eks-distro/kubernetes/pause

(1.4B+ downloads)

by Amazon EKS Distribution Verified Account

public.ecr.aws/eks-distro/kubernetes/pause:v1.26.5-eks-1-26-latest

Copy

Updated 5 days ago

Amazon EKS Distro Kubernetes pause image

OS/Arch: Linux, x86-64, ARM 64

About | Usage | Image tags

Amazon EKS Distro image for Kubernetes pause container

The Kubernetes pause container serves as the parent container for all of the containers in a pod. The pause container has two core responsibilities. First, it serves as the basis of Linux namespace sharing in the pod. Second, with PID (process ID) namespace sharing enabled, it serves as PID 1 for each pod and reaps zombie processes. It is responsible for creating shared network, assigning IP addresses within the pod for all containers inside this pod. If the pause container is terminated, Kubernetes will consider the whole pod as terminated and kill it and reschedule a new one.

工作 (/tags/#工作)　Kubernetes (/tags/#Kubernetes)　Docker (/tags/#Docker)

Kubernetes之Pause容器

The Pause Container Of Kubernetes

Posted by ChenJian on October 17, 2017

pause根容器

在接触Kubernetes的初期，便知道集群搭建需要下载一个 `gcr.io/google_containers/pause-amd64:3.0 (http://gcr.io/google_containers/pause-amd64:3.0)` 镜像，然后每次启动一个容器，都会伴随一个pause容器的启动。

但这个 `pause` 容器的功能是什么，它是如何做出来的，以及为何都伴随容器启动等等。这些问题一直在我心里，如今有缘学习相关内容。

`pause` 源码在kubernetes项目(**v1.6.7版本**)的 `kubernetes/build/pause/` 中。

```
git clone -b v1.6.7 https://github.com/kubernetes/kubernetes.git (http://github.com/kubernetes/ku

ll kubernetes/build/pause
<<'COMMENT'
Dockerfile  Makefile  orphan.c  pause.c
COMMENT
```

pause的源码

四个文件中， pause.c 是 pause 的源码，用 c语言 编写，如下(除去注释):

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

static void sigdown(int signo) {
  psignal(signo, "Shutting down, got signal");
  exit(0);
}

static void sigreap(int signo) {
  while (waitpid(-1, NULL, WNOHANG) > 0)
    ;
}

int main() {
  if (getpid() != 1)
    /* Not an error because pause sees use outside of infra containers. */
    fprintf(stderr, "Warning: pause should be the first process in a pod\n");

  if (sigaction(SIGINT, &(struct sigaction){.sa_handler = sigdown}, NULL) < 0)
    return 1;
  if (sigaction(SIGTERM, &(struct sigaction){.sa_handler = sigdown}, NULL) < 0)
    return 2;
  if (sigaction(SIGCHLD, &(struct sigaction){.sa_handler = sigreap,
                                             .sa_flags = SA_NOCLDSTOP},
                NULL) < 0)
    return 3;

  for (;;)
    pause();
  fprintf(stderr, "Error: infinite loop terminated\n");
  return 42;
}
```

可以看出来很简单。目前这段代码讲什么，还是没看懂。

pause的Dockerfile

剩余的文件 `orphan.c` 是个测试文件，不用管。 `Makefile` 用于制作 `pause` 镜像，制作镜像的模板便是 `Dockerfile` 。先看看这个 `Dockerfile` (除去注释)：

```
FROM scratch
ARG ARCH
ADD bin/pause-${ARCH} /pause
ENTRYPOINT ["/pause"]
```

- FROM scratch

  基础镜像是一个空镜像(an explicitly empty image)

- ARG ARCH

  等待在docker build –build-arg时提供的ARCH参数

- ADD bin/pause-${ARCH} /pause

  添加外部文件到内部

- ENTRYPOINT [“/pause”]

  开启容器，运行命令

可以看出这个 `bin/pause-${ARCH}` 非常关键，但是如何制作出来呢?

pause的Makefile

- ARCH值

```
# Architectures supported: amd64, arm, arm64, ppc64le and s390x
ARCH ?= amd64

ALL_ARCH = amd64 arm arm64 ppc64le s390x
```

可以看出架构支持很多类型，默认为 amd64

- 制作pause二进制文件

```
TAG = 3.0
CFLAGS = -Os -Wall -Werror -static
BIN = pause
SRCS = pause.c
KUBE_CROSS_IMAGE ?= gcr.io/google_containers/kube-cross (http://gcr.io/google_containers/kube-cro
KUBE_CROSS_VERSION ?= $(shell cat ../build-image/cross/VERSION)

bin/$(BIN)-$(ARCH): $(SRCS)
        mkdir -p bin
        docker run --rm -u $$(id -u):$$(id -g) -v $$(pwd):/build \
                $(KUBE_CROSS_IMAGE):$(KUBE_CROSS_VERSION) \
                /bin/bash -c "\
                        cd /build && \
                        $(TRIPLE)-gcc $(CFLAGS) -o $@ $^ && \
                        $(TRIPLE)-strip $@"
```

可以看出这分为两步，

- - 运行 gcr.io/google_containers/kube-cross:xxxx (http://gcr.io/google_containers/kube-cross:xxxx) 容器

    这个镜像的制作，可在kubernetes/build/build-image/cross路径下，
    其中的Makefile很简单。Dockerfile的基础镜像是golang:1.7.6，
    可以看出这个镜像目的是This file creates a standard build environment for building cross platform go binary for the architecture kubernetes cares about.
    该镜像也包含后续所需的gcc工具。

-
  - 制作二进制文件

    > 通过挂载，在容器内部制作pause二进制文件。

- 制作pause镜像

```
TAG = 3.0
REGISTRY ?= gcr.io/google_containers (http://gcr.io/google_containers)
IMAGE = $(REGISTRY)/pause-$(ARCH)

.container-$(ARCH): bin/$(BIN)-$(ARCH)
	docker build --pull -t $(IMAGE):$(TAG) --build-arg ARCH=$(ARCH) .
```

一个很简单的制作过程。

制作pause镜像

这里绕开制作 cross 镜像，直接做 pause 镜像。

```
cd kubernetes/build/pause
mkdir -p bin

sudo gcc -Os -Wall -Werror -static -o pause pause.c

ls -hl
<<'COMMENT'
total 876K
drwxr-xr-x. 2 root root    6 Oct 16 19:13 bin
-rw-r--r--. 1 root root  679 Oct 11 15:19 Dockerfile
-rw-r--r--. 1 root root 2.9K Oct 11 15:19 Makefile
-rw-r--r--. 1 root root 1.1K Oct 11 15:19 orphan.c
-rwxr-xr-x. 1 root root 858K Oct 16 19:11 pause
-rw-r--r--. 1 root root 1.6K Oct 11 15:19 pause.c
COMMENT

file pause
<<'COMMENT'
pause: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux
COMMENT

nm pause
<<'COMMENT'
00000000004183d0 T abort
00000000006c2860 B __abort_msg
00000000004530c0 W access
00000000004530c0 T __access
0000000000492a50 t add_fdes
0000000000461bb0 t add_module.isra.1
00000000004569a0 t add_name_to_object.isra.3
00000000006c1728 d adds.8351
0000000000418ea0 T __add_to_environ
000000000048aac0 t add_to_global
00000000006c2460 V __after_morecore_hook
0000000000416350 t alias_compare
0000000000409120 W aligned_alloc
00000000006c24d0 b aligned_heap_area
00000000004523f0 T __alloc_dir
```

```
000000000049dd50 r archfname
...
COMMENT

# 开始Strip
strip pause

ls -lh pause
<<'COMMENT'
-rwxr-xr-x. 1 root root 781K Oct 16 19:22 pause
COMMENT

file pause
<<'COMMENT'
pause: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux
COMMENT

nm pause
<<'COMMENT'
nm: pause: no symbols
COMMENT

cp pause bin/pause-amd64

docker build --pull -t gcr.io/google_containers/ (http://gcr.io/google_containers/)pause-amd64:3.

<<'COMMENT'
Sending build context to Docker daemon  1.612MB
Step 1/4 : FROM scratch
 --->
Step 2/4 : ARG ARCH
 ---> Running in 6eec4bcd21b7
 ---> 30b135219bee
Removing intermediate container 6eec4bcd21b7
Step 3/4 : ADD bin/pause-${ARCH} /pause
 ---> acda3361fddc
Removing intermediate container 79a21fb7baca
Step 4/4 : ENTRYPOINT /pause
```

```
 ---> Running in dd1d266bb882
 ---> 18620a113848
Removing intermediate container dd1d266bb882
Successfully built 18620a113848
Successfully tagged gcr.io/google_containers/ (http://gcr.io/google_containers/)pause-amd64:3.0
COMMENT

docker images

<<'COMMENT'
REPOSITORY                                TAG                 IMAGE ID            CREATED
gcr.io/google_containers/ (http://gcr.io/google_containers/)pause-amd64   3.0                 186
busybox                                   latest              2b8fd9751c4c        15 months ago
COMMENT
```

- strip

  通过上面的对比，可以看出strip后，pause文件由858K瘦身到781K。strip执行前后，不改变程序的执行能力。在开发过程中，strip用于产品的发布，调试均用未strip的程序。

- file

  通过file命令可以看到pause的strip状态

- nm

  通过nm命令，可以看到strip后的pause文件没有符号信息

pause容器的工作

可知kubernetes的pod抽象基于Linux的namespace和cgroups，为容器提供了良好的隔离环境。在同一个pod中，不同容器犹如在localhost中。

在Unix系统中，PID为1的进程为init进程，即所有进程的父进程。它很特殊，维护一张进程表，不断地检查进程状态。例如，一旦某个子进程由于父进程的错误而变成了“孤儿进程”，其便会被init进程进行收养并最终回收资源，从而结束进程。

或者，某子进程已经停止但进程表中仍然存在该进程，因为其父进程未进行wait syscall进行索引，从而该进程变成“僵尸进程”，这种僵尸进程存在时间较短。不过如果父进程只wait，而未syscall的话，僵尸进程便会存在较长时间。

同时，init进程不能处理某个信号逻辑，拥有“信号屏蔽”功能，从而防止init进程被误杀。

容器中使用pid namespace来对pid进行隔离，从而每个容器中均有其独立的init进程。例如对于寄主机上可以用个发送SIGKILL或者SIGSTOP(也就是docker kill 或者docker stop)来强制终止容器的运行，即终止容器内的init进程。一旦init进程被销毁， 同一pid namespace下的进程也随之被销毁，并容器进程被回收相应资源。

kubernetes中的pause容器便被设计成为每个业务容器提供以下功能：

- 在pod中担任Linux命名空间共享的基础；
- 启用pid命名空间，开启init进程。

实践操作

已有刚做好的 `pause` 镜像和 `busybox` 镜像

```
docker images

<<'COMMENT'
REPOSITORY                              TAG                 IMAGE ID            CREATED
gcr.io/google_containers/ (http://gcr.io/google_containers/)pause-amd64   3.0                 186
busybox                                 latest              2b8fd9751c4c        15 months ago
COMMENT

docker run -idt --name pause gcr.io/google_containers/ (http://gcr.io/google_containers/)pause-ar
<<'COMMENT'
7f6e459df5644a1db4bc9ad2206a0f99e40312de1892695f8a09d52faa9c1073
COMMENT

docker ps -a
<<'COMMENT'
CONTAINER ID        IMAGE                                           COMMAND             CREATED
7f6e459df564        gcr.io/google_containers/ (http://gcr.io/google_containers/)pause-amd64:3.0
COMMENT

docker run -idt --name busybox --net=container:pause --pid=container:pause --ipc=container:pause
<<'COMMENT'
ad3029c55476e431101473a34a71516949d1b7de3afe3d505b51d10c436b4b0f
COMMENT

docker ps -a
<<'COMMENT'
CONTAINER ID        IMAGE                                           COMMAND             CREATED
ad3029c55476        busybox                                         "sh"                36 seconds ago
7f6e459df564        gcr.io/google_containers/ (http://gcr.io/google_containers/)pause-amd64:3.0
COMMENT

docker exec -it ad3029c55476 /bin/sh
<<'COMMENT'
/ # ps aux
PID   USER     TIME   COMMAND
    1 root       0:00 /pause
    5 root       0:00 sh
    9 root       0:00 /bin/sh
```

```
   13 root       0:00 ps aux
COMMENT
```

可以看出来，busybox中的 `PID 1` 由 `pause` 容器提供。

参考博文

1. scratch镜像 (https://hub.docker.com/r/library/scratch/)
2. gcc参数问题 (https://zhidao.baidu.com/question/501561674.html)
3. linux中的strip命令简介 (http://blog.csdn.net/stpeace/article/details/47090255)
4. Kubernetes中的Pod的到底是什么 (http://dockone.io/article/2682)
5. Kubernetes之“暂停”容器 (http://dockone.io/article/2785)

Pause 容器

Pause 容器，又叫 Infra 容器，本文将探究该容器的作用与原理。

我们知道在 kubelet 的配置中有这样一个参数：

```
KUBELET_POD_INFRA_CONTAINER=--pod-infra-container-image=registry.access.redhat.com/rhel7/pod-infrastructure:latest
```

上面是 openshift 中的配置参数，kubernetes 中默认的配置参数是：

```
KUBELET_POD_INFRA_CONTAINER=--pod-infra-container-image=gcr.io/google_containers/pause-amd64:3.0
```

Pause 容器，是可以自己来定义，官方使用的 `gcr.io/google_containers/pause-amd64:3.0` 容器的代码见 Github，使用 C 语言编写。

Pause 容器特点

- 镜像非常小，目前在 700KB 左右
- 永远处于 Pause (暂停) 状态

Pause 容器背景

像 Pod 这样一个东西，本身是一个逻辑概念。那在机器上，它究竟是怎么实现的呢？这就是我们要解释的一个问题。

既然说 Pod 要解决这个问题，核心就在于如何让一个 Pod 里的多个容器之间最高效的共享某些资源和数据。

因为容器之间原本是被 Linux Namespace 和 cgroups 隔开的，所以现在实际要解决的是怎么去打破这个隔离，然后共享某些事情和某些信息。这就是 Pod 的设计要解决的核心问题所在。

所以说具体的解法分为两个部分：网络和存储。

Pause 容器就是为解决 Pod 中的网络问题而生的。

Pause 容器实现

Pod 里的多个容器怎么去共享网络？下面是个例子：

比如说现在有一个 Pod，其中包含了一个容器 A 和一个容器 B，它们两个就要共享 Network Namespace。在 Kubernetes 里的解法是这样的：它会在每个 Pod 里，额外起一个 Infra container 小容器来共享整个 Pod 的 Network Namespace。

Infra container 是一个非常小的镜像，大概 700KB 左右，是一个 C 语言写的、永远处于 “暂停” 状态的容器。由于有了这样一个 Infra container 之后，其他所有容器都会通过 Join Namespace 的方式加入到 Infra container 的 Network Namespace 中。

所以说一个 Pod 里面的所有容器，它们看到的网络视图是完全一样的。即：它们看到的网络设备、IP 地址、Mac 地址等等，跟网络相关的信息，其实全是一份，这一份都来自于 Pod 第一次创建的这个 Infra container。这就是 Pod 解决网络共享的一个解法。

在 Pod 里面，一定有一个 IP 地址，是这个 Pod 的 Network Namespace 对应的地址，也是这个 Infra container 的 IP 地址。所以大家看到的都是一份，而其他所有网络资源，都是一个 Pod 一份，并且被 Pod 中的所有容器共享。这就是 Pod 的网络实现方式。

由于需要有一个相当于说中间的容器存在，所以整个 Pod 里面，必然是 Infra container 第一个启动。并且整个 Pod 的生命周期是等同于 Infra container 的生命周期的，与容器 A 和 B 是无关的。这也是为什么在 Kubernetes 里面，它是允许去单独更新 Pod 里的某一个镜像的，即：做这个操作，整个 Pod 不会重建，也不会重启，这是非常重要的一个设计。

Pause 容器的作用

我们检查 node 节点的时候会发现每个 node 上都运行了很多的 pause 容器，例如如下。

```
$ docker ps
CONTAINER ID        IMAGE                                                                                                                                COMMAND
2c7d50f1a7be        docker.io/jimmysong/heapster-grafana-amd64@sha256:d663759b3de86cf62e64a43b021f133c383e8f7b0dc2bdd78115bc95db371c9a                    "/run.sh"
5df93dea877a        docker.io/jimmysong/heapster-influxdb-amd64@sha256:a217008b68cb49e8f038c4eeb6029261f02adca81d8eae8c5c01d030361274b8                  "influxd --
9cec6c0ef583        jimmysong/pause-amd64:3.0                                                                                                            "/pause"
54d06e30a4c7        docker.io/jimmysong/kubernetes-dashboard-amd64@sha256:668710d034c4209f8fa9a342db6d8be72b6cb5f1f3f696cee2379b8512330be4               "/dashboard
5a5ef33b0d58        jimmysong/pause-amd64:3.0                                                                                                            "/pause"
```

kubernetes 中的 pause 容器主要为每个业务容器提供以下功能：

- 在 pod 中担任 Linux 命名空间共享的基础；
- 启用 pid 命名空间，开启 init 进程。

这篇文章做出了详细的说明，pause 容器的作用可以从这个例子中看出，首先见下图：

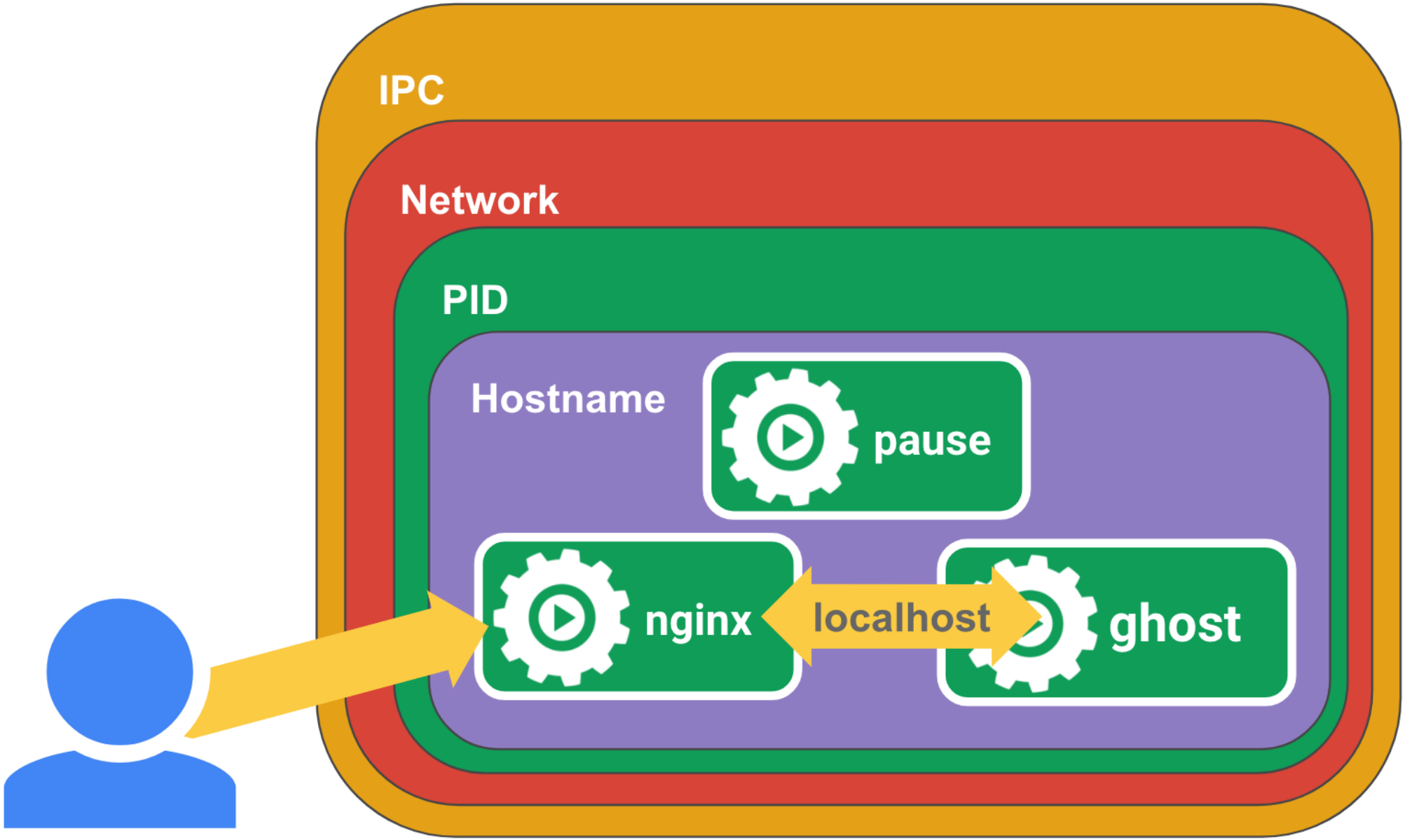


图 4.2.4.1：Pause容器

我们首先在节点上运行一个 pause 容器。

```
docker run -d --name pause -p 8880:80 --ipc=shareable jimmysong/pause-amd64:3.0
```

然后再运行一个 nginx 容器，nginx 将为 `localhost:2368` 创建一个代理。

```
$ cat <<EOF >> nginx.conf
error_log stderr;
events { worker_connections  1024; }
http {
    access_log /dev/stdout combined;
    server {
        listen 80 default_server;
        server_name example.com www.example.com;
        location / {
            proxy_pass http://127.0.0.1:2368;
        }
    }
}
EOF
$ docker run -d --name nginx -v `pwd`/nginx.conf:/etc/nginx/nginx.conf --net=container:pause --ipc=container:pause --pid=container:pause nginx
```

然后再为 ghost 创建一个应用容器，这是一款博客软件。

```
$ docker run -d --name ghost --net=container:pause --ipc=container:pause --pid=container:pause ghost
```

现在访问 http://localhost:8880/ 就可以看到 ghost 博客的界面了。

解析

pause 容器将内部的 80 端口映射到宿主机的 8880 端口，pause 容器在宿主机上设置好了网络 namespace 后，nginx 容器加入到该网络 namespace 中，我们看到 nginx 容器启动的时候指定了 `--net=container:pause`，ghost 容器同样加入到了该网络 namespace 中，这样三个容器就共享了网络，互相之间就可以使用 `localhost` 直接通信，`--ipc=container:pause --pid=container:pause` 就是三个容器处于同一个 namespace 中，init 进程为 `pause`，这时我们进入到 ghost 容器中查看进程情况。

```
# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0   1024     4 ?        Ss   13:49   0:00 /pause
root          5  0.0  0.1  32432  5736 ?        Ss   13:51   0:00 nginx: master p
systemd+      9  0.0  0.0  32980  3304 ?        S    13:51   0:00 nginx: worker p
node         10  0.3  2.0 1254200 83788 ?       Ssl  13:53   0:03 node current/in
root         79  0.1  0.0   4336   812 pts/0    Ss   14:09   0:00 sh
root         87  0.0  0.0  17500  2080 pts/0    R+   14:10   0:00 ps aux
```

在 ghost 容器中同时可以看到 pause 和 nginx 容器的进程，并且 pause 容器的 PID 是 1。而在 Kubernetes 中容器的 PID=1 的进程即为容器本身的业务进程。

参考

- The Almighty Pause Container - ianlewis.org
- Kubernetes 之 Pause 容器 - o-my-chenjian.com

Lifecycle of pause container in a pod

Asked 1 year, 9 months ago Modified 1 year, 9 months ago Viewed 2k times

1

In k8s, a pod starts with a container named `pause`.
Pause container helps other containers share network namespace.
I know it but have a question.

What is the lifecycle of the pause container?
What I want to know is that when a pod gets `crashloopbackoff` or temporally doesn't work, does the pause container also stop?

If not, does the pause container maintains its own Linux namespace?

kubernetes

Share Improve this question Follow

edited Sep 15, 2021 at 8:26

CodeWizard
125k • 21 • 141 • 163

asked Sep 15, 2021 at 6:56

Togomi
129 • 10

2 Answers

Sorted by: Highest score (default)

1

> When a pod gets `crashloopbackoff` or temporally doesn't work, does the pause container also stop?

No. A CrashloopBackOff is independent of `pause` containers. I have reproduced this situation in Minikube with docker driver with following yaml:

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: dummy-pod
spec:
  containers:
    - name: dummy-pod
      image: ubuntu
  restartPolicy: Always
```

Command `kubectl get pods` returns for me:

```
NAME        READY   STATUS             RESTARTS         AGE
dummy-pod   0/1     CrashLoopBackOff   7 (2m59s ago)    14m
```

Then I have logged into the node on which the crashed pod exist and run `docker ps`:

```
CONTAINER ID   IMAGE                  COMMAND                  CREATED
STATUS         PORTS     NAMES
7985cf2b01ad   k8s.gcr.io/pause:3.5   "/pause"                 14 minutes ago
Up 14 minutes             k8s_POD_dummy-pod_default_0f278cd1-6225-4311-98c9-
e154bf9b42a3_0
18eeb073fe71   6e38f40d628d           "/storage-provisioner"   16 minutes ago
Up 16 minutes             k8s_storage-provisioner_storage-provisioner_kube-
system_5c3cec65-5a2d-4881-aa34-d98e1098f17f_1
b7dd2640584d   8d147537fb7d           "/coredns -conf /etc…"   17 minutes ago
Up 17 minutes             k8s_coredns_coredns-78fcd69978-h28mp_kube-
system_f62eec5a-290c-4a42-b488-e1475d7f6ff2_0
d3acb4e61218   36c4ebbc9d97           "/usr/local/bin/kube…"   17 minutes ago
Up 17 minutes             k8s_kube-proxy_kube-proxy-bf75s_kube-system_39dc64cc-
2eab-497d-bf13-b5e6d1dbc9cd_0
083690fe3672   k8s.gcr.io/pause:3.5   "/pause"                 17 minutes ago
Up 17 minutes             k8s_POD_coredns-78fcd69978-h28mp_kube-system_f62eec5a-
290c-4a42-b488-e1475d7f6ff2_0
df0186291c8c   k8s.gcr.io/pause:3.5   "/pause"                 17 minutes ago
Up 17 minutes             k8s_POD_kube-proxy-bf75s_kube-system_39dc64cc-2eab-
497d-bf13-b5e6d1dbc9cd_0
06fdfb5eab54   k8s.gcr.io/pause:3.5   "/pause"                 17 minutes ago
Up 17 minutes             k8s_POD_storage-provisioner_kube-system_5c3cec65-5a2d-
4881-aa34-d98e1098f17f_0
183f6cc10573   aca5ededae9c           "kube-scheduler --au…"   17 minutes ago
Up 17 minutes             k8s_kube-scheduler_kube-scheduler-minikube_kube-
system_6fd078a966e479e33d7689b1955afaa5_0
```

```
2d032a2ec51d   f30469a2491a           "kube-apiserver --ad…"   17 minutes ago
Up 17 minutes             k8s_kube-apiserver_kube-apiserver-minikube_kube-
system_4889789e825c65fc82181cf533a96c40_0
cd157b628bc5   6e002eb89a88           "kube-controller-man…"   17 minutes ago
Up 17 minutes             k8s_kube-controller-manager_kube-controller-manager-
minikube_kube-system_f8d2ab48618562b3a50d40a37281e35e_0
a2d5608e5bac   004811815584           "etcd --advertise-cl…"   17 minutes ago
Up 17 minutes             k8s_etcd_etcd-minikube_kube-
system_08a3871e1baa241b73e5af01a6d01393_0
e9493a3f2383   k8s.gcr.io/pause:3.5   "/pause"                 17 minutes ago
Up 17 minutes             k8s_POD_kube-apiserver-minikube_kube-
system_4889789e825c65fc82181cf533a96c40_0
1088a8210eed   k8s.gcr.io/pause:3.5   "/pause"                 17 minutes ago
Up 17 minutes             k8s_POD_kube-scheduler-minikube_kube-
system_6fd078a966e479e33d7689b1955afaa5_0
f551447a77b6   k8s.gcr.io/pause:3.5   "/pause"                 17 minutes ago
Up 17 minutes             k8s_POD_etcd-minikube_kube-
system_08a3871e1baa241b73e5af01a6d01393_0
c8414ee790d8   k8s.gcr.io/pause:3.5   "/pause"                 17 minutes ago
Up 17 minutes             k8s_POD_kube-controller-manager-minikube_kube-
system_f8d2ab48618562b3a50d40a37281e35e_0
```

The `pause` containers are independent in the pod. The pause container is a container which holds the network namespace for the pod. It does nothing. It doesn't stop even if the pod is in the CrashLoopBackOff state. If the pause container is dead, kubernetes consider the pod died and kill it and reschedule a new one. There was no such situation here.

See also an explanation of `pause` containers.

Share Improve this answer Follow

answered Sep 15, 2021 at 11:25

Mikołaj Głodziak
4,597 • 6 • 27

1

The pause container is a fully independent container like the others in the pod (other than the places where the namespaces overlap, as you mentioned). It starts when the pod is started up by the kubelet and is torn down when the pod is gone (deleted, scheduled elsewhere, whatever).

Share Improve this answer Follow

answered Sep 15, 2021 at 8:04

The Almighty Pause Container

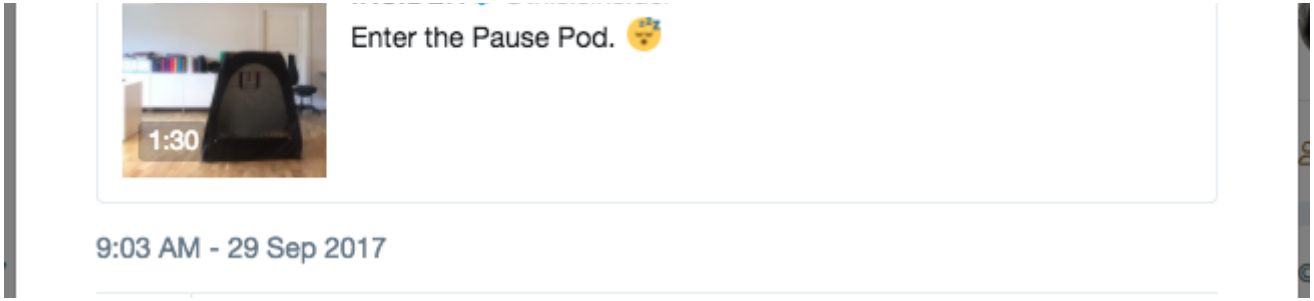
Oct 10, 2017

kubernetes docker

When checking out the nodes of your Kubernetes cluster, you may have noticed some containers called "pause" running when you do a `docker ps` on the node.

```
$ docker ps
CONTAINER ID        IMAGE                                  COMMAND ...
...
3b45e983c859        gcr.io/google_containers/pause-amd64:3.0    "/pause" ...
...
dbfc35b00062        gcr.io/google_containers/pause-amd64:3.0    "/pause" ...
...
c4e998ec4d5d        gcr.io/google_containers/pause-amd64:3.0    "/pause" ...
...
508102acf1e7        gcr.io/google_containers/pause-amd64:3.0    "/pause" ...
```

What are these "pause" containers and why are there so many of them? What's going on?



(https://twitter.com/IanMLewis/status/913554746115424256)

In order to answer these questions, we need to take a step back and look at how pods in Kubernetes are implemented, particularly with the Docker/containerd runtime. If you haven't already done so, please read my previous post (https://www.ianlewis.org/en/what-are-kubernetes-pods-anyway) on Kubernetes pods.

Docker supports containers, which are great for deploying single units of software. However, this model can become a bit cumbersome when you want to run multiple pieces of software together. You often see this when developers create Docker images that use supervisord as an entrypoint to start and manage multiple processes. For production systems, many have found that it is instead more useful to deploy those applications in groups of containers that are partially isolated and partially share an environment.

Kubernetes provides a clean abstraction called pods for just this use case. It hides the complexity of Docker flags and the need to babysit the containers, shared volumes, and the like. It also hides differences between container runtimes. For example, rkt supports pods natively (https://coreos.com/rkt/docs/latest/app-container.html#pods) so there is less work for Kubernetes to do but you as the user of Kubernetes don't have to worry about it.

In principle, anyone can configure Docker to control the level of sharing between groups of containers -- you just have to create a parent container, know exactly the right flag setting to use to create new containers that share the same environment, and then manage the lifetime of those containers. Managing the lifetimes of all of these pieces can get pretty complex.

In Kubernetes, the pause container serves as the "parent container" for all of the containers in your pod. The pause container has two core responsibilities. First, it serves as the basis of Linux namespace sharing in the pod. And second, with PID (process ID) namespace sharing enabled, it serves as PID 1 for each pod and reaps zombie processes.

Sharing namespaces

In Linux, when you run a new process, the process inherits its namespaces from the parent process. The way that you run a process in a new namespace is by "unsharing" the namespace with the parent process thus creating a new namespace. Here is an example using the `unshare` tool to run a shell in new PID, UTS, IPC, and mount namespaces.

```
sudo unshare --pid --uts --ipc --mount -f chroot rootfs /bin/sh
```

Once the process is running, you can add other processes to the process' namespace to form a pod. New processes can be added to an existing namespace using the `setns` system call.

Containers in a pod share namespaces among them. Docker lets you automate the process a bit so let's look at an example of how to create a pod from scratch by using the pause container and sharing namespaces. First we will need to start the pause container with Docker so that we can add our containers to the pod.

```
docker run -d --name pause -p 8080:80 gcr.io/google_containers/pause-amd64:3.0
```

Then we can run the containers for our pod. First we will run nginx. This will set up nginx to proxy requests to its localhost on port 2368.

> *“Note that we also mapped the host port 8080 to port 80 on the pause container rather than the nginx container because the pause container sets up the initial network namespace that nginx will be joining.*
>
> *”*

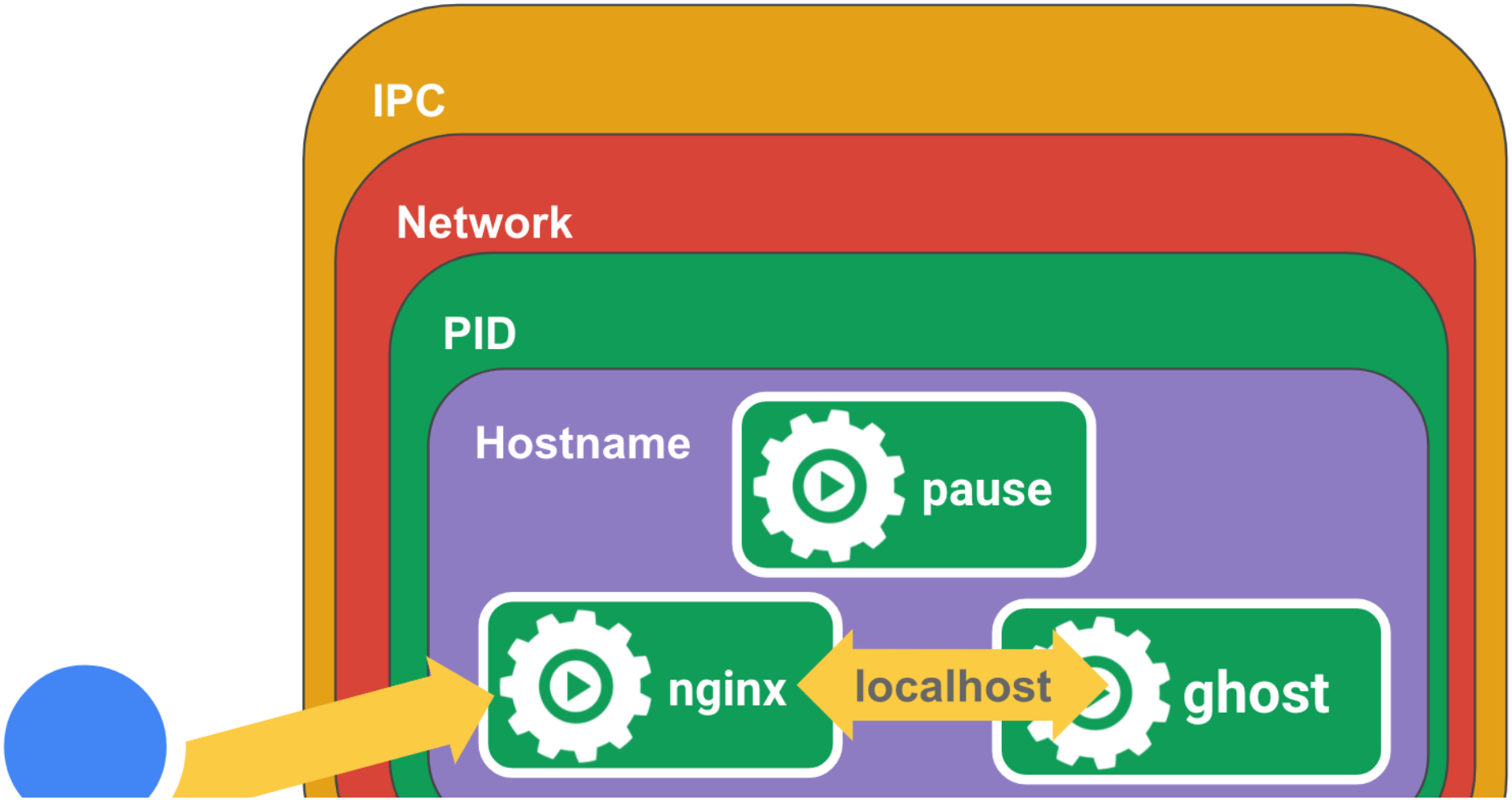
```
$ cat <<EOF >> nginx.conf
> error_log stderr;
> events { worker_connections  1024; }
> http {
>     access_log /dev/stdout combined;
>     server {
>         listen 80 default_server;
>         server_name example.com www.example.com;
>         location / {
>             proxy_pass http://127.0.0.1:2368;
>         }
>     }
> }
```

```
> EOF
$ docker run -d --name nginx -v `pwd`/nginx.conf:/etc/nginx/nginx.conf --net=container:pause --ipc=container:pause --pid=container:pause nginx
```

And then we will create another container for the ghost (https://github.com/TryGhost/Ghost) blog application which serves as our application server.

```
$ docker run -d --name ghost --net=container:pause --ipc=container:pause --pid=container:pause ghost
```

In both cases we specify the pause container as the container whose namespaces we want to join. This will effectively create our pod. If you access `http://localhost:8080/` you should be able to see ghost running through an nginx proxy because the network namespace is shared among the pause, nginx, and ghost containers.



If you think all of this is complex, you're right; it is. And we haven't even gotten into how to monitor and manage the lifetimes of these containers. The nice thing about Kubernetes is that through pods, Kubernetes manages all of this for you.

Reaping Zombies

In Linux, processes in a PID namespace form a tree with each process having a parent process. Only one process at the root of the tree doesn't really have a parent. This is the "init" process, which has PID 1.

Processes can start other processes using the `fork` and `exec` syscalls. When they do this, the new process' parent is the process that called the `fork` syscall. `fork` is used to start another copy of the running process and `exec` is used to replace the current process with a new one, keeping the same PID (in order to run a totally separate application you need to run the `fork` *and* `exec` syscalls. A process creates a new copy of itself as a child process with a new PID using `fork` and then when the child process runs it checks if it's the child process and runs `exec` to replace itself with the one you actually want to run. Most languages provide a way to do this via a single function). Each process has an entry in the OS process table. This records info about the process' state and exit code. When a child process has finished running, its process table entry remains until the parent process has retrieved its exit code using the `wait` syscall. This is called "reaping" zombie processes.

> “*CC0 Creative Commons (https://creativecommons.org/publicdomain/zero/1.0/deed.en) https://pixabay.com/en/zombie-warning-road-sign-roadsign-147945/*
>
> ”

Zombie processes are processes that have stopped running but their process table entry still exists because the parent process hasn't retrieved it via the `wait` syscall. Technically each process that terminates is a zombie for a very short period of time but they could live for longer.

Longer lived zombie processes occur when parent processes don't call the `wait` syscall after the child process has finished. One situation where this occurs is when the parent process is poorly written and simply omits the `wait` call or when the parent process dies before the child and the new parent process does not call `wait` on it. When a process' parent dies before the child, the OS assigns the child process to the "init" process or PID 1. i.e. The init process "adopts" the child process and becomes its parent. This means that now when the child process exits the new parent (init) must call `wait`to get its exit code or its process table entry remains forever and it becomes a zombie.

In containers, one process must be the init process for each PID namespace. With Docker, each container usually has its own PID namespace and the ENTRYPOINT process is the init process. However, as I noted in my previous post (https://www.ianlewis.org/en/what-are-kubernetes-pods-anyway) on Kubernetes pods, a container can be made to run in another container's namespace. In this case, one container must assume the role of the init process, while others are added to the namespace as children of the init process.

In the post on Kubernetes pods, I ran nginx in a container, and added ghost to the PID namespace of the nginx container.

```
$ docker run -d --name nginx -v `pwd`/nginx.conf:/etc/nginx/nginx.conf -p 8080:80 nginx
```

```
$ docker run -d --name ghost --net=container:nginx --ipc=container:nginx --pid=container:nginx ghost
```

In this case, nginx is assuming the role of PID 1 and ghost is added as a child process of nginx. This is mostly fine, but technically nginx is now responsible for any children that ghost orphans. If, for example, ghost forks itself or runs child processes using `exec`, and crashes before the child finishes, then those children will be adopted by nginx. However, nginx is not designed to be able to run as an init process and reap zombies. That means we could potentially have lots of them and they will last for the life of that container.

In Kubernetes pods, containers are run in much the same way as above, but there is a special pause container that is created for each pod. This pause container runs a very simple process that performs no function but essentially sleeps forever (see the `pause()` call below). It's so simple that I can include the full source code as of this writing here:

```
/*
Copyright 2016 The Kubernetes Authors.
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
    http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

static void sigdown(int signo) {
  psignal(signo, "Shutting down, got signal");
  exit(0);
}
```

```
static void sigreap(int signo) {
  while (waitpid(-1, NULL, WNOHANG) > 0);
}

int main() {
  if (getpid() != 1)
    /* Not an error because pause sees use outside of infra containers. */
    fprintf(stderr, "Warning: pause should be the first process\n");

  if (sigaction(SIGINT, &(struct sigaction){.sa_handler = sigdown}, NULL) < 0)
    return 1;
  if (sigaction(SIGTERM, &(struct sigaction){.sa_handler = sigdown}, NULL) < 0)
    return 2;
  if (sigaction(SIGCHLD, &(struct sigaction){.sa_handler = sigreap,
                                             .sa_flags = SA_NOCLDSTOP},
                NULL) < 0)
    return 3;

  for (;;)
    pause();
  fprintf(stderr, "Error: infinite loop terminated\n");
  return 42;
}
```

As you can see, it doesn't simply sleep. It performs one other important function. It assumes the role of PID 1 and will reap any zombies by calling `wait` on them when they are orphaned by their parent process (see `sigreap`). This way we don't get zombies piling up in the PID namespaces of our Kubernetes pods.

Some Context on PID Namespace Sharing

It's useful to note that there has been a lot of back-and-forth on PID namespace sharing. Reaping zombies is only done by the pause container if you have PID namespace sharing enabled, and currently it is only available in Kubernetes 1.7+. It is enabled by default if running Kubernetes 1.7 with Docker 1.13.1+ unless disabled with a kubelet flag (https://kubernetes.io/docs/admin/kubelet/) (`--docker-disable-shared-pid=true`). This was reverted (https://github.com/kubernetes/kubernetes/pull/51634) in Kubernetes 1.8 and now it's disabled by default unless enabled by a kubelet flag (`--docker-disable-`

(https://github.com/kubernetes/kubernetes/pull/51634) in Kubernetes 1.8 and now it's disabled by default unless enabled by a kubelet flag (`--docker-disable-shared-pid=false`). See the discussion on adding support for PID namespace sharing in this GitHub issue (https://github.com/kubernetes/kubernetes/issues/1615).

If PID namespace sharing is not enabled then each container in a Kubernetes pod will have its own PID 1 and each one will need to reap zombie processes itself. Many times it isn't a problem because the application doesn't spawn other processes, but zombie processes using up memory is an often overlooked issue. Because of this, and because PID namespace sharing enables you to send signals between containers in the same pod, I really hope that PID namespace sharing becomes the default in Kubernetes.

Join us in the Community

Hopefully this post helped in illuminating a core part of Kubernetes. Let me know if this post was helpful in the comments below or on Twitter (https://twitter.com/IanMLewis). If you are interested in Kubernetes and want to join us in the community, you can do that in a number of ways:

- Post and answer questions on Stack Overflow (http://stackoverflow.com/questions/tagged/kubernetes)
- Follow @Kubernetesio (https://twitter.com/kubernetesio) on Twitter (While you're at it follow me (https://twitter.com/IanMLewis) too!)
- Join the Kubernetes Slack (http://slack.k8s.io/) and chat with us. (I'm ianlewis so say Hi!)
- Contribute to the Kubernetes project on GitHub (https://github.com/kubernetes/kubernetes)

Hope to see you soon!

> *I'd like to thank Eric Tune (https://twitter.com/erictune4), Ahmet Alp Balkan (https://twitter.com/ahmetb), Craig Box (https://twitter.com/craigbox), Mete Atamel (https://twitter.com/meteatamel), Nikhil Jindal (https://twitter.com/nikhiljindal181), and Jack Wilber for reviewing this post.*

28 Comments

1 Login ▾

12 Share

Best Newest Oldest

A

Att A

6 years ago edited

Ian , I hope you do a post on Kubernetes networking . Things like how the cluster-cidr on the controller relates to the pod-cidr on the kubelet are so confusing.

23 0 Share ›

Ian Lewis Mod Att A

6 years ago edited

Thanks for the suggestion! I'll definitely add it to my list of potential topics. In the meantime you can check out some of these articles. I don't think they get at exactly what you want but hopefully should help you and others who come across this comment.

https://kubernetes.io/docs/...
https://jvns.ca/blog/2016/1...
https://jvns.ca/blog/2017/1...
https://ahmet.im/blog/kuber...

5 0 Share ›

S

SteveCoffman

6 years ago

Is this obviated by having all your containers add an entrypoint with tini or dumb-init?

1 0 Share ›

Ian Lewis **Mod** → SteveCoffman

6 years ago

Right. Each container could be started with tini or dumb-init as PID 1 and they would reap zombies for you. Ideally you wouldn't have to do that though. Particularly as you would rely on each person deploying apps to your cluster to create their images properly.

1 0 Share ›

Ijaz Ahmed → Ian Lewis

6 years ago

Hi, I have been using dumb-init so for in containers , but may be , in new docker version , it is no longer needed.

0 0 Share ›

Ian Lewis **Mod** → Ijaz Ahmed

6 years ago

If you have PID namespace sharing enabled in Kubernetes and the Docker version is >1.13.1 then yah, you wouldn't need dumb-init.

0 0 Share ›

S

S K

5 years ago edited

Awesome post. Given the way u explained the concepts. Will love to read more on other topics.

0 0 Share ›

Lee Calcote

5 years ago

Great post, @Ian Lewis. I understand your perspective is that PID namespace sharing would ideally be enabled by default in order to avoid zombie processes consuming memory and to allow signals to be sent between containers in the same pod. What are examples of when it would be convenient for a container to signal another? Which signals might these be?

0 0 Share ›

Ian Lewis Mod → Lee Calcote

5 years ago

I'm thinking signals like TERM or HUP could be sent from a sidecar container to another container in the same pod to tell it to restart it to reload it's configuration.

1 0 Share ›

A.

6 years ago

Not that Zombies shouldn't be managed, but my understanding was that the only resources they kept consuming was an entry in the process descriptor list?

0 0 Share ›

Ian Lewis Mod → A.

6 years ago

You're right and I specifically addressed that point in the post. I described it as an entry in the process table. Do you think that I addressed the point inadequately?

4 0 Share ›

Jon

6 years ago

This was an awesome post! Thank you for this explanation, it was really helpful. I am trying to reproduce this for a demo and wonder how you can get the ports for nginx exposed when using the --net:container option?

Docker run does not allow exposure of ports when using this mode unless I am missing something?

From the Docker docs:
With the network set to container a container will share the network stack of another container. The other container's name must be provided in the format of --network container:<name|id>. Note that --add-host --hostname --dns --dns-search --dns-option and --mac-address are invalid in container netmode, and --publish --publish-all --expose are also invalid in container netmode

0 0 Share ›

Ian Lewis Mod → Jon

6 years ago

When you use --net you are joining another containers network namespace so AFAIK you need to expose the ports you want to expose on that container.

0 0 Share ›

Jon → Ian Lewis

6 years ago

Yes, I see, perfect. So the above docker commands, the port exposure should be on the pause container then?

docker run -d --name pause gcr.io/google_containers/pa... -p 8080:80

0 0 Share ›

Ian Lewis Mod → Jon

6 years ago

All fixed. Make sure to put the -p option before the image name in the command so it's not passed as an argument to the pause process.

0 0 Share ›

Jon → Ian Lewis

6 years ago

Works like a charm, thanks for the quick response. And again, thanks for this post!

0 0 Share ›

Ian Lewis Mod → Jon

6 years ago

My pleasure!

0 0 Share ›

Ian Lewis Mod → Jon

6 years ago

Ah, right, good catch. That's likely a copy and paste error. I'll fix it.

0 0 Share ›

Nic Cope

6 years ago

"Processes can start other processes using the fork or exec syscall. When they do this, the new process' parent is the process that called the fork or exec syscall. fork is used to start another copy of the running process and exec is used to start different process."

I think this wording is a little misleading, exec never creates new processes (i.e. new

I think this wording is a little misleading: exec never creates new processes (i.e. new PIDs or entries in the process table). It instead switches out the context of the existing process to whatever you want to run. So it's more like "Processes can start other processes using the fork and exec syscalls".

0 0 Share ›

Ian Lewis Mod → Nic Cope

6 years ago

Fair enough. I updated the wording. Hopefully it's a little less misleading.

1 0 Share ›

Richard Bown → Ian Lewis

5 years ago

I just read the updated version and still think it's misleading - I came to the same conclusion "huh, exec doesn't create a new process?!"

0 0 Share ›

Ian Lewis Mod → Richard Bown

5 years ago

Yah. I guess I wasn't trying to go to deep into the weeds since the post is mostly geared towards folks who don't use the syscalls directly and I cared mostly about wait but I see now that it's necessary. Re-written with full explanation about what exactly fork and exec do and how a new, totally different process is started.

2 0 Share ›

Vincent De Smet

6 years ago edited

also, if you want a truly offline cluster - use `--pod-infra-container-image string` to control which image is used for the pause container (can host one privately)

0 0 Share ›

Rory McCune

6 years ago

This is a really awesome post, thanks for making it :)

One quick question, in the `docker run` examples where --net is there twice is that