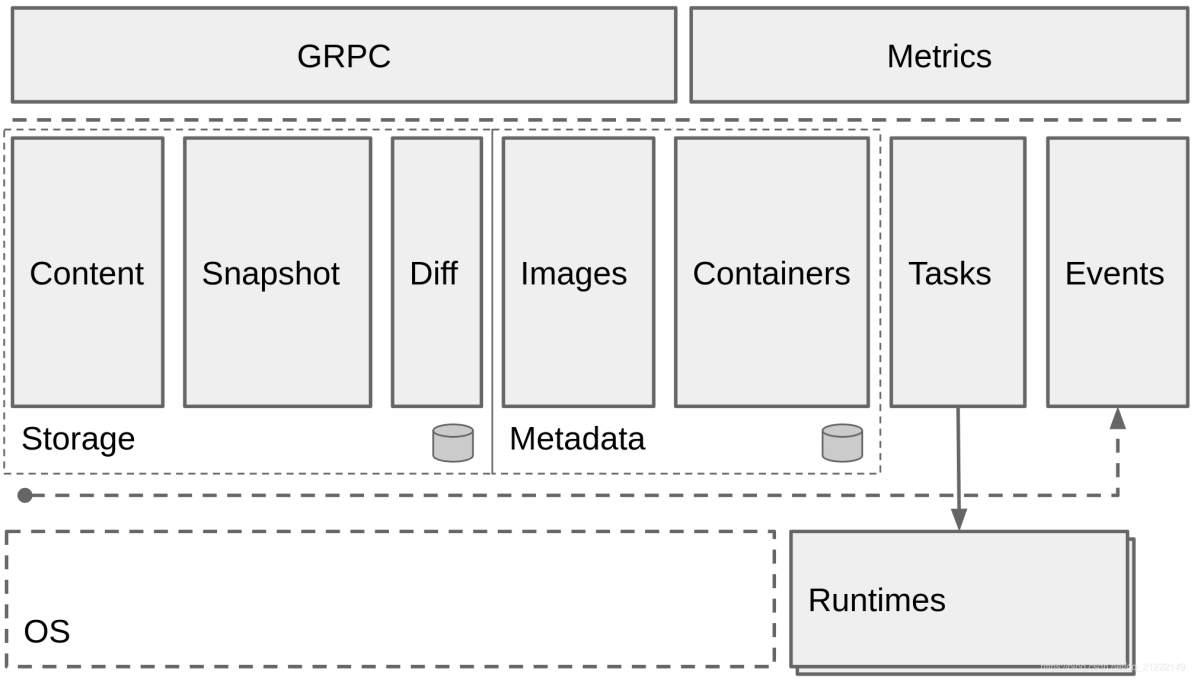
## Containerd

```
/usr/bin/docker-containerd
```

containerd是容器技术标准化之后的产物，为了能够兼容OCI标准，将容器运行时及其管理功能从Docker Daemon剥离。理论上，即使不运行dockerd，也能够直接通过containerd来管理容器。（当然，containerd本身也只是一个守护进程，容器的实际运行时由后面介绍的runC控制。）
最近，Docker刚刚宣布开源containerd。从其项目介绍页面可以看出，containerd主要职责是**镜像管理**（镜像、元信息等）、**容器执行**（调用最终运行时组件执行）。

containerd向上为Docker Daemon提供了gRPC接口，使得Docker Daemon屏蔽下面的结构变化，确保原有接口向下兼容。向下通过containerd-shim结合runC，使得引擎可以独立升级，避免之前Docker Daemon升级会导致所有容器不可用的问题。




containerd fully leverages the **OCI runtime specification1, image format specifications and OCI reference implementation (runc)**.

containerd includes a daemon exposing gRPC API over a local UNIX socket. The API is a low-level one designed for higher layers to wrap and extend. Containerd uses RunC to run containers according to the OCI specification.

## docker-shim

```
1 /usr/bin/docker-containerd-shim
```

每启动一个容器都会起一个新的docker-shim的一个进程.
他直接通过指定的三个参数来创建一个容器:

1. 容器id
2. boundle目录（containerd的对应某个容器生成的目录，一般位于：/var/run/docker/libcontainerd/containerID）
3. 运行是二进制（默认为runc）来调用runc的api（比如创建容器时，最后拼装的命令如下：runc create。。。）

他的作用是：

1. 它允许容器运行时(即 runC)在启动容器之后退出，简单说就是不必为每个容器一直运行一个容器运行时(runC)
2. 即使在 containerd 和 dockerd 都挂掉的情况下，容器的标准 IO 和其它的文件描述符也都是可用的
3. 向 containerd 报告容器的退出状态
   前两点尤其重要，有了它们就可以在不中断容器运行的情况下升级或重启 dockerd(这对于生产环境来说意义重大)。

## runc (OCI reference implementation)

```
/usr/bin/docker-runc
```

OCI定义了容器运行时标准OCI Runtime Spec support (aka runC)，runC是Docker按照开放容器格式标准（OCF, Open Container Format）制定的一种具体实现。

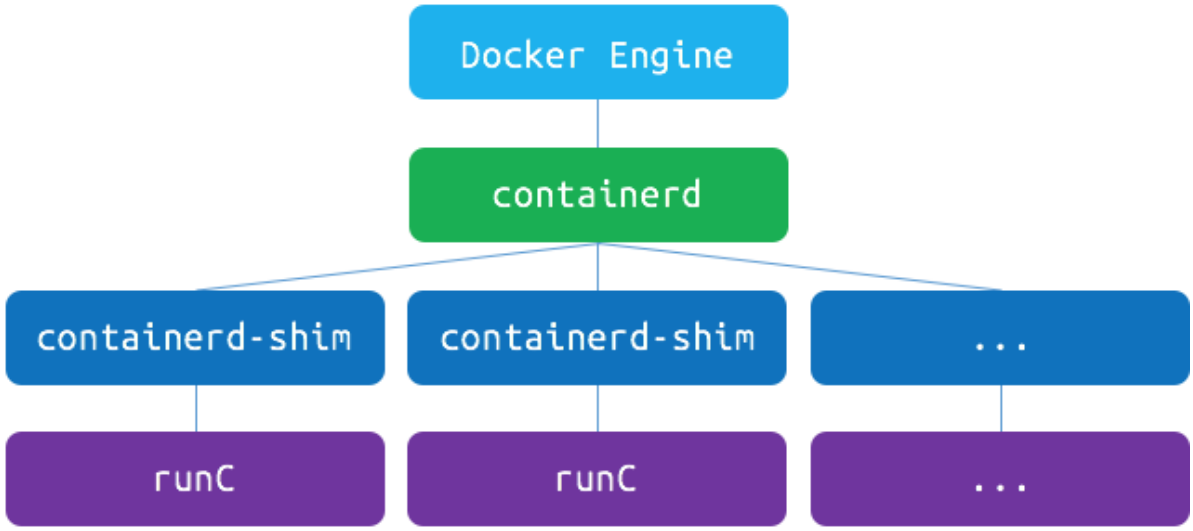
runC是从Docker的**libcontainer**中迁移而来的，实现了容器启停、资源隔离等功能。

Docker默认提供了docker-runc实现，事实上，通过containerd的封装，可以在Docker Daemon启动的时候指定runc的实现。

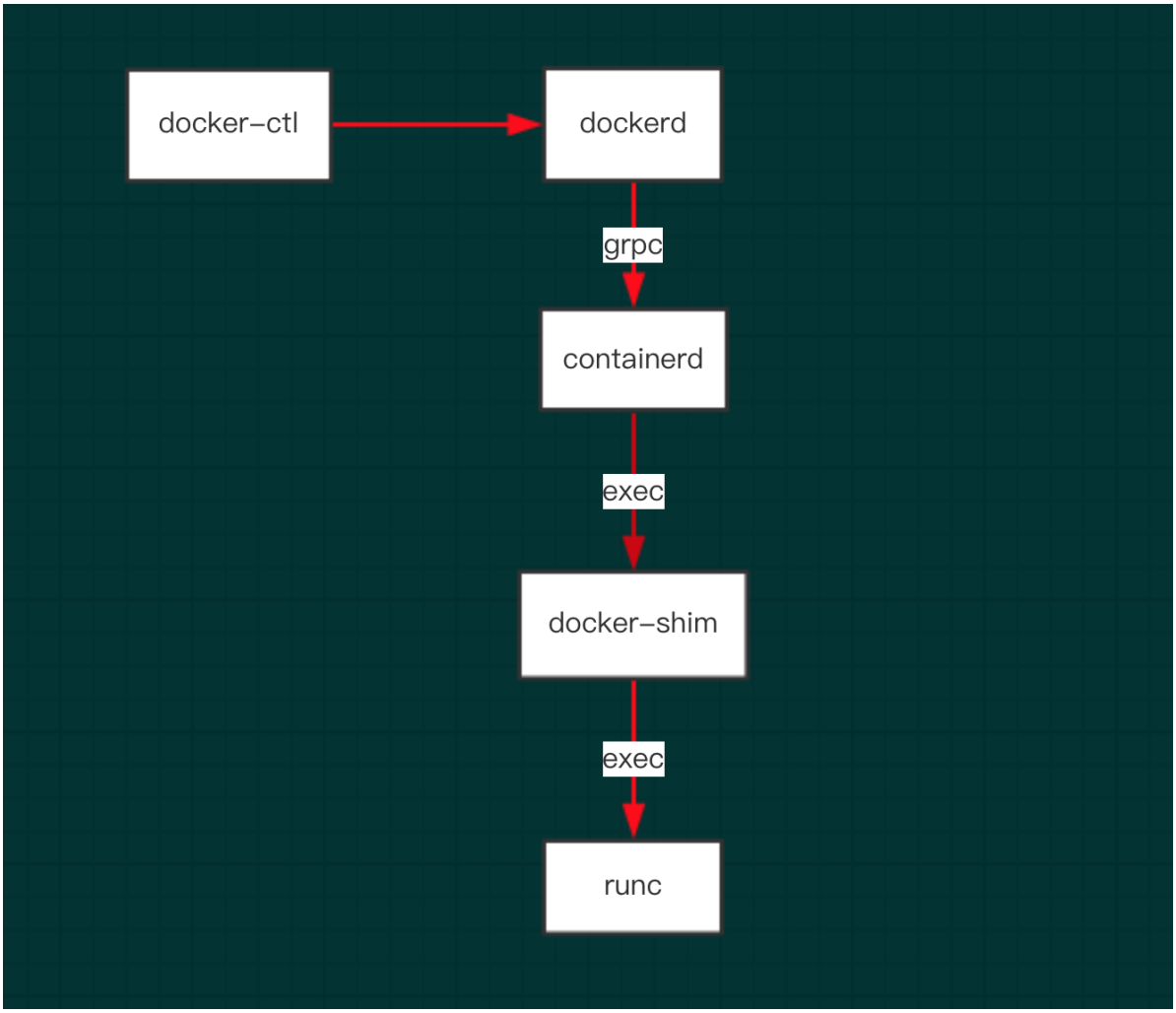
我们可以通过启动Docker Daemon时增加–add-runtime参数来选择其他的runC现。例如：

```
docker daemon --add-runtime "custom=/usr/local/bin/my-runc-replacement"
```

## Docker、containerd, containerd-shim和runc之间的关系




他们之间的关系如下图：




我们可以通过启动一个Docker容器，来观察进程之间的关联。

## 通过docker 而通过runc来启动一个container的过程

### 查看进程信息

利用`docker top`命令，可以让我们从宿主机操作系统中看到容器的进程信息。

```
[root@VM-4-17-centos ~]# docker top centos6-2
UID                 PID                 PPID                C                   STIME               TTY                 TIME                CMD
root                4962                4948                0                   16:24               pts/0               00:00:00            /usr/sbin/sshd -D


```

### 查看子进程信息

```
[root@VM-4-17-centos containerd]# ps aux | grep 4948
root      4948  0.0  0.0  12212  3696 ?        Sl   16:24   0:00 docker-containerd-shim -namespace moby -workdir /var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd -address /var/run/docker/containerd/docker-containerd.sock -containerd-binary /usr/bin/docker-containerd -runtime-root /var/run/docker/runtime-runc
root     27040  0.0  0.0 115932  1004 pts/0    S+   18:32   0:00 grep --color=auto 4948
```

### 查看进程树

```
pstree -l -a -A 4948 -p
```

输出结果如下：

```
[root@VM-4-17-centos containerd]# pstree -l -a -A 4948 -p
docker-containe,4948 -namespace moby -workdir /var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd -address /var/run/docker/containerd/docker-containerd.sock -containerd-binary /usr/bin/docker-containerd -runtime-root /var/run/docker/runtime-runc
  |-sshd,4962 -D
  |-{docker-containe},4949
  |-{docker-containe},4950
  |-{docker-containe},4951
  |-{docker-containe},4952
  |-{docker-containe},4953
  |-{docker-containe},4954
  `-{docker-containe},1593
```

虽然`pstree`命令截断了命令，但我们还是能够看出，

当Docker daemon启动之后，dockerd和docker-containerd进程一直存在。

当启动容器之后，docker-containerd进程（也是这里介绍的containerd组件）会创建docker-containerd-shim进程，其中的参数460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd就是要启动容器的id。

最后docker-containerd-shim子进程，已经是实际在容器中运行的进程（既sleep 1000）。

docker-containerd-shim另一个参数，是一个和容器相关的目录/var/run/docker/containerd/460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd，里面的内容有：

```
[root@VM-4-17-centos containerd]# ll /var/run/docker/containerd/460d688239304172f39bb9586bfc5959e0c3db64e7c3a0937f1003f94408ebbd
total 0
prwx------ 1 root root 0 Nov  3 16:24 init-stdin
prwx------ 1 root root 0 Nov  3 16:24 init-stdout

```

其中包括了容器配置和标准输入、标准输出、标准错误三个管道文件。

### docker-shim

docker-shim是一个真实运行的容器的真实垫片载体，

每启动一个容器都会起一个新的docker-shim的一个进程，

他直接通过指定的三个参数：

- 容器id，
- boundle目录（containerd的对应某个容器生成的目录，一般位于：/var/run/docker/libcontainerd/containerID），
- 运行时二进制（默认为runc）

  调用runc的api创建一个容器（比如创建容器：最后拼装的命令如下：runc create 。。。。。）

### RunC

OCI定义了容器运行时标准，

runC是Docker按照开放容器格式标准（OCF, Open Container Format）制定的一种具体实现。

runC是从Docker的libcontainer中迁移而来的，实现了容器启停、资源隔离等功能。

Docker默认提供了docker-runc实现，事实上，通过containerd的封装，可以在Docker Daemon启动的时候指定runc的实现。

## CRI

kubernetes在初期版本里，就对多个容器引擎做了兼容，因此可以使用docker、rkt对容器进行管理。

以docker为例，kubelet中会启动一个docker manager，通过直接调用docker的api进行容器的创建等操作。

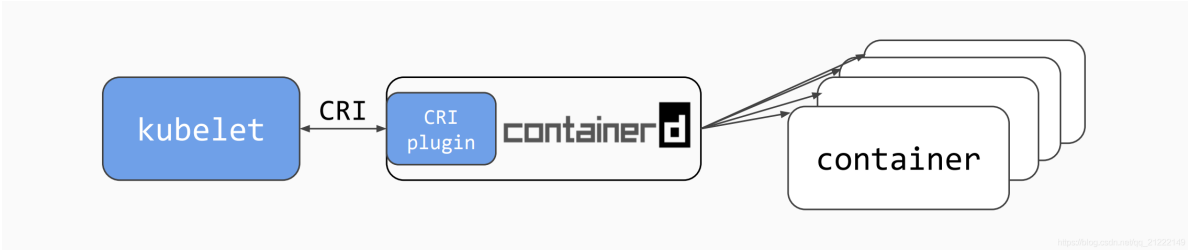
在k8s 1.5版本之后，kubernetes推出了自己的运行时接口api–CRI(container runtime interface)。cri接口的推出，隔离了各个容器引擎之间的差异，而通过统一的接口与各个容器引擎之间进行互动。

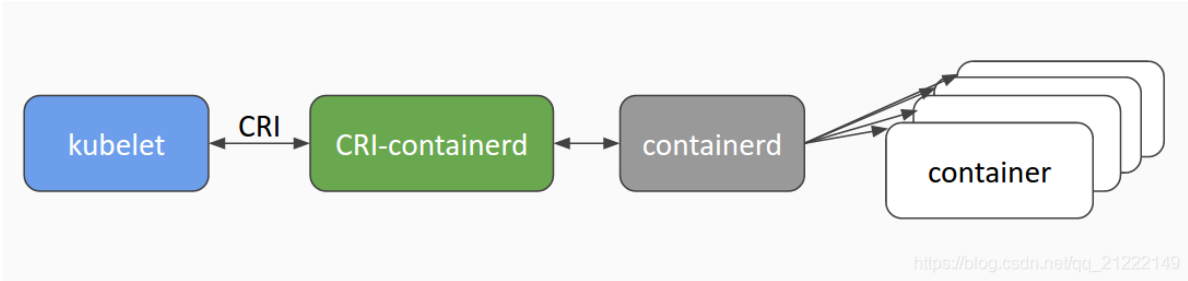
与oci不同，cri与kubernetes的概念更加贴合，并紧密绑定。

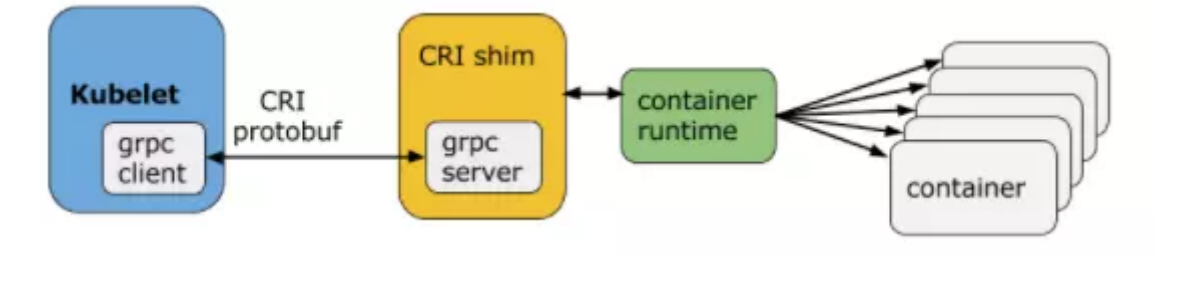
cri不仅定义了容器的生命周期的管理，还引入了k8s中pod的概念，并定义了管理pod的生命周期。

在kubernetes中，pod是由一组进行了资源限制的，在隔离环境中的容器组成。而这个隔离环境，称之为PodSandbox。在cri开始之初，主要是支持docker和rkt两种。其中kubelet是通过cri接口，调用docker-shim，并进一步调用docker api实现的。

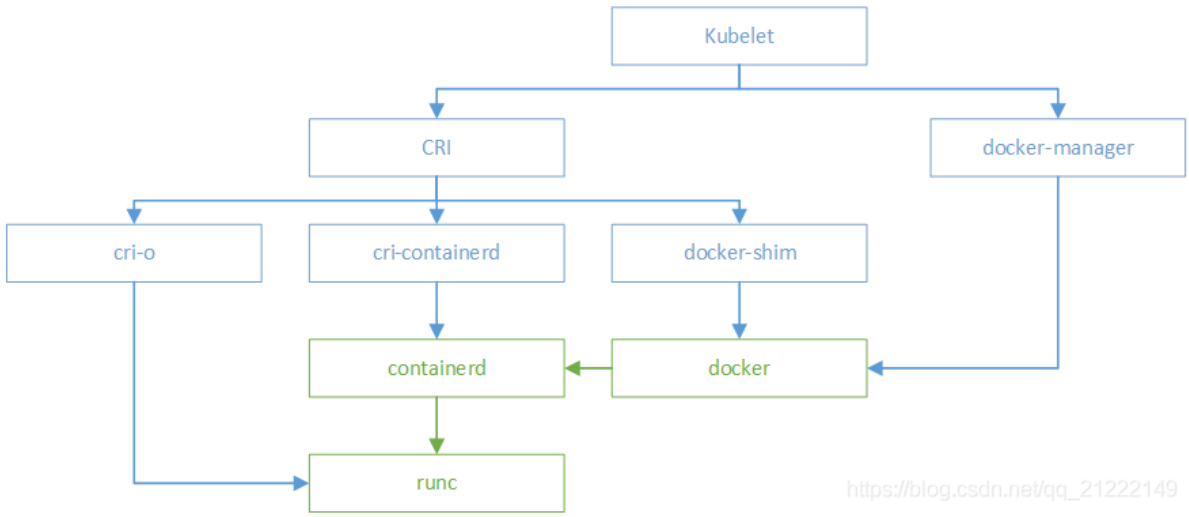
如上文所述，docker独立出来了containerd。kubernetes也顺应潮流，孵化了cri-containerd项目，用以将containerd接入到cri的标准中。




为了进一步与oci进行兼容，kubernetes还孵化了cri-o，成为了架设在cri和oci之间的一座桥梁。通过这种方式，可以方便更多符合oci标准的容器运行时，接入kubernetes进行集成使用。可以预见到，通过cri-o，kubernetes在使用的兼容性和广泛性上将会得到进一步加强。




# docker-compose

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 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。

首先检查 版本

```
[root@k8s-master ~]#  /usr/local/bin/docker-compose -version
docker-compose version 1.25.1, build a82fef07

```

如果安装好了，就ok了，如果没有安装，则安装docker

1.从github上下载docker-compose二进制文件安装

下载最新版的docker-compose文件

```
# curl -L https://github.com/docker/compose/releases/download/1.25.1/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

> 注明：离线安装包已经提供。上传后，复制到/usr/local/bin/即可

```
cp /root/docker-compose  /usr/local/bin/

```

添加可执行权限

```
# chmod +x /usr/local/bin/docker-compose
```

测试安装结果

```


[root@localhost ~]# docker-compose --version
docker-compose version 1.25.1, build a82fef07


cp /root/docker-compose  /usr/local/bin/
chmod +x /usr/local/bin/docker-compose
 docker-compose --version
```

# 总结：docker改变了什么

- 面向产品：产品交付
- 面向开发：简化环境配置
- 面向测试：多版本测试
- 面向运维：环境一致性
- 面向架构：自动化扩容（微服务）

# 聊聊：docker是怎么工作的?

实际上docker使用了常见的CS架构，也就是client-server模式，docker client负责处理用户输入的各种命令，比如docker build、docker run，真正工作的其实是server，也就是docker demon，值得注意的是，docker client和docker demon可以运行在同一台机器上。

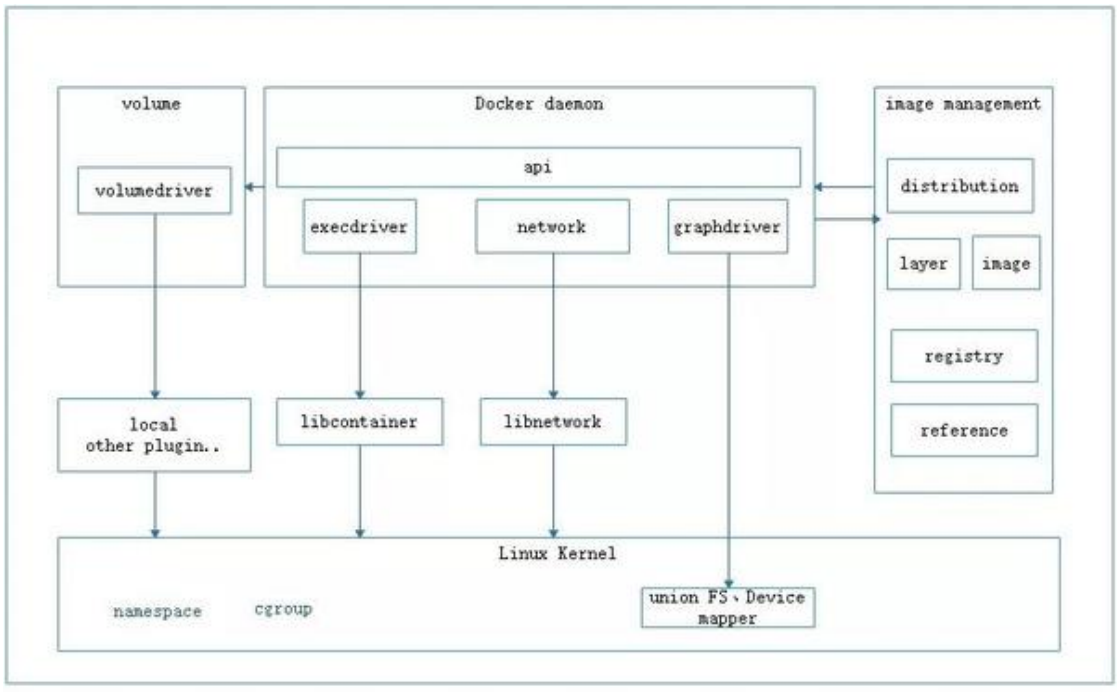
Docker是一个Client-Server结构的系统，Docker守护进程运行在主机上， 然后通过Socket连接从客户端访问，守护进程从客户端接受命令并管理运行在主机上的容器。守护进程和客户端可以运行在同一台机器上。

# 聊聊：docker的工作原理是什么，讲一下

docker是一个Client-Server结构的系统，docker守护进程运行在宿主机上，

守护进程从客户端接受命令并管理运行在主机上的容器，容器是一个运行时环境，这就是我们说的集装箱。

# 聊聊：docker架构




- distribution 负责与docker registry交互，上传洗澡镜像以及v2 registry 有关的源数据
- registry负责docker registry有关的身份认证、镜像查找、镜像验证以及管理registry mirror等交互操作
- image 负责与镜像源数据有关的存储、查找，镜像层的索引、查找以及镜像tar包有关的导入、导出操作
- reference负责存储本地所有镜像的repository和tag名，并维护与镜像id之间的映射关系
- layer模块负责与镜像层和容器层源数据有关的增删改查，并负责将镜像层的增删改查映射到实际存储镜像层文件的graphdriver模块

- graghdriver是所有与容器镜像相关操作的执行者

# 聊聊：docker的组成包含哪几大部分

一个完整的docker有以下几个部分组成：

1、docker client，客户端，为用户提供一系列可执行命令，用户用这些命令实现跟 docker daemon 交互；

2、docker daemon，守护进程，一般在宿主主机后台运行，等待接收来自客户端的请求消息；

3、docker image，镜像，镜像run之后就生成为docker容器；

4、docker container，容器，一个系统级别的服务，拥有自己的ip和系统目录结构；运行容器前需要本地存在对应的镜像，如果本地不存在该镜像则就去镜像仓库下载。

docker 使用客户端-服务器 (C/S) 架构模式，使用远程api来管理和创建docker容器。docker 容器通过 docker 镜像来创建。容器与镜像的关系类似于面向对象编程中的对象与类。

# 聊聊：docker技术的三大核心概念是什么?

**镜像：**

镜像是一种轻量级、可执行的独立软件包，它包含运行某个软件所需的所有内容，我们把应用程序和配置依赖打包好形成一个可交付的运行环境(包括代码、运行时需要的库、环境变量和配置文件等)，这个打包好的运行环境就是image镜像文件。
**容器：**

容器是基于镜像创建的，是镜像运行起来之后的一个实例，容器才是真正运行业务程序的地方。如果把镜像比作程序里面的类，那么容器就是对象。

**镜像仓库：**

存放镜像的地方，研发工程师打包好镜像之后需要把镜像上传到镜像仓库中去，然后就可以运行有仓库权限的人拉取镜像来运行容器了。

# 聊聊：基本的Docker使用流程

1. 一切都从Dockerfile开始。Dockerfile是镜像的源代码。
2. 创建Dockerfile后，您可以构建它以创建容器的镜像。镜像只是“源代码”的“编译版本”，即Dockerfile。
3. 获得容器的镜像后，应使用注册表重新分发容器。注册表就像一个git存储库 - 你可以推送和拉取镜像。
4. 接下来，您可以使用该镜像来运行容器。在许多方面，正在运行的容器与虚拟机（但没有管理程序）非常相似。

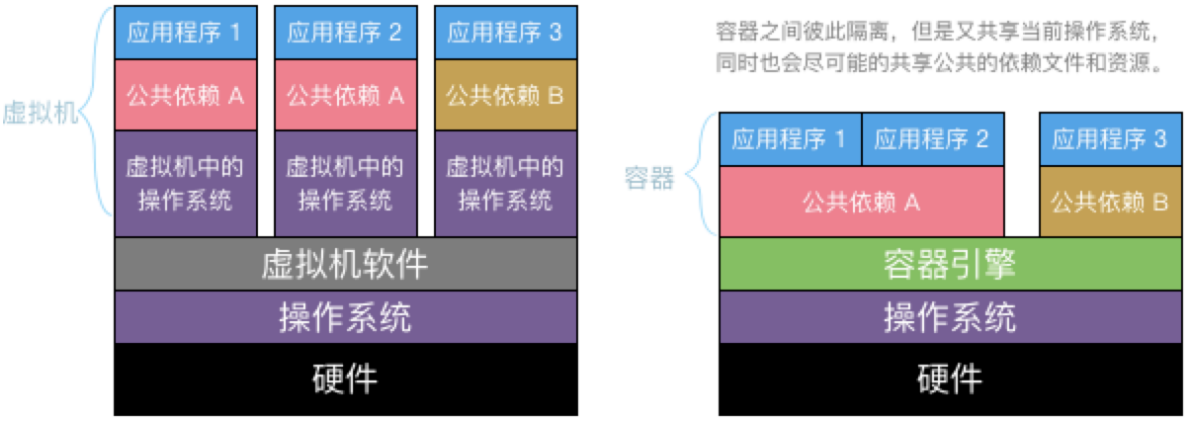
## 聊聊：Docker 安全么?

Docker 利用了 Linux 内核中很多安全特性来保证不同容器之间的隔离，并且通过签名机制来对镜像进行验证。大量生产环境的部署证明，Docker 虽然隔离性无法与虚拟机相比，但仍然具有极高的安全性。

## 聊聊：Docker 与 虚拟机 有何不同?

Docker 不是虚拟化方法。它依赖于实际实现基于容器的虚拟化或操作系统级虚拟化的其他工具。为此，Docker 最初使用 LXC 驱动程序，然后移动到libcontainer 现在重命名为 runc。Docker 主要专注于在应用程序容器内自动部署应用程序。应用程序容器旨在打包和运行单个服务，而系统容器则设计为运行多个进程，如虚拟机。因此，Docker 被视为容器化系统上的容器管理或应用程序部署工具。

- 容器不需要引导操作系统内核，因此可以在不到一秒的时间内创建容器。此功能使基于容器的虚拟化比其他虚拟化方法更加独特和可取。
- 由于基于容器的虚拟化为主机增加了很少或没有开销，因此基于容器的虚拟化具有接近本机的性能。
- 对于基于容器的虚拟化，与其他虚拟化不同，不需要其他软件。
- 主机上的所有容器共享主机的调度程序，从而节省了额外资源的需求。
- 与虚拟机映像相比，容器状态（Docker 或 LXC 映像）的大小很小，因此容器映像很容易分发。
- 容器中的资源管理是通过 cgroup 实现的。Cgroups 不允许容器消耗比分配给它们更多的资源。虽然主机的所有资源都在虚拟机中可见，但无法使用。这可以通过在容器和主机上同时运行 top 或 htop 来实现。所有环境的输出看起来都很相似。




| 特性 | Docker | 虚拟机 |
| --- | --- | --- |
| 启动速度 | 秒级 | 分钟级 |
| 交付/部署 | 开发、测试、生产环境一致 | 无成熟体系 |
| 性能 | 近似物理机 | 性能损耗大 |
| 体量 | 极小（MB） | 较大（GB） |
| 迁移/扩展 | 跨平台，可复制 | 较为复杂 |

# 聊聊：docker与传统虚拟机的区别什么?

1、传统虚拟机是需要安装整个操作系统的，然后再在上面安装业务应用，启动应用，通常需要几分钟去启动应用，而docker是直接使用镜像来运行业务容器的，其容器启动属于秒级别；

2、Docker需要的资源更少，Docker在操作系统级别进行虚拟化，Docker容器和内核交互，几乎没有性能损耗，而虚拟机运行着整个操作系统，占用物理机的资源就比较多;

3、Docker更轻量，Docker的架构可以共用一个内核与共享应用程序库，所占内存极小;同样的硬件环境，Docker运行的镜像数远多于虚拟机数量，对系统的利用率非常高;

4、与虚拟机相比，Docker隔离性更弱，Docker属于进程之间的隔离，虚拟机可实现系统级别隔离;

5、Docker的安全性也更弱，Docker的租户root和宿主机root相同，一旦容器内的用户从普通用户权限提升为root权限，它就直接具备了宿主机的root权限，进而可进行无限制的操作。虚拟机租户root权限和宿主机的root虚拟机权限是分离的，并且虚拟机利用如Intel的VT-d和VT-x的ring-1硬件隔离技术，这种技术可以防止虚拟机突破和彼此交互，而容器至今还没有任何形式的硬件隔离;

6、Docker的集中化管理工具还不算成熟，各种虚拟化技术都有成熟的管理工具，比如：VMware vCenter提供完备的虚拟机管理能力;

7、Docker对业务的高可用支持是通过快速重新部署实现的，虚拟化具备负载均衡，高可用、容错、迁移和数据保护等经过生产实践检验的成熟保障机制，Vmware可承诺虚拟机99.999%高可用，保证业务连续性;

8、虚拟化创建是分钟级别的，Docker容器创建是秒级别的，Docker的快速迭代性，决定了无论是开发、测试、部署都可以节省大量时间;

9、虚拟机可以通过镜像实现环境交付的一致性，但镜像分发无法体系化，Docker在Dockerfile中记录了容器构建过程，可在集群中实现快速分发和快速部署。

# 聊聊：Docker与LXC ( Linux Container)有何不同?

LXC利用Linux上相关技术实现了容器支持； Docker早期版本中使用了LXC技术，后期演化为新的libcontainer, 在如下的几个方面进行了改进：

- 移植性： 通过抽象容器配置， 容器可以实现从一个平台移植到另一个平台；
- 镜像系统： 基于AUFS的镜像系统为容器的分发带来了很多的便利， 同时共同的镜像层只需要存储一份，实现高效率的存储；
- 版本管理： 类似于Git的版本管理理念， 用户可以更方便地创建、 管理镜像文件；
- 仓库系统： 仓库系统大大降低了镜像的分发和管理的成本；
- 周边工具： 各种现有工具（配置管理、 云平台）对Docker的支持， 以及基于Docker 的PaaS、CI等系统， 让Docker的应用更加方便和多样化。

# 聊聊：什么是 Docker 镜像?

Docker 镜像是 Docker 容器的源代码，Docker 镜像用于创建容器。使用build 命令创建镜像。

## 聊聊：什么是 Docker 容器?

Docker 容器包括应用程序及其所有依赖项，作为操作系统的独立进程运行。

## 聊聊：Docker 容器有几种状态?

四种状态：运行、已暂停、重新启动、已退出。

## 聊聊：Dockerfile 中最常见的指令是什么?

- FROM：指定基础镜像
- LABEL：功能是为镜像指定标签
- RUN：运行指定的命令
- CMD：容器启动时要运行的命令

## 聊聊：Dockerfile 中的命令 COPY 和 ADD 命令有什么区别?

COPY 与 ADD 的区别 COPY 的 SRC 只能是本地文件，其他用法一致。 8. 解释一下 Dockerfile 的 ONBUILD 指令? 当镜像用作另一个镜像构建的基础时，ONBUILD 指令向镜像添加将在稍后执行的触发指令。如果要构建将用作构建其他镜像的基础的镜像（例如，可以使用特定于用户的配置自定义的应用程序构建环境或守护程序），这将非常有用。

## 聊聊：docker常用命令?

- docker pull 拉取或者更新指定镜像
- docker push 将镜像推送至远程仓库
- docker rm 删除容器
- docker rmi 删除镜像
- docker images 列出所有镜像
- docker ps 列出所有容器

**1、如何列出可运行的容器?**
docker ps

**2、启动nginx容器（随机端口映射），并挂载本地文件目录到容器html的命令是?**
docker run -d -P --name nginx2 -v /home/nginx:/usr/share/nginx/html nginx

**3、进入容器的方法有哪些?**
1、使用 docker attach 命令
2、使用 exec 命令，例如docker exec -i -t 784fd3b294d7 /bin/bash

**4、容器与主机之间的数据拷贝命令是?**
docker cp 命令用于容器与主机之间的数据拷贝
主机到容器：
docker cp /www 96f7f14e99ab:/www/
容器到主机：
docker cp 96f7f14e99ab:/www /tmp/

**5、当启动容器的时候提示：exec format error？如何解决问题**
检查启动命令是否有可执行权限，进入容器手工运行脚本进行排查。

**6、本地的镜像文件都存放在哪里?**
与 Docker 相关的本地资源都存放在/var/lib/docker/目录下，其中 container 目录存放容器信息，graph 目录存放镜像信息，aufs 目录下存放具体的内容文件。

**7、如何退出一个镜像的 bash，而不终止它?**
按 Ctrl-p Ctrl-q。

**8、退出容器时候自动删除?**
使用 –rm 选项，例如 sudo docker run –rm -it ubuntu

**9、如何批量清理临时镜像文件?**
可以使用 sudo docker rmi $(sudo docker images -q -f danging=true)命令

**10、如何查看镜像支持的环境变量?**
使用 sudo docker run IMAGE env

**11、本地的镜像文件都存放在哪里**
于 Docker 相关的本地资源存放在/var/lib/docker/目录下，其中 container 目录存放容器信息，graph 目录存放镜像信息，aufs 目录下存放具体的镜像底层文件。

**12、容器退出后，通过 docker ps 命令查看不到，数据会丢失么?**
容器退出后会处于终止（exited）状态，此时可以通过 docker ps -a 查看，其中数据不会丢失，还可以通过 docker start 来启动，只有删除容器才会清除数据。

**13、如何停止所有正在运行的容器?**
使用 docker kill $(sudo docker ps -q)

**14、如何清理批量后台停止的容器?**
答：使用 docker rm $（sudo docker ps -a -q）

**15、如何临时退出一个正在交互的容器的终端，而不终止它?**
按 Ctrl+p，后按 Ctrl+q，如果按 Ctrl+c 会使容器内的应用进程终止，进而会使容器终止。

# 说说: Dockerfile的基本指令有哪些?

FROM 指定基础镜像（必须为第一个指令，因为需要指定使用哪个基础镜像来构建镜像）；

MAINTAINER 设置镜像作者相关信息，如作者名字，日期，邮件，联系方式等；

COPY 复制文件到镜像；

ADD 复制文件到镜像（ADD与COPY的区别在于，ADD会自动解压tar、zip、tgz、xz等归档文件，而COPY不会，同时ADD指令还可以接一个url下载文件地址，一般建议使用COPY复制文件即可，文件在宿主机上是什么样子复制到镜像里面就是什么样子这样比较好）；

ENV 设置环境变量；

EXPOSE 暴露容器进程的端口，仅仅是提示别人容器使用的哪个端口，没有过多作用；

VOLUME 数据卷持久化，挂载一个目录；

WORKDIR 设置工作目录，如果目录不在，则会自动创建目录；

RUN 在容器中运行命令，RUN指令会创建新的镜像层，RUN指令经常被用于安装软件包；

CMD 指定容器启动时默认运行哪些命令，如果有多个CMD，则只有最后一个生效，另外，CMD指令可以被docker run之后的参数替换；

ENTRYOINT 指定容器启动时运行哪些命令，如果有多个ENTRYOINT，则只有最后一个生效，另外，如果

Dockerfile中同时存在CMD和ENTRYOINT，那么CMD或docker run之后的参数将被当做参数传递给ENTRYOINT；

# 说说: 如何进入容器？使用哪个命令

进入容器有两种方法：docker attach、docker exec；
docker attach命令是attach到容器启动命令的终端，docker exec 是另外在容器里面启动一个TTY终端。

```
docker run -d centos /bin/bash -c "while true;do sleep 2;echo I_am_a_container;done"
3274412d88ca4f1d1292f6d28d46f39c14c733da5a4085c11c6a854d30d1cde0
docker attach 3274412d88ca4f                              #attach进入容器
Ctrl + c  退出，Ctrl + c会直接关闭容器终端，这样容器没有进程一直在前台运行就会死掉了
Ctrl + pq 退出（不会关闭容器终端停止容器，仅退出）

docker exec -it 3274412d88ca /bin/bash                    #exec进入容器
[root@3274412d88ca /]# ps -ef                             #进入到容器了开启了一个bash进程
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 05:31 ?        00:00:01 /bin/bash -c while true;do sleep 2;echo I_am_a_container;done
root         306       0  1 05:41 pts/0    00:00:00 /bin/bash
root         322       1  0 05:41 ?        00:00:00 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 2
root         323     306  0 05:41 pts/0    00:00:00 ps -ef
[root@3274412d88ca /]#exit                                #退出容器，仅退出我们自己的bash窗口

```

小结：

attach是直接进入容器启动命令的终端，不会启动新的进程；

exec则是在容器里面打开新的终端，会启动新的进程；一般建议已经exec进入容器。

# 聊聊：什么是 Docker Swarm?

Docker Swarm 是 Docker 的本机群集。它将 Docker 主机池转变为单个虚拟Docker 主机。Docker Swarm 提供标准的 Docker API，任何已经与 Docker守护进程通信的工具都可以使用 Swarm 透明地扩展到多个主机。

# 聊聊：容器内部机制?

每个容器都在自己的命名空间中运行，但使用与所有其他容器完全相同的内核。发生隔离是因为内核知道分配给进程的命名空间，并且在API调用期间确保进程只能访问其自己的命名空间中的资源。

## 聊聊：什么是Docker Hub?

Docker hub是一个基于云的注册表服务，允许您链接到代码存储库，构建镜像并测试它们，存储手动推送的镜像以及指向Docker云的链接，以便您可以将镜像部署到主机。它为整个开发流程中的容器镜像发现，分发和变更管理，用户和团队协作以及工作流自动化提供了集中资源。

## 聊聊：镜像与 UnionFS

Linux 的命名空间和控制组分别解决了不同资源隔离的问题，前者解决了进程、网络以及文件系统的隔离，后者实现了 CPU、内存等资源的隔离，但是在 Docker 中还有另一个非常重要的问题需要解决 - 也就是镜像。

Docker 镜像其实本质就是一个压缩包，我们可以使用命令将一个 Docker 镜像中的文件导出，你可以看到这个镜像中的目录结构与 Linux 操作系统的根目录中的内容并没有太多的区别，可以说 Docker 镜像就是一个文件。

## 存储驱动

Docker 使用了一系列不同的存储驱动管理镜像内的文件系统并运行容器，这些存储驱动与

Docker 卷（volume）有些不同，存储引擎管理着能够在多个容器之间共享的存储。

## 聊聊：docker容器之间怎么隔离?

Linux中的PID、IPC、网络等资源是全局的，而NameSpace机制是一种资源隔离方案，在该机制下这些资源就不再是全局的了，而是属于某个特定的NameSpace，各个NameSpace下的资源互不干扰。

虽然有了NameSpace技术可以实现资源隔离，但进程还是可以不受控的访问系统资源，比如CPU、内存、磁盘、网络等，为了控制容器中进程对资源的访问，Docker采用control groups技术(也就是cgroup)，有了cgroup就可以控制容器中进程对系统资源的消耗了，比如你可以限制某个容器使用内存的上限、可以在哪些CPU上运行等等。

有了这两项技术，容器看起来就真的像是独立的操作系统了。

## 聊聊：仓库( Repository)、 注册服务器( Registry)、 注册索引( Index)有何关系?

仓库是存放一组关联镜像的集合，比如同一个应用的不同版本的镜像。注册服务器是存放实际的镜像文件的地方。注册索引则负责维护用户的账号、权限、搜索、标签等的管理。因此，注册服务器利用注册索引来实现认证等管理。

## 聊聊：如何在生产中监控 Docker?

Docker 提供 docker stats 和 docker 事件等工具来监控生产中的 Docker。

我们可以使用这些命令获取重要统计数据的报告。

Docker 统计数据：当我们使用容器 ID 调用 docker stats 时，我们获得容器的CPU，内存使用情况等。它类似于 Linux 中的 top 命令。

Docker 事件：Docker 事件是一个命令，用于查看 Docker 守护程序中正在进行的活动流。 一些常见的 Docker 事件：attach，commit，die，detach，rename，destroy 等。我们还可以使用各种选项来限制或过滤我们感兴趣的事件。

## 聊聊：如何在多个环境中使用Docker?

可以进行以下更改：

- 删除应用程序代码的任何卷绑定，以便代码保留在容器内，不能从外部更改
- 绑定到主机上的不同端口
- 以不同方式设置环境变量（例如，减少日志记录的详细程度，或启用电子邮件发送）
- 指定重启策略（例如，重启：始终）以避免停机
- 添加额外服务（例如，日志聚合器）

因此，您可能希望定义一个额外的Compose文件，例如production.yml，它指定适合生产的配置。此配置文件只需要包含您要从原始Compose文件中进行的更改。

## 聊聊：Docker能在非Linux平台（比如`macOS`或`Windows`)上运行么?

可以。

macOS目前需要使用 docker for mac等软件创建一个轻量级的Linux虚拟机层。 由于成熟度不高，暂时不推荐在Windows环境中使用Docker。

# 说说: centos镜像几个G，但是docker centos镜像才几百兆，这是为什么?

一个完整的Linux操作系统包含Linux内核和rootfs根文件系统，即我们熟悉的/dev、/proc/、/bin等目录。

我们平时看到的centOS除了rootfs，还会选装很多软件，服务，图形桌面等，所以centOS镜像有好几个G也不足为奇。
而对于容器镜像而言，所有容器都是共享宿主机的Linux 内核的，

而对于docker镜像而言，docker镜像只需要提供一个很小的rootfs根文件系统即可，只需要包含即我们熟悉的/dev、/proc/、/bin等目录，这是最基本的命令，工具，程序库即可，

所以，docker镜像才会这么小。

# 说说: 镜像的分层结构以及为什么要使用镜像的分层结构?

一个新的镜像其实是从 base 镜像一层一层叠加生成的。

每安装一个软件，dockerfile中使用RUM命令，就会在现有镜像的基础上增加一层，这样一层一层的叠加最后构成整个镜像。

所以我们docker pull拉取一个镜像的时候会看到docker是一层层拉去的。

分层机构最大的一个好处就是： 共享资源。

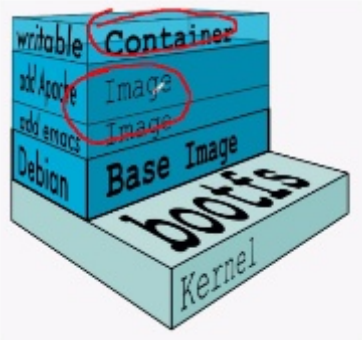
比如：有多个镜像都从相同的 base 镜像构建而来，那么 Docker Host 只需在磁盘上保存一份 base 镜像；

同时内存中也只需加载一份 base 镜像，就可以为所有容器服务了。而且镜像的每一层都可以被共享。

## 说说: 容器的copy-on-write特性，修改容器里面的内容会修改镜像吗?

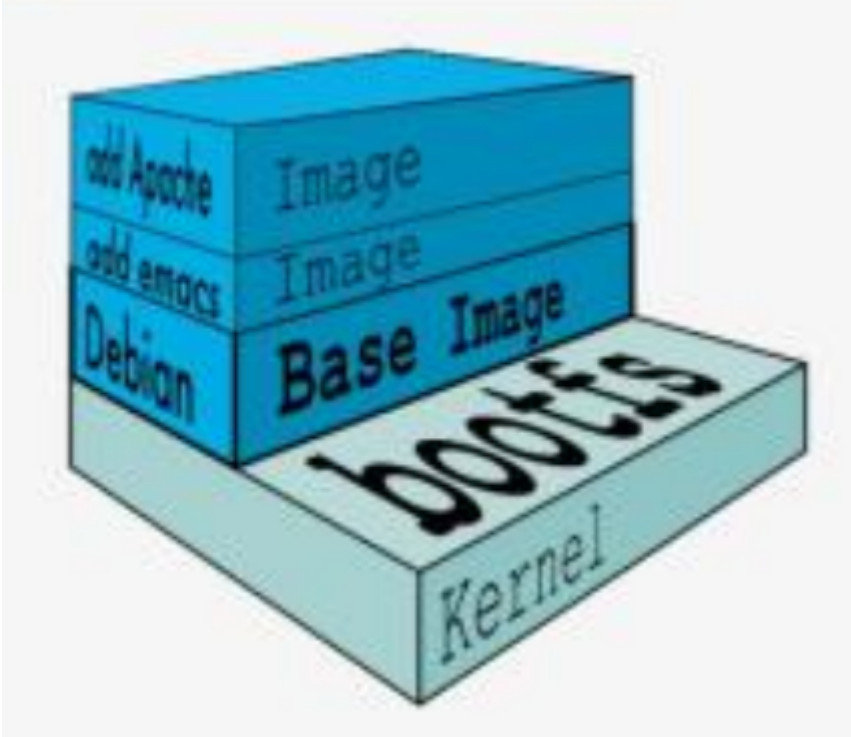
我们知道，镜像是分层的，镜像的每一层都可以被共享，同时，镜像是只读的。

当一个容器启动时，一个新的可写层被加载到镜像的顶部，这一层通常被称作“容器层”，“容器层”之下的都叫“镜像层”。




实际上，docker hub中99%的镜像都是通过在base镜像中安装和配置需要的软件构建出来的

新的镜像是从base镜像一层一层叠加生成的，每安装一个软件，就在现有的基础增加一层




为什么docker镜像要采用这种分层结构呢?

最大的一个好处是：共享资源

比如:有多个镜像都从相同的base镜像构建而来，那么docker host只需在磁盘上保存一份base镜像：同时内存中也只需加载一份base镜像，就可以为所有容器服务了，而其镜像的每一层都可以被共享

问题是：如果多个容器共享一份基础镜像，当某个容器修改了基础镜像的内容，比如/etc下的文件，这时其他容器的/etc是否也会被修改？？？ 答案是不会

修改会被限制在单个容器内, 这就是 容器copy-on-write特性

所有对容器的改动 - 无论添加、删除、还是修改文件，都只会发生在容器层中，

因为只有容器层是可写的，容器层下面的所有镜像层都是只读的。

镜像层数量可能会很多，所有镜像层会联合在一起组成一个统一的文件系统。

如果不同层中有一个相同路径的文件，比如 /a，上层的 /a 会覆盖下层的 /a，也就是说用户只能访问到上层中的文件 /a。

在容器层中，用户看到的是一个叠加之后的文件系统。

添加文件时：在容器中创建文件时，新文件被添加到容器层中。
读取文件：在容器中读取某个文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，立即将其复制到容器层，然后打开并读入内存。
修改文件：在容器中修改已存在的文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，立即将其复制到容器层，然后修改之。
删除文件：在容器中删除文件时，Docker 也是从上往下依次在镜像层中查找此文件。找到后，会在容器层中记录下此删除操作。

只有当需要修改时才复制一份数据，这种特性被称作 Copy-on-Write。可见，容器层保存的是镜像变化的部分，不会对镜像本身进行任何修改。

# 说说: Dockerfile的整个构建镜像过程

1、首先，创建一个目录用于存放应用程序以及构建过程中使用到的各个文件等；

2、然后，在这个目录下创建一个Dockerfile文件，一般建议Dockerfile的文件名就是Dockerfile；

3、编写Dockerfile文件，编写指令，如，使用FORM指令指定基础镜像，COPY指令复制文件，RUN指令指定要运行的命令，ENV设置环境变量，EXPOSE指定容器要暴露的端口，WORKDIR设置当前工作目录，CMD容器启动时运行命令，等等指令构建镜像；

4、Dockerfile编写完成就可以构建镜像了，使用`docker build -t 镜像名:tag .`命令来构建镜像，最后一个点是表示当前目录，docker会默认寻找当前目录下的Dockerfile文件来构建镜像，如果不使用默认，可以使用-f参数来指定dockerfile文件，如：`docker build -t 镜像名:tag -f /xx/xxx/Dockerfile`；

5、使用docker build命令构建之后，docker就会将当前目录下所有的文件发送给docker daemon，顺序执行Dockerfile文件里的指令，在这过程中会生成临时容器，在临时容器里面安装RUN指定的命令，安装成功后，docker底层会使用类似于docker commit命令来将容器保存为镜像，然后删除临时容器，以此类推，一层层的构建镜像，运行临时容器安装软件，直到最后的镜像构建成功。

# 说说: Dockerfile构建镜像出现异常，如何排查?

首先，Dockerfile是一层一层的构建镜像，期间会产生一个或多个临时容器，

构建过程中其实就是在临时容器里面安装应用，如果因为临时容器安装应用出现异常导致镜像构建失败，这时容器虽然被清理掉了，但是期间构建的中间镜像还在，

那么我们可以根据异常时上一层已经构建好的临时镜像，将临时镜像运行为容器，然后在容器里面运行安装命令来定位具体的异常。

# 极兔一面：Dockerfile如何优化？注意：千万不要只说减少层数

在40岁老架构师 尼恩的**读者交流群(50+)**中，面试题是一个非常、非常高频的交流话题。

最近，有小伙伴面试极兔时，遇到一个面试题：

> 如果优化 Dockerfile?

小伙伴没有回答好，只是提到了减少镜像层数。

一般来说，面试的小伙伴，大部分都会说

1. 使用更小的基础镜像， 比如 `alpine`.
2. 减少镜像层数， 比如 使用 `&&` 符号将命令链接起来。
3. 给基础镜像打上 **安全补丁。**

但这些，其实都是单点的优化。优化 Dockerfile 的核心是 **合理分层、构建一个精良的基础镜像。**

这里尼恩给大家做一下系统化、体系化的梳理，使得大家可以充分展示一下大家雄厚的 “技术肌肉”，**让面试官爱到 “不能自已、口水直流”。**

也一并把这个题目以及参考答案，收入咱们的 《尼恩Java面试宝典》V46版本，供后面的小伙伴参考，提升大家的 3高 架构、设计、开发水平。

## 为什么要优化镜像

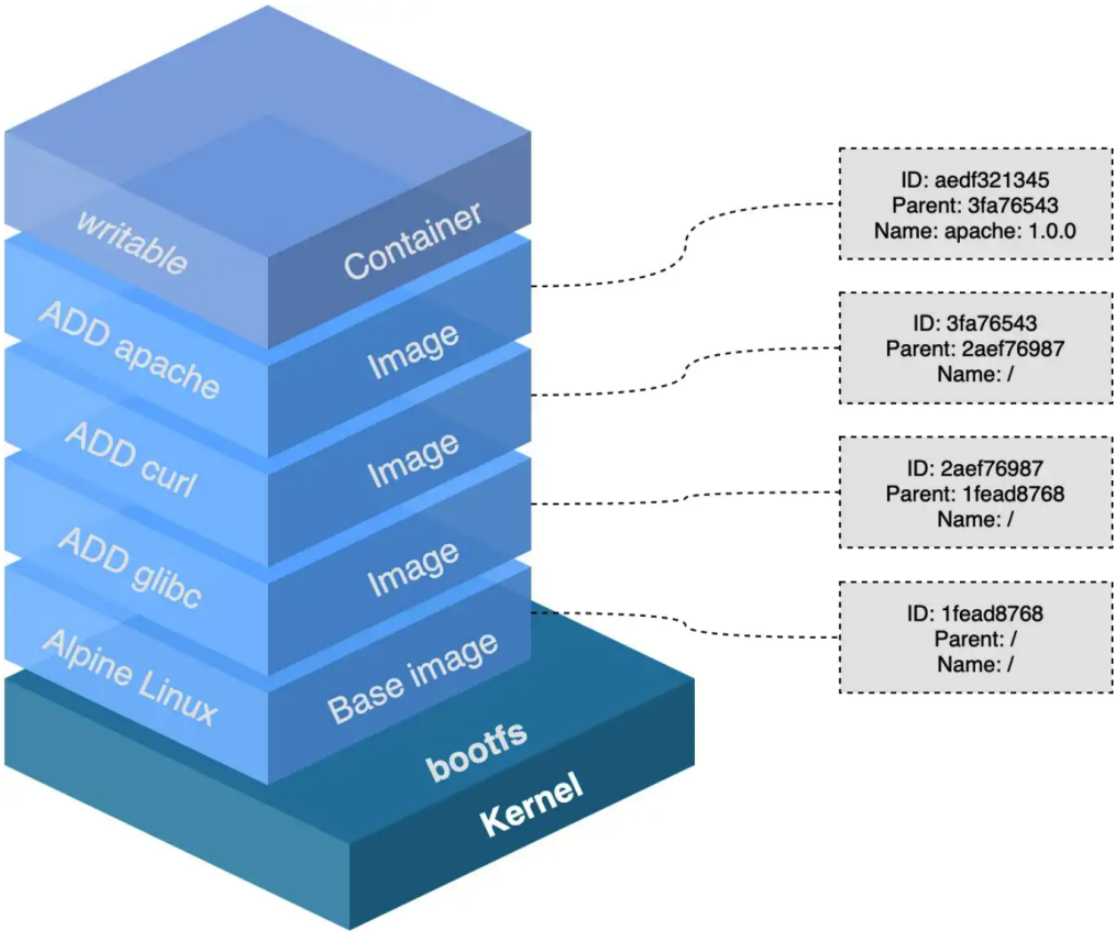
首先，回到起点。为啥要优化 镜像？优化镜像的好处是：

- **一个小镜像有什么好处**: 分发更快，存储更少，加载更快。
- **镜像臃肿带来了什么问题**: 存储过多，分发更慢且浪费带宽更多。

### 镜像的构成

其次，来看看镜像的构成。从两个维度来看：

- **俯瞰镜像**: 就是一个删减版的操作系统。
- **侧看镜像**: 由一层层的 `layer` 堆叠而成




Docker image layer

## 那么问题来了

应该如何优化镜像?

## 举个例子 docker build

- **Dockerfile v1**

```
# v1
FROM nginx:1.15-alpine
RUN echo "hello"
RUN echo "demo best practise"
ENTRYPOINT [ "/bin/sh" ]
```

- **Dockerfile v2**

```
# v2
FROM nginx:1.15-alpine
RUN echo "hello"
RUN echo "demo best practise"
ENTRYPOINT [ "/bin/sh" ]
```

## 1st build

全新构建

```
# docker build -t demo:0.0.1 .
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM nginx:1.15-alpine
 ---> 9a2868cac230
Step 2/4 : RUN echo "hello"
 ---> Running in d301b4b3ed55
hello
Removing intermediate container d301b4b3ed55
 ---> 6dd2a7773bbc
Step 3/4 : RUN echo "demo best practise"
 ---> Running in e3084037668e
demo best practise
Removing intermediate container e3084037668e
 ---> 4588ecf9837a
Step 4/4 : ENTRYPOINT [ "/bin/sh" ]
 ---> Running in d63f460347ff
Removing intermediate container d63f460347ff
 ---> 77b52d828f21
Successfully built 77b52d828f21
Successfully tagged demo:0.0.1
```

## 2nd build

Dockerfile 与 `1st build` 完全一致， 命令仅修改 build tag , 从 `0.0.1` 到 `0.0.2`

```
# docker build -t demo:0.0.2 .
Sending build context to Docker daemon  4.096kB
Step 1/4 : FROM nginx:1.15-alpine
 ---> 9a2868cac230
Step 2/4 : RUN echo "hello"
 ---> Using cache
 ---> 6dd2a7773bbc
Step 3/4 : RUN echo "demo best practise"
 ---> Using cache
 ---> 4588ecf9837a
Step 4/4 : ENTRYPOINT [ "/bin/sh" ]
 ---> Using cache
 ---> 77b52d828f21
Successfully built 77b52d828f21
Successfully tagged demo:0.0.2
```

可以看到，

1. 每层 layer 都使用 cache ( `---> Using cache` )， 并未重新构建。
2. 我们可以通过 `docker image ls |grep demo` 看到， `demo:0.0.1` 与 `demo:0.0.2` 的 layer hash 是相同。

所以从根本上来说， 这两个镜像就是同一个镜像，虽然都是 build 出来的。

### 3rd build

这次， 我们将Dockerfile 02的 第三层 `RUN echo "demo best practise"` 变更为 `RUN echo "demo best practise 02"`

```
docker build -t demo:0.0.3 .
Sending build context to Docker daemon  4.608kB
Step 1/4 : FROM nginx:1.15-alpine
 ---> 9a2868cac230
Step 2/4 : RUN echo "hello"
 ---> Using cache
 ---> 6dd2a7773bbc
Step 3/4 : RUN echo "demo best practise 02"
 ---> Running in c55f94e217bd
demo best practise 02
Removing intermediate container c55f94e217bd
 ---> 46992ea04f49
Step 4/4 : ENTRYPOINT [ "/bin/sh" ]
 ---> Running in f176830cf445
Removing intermediate container f176830cf445
 ---> 2e2043b7f3cb
Successfully built 2e2043b7f3cb
Successfully tagged demo:0.0.3
```

可以看到，

1. 第二层仍然使用 `cache`
2. 但是第三层已经生成了新的 hash 了
3. 虽然第四层的操作没有变更，但是由于上层的镜像已经变化了，所以第四层本身也发生了变化。

> 注意: 每层在 `build` 的时候都是依赖于上册 `---> Running in f176830cf445`。

### 4th build

第四次构建， 这次使用 `--no-cache` 不使用缓存， 模拟在另一台电脑上进行 build 。

```
# docker build -t demo:0.0.4 --no-cache .
Sending build context to Docker daemon  5.632kB
Step 1/4 : FROM nginx:1.15-alpine
 ---> 9a2868cac230
Step 2/4 : RUN echo "hello"
 ---> Running in 7ecbed95c4cd
hello
Removing intermediate container 7ecbed95c4cd
 ---> a1c998781f2e
Step 3/4 : RUN echo "demo best practise 02"
 ---> Running in e90dae9440c2
demo best practise 02
Removing intermediate container e90dae9440c2
 ---> 09bf3b4238b8
Step 4/4 : ENTRYPOINT [ "/bin/sh" ]
 ---> Running in 2ec19670cb14
Removing intermediate container 2ec19670cb14
 ---> 9a552fa08f73
Successfully built 9a552fa08f73
Successfully tagged demo:0.0.4
```

可以看到，

1. 虽然和 `3rd build` 使用的 `Dockerfile` 相同， 但由于没有缓存，每一层都是重新 build 的。
2. 虽然 `demo:0.0.3` 和 `demo:0.0.4` 在功能上是一致的。但是 **他们的 layer 不同， 从根本上来说，他们是不同的镜像。**

## 结论

### 1. 合理分层、构建一个精良的基础镜像

1. 一个相对固定的 `build` 环境
2. 善用 `cache`
3. 构建 `自己的基础镜像`：其中就包括了

   a. 安全补丁
   b. 权限限制
   c. 基础库依赖安装
   d. 等...

### 2. 精简为美：一屋不扫何以扫天下

1. 使用 `.dockerignore` 保持 `context` 干净
2. 容器镜像环境清理
   a. 缓存清理
   b. `multi stage build`

尼恩提示：以上答案，所包含的技术细节比较多，具体请参见《尼恩*Java*面试宝典》最新版。

# 参考文献：


https://blog.csdn.net/qq_21222149/article/details/89201744

https://blog.csdn.net/warrior_0319/article/details/80073720
http://www.sel.zju.edu.cn/?p=840
http://alexander.holbreich.org/docker-components-explained/
https://www.cnblogs.com/sparkdev/p/9129334.htmls

- `docker storage driver`: https://docs.docker.com/storage/storagedriver/
- `dockerfile best practices`: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- `multi-stage`: https://docs.docker.com/develop/develop-images/multistage-build/

# 免费领取11个技术圣经PDF

## 技术自由圈 7个 学习圣经 PDF

1. 《SpringCloud Alibaba 学习圣经》 377页 PDF
   v1 版，发布日期：2023年03月15日
2. 《Docker 学习圣经》 145页 PDF
   v2 版，发布日期：2023年03月12日
3. 《Kubernets 学习圣经》 530页 PDF
   v1 版，发布日期：2023年04月11日
4. 《响应式圣经》 114页 PDF
   v2 版，发布日期：2023年02月14日
5. 《缓存之王 Caffeine 红宝书》 175页 PDF
   v2 版，发布日期：2022年11月30日
6. 《队列之王 Disruptor 红宝书》 75页 PDF
   v1 版，发布日期：2022年10月05日
7. 《Prometheus+grafna 红宝书》 100页 PDF
   v1 版，发布日期：2022年10月08日

## 技术自由圈 3个 高并发圣经 PDF

8. 《Java高并发核心编程 卷1 加强版：NIO、Netty、Redis、ZooKeeper》 609页 PDF
   加强版，发布日期：2023年01月01日
9. 《Java高并发核心编程 卷2 加强版：多线程、锁、JMM、JUC、高并发设计模式》 448页 PDF
   加强版，发布日期：2022年11月14日
10. 《Java高并发核心编程 卷3 加强版：亿级用户Web应用架构与实战》 481页 PDF
    加强版，发布日期：2022年11月14日

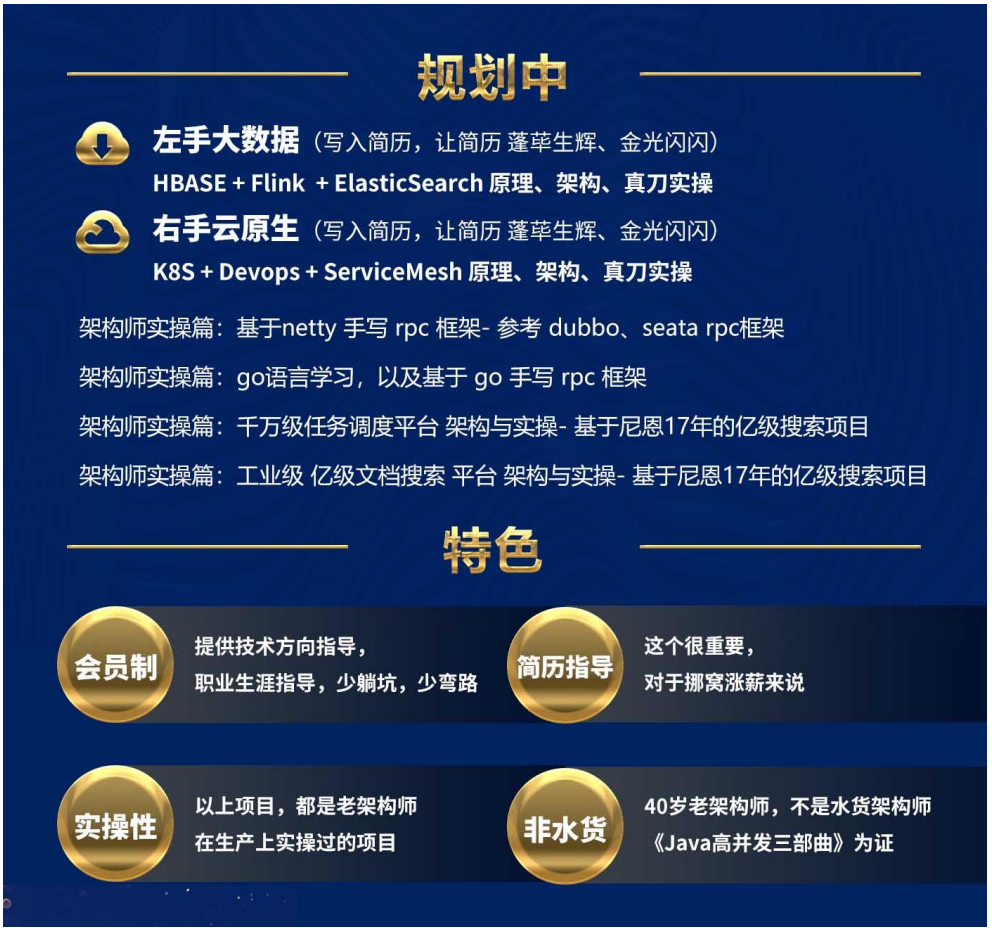
## 技术自由圈 面试圣经 40个 面试题 PDF

11. 《尼恩 Java 面试宝典》40个专题 4000页 PDF
    v60 版，发布日期：2023年03月11日

领取方式： 技术自由圈 公众号

# 硬核推荐：尼恩Java硬核架构班

详情：https://www.cnblogs.com/crazymakercircle/p/9904544.html


规划中
左手大数据（写入简历，让简历 蓬荜生辉、金光闪闪）
HBASE + Flink + ElasticSearch 原理、架构、真刀实操
右手云原生（写入简历，让简历 蓬荜生辉、金光闪闪）
K8S + Devops + ServiceMesh 原理、架构、真刀实操
架构师实操篇：基于netty 手写 rpc 框架- 参考 dubbo、seata rpc框架
架构师实操篇：go语言学习，以及基于 go 手写 rpc 框架
架构师实操篇：千万级任务调度平台 架构与实操- 基于尼恩17年的亿级搜索项目
架构师实操篇：工业级 亿级文档搜索 平台 架构与实操- 基于尼恩17年的亿级搜索项目
特色
会员制
提供技术方向指导，
职业生涯指导，少躺坑，少弯路
简历指导
这个很重要，
对于挪窝涨薪来说
实操性
以上项目，都是老架构师
在生产上实操过的项目
非水货
40岁老架构师，不是水货架构师
《Java高并发三部曲》为证

# 架构班（社群 VIP）的起源：

最初的视频，主要是给读者加餐。很多的读者，需要一些高质量的实操、理论视频，所以，我就围绕书，和底层，做了几个实操、理论视频，然后效果还不错，后面就做成迭代模式了。

# 架构班（社群 VIP）的功能：

提供高质量实操项目整刀真枪的架构指导、快速提升大家的：

- 开发水平
- 设计水平
- 架构水平

弥补业务中 CRUD 开发短板，帮助大家尽早脱离具备 3 高能力，掌握：

- 高性能
- 高并发
- 高可用

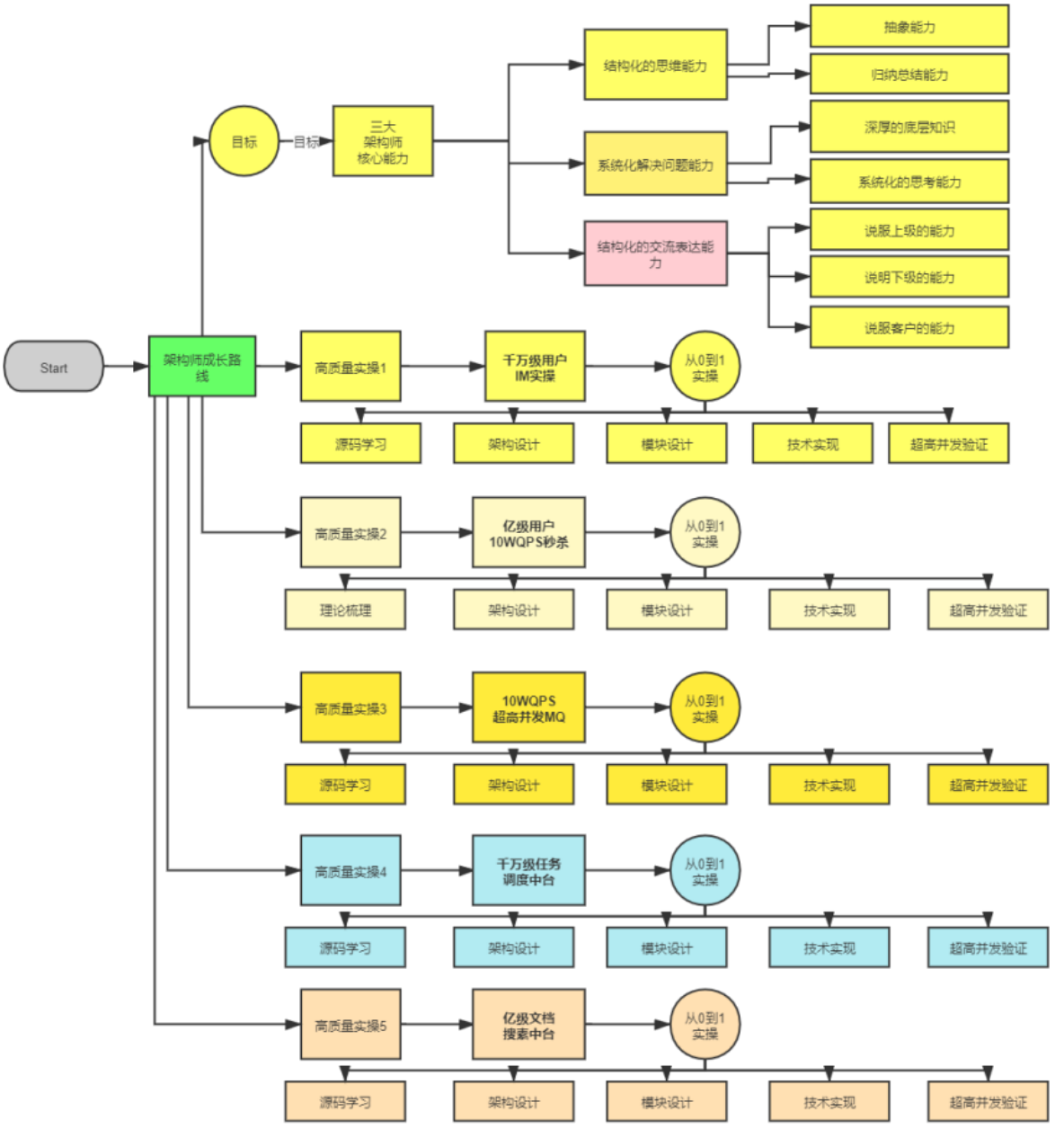
作为一个高质量的架构师成长、人脉社群，把所有的卷王聚焦起来，一起卷：

- 卷高并发实操
- 卷底层原理
- 卷架构理论、架构哲学
- 最终成为顶级架构师，实现人生理想，走向人生巅峰

# 架构班（社群 VIP）的目的：

- 高质量的实操，大大提升简历的含金量，吸引力，增强面试的召唤率
- 为大家提供九阳真经、葵花宝典，快速提升水平
- 进大厂、拿高薪
- 一路陪伴，提供助学视频和指导，辅导大家成为架构师
- 自学为主，和其他卷王一起，卷高并发实操，卷底层原理、卷大厂面试题，争取狠卷 3 月成高手，狠卷 3 年成为顶级架构师

# N 个超高并发实操项目：简历压轴、个顶个精彩



# 【样章】第 17 章：横扫全网 Rocketmq 视频第 2 部曲：工业级 rocketmq 高可用（HA）底层原理和实操

工业级 rocketmq 高可用底层原理，包含：消息消费、同步消息、异步消息、单向消息等不同消息的底层原理和源码实现；消息队列非常底层的主从复制、高可用、同步刷盘、异步刷盘等底层原理。

工业级 rocketmq 高可用底层原理和搭建实操，包含：高可用集群的搭建。

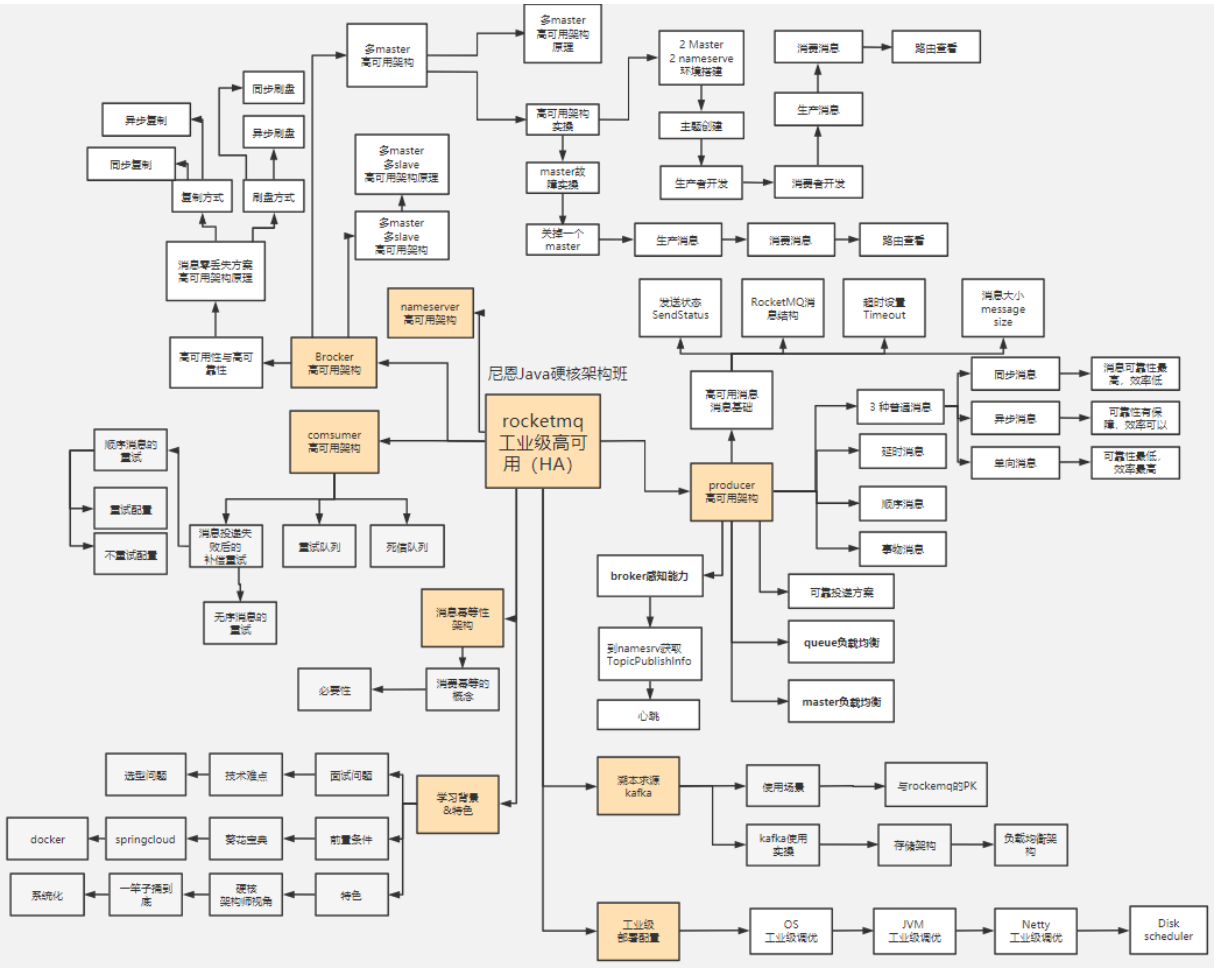
解决以下难题：

1、技术难题：RocketMQ 如何最大限度的保证消息不丢失的呢？RocketMQ 消息如何做到高可靠投递？

2、技术难题：基于消息的分布式事务，核心原理不理解

3、选型难题： kafka or rocketmq ，该娶谁？

下图链接：https://www.processon.com/view/6178e8ae0e3e7416bde9da19



# 成功案例：2 年翻 3 倍，35 岁卷王成功转型为架构师

详情：http://topcoder.cloud/forum.php?mod=forumdisplay&fid=43&page=1

最新　最后发表　热门　精华

成功案例：[1057号卷王] 3年小伙拿到外企offer，薪酬涨了200%
卷王1号　超级版主　前天 17:41

成功案例：[645号卷王] 4年经验卷王逆袭，被毕业后，反涨24W
卷王1号　超级版主　2022-9-21

成功案例：[878号卷王] 小伙8年经验，年薪60W
卷王1号　超级版主　2022-8-13

年薪70W案例：通过尼恩的指导，小伙伴年薪从40W涨到70W
卷王1号　超级版主　2022-2-11

成功案例：[493号卷王] 5年小伙拿满意offer，就业寒冬季逆涨30%
卷王1号　超级版主　前天 17:43

成功案例：[250号卷王] 就业极寒时代，收offer 涨25%
卷王1号　超级版主　前天 17:38

成功案例：[612号卷王] 就业极寒时代，从外包到自研
卷王1号　超级版主　前天 17:15

成功案例：[913号卷王] 热烈祝贺6年经验卷王，年薪40W
卷王1号　超级版主　2022-9-21

成功案例：[959号卷王] 4年经验卷王，喜获百度、Boss直聘等N个优质offer，最高涨100%
卷王1号　超级版主　2022-9-21

成功案例：[529号卷王] 5年经验卷王喜收2大offer，最高涨5K
卷王1号　超级版主　2022-9-21

成功案例：[811号卷王] 热烈祝贺7年经验卷王，薪酬涨30%
卷王1号　超级版主　2022-9-21

成功案例：[287号卷王] 不惧大寒潮，卷王逆市收4 offer，涨30%，可喜可贺
卷王1号　超级版主　2022-5-30

成功案例：[1002号卷王] 5月份"被毕业"，改简历后，斩获顶级央企Offer，涨薪7000+
卷王1号　超级版主　2022-7-5

成功案例：[7号卷王] 热烈祝贺小伙伴涨薪120%

卷王1号 超级版主 2022-8-13

成功案例：[134号卷王] 大三小伙卷1年，斩获顶级央企Offer，成功逆袭

卷王1号 超级版主 2022-7-6

成功案例：[1008号卷王] 5年经验卷王收42W offer，月涨8000，可喜可贺

卷王1号 超级版主 2022-5-30

成功案例：[453号卷王] 非全日制 6年卷王喜提3 offer，年薪30W，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[924号卷王] 6年卷王喜提4 offer，最高涨薪9000，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[15号卷王] 4年卷王入职 微软，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-12

成功案例：[527号卷王] 4年卷王喜提2 offer，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-13

成功案例：[788号卷王] 3年卷王喜提优质Offer，涨薪60%

卷王1号 超级版主 2022-5-11

成功案例：热烈祝贺：非全日制卷王，喜提2个心仪offer，面3家过2家

卷王1号 超级版主 2022-4-21

成功案例：[693号卷王] 二线城市6年卷王喜提4大优质Offer，含央企offer，最高薪酬35W

卷王1号 超级版主 2022-4-16

成功案例：[85号卷王] 双非2本小伙，春招大捷，喜提9个offer，最高薪酬近30万

卷王1号 超级版主 2022-4-14

成功案例：[741号卷王] 卷王逆袭！6年小伙从很少面试机会到搞定35K*14薪Offer

卷王1号 超级版主 2022-4-12

成功案例：[642号卷王] 热烈祝贺，6年卷王喜提优质国企offer

卷王1号 超级版主 2022-4-7

成功案例：[796号卷王] 热烈祝贺，36岁卷王喜提52万优质offer

卷王1号 超级版主 2022-3-25

成功案例：[15号卷王] 小伙卷1年，涨薪9K+，喜收ebay等多个优质offer

卷王1号 超级版主 2022-3-24

成功案例：[821号卷王] 小伙狠卷3个月，喜提10多个offer

卷王1号 超级版主 2022-3-21

成功案例：[736号卷王] 3年半经验收22k offer，但是小伙志存高远，冲击25k+

卷王1号 超级版主 2022-3-20

成功案例：热烈祝贺1群小卷王offer拿到手软，甚至拒了阿里offer

卷王1号 超级版主 2022-3-16

简历案例：简历一改，腾讯的邀请就来了！热烈祝贺，小伙收到一大堆面试邀请

卷王1号 超级版主 2022-3-10

成功案例：祝贺我圈两大超级卷王，一个过了阿里HR面，一个过了阿里2面

卷王1号 超级版主 2022-3-10

成功案例：小伙伴php转Java，卷1.5年Java，涨薪50%，喜收多个优质offer

卷王1号 超级版主 2022-3-10

成功案例：4年小伙狠卷半年，拿到 移动、京东 两大顶级offer

尼恩 超级版主 2022-3-5

成功案例：[267号卷王] 助力3年经验卷王，拿到蜂巢的17k x 14薪的offer

卷王1号 超级版主 2022-2-27

成功案例：[143号卷王] 二本院校00后卷神，毕业没到一年跳到字节，年薪45W

卷王1号 超级版主 2022-2-27

成功案例：[494号卷王] 尼恩分布式事务助力卷王拿到 中信银行offer

卷王1号 超级版主 2022-2-27

成功案例：[76号卷王] 2线城市卷王，狠卷1.5年，喜收22K offer

卷王1号 超级版主 2022-2-27

成功案例：[429号卷王] 小伙伴在社群卷5个月，涨8k+

卷王1号 超级版主 2022-2-27

成功案例：[154号卷王] 双非学校毕业卷王，连拿 京东到家&滴滴 两个大厂Offer

卷王1号 超级版主 2022-2-27

成功案例：[232号卷王] 涨薪10K，继续卷向食物链顶端

卷王1号 超级版主 2022-2-27

成功案例：狠卷1年技术，喜收 腾讯、阿里、微软三大Offer，最高年薪56W

卷王1号 超级版主 2022-2-27

成功案例：[449号卷王] 应届毕业卷王喜收 滴滴offer, 年薪33W

卷王1号 超级版主 2022-2-27

成功案例：[551号卷王] 小伙伴学完后，成功进入大厂，并且推荐自己的朋友加VIP学习

卷王1号 超级版主 2022-2-10

成功案例：[214号卷王] 助力2年经验卷王，成功拿到17K月薪

卷王1号 超级版主 2022-2-10

成功案例：[92号卷王] 课程实操助力社群小伙伴喜收 喜马拉雅Offer

卷王1号 超级版主 2022-2-10

成功案例：社群卷王小伙伴成功过了滴滴三面 获滴滴Offer

卷王1号 超级版主 2022-2-10

[612号卷王]滴滴小伙伴，蹲点考察半年，觉得靠谱后加入 疯狂创客圈

卷王1号 超级版主 2022-2-10

成功案例：[732号卷王] 尼恩助力3年经验卷王收获 京东offer, 年薪35W

卷王1号 超级版主 2022-2-27

成功案例：[558号卷王] 2年经验卷王，喜收 网易和阿里子公司两个优质offer

卷王1号 超级版主 2022-2-27

成功案例：[569号卷王] 双非应届生卷王, 喜收字节跳动实习offer

卷王1号 超级版主 2022-2-25

成功案例：[420号卷王] 狠卷1年，卷王涨薪80%，涨薪12000元!

卷王1号 超级版主 2022-2-25

成功案例：[76号卷王] 通过尼恩1年半的指导，专科学历小伙伴从0.8K涨到22K

卷王1号 超级版主 2022-2-10

# 简历优化后的成功涨薪案例（VIP含免费简历优化）


9年 小伙伴拿到
年薪90W offer
9月11日改简历
11月29日晒offer
秘诀:
简历指导+ 狠狠卷
这个是给你微调的
省略号的地方，需要你再补充一点
上面你留着这个就行
Java 开发 9年-修改.docx
34.1 KB
薪资高 稳
主要的工作是啥?
提升了自己的实力，就不用怕
易所 关于数字货 的 易
po 也有 打盹的时候，该裁员，照样一个不少
年包比 po 多 19w
这么多
po 估计有 70 万
那你不有 90 个 W?
po 给我 68
就是吗


小伙8年经验
年薪60w
7月12日改简历
8月10日晒offer
秘诀:
改简历 + 狠狠卷
明天晚上哈
好哒
恩哥，今晚还改简历吗?
今晚还在外边应酬，估计回去比较晚
要不，咱们延迟到明天，如何
明天白天也行
好的，白天吧，答应别人明天给他们简历了。
OK
那就上午 11 点左右哈
好的
涨幅多少呀?
或者，额度多少呀
之前 36*15，现在这个 39*15
今年行情不太好，还有一些 offer 基本都是平薪，或降薪的。
OK
这个马上来
刚在指导简历
哈哈
稍等哈
辛苦啊恩哥
这次找工作，您的 起到效果了。
我这次复习基本看的都是咱们课程的 。


6年小伙伴
年薪40w
9月6日改简历
9月21日晒offer
秘诀:
简历指导+ 狠狠卷
Java - 6年.docx
24.7 KB
恩哥，简历我改好了，您再帮我看一下
Java - 6年(1).docx
24.5 KB
恩哥，看第二个吧
给你前面调整了一下
涨这多
今年这行情，也算可以了
总包多少呀，让我也陶醉一下
大概 40w 吧
谢谢恩哥的指导和鼓励
是在深圳
深圳的行情尤其难
能有面试电话就不错啦
是的
太牛啦
哈哈哈哈，恩哥的鼓励指导也很重要


5年小伙喜提3个offer
年薪 35个W
5月22日改简历
11月29日晒offer
我简历还没写出来 不然可以给老师您看看
算法方面，可以找点 刷刷，当作复习
项目的话，再好好改造一下
好的 谢谢老师
java 开发-本科统招-2022-5 年经验.docx
36.0 KB
OK
我排好号，回头找你哈
秘诀:
简历指导+ 狠狠卷
尼恩老师晚上好，汇报下最近的 offer 情况。最近面试收到了三个 offer，两个是平薪，一个是跨境电商公司的 offer，涨幅暂时不到 20%。通过这次面试也让我知道自己距离高级开发还有一点点距离，还要再多看才能突破。
最后结合自己的情况，先选择去跨境电商的公司再提升下
之前是 20k*13.5，跨境电商这个是说 23k*（14-15）
恭喜恭喜
就业寒冬，能提到 3 个 offer，已经很 ，已经是高手了
很多小伙伴，面试电话一个都接不到，简历海投 7000 份，只收到 3 次面试机会，没有一个机会拿到最终 offer
您这个年薪，算下来也有 35W 了吧
年终部分还要确认下，大部分人听说只有两个月
这个时间点，拿到这个水平，挺不错的啦
持续加油卷哈
嗯嗯！看看明年自己有没有能力冲击高开
把你我视频都吃透
辛苦老师再帮忙指导下哈


1.5年小伙搞定15K offer
就业寒冬涨100%
5月7日改简历
11月21日晒offer
秘诀：
简历指导+狠狠卷
多谢啦
你好，好多公司都觉得我的简历太简单了，我感觉是不是没法优化了
简历.pdf
397.4 KB
好的
晚上指导你改
推送中台需要加不
我还没做啊，准备八股文暂时没时间搞这些，入职了会开始搞
OK
还有其他的项目吗
一个项目，太少啦
那种练习项目也行
他那个不止两倍吧，我才是两倍，原先7k，现在15k
那也很牛
都翻倍了，都是牛人
正常就30%
慢慢来
下一步可以瞄准25k
其实我工作一年
那你就更牛逼拉
一年经验，搞定15k offer，舍你其谁


卷王逆袭成功案例
6年小伙从很少面试机会到
搞定35K*14薪
3月5日改简历
4月11日拿offer
面试邀请少
小伙很苦恼
理想很丰满
目标35K
我要努力争取35+
我指导简历，非vip收费3000，值这个价吧？
一个月拿到了
理想的offer
面试法宝
rocketmq四部曲

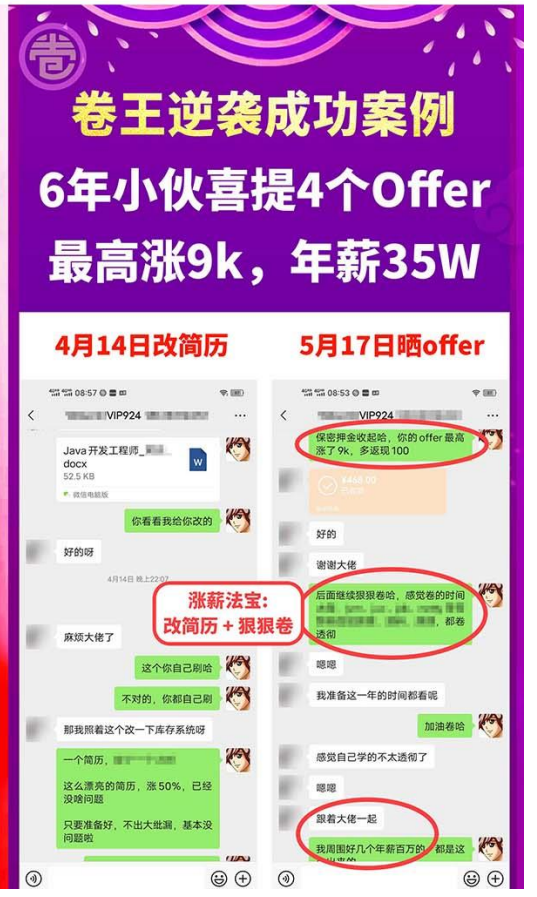

6年 经验小伙伴
喜收25K offer
3月12日改简历
12月1日晒offer
秘诀：
简历指导+狠狠卷
老师
咱们以这个项目为模板改哈
嗯好
项目有多少人，你带领多少人？
你工作几年了？
6年了
项目有多少人，你带领多少人？
语音交流一下
14:48
好
你方便吗
OK
不知道往哪个方向发展
改简历挪窝，20 涨到 25 k，高级开发，P7 带的后端团队
任务重，道路远，并不是没有方向


7年经验卷王
薪酬涨30%
7月11日改简历
9月1日晒offer
秘诀：
改简历 + 狠狠卷
咱们开始修改哈
要语音还是打字也行，还没到家
解决过最复杂的难题是啥？说3个哈
看看，有没有问题？
没问题
看看，有没有问题？
只能挪窝
下次你喊足一点哦
只要本事好，拿 offer 比较容易的
喊了，但是我流水太差了。。。我这家是补贴，银行不认这玩意，只能在上家的基础上加到 30% 封顶（特别优秀除外，我只是不错。。。）
入职了，最终并不能大幅涨薪，还是旧的太保，只涨了 30%
恭喜一下
现在不比往年
能涨 30% 是大牛
拿到 offer 都算大牛
现在很多人投几百发，连个电话都没有
你已经很


4年经验卷王逆袭
被毕业后，反涨24W
7月改简历
8月30日晒offer
秘诀:
改简历 + 狠狠卷
这就是你的简历
差得太多啦
ok
总共写了四个项目，最近的一年的还没补充上
你是在职，还是离职呢?
离职
原因大概是啥?
这个地方要这样写
项目被终止
方便语音沟通不
ok
是的 经你这么指导，非常重要
老哥 我八月十号开始找工作，今天已经入职了
现金基本持平，股票 +24W
总计涨了多少呢
能涨24W
股票这个吧只能到手了才算
也不错啦
很多小伙伴，面试机会都没有
感谢老哥的指导，继续跟你卷技术
继续狠狠卷哈，马上就技术自由拉

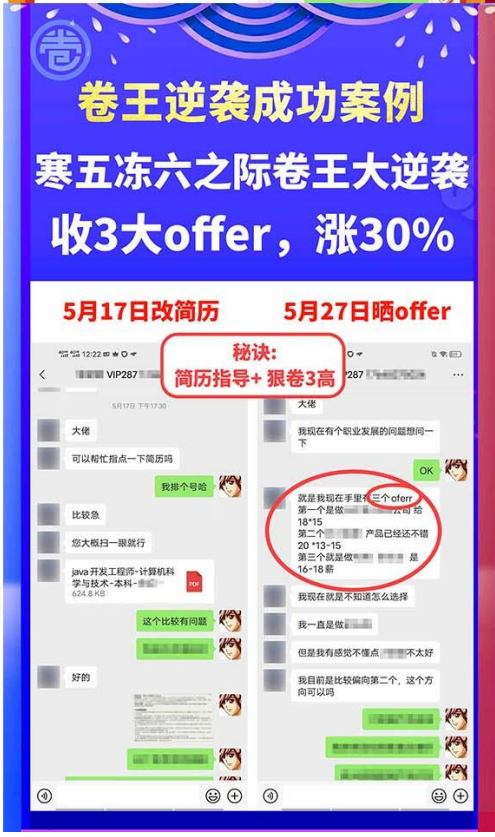

小伙5月份"被毕业"，改简历后
斩获顶级央企Offer
涨薪7000+
5月29日改简历
7月5日晒offer
秘诀:
简历指导+ 狠卷3高
快速看书，就要不求甚解，把目录和 场景大概一下，然后重点的地方，用到的地方，再去回炉
尼恩
我被 " 毕业 " 了
这周末或下周我你改一下简历
毕业了没有关系
ok，发我吧
it行业，跳来跳去，太频繁啦
嗯，其实有点心理准备
简历
不太会写简历
尼恩
我拿到 的offer 了
涨20%，
不还价的
看起来 不差钱呀
超过了8000没
平均算下来
7000多
好的
有啥面试的心得吗
可以分享给其他小伙伴的


卷王逆袭成功案例
武汉6年喜收4个优质offer
最高的年薪35W
2月9日改简历
4月15日晒offer
面试法宝:
改简历 + 实操
尼恩老师，新年好!
能帮忙修改下简历吗?
金三银四准备跳了
可以的
java 开发-6 年-简历-
.pdf
拜托了，恩师。希望能拿 25k 回来给你报喜
好，我加一下
还有吗?
恩哥，决赛圈 offer 了
截图是以我目前认知能写出来的评分了，麻烦帮我参考下
选择大于努力，恩师助我上岸
这么多 offer，我看着哈
都是恩哥指点有方，本来还有个新能源汽车的，35w 给拒了，主要太远了
跟着恩哥卷的时间太短了，目前实力也只能到这儿了
这里边有个大数据的，感觉也不错


卷王逆袭成功案例
6年小伙喜提4个Offer
最高涨9k，年薪35W
4月14日改简历
5月17日晒offer
涨薪法宝:
改简历 + 狠狠卷
Java 开发工程师_
.docx
你看看我给你改的
好的呀
麻烦大佬了
这个你自己刷哈
不对的，你都自己刷
那我照着这个改一下库存系统呀
一个简历，
这么漂亮的简历，涨 50%，已经没啥问题
只要准备好，不出大纰漏，基本没问题啦
保密拼金收起哈，你的 offer 最高涨了 9k，多返现 100
好的
谢谢大佬
后面继续狠狠卷哈，感觉卷的时间 都卷透彻
嗯嗯
我准备这一年的时间都看呢
加油卷哈
感觉自己学的不太透彻了
嗯嗯
跟着大佬一起
我周围好几个年薪百万的，都是这


卷王逆袭成功案例
5年经验小伙收2个offer
最高涨薪8k，年薪42W
5月9日改简历
5月30日晒offer
秘诀:
简历指导+ 狠卷3高
以此为样
大家狠狠卷
打造最卷IT社群


卷王逆袭成功案例
非全日制 6年经验卷王
喜提3个Offer，年包30W
5月9日改简历
5月18日晒offer
面试法宝:
改简历 + 狠狠卷


卷王逆袭成功案例
寒五冻六之际卷王大逆袭
收3大offer，涨30%
5月17日改简历
5月27日晒offer
秘诀:
简历指导+ 狠卷3高


卷王逆袭成功案例
4年卷王入职微软，涨50%
3月7日改简历
5月12日晒offer
涨薪法宝:
改简历 + 狠狠卷


4年小伙喜收百度、Boss直聘
等N个顶级Offer
最高涨幅100%
6月27日改简历
9月19日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
4年卷王入收2个offer，涨50%
3月23日改简历
5月12日晒offer
涨薪法宝:
改简历＋狠狠卷


小伙大三暑期很焦虑
跟着尼恩卷一年
校招斩获顶级央企Offer
去年5月19日加入VIP群
今年7月5日晒offer
秘诀:
狠狠卷 书+视频


小伙高中学历
薪酬涨120%
5月6日改简历
7月22日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
非全日制卷王 面试3家
收2个offer 涨薪30%
4月13日改简历
4月21日晒offer
面试法宝：
改简历 + 面试题


5年卷王喜收2大Offer
最高涨5K
5月19日改简历
9月13日晒offer
秘诀：
改简历 + 狠狠卷


卷王逆袭成功案例
3年经验卷王，涨60%
4月16日改简历
5月11日晒offer
涨薪法宝：
改简历 + 狠狠卷


卷王逆袭成功案例
双非二本小伙春招大翻身
喜提9大offer
2月22日改简历
4月13日晒offer
面试法宝：
改简历 + IM实操
9大offer 最高年薪30万

# 修改简历找尼恩（资深简历优化专家）

- 如果面试表达不好，尼恩会提供 简历优化指导
- 如果项目没有亮点，尼恩会提供 项目亮点指导
- 如果面试表达不好，尼恩会提供 面试表达指导

**作为 40 岁老架构师，尼恩长期承担技术面试官的角色：**

- **从业以来，“阅历”无数，对简历有着点石成金、改头换面、脱胎换骨**的指导能力。
- **尼恩指导过刚刚就业的小白，也指导过 P8 级的老专家，都指导他们上岸。**

如何联系尼恩。尼恩微信，请参考下面的地址：

语雀：https://www.yuque.com/crazymakercircle/gkkw8s/khigna
码云：https://gitee.com/crazymaker/SimpleCrayIM/blob/master/疯狂创客圈总目录.md