25 为什么说容器是个单进程模型

更新时间：2020-09-11 10:30:08

> 一个人追求的目标越高，他的才力就发展得越快，对社会就越有益。——高尔基

过去两年很多大公司的一个主要技术方向就是将应用上云，在这个过程中的一个典型错误用法就是将容器当成虚拟机来使用，将一堆进程启动在一个容器内。但是容器和虚拟机对进程的管理能力是有着巨大差异的。不管在容器中还是虚拟机中都有一个一号进程，虚拟机中是 systemd 进程，容器中是 entrypoint 启动进程，然后所有的其他线程都是一号进程的子进程，或者子进程的子进程，递归下去。这里的主要差异就体现在 systemd 进程对僵尸进程回收的能力。

1. 僵尸进程

说到僵尸进程，这里简单介绍一下 Linux 系统中的进程状态，我们可以通过 ps 或者 top 等命令查看系统中的进程，比如通过 ps aux 在我的 ecs 虚拟机上面得到如下的输出。

```
[root@emr-header-1 ~]# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0 190992  3568 ?        Ss   Mar16 289:04 /usr/lib/systemd/systemd --switched-root --system --de
root         2  0.0  0.0      0     0 ?        S    Mar16   0:05 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    Mar16  13:01 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S<   Mar16   0:00 [kworker/0:0H]
root         7  0.0  0.0      0     0 ?        S    Mar16  14:41 [migration/0]
root         8  0.0  0.0      0     0 ?        S    Mar16   0:00 [rcu_bh]
root         9  0.0  0.0      0     0 ?        S    Mar16 243:19 [rcu_sched]
root        10  0.0  0.0      0     0 ?        S    Mar16   0:50 [watchdog/0]
root        11  0.0  0.0      0     0 ?        S    Mar16   0:39 [watchdog/1]
root        12  0.0  0.0      0     0 ?        S    Mar16  23:51 [migration/1]
root        13  0.0  0.0      0     0 ?        S    Mar16  15:44 [ksoftirqd/1]
root        15  0.0  0.0      0     0 ?        S<   Mar16   0:00 [kworker/1:0H]
```

我们可以看到排在第一位的就是前面说到的 1 号进程 systemd。其中的 STAT 那一列就是进程状态，这里的状态都是和 S 有关的，但是正常还有 R、D、Z 等状态。各个状态的含义简单描述如下：

- S : Interruptible Sleep，中文可以叫做可中断的睡眠状态，表示进程因为等待某个资源或者事件就绪而被系统暂时挂起。当资源或者事件 Ready 的时候，进程轮转到 R 状态；
- R : 也就是 Running，有时候也可以指代 Runnable，表示进程正在运行或者等待运行；
- Z : Zombie，也就是僵尸进程。我们知道每个进程都是会占用一定的资源的，比如 pid 等，如果进程结束，资源没有被回收就会变成僵尸进程；
- D : Disk Sleep，也就是 Uninterruptible Sleep，不可中断的睡眠状态，一般是进程在等待 IO 等资源，并且不可中断。D 状态相信很多人在实践中第一次接触就是 ps 卡住。D 状态一般在 IO 等资源就绪之后就会轮转到 R 状态，如果进程处于 D 状态比较久，这个时候往往是 IO 出现问题，解决办法大部分情况是重启机器；
- I : Idle，也就是空闲状态，不可中断的睡眠的内核线程。和 D 状态进程的主要区别是可能实际上不会造成负载升高。

关于僵尸进程，这里继续讨论一下。对于正常的使用情况，子进程的创建一般需要父进程通过系统调用 `wait()` 或者 `waitpid()` 来等待子进程结束，从而回收子进程的资源。除了这种方式外，还可以通过异步的方式来进行回收，这种方式的基础是子进程结束之后会向父进程发送 SIGCHLD 信号，基于此父进程注册一个 SIGCHLD 信号的处理函数来进行子进程的资源回收就可以了。记住这两种方式，后面还会涉及到。

僵尸进程的最大危害是对资源的一种永久性占用，比如进程号，系统会有一个最大的进程数 n 的限制，也就意味一旦 1 到 n 进程号都被占用，系统将不能创建任何进程和线程（进程和线程对于 OS 而言，使用同一种数据结构来表示，task_struct）。这个时候对于用户的一个直观感受就是 shell 无法执行任何命令，这个原因是 shell 执行命令的本质是 fork。

```
[root@emr-header-1 ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 63471
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 131070
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 63471
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

2. 孤儿进程

前面说到如果子进程先于父进程退出，并且父进程没有对子进程残留的资源进行回收的话将会产生僵尸进程。这里引申另外一种情况，父进程先于子进程退出的话，那么子进程的资源谁来回收呢？

父进程先于子进程退出，这个时候我们一般将还在运行的子进程称为孤儿进程，但是实际上孤儿进程并没有一个明确的定义，他的状态还是处于上面讨论的几种进程状态中。那么孤儿进程的资源谁来回收呢？类 Unix 系统针对这种情况会将这些孤儿进程的父进程置为 1 号进程也就是 systemd 进程，然后由 systemd 来对孤儿进程的资源进行回收。

3. 单进程模型的本质

看完上面两节大家应该知道了虚拟机或者一个完整的 OS 是如何避免僵尸进程的。但是，在容器中，1 号进程一般是 entry point 进程，针对上面这种 **将孤儿进程的父进程置为 1 号进程进而避免僵尸进程** 处理方式，容器是处理不了的。进而就会导致容器中在孤儿进程这种异常场景下僵尸进程无法彻底处理的窘境。

所以说，容器的单进程模型的本质其实是容器中的 1 号进程并不具有管理多进程、多线程等复杂场景下的能力。如果一定在容器中处理这些复杂情况的，那么需要开发者对 entry point 进程赋予这种能力。这无疑是加重了开发者的心智负担，这是任何一项大众技术或者平台框架都不愿看到的尴尬之地。

4. 如何避免

除了第二节讨论的开发者自己赋予 entrypoint 进程管理多进程的能力，这里我更推荐借助 Kubernetes （下面简称 k8s）来做这件事情。我想现在应该也没有人对容器进行人工管理了，大部分人应该都转向了容器编排和调度工具 k8s 阵营了（对于那些还在使用 swarm 的一小波人，我劝你们早日弃暗投明 :)）。

k8s 中可以将多个容器编排到一个 pod 里面，共享同一个 Linux NameSpace。这项技术的本质是使用 k8s 提供一个 pause 镜像，展开来说就是先用 pause 镜像实例化出 NameSpace，然后其他容器加入这个 NameSpace 从而实现 NameSpace 共享。突然意识到这块需要有容器和 NameSpace 的技术背景，限于篇幅，希望你可以自行搜索这种技术背景。或者我下一篇文章讨论一下容器技术的本质。

言归正传，我们来介绍一下 pause。pause 是 k8s 在 1.10 版本引入的技术，要使用 pause，我们只需要在 pod 创建的 yaml 中指定 shareProcessNamespace 参数为 true，如下。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  shareProcessNamespace: true
  containers:
  - name: nginx
    image: nginx
  - name: shell
    image: busybox
    securityContext:
      capabilities:
        add:
        - SYS_PTRACE
    stdin: true
    tty: true
```

创建 pod。

```
kubectl apply -f share-process-namespace.yaml
```

attach 到 pod 中，ps 查看进程列表。

```
/ # ps ax
PID   USER     TIME  COMMAND
    1 root      0:00 /pause
    8 root      0:00 nginx: master process nginx -g daemon off;
   14 101       0:00 nginx: worker process
   15 root      0:00 sh
   21 root      0:00 ps ax
```

我们可以看到 pod 中的 1 号进程变成了 /pause，其他容器的 entrypoint 进程都变成了 1 号进程的子进程。这个时候开始逐渐逼近事情的本质了：/pause 进程是如何处理 ***将孤儿进程的父进程置为 1 号进程进而避免僵尸进程*** 的呢？我们看一下源码，git repo: pause.c

```c
#define STRINGIFY(x) #x
#define VERSION_STRING(x) STRINGIFY(x)

#ifndef VERSION
#define VERSION HEAD
#endif

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(int argc, char **argv) {
  int i;
  for (i = 1; i < argc; ++i) {
    if (!strcasecmp(argv[i], "-v")) {
      printf("pause.c %s\n", VERSION_STRING(VERSION));
      return 0;
    }
  }

  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;
}
```

重点关注一下 35 行和 13 行，这个不就是我们上面说的：

> 除了这种方式外，还可以通过异步的方式来进行回收，这种方式的基础是子进程结束之后会向父进程发送 SIGCHLD 信号，基于此父进程注册一个 SIGCHLD 信号的处理函数来进行子进程的资源回收就可以了。

SIGCHLD 信号的处理函数核心就是这一行 `while (waitpid(-1, NULL, WNOHANG) > 0)` ，其中 WNOHANG 参数是为了让父进程直接返回不阻塞。

5. 总结

容器化改造的路非常漫长，对于很多业务同学在改造的过程中由于一些思维的惯性就想把容器当成一个虚拟机来使用，这个可能会导致非常多的问题。或许我们可以探究一些容器的设计模式，以便进行更好的实践。

}

24 Docker 最佳实践：tag 如何使用？

26 大话容器设计模式