容器服务

故障处理

产品文档

腾讯云

文档目录

故障处理
磁盘爆满

最近更新时间：2020-04-24 16:17:12

本文档介绍 TKE 集群中多场景下可能发生的磁盘爆满问题，并给出对应的排查思路及解决方案，请按照下文中的步骤进行排查并解决。

可能原因

Kubelet 支持 gc 和驱逐机制，可通过 `--image-gc-high-threshold`、`--image-gc-low-threshold`、`--eviction-hard`、`--eviction-soft` 及 `--eviction-minimum-reclaim` 等参数进行控制以实现磁盘空间的释放。当配置不正确，或者节点上有其它非 K8S 管理的进程在不断写数据到磁盘，将会占用大量空间时将导致磁盘爆满。

磁盘爆满将影响 K8S 运行，主要涉及 kubelet 和容器运行时两个关键组件。请按照以下步骤进行分析：

1. 执行 `df` 命令，查看 kubelet 和容器运行时所使用的目录是否存在于该磁盘。
2. 对应实际结果，选择以下方式进行问题精确定位：
   - 容器运行时使用的目录所在磁盘爆满
   - Kubelet 使用的目录所在磁盘爆满

问题定位及解决思路

">

容器运行时使用的目录所在磁盘爆满

如果容器运行时使用的目录所在磁盘空间爆满，将可能会造成容器运行时无响应。例如，当前容器运行时为 docker，则执行 docker 相关的命令将会一直 hang 住，kubelet 日志也将看到 PLEG unhealthy，而 CRI 调用 timeout 也将导致容器无法创建或销毁，外在现象通常表现为 Pod 一直 ContainerCreating 或一直 Terminating。

docker 默认使用的目录

- `/var/run/docker`：用于存储容器运行状态，通过 dockerd 的 `--exec-root` 参数指定。
- `/var/lib/docker`：用于持久化容器相关的数据。例如，容器镜像、容器可写层数据、容器标准日志输出及通过 docker 创建的 volume 等。

Pod 启动过程事件

Pod 在启动过程中，可能会出现以下类似事件：

```
Warning FailedCreatePodSandBox 53m kubelet, 172.22.0.44 Failed create pod sandbox: rpc error: cod
e = DeadlineExceeded desc = context deadline exceeded
```

```
Warning FailedCreatePodSandBox 2m (x4307 over 16h) kubelet, 10.179.80.31 (combined from similar e
vents): Failed create pod sandbox: rpc error: code = Unknown desc = failed to create a sandbox fo
r pod "apigateway-6dc48bf8b6-l8xrw": Error response from daemon: mkdir /var/lib/docker/aufs/mnt/1
f09d6c1c9f24e8daaea5bf33a4230de7dbc758e3b22785e8ee21e3e3d921214-init: no space left on device
```

```
Warning Failed 5m1s (x3397 over 17h) kubelet, ip-10-0-151-35.us-west-2.compute.internal (combined
from similar events): Error: container create failed: container_linux.go:336: starting container
process caused "process_linux.go:399: container init caused ¥"rootfs_linux.go:58: mounting ¥¥¥"/s
ys¥¥¥" to rootfs ¥¥¥"/var/lib/dockerd/storage/overlay/051e985771cc69f3f699895a1dada9ef6483e912b46
a99e004af7bb4852183eb/merged¥¥¥" at ¥¥¥"/var/lib/dockerd/storage/overlay/051e985771cc69f3f699895a
1dada9ef6483e912b46a99e004af7bb4852183eb/merged/sys¥¥¥" caused ¥¥¥"no space left on device¥¥¥"¥""
```

Pod 删除过程事件

Pod 在删除过程中，可能会出现以下类似事件：

```
Normal Killing 39s (x735 over 15h) kubelet, 10.179.80.31 Killing container with id docker://apiga
teway:Need to kill Pod
```

">

Kubelet 使用的目录所在磁盘爆满

kubelet 默认使用的目录

`/var/lib/kubelet` ：通过 kubelet 的 `--root-dir` 参数指定，用于存储插件信息、Pod 相关的状态以及挂载的 volume（例如， `emptyDir` 、 `ConfigMap` 及 `Secret` ）。

Pod 事件

Kubelet 使用的目录所在磁盘空间爆满（通常是系统盘），新建 Pod 时无法成功进行 mkdir，导致 Sandbox 也无法创建成功，Pod 通常会出现以下类似事件：

```
Warning UnexpectedAdmissionError 44m kubelet, 172.22.0.44 Update plugin resources failed due to f
ailed to write checkpoint file "kubelet_internal_checkpoint": write /var/lib/kubelet/device-plugi
ns/.728425055: no space left on device, which is unexpected.
```

处理步骤

当容器运行时为 docker 时发生磁盘爆满问题，dockerd 也会因此无法正常响应，在停止时会卡住，从而导致无法直接重启 dockerd 来释放空间。需要先手动清理部分文件腾出空间以确保 dockerd 能够停止并重启。恢复步骤如下：

1. 手动删除 docker 的部分 log 文件或可写层文件。通常删除 log 文件，示例如下：

```
$ cd /var/lib/docker/containers
$ du -sh * # 找到比较大的目录
$ cd dda02c9a7491fa797ab730c1568ba06cba74cecd4e4a82e9d90d00fa11de743c
$ cat /dev/null > dda02c9a7491fa797ab730c1568ba06cba74cecd4e4a82e9d90d00fa11de743c-json.log.9
# 删除log文件
```

   - 删除文件时，建议使用 `cat /dev/null >` 方式进行删除，不建议使用 `rm` 。使用 `rm` 方式删除的文件，不能够被 docker 进程释放掉，该文件所占用的空间也就不会被释放。
   - log 的后缀数字越大表示时间越久远，建议优先删除旧日志。

2. 执行以下命令，将该 Node 标记为不可调度，并将其已有的 Pod 驱逐到其它节点。

```
kubectl drain <node-name>
```

   该步骤可以确保 dockerd 重启时将原节点上 Pod 对应的容器删掉，同时确保容器相关的日志（标准输出）与容器内产生的数据文件（未挂载 volume 及可写层）也会被清理。
3. 执行以下命令，重启 dockerd。

```
systemctl restart dockerd
# or systemctl restart docker
```

4. 等待 dockerd 重启恢复，Pod 调度到其它节点后，排查磁盘爆满原因并进行数据清理和规避操作。
5. 执行以下命令，取消节点不可调度标记。

```
kubectl uncordon <node-name>
```

如何规避磁盘爆满？

需确保 kubelet 的 gc 和驱逐相关参数进行配置正确。若配置无问题，即便磁盘达到爆满地步，此时事发节点上的 Pod 也已自动驱逐到其它节点上，不会出现 Pod 一直 ContainerCreating 或 Terminating 的问题。

高负载

最近更新时间：2020-04-24 16:17:12

本文档介绍如何在 TKE 集群中，通过工具定位异常是否由高负载造成，请按照以下步骤进行问题排查。

现象描述

节点高负载将会导致进程无法获得足够运行所需的 CPU 时间片，通常表现为网络 timeout、健康检查失败或服务不可用。

问题定位及解决思路

有时节点在低 cpu ‘us’ (user) 、高 cpu ‘id’ (idle) 的条件下，仍会出现负载很高的情况。通常是由于文件 IO 性能达到瓶颈，导致 IO Wait 过多，使节点整体负载升高，影响其它进程的性能。
本文以 top、atop 及 iotop 工具为例，来判断磁盘 I/O 是否正在降低应用性能。

查看平均负载及等待时间

1. 登录节点，执行 `top` 命令查看当前负载。返回结果如下：

> 可通过较高的 `load average` 值得知该节点正在承接大量的请求，也可以通过 `Cpu(s)` 、 `Mem` 、 `%CPU` 及 `%MEM` 列的数据获取哪些进程正在占用大多数资源。

```
top - 19:42:06 up 23:59, 2 users, load average: 34.64, 35.80, 35.76
Tasks: 679 total, 1 running, 678 sleeping, 0 stopped, 0 zombie
Cpu(s): 15.6%us, 1.7%sy, 0.0%ni, 74.7%id, 7.9%wa, 0.0%hi, 0.1%si, 0.0%st
Mem: 32865032k total, 30989168k used, 1875864k free, 370748k buffers
Swap: 8388604k total, 5440k used, 8383164k free, 7982424k cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
9783 mysql 20 0 17.3g 16g 8104 S 186.9 52.3 3752:33 mysqld
5700 nginx 20 0 1330m 66m 9496 S 8.9 0.2 0:20.82 php-fpm
6424 nginx 20 0 1330m 65m 8372 S 8.3 0.2 0:04.97 php-fpm
6573 nginx 20 0 1330m 64m 7368 S 8.3 0.2 0:01.49 php-fpm
5927 nginx 20 0 1320m 56m 9272 S 7.6 0.2 0:12.54 php-fpm
5956 nginx 20 0 1330m 65m 8500 S 7.6 0.2 0:12.70 php-fpm
6126 nginx 20 0 1321m 57m 8964 S 7.3 0.2 0:09.72 php-fpm
```

```
6127 nginx 20 0 1319m 54m 9520 S 6.6 0.2 0:08.73 php-fpm
6131 nginx 20 0 1320m 56m 9404 S 6.6 0.2 0:09.43 php-fpm
6174 nginx 20 0 1321m 56m 8444 S 6.3 0.2 0:08.92 php-fpm
5790 nginx 20 0 1319m 54m 9468 S 5.6 0.2 0:17.33 php-fpm
6575 nginx 20 0 1320m 55m 8212 S 5.6 0.2 0:02.11 php-fpm
6160 nginx 20 0 1310m 44m 8296 S 4.0 0.1 0:10.05 php-fpm
5597 nginx 20 0 1310m 46m 9556 S 3.6 0.1 0:21.03 php-fpm
5786 nginx 20 0 1310m 45m 8528 S 3.6 0.1 0:15.53 php-fpm
5797 nginx 20 0 1310m 46m 9444 S 3.6 0.1 0:14.02 php-fpm
6158 nginx 20 0 1310m 45m 8324 S 3.6 0.1 0:10.20 php-fpm
5698 nginx 20 0 1310m 46m 9184 S 3.3 0.1 0:20.62 php-fpm
5779 nginx 20 0 1309m 44m 8336 S 3.3 0.1 0:15.34 php-fpm
6540 nginx 20 0 1306m 40m 7884 S 3.3 0.1 0:02.46 php-fpm
5553 nginx 20 0 1300m 36m 9568 S 3.0 0.1 0:21.58 php-fpm
5722 nginx 20 0 1310m 45m 8552 S 3.0 0.1 0:17.25 php-fpm
5920 nginx 20 0 1302m 36m 8208 S 3.0 0.1 0:14.23 php-fpm
6432 nginx 20 0 1310m 45m 8420 S 3.0 0.1 0:05.86 php-fpm
5285 nginx 20 0 1302m 38m 9696 S 2.7 0.1 0:23.41 php-fpm
```

2. 在返回结果界面，可查看核的 `wa` 值， `wa` （wait）表示 IO WAIT 的 CPU 占用。默认显示所有核的平均值，按**1**查看每个核的 `wa` 值。如下所示：

> `wa` 通常为0%，如果经常浮动在1%之上，说明存储设备的速度已经太慢，无法跟上 CPU 的处理速度。

```
top - 19:42:08 up 23:59, 2 users, load average: 34.64, 35.80, 35.76
Tasks: 679 total, 1 running, 678 sleeping, 0 stopped, 0 zombie
Cpu0 : 29.5%us, 3.7%sy, 0.0%ni, 48.7%id, 17.9%wa, 0.0%hi, 0.1%si, 0.0%st
Cpu1 : 29.3%us, 3.7%sy, 0.0%ni, 48.9%id, 17.9%wa, 0.0%hi, 0.1%si, 0.0%st
Cpu2 : 26.1%us, 3.1%sy, 0.0%ni, 64.4%id, 6.0%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu3 : 25.9%us, 3.1%sy, 0.0%ni, 65.5%id, 5.4%wa, 0.0%hi, 0.1%si, 0.0%st
Cpu4 : 24.9%us, 3.0%sy, 0.0%ni, 66.8%id, 5.0%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu5 : 24.9%us, 2.9%sy, 0.0%ni, 67.0%id, 4.8%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu6 : 24.2%us, 2.7%sy, 0.0%ni, 68.3%id, 4.5%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu7 : 24.3%us, 2.6%sy, 0.0%ni, 68.5%id, 4.2%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu8 : 23.8%us, 2.6%sy, 0.0%ni, 69.2%id, 4.1%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu9 : 23.9%us, 2.5%sy, 0.0%ni, 69.3%id, 4.0%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu10 : 23.3%us, 2.4%sy, 0.0%ni, 68.7%id, 5.6%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu11 : 23.3%us, 2.4%sy, 0.0%ni, 69.2%id, 5.1%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu12 : 21.8%us, 2.4%sy, 0.0%ni, 60.2%id, 15.5%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu13 : 21.9%us, 2.4%sy, 0.0%ni, 60.6%id, 15.2%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu14 : 21.4%us, 2.3%sy, 0.0%ni, 72.6%id, 3.7%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu15 : 21.5%us, 2.2%sy, 0.0%ni, 73.2%id, 3.1%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu16 : 21.2%us, 2.2%sy, 0.0%ni, 73.6%id, 3.0%wa, 0.0%hi, 0.0%si, 0.0%st
```

```
Cpu17 : 21.2%us, 2.1%sy, 0.0%ni, 73.8%id, 2.8%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu18 : 20.9%us, 2.1%sy, 0.0%ni, 74.1%id, 2.9%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu19 : 21.0%us, 2.1%sy, 0.0%ni, 74.4%id, 2.5%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu20 : 20.7%us, 2.0%sy, 0.0%ni, 73.8%id, 3.4%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu21 : 20.8%us, 2.0%sy, 0.0%ni, 73.9%id, 3.2%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu22 : 20.8%us, 2.0%sy, 0.0%ni, 74.4%id, 2.8%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu23 : 20.8%us, 1.9%sy, 0.0%ni, 74.4%id, 2.8%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 32865032k total, 30209248k used, 2655784k free, 370748k buffers
Swap: 8388604k total, 5440k used, 8383164k free, 7986552k cached
```

监视磁盘 IO 统计信息

1. 执行命令 `atop` ，查看当前磁盘 IO 状态。本例中，磁盘 `sda` 显示 `busy 100%` ，表示已达到严重性能瓶颈。

```
ATOP - lemp 2017/01/23 19:42:32 ---------- 10s elapsed
PRC | sys 3.18s | user 33.24s | #proc 679 | #tslpu 28 | #zombie 0 | #exit 0 |
CPU | sys 29% | user 330% | irq 1% | idle 1857% | wait 182% | curscal 69% |
CPL | avg1 33.00 | avg5 35.29 | avg15 35.59 | csw 62610 | intr 76926 | numcpu 24 |
MEM | tot 31.3G | free 2.1G | cache 7.6G | dirty 41.0M | buff 362.1M | slab 1.2G |
SWP | tot 8.0G | free 8.0G | | | vmcom 23.9G | vmlim 23.7G |
DSK | sda | busy 100% | read 4 | write 1789 | MBw/s 2.84 | avio 5.58 ms |
NET | transport | tcpi 10357 | tcpo 9065 | udpi 0 | udpo 0 | tcpao 174 |
NET | network | ipi 10360 | ipo 9065 | ipfrw 0 | deliv 10359 | icmpo 0 |
NET | eth0 4% | pcki 6649 | pcko 6136 | si 1478 Kbps | so 4115 Kbps | erro 0 |
NET | lo ---- | pcki 4082 | pcko 4082 | si 8967 Kbps | so 8967 Kbps | erro 0 |

PID TID THR SYSCPU USRCPU VGROW RGROW RDDSK WRDSK ST EXC S CPUNR CPU CMD 1/12
9783 - 156 0.21s 19.44s 0K -788K 4K 1344K -- - S 4 197% mysqld
5596 - 1 0.10s 0.62s 47204K 47004K 0K 220K -- - S 18 7% php-fpm
6429 - 1 0.06s 0.34s 19840K 19968K 0K 0K -- - S 21 4% php-fpm
6210 - 1 0.03s 0.30s -5216K -5204K 0K 0K -- - S 19 3% php-fpm
5757 - 1 0.05s 0.27s 26072K 26012K 0K 4K -- - S 13 3% php-fpm
6433 - 1 0.04s 0.28s -2816K -2816K 0K 0K -- - S 11 3% php-fpm
5846 - 1 0.06s 0.22s -2560K -2660K 0K 0K -- - S 7 3% php-fpm
5791 - 1 0.05s 0.21s 5764K 5692K 0K 0K -- - S 22 3% php-fpm
5860 - 1 0.04s 0.21s 48088K 47724K 0K 0K -- - S 1 3% php-fpm
6231 - 1 0.04s 0.20s -256K -4K 0K 0K -- - S 1 2% php-fpm
6154 - 1 0.03s 0.21s -3004K -3184K 0K 0K -- - S 21 2% php-fpm
6573 - 1 0.04s 0.20s -512K -168K 0K 0K -- - S 4 2% php-fpm
6435 - 1 0.04s 0.19s -3216K -2980K 0K 0K -- - S 15 2% php-fpm
5954 - 1 0.03s 0.20s 0K 164K 0K 4K -- - S 0 2% php-fpm
6133 - 1 0.03s 0.19s 41056K 40432K 0K 0K -- - S 18 2% php-fpm
6132 - 1 0.02s 0.20s 37836K 37440K 0K 0K -- - S 11 2% php-fpm
6242 - 1 0.03s 0.19s -12.2M -12.3M 0K 4K -- - S 12 2% php-fpm
6285 - 1 0.02s 0.19s 39516K 39420K 0K 0K -- - S 3 2% php-fpm
6455 - 1 0.05s 0.16s 29008K 28560K 0K 0K -- - S 14 2% php-fpm
```

2. 查看占用磁盘 IO 的进程，有如下两种方法：

- 继续在该界面按 **d**，可查看哪些进程正在使用磁盘 IO。返回结果如下：

```
ATOP - lemp 2017/01/23 19:42:46 --------- 2s elapsed
PRC | sys 0.24s | user 1.99s | #proc 679 | #tslpu 54 | #zombie 0 | #exit 0 |
CPU | sys 11% | user 101% | irq 1% | idle 2089% | wait 208% | curscal 63% |
CPL | avg1 38.49 | avg5 36.48 | avg15 35.98 | csw 4654 | intr 6876 | numcpu 24 |
MEM | tot 31.3G | free 2.2G | cache 7.6G | dirty 48.7M | buff 362.1M | slab 1.2G |
SWP | tot 8.0G | free 8.0G | | | vmcom 23.9G | vmlim 23.7G |
DSK | sda | busy 100% | read 2 | write 362 | MBw/s 2.28 | avio 5.49 ms |
NET | transport | tcpi 1031 | tcpo 968 | udpi 0 | udpo 0 | tcpao 45 |
NET | network | ipi 1031 | ipo 968 | ipfrw 0 | deliv 1031 | icmpo 0 |
NET | eth0 1% | pcki 558 | pcko 508 | si 762 Kbps | so 1077 Kbps | erro 0 |
NET | lo ---- | pcki 406 | pcko 406 | si 2273 Kbps | so 2273 Kbps | erro 0 |

PID TID RDDSK WRDSK WCANCL DSK CMD 1/5
9783 - 0K 468K 16K 40% mysqld
1930 - 0K 212K 0K 18% flush-8:0
5896 - 0K 152K 0K 13% nginx
880 - 0K 148K 0K 13% jbd2/sda5-8
5909 - 0K 60K 0K 5% nginx
5906 - 0K 36K 0K 3% nginx
5907 - 16K 8K 0K 2% nginx
5903 - 20K 0K 0K 2% nginx
5901 - 0K 12K 0K 1% nginx
5908 - 0K 8K 0K 1% nginx
5894 - 0K 8K 0K 1% nginx
5911 - 0K 8K 0K 1% nginx
5900 - 0K 4K 4K 0% nginx
5551 - 0K 4K 0K 0% php-fpm
5913 - 0K 4K 0K 0% nginx
5895 - 0K 4K 0K 0% nginx
6133 - 0K 0K 0K 0% php-fpm
5780 - 0K 0K 0K 0% php-fpm
6675 - 0K 0K 0K 0% atop
```

- 也可以执行命令 `iotop -oPa` 查看哪些进程占用磁盘 IO。返回结果如下：

```
Total DISK READ: 15.02 K/s | Total DISK WRITE: 3.82 M/s
PID PRIO USER DISK READ DISK WRITE SWAPIN IO> COMMAND
1930 be/4 root 0.00 B 1956.00 K 0.00 % 83.34 % [flush-8:0]
5914 be/4 nginx 0.00 B 0.00 B 0.00 % 36.56 % nginx: cache manager process
```

```
880 be/3 root 0.00 B 21.27 M 0.00 % 35.03 % [jbd2/sda5-8]
5913 be/2 nginx 36.00 K 1000.00 K 0.00 % 8.94 % nginx: worker process
5910 be/2 nginx 0.00 B 1048.00 K 0.00 % 8.43 % nginx: worker process
5896 be/2 nginx 56.00 K 452.00 K 0.00 % 6.91 % nginx: worker process
5909 be/2 nginx 20.00 K 1144.00 K 0.00 % 6.24 % nginx: worker process
5890 be/2 nginx 48.00 K 692.00 K 0.00 % 6.07 % nginx: worker process
5892 be/2 nginx 84.00 K 736.00 K 0.00 % 5.71 % nginx: worker process
5901 be/2 nginx 20.00 K 504.00 K 0.00 % 5.46 % nginx: worker process
5899 be/2 nginx 0.00 B 596.00 K 0.00 % 5.14 % nginx: worker process
5897 be/2 nginx 28.00 K 1388.00 K 0.00 % 4.90 % nginx: worker process
5908 be/2 nginx 48.00 K 700.00 K 0.00 % 4.43 % nginx: worker process
5905 be/2 nginx 32.00 K 1140.00 K 0.00 % 4.36 % nginx: worker process
5900 be/2 nginx 0.00 B 1208.00 K 0.00 % 4.31 % nginx: worker process
5904 be/2 nginx 36.00 K 1244.00 K 0.00 % 2.80 % nginx: worker process
5895 be/2 nginx 16.00 K 780.00 K 0.00 % 2.50 % nginx: worker process
5907 be/2 nginx 0.00 B 1548.00 K 0.00 % 2.43 % nginx: worker process
5903 be/2 nginx 36.00 K 1032.00 K 0.00 % 2.34 % nginx: worker process
6130 be/4 nginx 0.00 B 72.00 K 0.00 % 2.18 % php-fpm: pool www
5906 be/2 nginx 12.00 K 844.00 K 0.00 % 2.10 % nginx: worker process
5889 be/2 nginx 40.00 K 1164.00 K 0.00 % 2.00 % nginx: worker process
5894 be/2 nginx 44.00 K 760.00 K 0.00 % 1.61 % nginx: worker process
5902 be/2 nginx 52.00 K 992.00 K 0.00 % 1.55 % nginx: worker process
5893 be/2 nginx 64.00 K 972.00 K 0.00 % 1.22 % nginx: worker process
5814 be/4 nginx 36.00 K 44.00 K 0.00 % 1.06 % php-fpm: pool www
6159 be/4 nginx 4.00 K 4.00 K 0.00 % 1.00 % php-fpm: pool www
5693 be/4 nginx 0.00 B 4.00 K 0.00 % 0.86 % php-fpm: pool www
5912 be/2 nginx 68.00 K 300.00 K 0.00 % 0.72 % nginx: worker process
5911 be/2 nginx 20.00 K 788.00 K 0.00 % 0.72 % nginx: worker process
```

通过 `man iotop` 命令可以查看以下几个参数的含义。返回结果如下：

```
-o, --only
Only show processes or threads actually doing I/O, instead of showing all processes or threa
ds. This can be dynamically toggled by pressing o.
-P, --processes
Only show processes. Normally iotop shows all threads.

-a, --accumulated
Show accumulated I/O instead of bandwidth. In this mode, iotop shows the amount of I/O proce
sses have done since iotop started.
```

其他原因

节点上部署了其他非 K8S 管理的服务可能造成节点高负载问题。例如，在节点上安装不被 K8S 所管理的数据库。

内存碎片化

最近更新时间：2020-04-24 16:17:12

本文档介绍如何判断 TKE 集群中存在问题是否由内存碎片化引起，并给出解决方法，请按照以下步骤进行排查并解决。

问题分析

内存页分配失败，内核日志将会出现以下报错：

```
mysqld: page allocation failure. order:4, mode:0x10c0d0
```

- `mysqld` ：为被分配内存的程序。
- `order` ：表示需要分配连续页的数量（2^order），本例中4则表示 2^4 = 16 个连续的页。
- `mode` ：内存分配模式的标识，在内核源码文件 `include/linux/gfp.h` 中定义，通常是多个标识项与运算的结果。不同版本内核有一定区别。例如。在新版内核中 `GFP_KERNEL` 是 `__GFP_RECLAIM | __GFP_IO | __GFP_FS` 的运算结果，而 `__GFP_RECLAIM` 是 `___GFP_DIRECT_RECLAIM|___GFP_KSWAPD_RECLAIM` 的运算结果。

  - 当 order 值为0时，说明系统已经完全没有可用内存。
  - 当 order 值较大时，说明内存已碎片化，无法分配连续的大页内存。

现象描述

容器启动失败

K8S 会为每个 Pod 创建 netns 来隔离 network namespace，内核初始化 netns 时会为其创建 nf_conntrack 表的 cache，需要申请大页内存，如果此时系统内存已经碎片化，无法分配到足够的大页内存内核就会出现如下报错（ `v2.6.33 - v4.6` ）：

```
runc:[1:CHILD]: page allocation failure: order:6, mode:0x10c0d0
```

Pod 将会一直处于 ContainerCreating 状态，dockerd 启动容器也将失败。日志报错如下：

```
Jan 23 14:15:31 dc05 dockerd: time="2019-01-23T14:15:31.288446233+08:00" level=error msg="containerd: start container" error="oci runtime error: container_linux.go:247: starting container process caused ¥"process_linux.go:245: running exec setns process for init caused ¥¥¥"exit status 6¥¥¥"¥"¥n" id=5b9be8c5bb121264899fac8d9d36b02150269d41ce96ba6ad36d70b8640cb01c
```

```
Jan 23 14:15:31 dc05 dockerd: time="2019-01-23T14:15:31.317965799+08:00" level=error msg="Create container failed with error: invalid header field value ¥"oci runtime error: container_linux.go:247: starting container process caused ¥¥¥"process_linux.go:245: running exec setns process for init caused ¥¥¥¥¥¥¥"exit status 6¥¥¥¥¥¥¥"¥¥¥"¥¥n¥""
```

Kubelet 日志报错如下：

```
Jan 23 14:15:31 dc05 kubelet: E0123 14:15:31.352386 26037 remote_runtime.go:91] RunPodSandbox from runtime service failed: rpc error: code = 2 desc = failed to start sandbox container for pod "matchdataserver-1255064836-t4b2w": Error response from daemon: {"message":"invalid header field value ¥"oci runtime error: container_linux.go:247: starting container process caused ¥¥¥"process_linux.go:245: running exec setns process for init caused ¥¥¥¥¥¥¥"exit status 6¥¥¥¥¥¥¥"¥¥¥"¥¥n¥""}
Jan 23 14:15:31 dc05 kubelet: E0123 14:15:31.352496 26037 kuberuntime_sandbox.go:54] CreatePodSandbox for pod "matchdataserver-1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)" failed: rpc error: code = 2 desc = failed to start sandbox container for pod "matchdataserver-1255064836-t4b2w": Error response from daemon: {"message":"invalid header field value ¥"oci runtime error: container_linux.go:247: starting container process caused ¥¥¥"process_linux.go:245: running exec setns process for init caused ¥¥¥¥¥¥¥"exit status 6¥¥¥¥¥¥¥"¥¥¥"¥¥n¥""}
Jan 23 14:15:31 dc05 kubelet: E0123 14:15:31.352518 26037 kuberuntime_manager.go:618] createPodSandbox for pod "matchdataserver-1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)" failed: rpc error: code = 2 desc = failed to start sandbox container for pod "matchdataserver-1255064836-t4b2w": Error response from daemon: {"message":"invalid header field value ¥"oci runtime error: container_linux.go:247: starting container process caused ¥¥¥"process_linux.go:245: running exec setns process for init caused ¥¥¥¥¥¥¥"exit status 6¥¥¥¥¥¥¥"¥¥¥"¥¥n¥""}
Jan 23 14:15:31 dc05 kubelet: E0123 14:15:31.352580 26037 pod_workers.go:182] Error syncing pod 485fd485-1ed6-11e9-8661-0a587f8021ea ("matchdataserver-1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)"), skipping: failed to "CreatePodSandbox" for "matchdataserver-1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)" with CreatePodSandboxError: "CreatePodSandbox for pod ¥"matchdataserver-1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)¥" failed: rpc error: code = 2 desc = failed to start sandbox container for pod ¥"matchdataserver-1255064836-t4b2w¥": Error response from daemon: {¥"message¥":¥"invalid header field value ¥¥¥"oci runtime error: container_linux.go:247: starting container process caused ¥¥¥¥¥¥¥"process_linux.go:245: running exec setns process for init caused ¥¥¥¥¥¥¥¥¥¥¥¥¥¥¥"exit status 6¥¥¥¥¥¥¥¥¥¥¥¥¥¥¥"¥¥¥¥¥¥¥"¥¥¥¥n¥¥¥"¥"}"
Jan 23 14:15:31 dc05 kubelet: I0123 14:15:31.372181 26037 kubelet.go:1916] SyncLoop (PLEG): "matchdataserver-1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)", event: &pleg.PodLifecycleEvent{ID:"485fd485-1ed6-11e9-8661-0a587f8021ea", Type:"ContainerDied", Data:"5b9be8c5bb121264899fac8d9d36b02150269d41ce96ba6ad36d70b8640cb01c"}
Jan 23 14:15:31 dc05 kubelet: W0123 14:15:31.372225 26037 pod_container_deletor.go:77] Container "5b9be8c5bb121264899fac8d9d36b02150269d41ce96ba6ad36d70b8640cb01c" not found in pod's containers
Jan 23 14:15:31 dc05 kubelet: I0123 14:15:31.678211 26037 kuberuntime_manager.go:383] No ready sandbox for pod "matchdataserver-1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)" can be found. Need to start a new one
```

执行命令 `cat /proc/buddyinfo` 查看 slab，系统不存在大块内存时返回0数量较多。如下所示：

```
$ cat /proc/buddyinfo
Node 0, zone DMA 1 0 1 0 2 1 1 0 1 1 3
Node 0, zone DMA32 2725 624 489 178 0 0 0 0 0 0 0
Node 0, zone Normal 1163 1101 932 222 0 0 0 0 0 0 0
```

系统 OOM

内存碎片化严重，系统缺少大页内存，即使是当前系统总内存较多的情况下，也会出现给程序分配内存失败的现象。此时系统会做出内存不够用的误判，认为需要停止一些进程来释放内存，从而导致系统 OOM。

处理步骤

1. 周期性地或者在发现大块内存不足时，先进行 drop_cache 操作。

```
echo 3 > /proc/sys/vm/drop_caches
```

2. 必要时执行以下操作进行内存整理。

> 请谨慎执行此步骤，该操作开销较大，会造成业务卡顿。

```
echo 1 > /proc/sys/vm/compact_memory
```

排错方法

使用 Systemtap 定位 Pod 异常退出原因

最近更新时间：2020-04-24 16:17:12

本文介绍如何使用 Systemtap 工具定位 Pod 异常问题原因。

准备工作

请对应您使用节点的操作系统，按照以下步骤进行相关软件包安装：

Ubuntu 操作系统

1. 执行以下命令，安装 Systemtap。

```
apt install -y systemtap
```

2. 执行以下命令，检查所需待安装项。

```
stap-prep
```

返回示例结果如下：

```
Please install linux-headers-4.4.0-104-generic
You need package linux-image-4.4.0-104-generic-dbgsym but it does not seem to be available
Ubuntu -dbgsym packages are typically in a separate repository
Follow https://wiki.ubuntu.com/DebuggingProgramCrash to add this repository
apt install -y linux-headers-4.4.0-104-generic
```

3. 根据上述返回结果可知目前需要 dbgsym 包，但已有软件源中并不包含，需使用第三方软件源安装。 请执行以下命令，安装 dbgsym。

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys C8CAB6595FDFF622

codename=$(lsb_release -c | awk '{print $2}')
sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ ${codename} main restricted universe multiverse
deb http://ddebs.ubuntu.com/ ${codename}-security main restricted universe multiverse
```

```
deb http://ddebs.ubuntu.com/ ${codename}-updates main restricted universe multiverse
deb http://ddebs.ubuntu.com/ ${codename}-proposed main restricted universe multiverse
EOF

sudo apt-get update
```

4. 配置好软件源后，再次运行以下命令。

```
stap-prep
```

返回示例结果如下：

```
Please install linux-headers-4.4.0-104-generic
Please install linux-image-4.4.0-104-generic-dbgsym
```

5. 依次执行以下命令，安装提示所需的包。

```
apt install -y linux-image-4.4.0-104-generic-dbgsym
```

```
apt install -y linux-headers-4.4.0-104-generic
```

CentOS 操作系统

1. 执行以下命令，安装 Systemtap。

```
yum install -y systemtap
```

2. 向软件源 `/etc/yum.repos.d/CentOS-Debug.repo` 配置文件中输入以下内容并保存，本文以默认未安装 `debuginfo` 为例。

```
[debuginfo]
name=CentOS-$releasever - DebugInfo
baseurl=http://debuginfo.centos.org/$releasever/$basearch/
gpgcheck=0
enabled=1
protect=1
priority=1
```

3. 执行以下命令检查所需待安装项，并进行相关项的安装。

该命令将会安装 `kernel-debuginfo` 。

```
stap-prep
```

4. 执行以下命令，检查节点是否已安装多个版本 `kernel-devel` 。

```
rpm -qa | grep kernel-devel
```

返回结果如下：

```
kernel-devel-3.10.0-327.el7.x86_64
kernel-devel-3.10.0-514.26.2.el7.x86_64
kernel-devel-3.10.0-862.9.1.el7.x86_64
```

若存在多个版本，则只需保留与当前内核版本相同的版本。假设当前内核版本为 `3.10.0-862.9.1.el7.x86_64` ，结合上述返回结果需执行以下命令删除多余版本。

- 可通过执行 `uname -r` 命令查看内核版本。
- 需确保 `kernel-debuginfo` 和 `kernel-devel` 均已安装并且安装版本与当前内核版本相同。

```
rpm -e kernel-devel-3.10.0-327.el7.x86_64 kernel-devel-3.10.0-514.26.2.el7.x86_64
```

问题分析

若 Pod 因异常被停止时，可以使用 Systemtap 来监视进程的信号发送进行问题定位。该过程工作原理如下：

1. Systemtap 将脚本转换为 C 语言代码，调用 gcc 将其编译成 Linux 内核模块并通过 `modprobe` 加载到内核。
2. 根据脚本内容在内核进行 hook。此时即可通过 hook 信号的发送，找出容器进程停止的真正原因。

问题定位

步骤1：获取异常 Pod 中重新自动重启的容器 pid

1. 执行以下命令，获取容器 ID。

```
kubectl describe pod <pod name>
```

返回结果如下：

```
......
Container ID: docker://5fb8adf9ee62afc6d3f6f3d9590041818750b392dff015d7091eaaf99cf1c945
......
Last State: Terminated
Reason: Error
Exit Code: 137
Started: Thu, 05 Sep 2019 19:22:30 +0800
Finished: Thu, 05 Sep 2019 19:33:44 +0800
```

2. 执行以下命令，通过获取到的 Container ID 反查容器主进程的 pid。

```
docker inspect -f "{{.State.Pid}}" 5fb8adf9ee62afc6d3f6f3d9590041818750b392dff015d7091eaaf99cf1c945
```

返回结果如下：

```
7942
```

步骤2：根据容器退出状态码缩小排查范围

通过步骤1中返回信息中的 `Exit Code` 可以获取容器上次退出的状态码。本文以137为例，进行以下分析：

- 如果进程是被外界中断信号停止的，则退出状态码将在129 - 255之间。
- 退出状态码为137则表示进程是被 `SIGKILL` 信号停止的，但此时仍不能确定进程停止原因。

步骤3：使用 Systemtap 脚本定位异常原因

假设引发异常的问题可复现，则可以使用 Systemtap 脚本监视容器停止原因。

1. 创建 `sg.stp` 文件，输入以下 Systemtap 脚本内容并保存。

```
global target_pid = 7942
probe signal.send{
if (sig_pid == target_pid) {
printf("%s(%d) send %s to %s(%d)¥n", execname(), pid(), sig_name, pid_name, sig_pid);
printf("parent of sender: %s(%d)¥n", pexecname(), ppid())
printf("task_ancestry:%s¥n", task_ancestry(pid2task(pid()), 1));
}
}
```

> 变量 `pid` 的值需替换为 步骤2 中获取的容器主进程 pid，本为以7942为例。

2. 执行以下命令，运行脚本。

```
stap sg.stp
```

当容器进程因异常停止时，脚本可捕捉到事件，并执行输出如下：

```
pkill(23549) send SIGKILL to server(7942)
parent of sender: bash(23495)
task_ancestry:swapper/0(0m0.000000000s)=>systemd(0m0.080000000s)=>vGhyM0(19491m2.579563677s)=>
sh(33473m38.074571885s)=>bash(33473m38.077072025s)=>bash(33473m38.081028267s)=>bash(33475m4.81
7798337s)=>pkill(33475m5.202486630s)
```

解决方法

通过观察 `task_ancestry` ，可以看到停止进程的所有父进程。例如，此处可以看到一个名为 `vGhyM0` 的异常进程，该现象通常表明程序中存在木马病毒，需要进行相关病毒清理工作以确保容器正常运行。

通过 Exit Code 定位 Pod 异常退出原因

最近更新时间：2020-09-23 18:28:14

本文介绍如何根据 Pod 异常状态信息中的 Exit Code 进一步定位问题。

查看 Pod 异常状态信息

执行以下命令，查看异常 Pod 状态信息。

```
kubectl describe pod <pod name>
```

返回结果如下：

```
Containers:
kubedns:
Container ID: docker://5fb8adf9ee62afc6d3f6f3d9590041818750b392dff015d7091eaaf99cf1c945
Image: ccr.ccs.tencentyun.com/library/kubedns-amd64:1.14.4
Image ID: docker-pullable://ccr.ccs.tencentyun.com/library/kubedns-amd64@sha256:40790881bbe9ef4ae
4ff7fe8b892498eecb7fe6dcc22661402f271e03f7de344
Ports: 10053/UDP, 10053/TCP, 10055/TCP
Host Ports: 0/UDP, 0/TCP, 0/TCP
Args:
--domain=cluster.local.
--dns-port=10053
--config-dir=/kube-dns-config
--v=2
State: Running
Started: Tue, 27 Aug 2019 10:58:49 +0800
Last State: Terminated
Reason: Error
Exit Code: 255
Started: Tue, 27 Aug 2019 10:40:42 +0800
Finished: Tue, 27 Aug 2019 10:58:27 +0800
Ready: True
Restart Count: 1
```

在返回结果的容器列表 `Last State` 字段中， `Exit Code` 为程序上次退出时的状态码，该值不为0即表示程序异常退出，可根据退出状态码进一步分析异常原因。

退出状态码说明

- 状态码需在0 - 255之间。
- 0表示正常退出。
- 若因外界中断导致程序退出，则状态码区间为129 - 255。例如，操作系统给程序发送中断信号 `kill -9` 或 `ctrl+c`，导致程序状态变为 `SIGKILL` 或 `SIGINT`。
- 通常因程序自身原因导致的异常退出，状态码区间在1 - 128。在某些场景下，也允许程序设置使用129 - 255区间的状态码。
- 若指定的退出状态码不在0 - 255之间（例如，设置 `exit(-1)`），此时将会自动执行转换，最终呈现的状态码仍会在0 - 255之间。
  若将退出时状态码记为 `code`，则不同情况下转换方式如下：
  - 当指定的退出时状态码为负数，转换公式为：
    ```
    256 - (|code| % 256)
    ```
  - 当指定的退出时状态码为正数，转换公式为：
    ```
    code % 256
    ```

常见异常状态码

- **137**：表示程序被 `SIGKILL` 中断信号杀死。异常原因可能为：
  - 通常是由于 Pod 中容器内存达到了其资源限制（`resources.limits`）。例如，内存溢出（OOM）。由于资源限制是通过 Linux 的 cgroup 实现的，当某个容器内存达到资源限制， cgroup 就会将其强制停止（类似于 `kill -9`），此时通过 `describe pod` 可以看到 Reason 是 `OOMKilled`。
  - 宿主机本身资源不够用（OOM），则内核会选择停止一些进程来释放内存。

    说明：

    无论是 cgroup 限制，还是因为节点机器本身资源不够导致的进程停止，都可以从系统日志中找到记录。方法如下：
    Ubuntu 系统日志存储在目录 `/var/log/syslog`，CentOS 系统日志存储在目录 `/var/log/messages` 中，两者系统日志均可通过 `journalctl -k` 命令进行查看。

  - livenessProbe（存活检查）失败，使得 kubelet 停止 Pod。
  - 被恶意木马进程停止。
- **1和255**：通常表示一般错误，具体原因需要通过容器日志进一步定位。例如，可能是设置异常退出使用 `exit(1)` 或 `exit(-1)` 导致的，而-1将会根据规则转换成255。

Linux 标准中断信号

Linux 程序被外界中断时会发送中断信号，程序退出时的状态码为中断信号值加128。例如， `SIGKILL` 的中断信号值为9，那么程序退出状态码则为9 + 128 = 137。更多标准信号值参考如下表：

信号 Signal	状态码 Value	动作 Action	描述 Comment
`SIGHUP`	1	Term	Hangup detected on controlling terminal or death of controlling process
`SIGINT`	2	Term	Interrupt from keyboard
`SIGQUIT`	3	Core	Quit from keyboard
`SIGILL`	4	Core	Illegal Instruction
`SIGABRT`	6	Core	Abort signal from abort(3)
`SIGFPE`	8	Core	Floating-point exception
`SIGKILL`	9	Term	Kill signal
`SIGSEGV`	11	Core	Invalid memory reference
`SIGPIPE`	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
`SIGALRM`	14	Term	Timer signal from alarm(2)
`SIGTERM`	15	Term	Termination signal
`SIGUSR1`	30,10,16	Term	User-defined signal 1
`SIGUSR2`	31,12,17	Term	User-defined signal 2
`SIGCHLD`	20,17,18	Ign	Child stopped or terminated
`SIGCONT`	19,18,25	Cont	Continue if stopped
`SIGSTOP`	17,19,23	Stop	Stop process
`SIGTSTP`	18,20,24	Stop	Stop typed at terminal
`SIGTTIN`	21,21,26	Stop	Terminal input for background process
`SIGTTOU`	22,22,27	Stop	Terminal output for background process

C/C++ 退出状态码

`/usr/include/sysexits.h` 中进行了退出状态码标准化（仅限 C/C++），如下表：

定义	状态码	描述
`#define EX_OK`	0	successful termination
`#define EX__BASE`	64	base value for error messages
`#define EX_USAGE`	64	command line usage error
`#define EX_DATAERR`	65	data format error
`#define EX_NOINPUT`	66	cannot open input
`#define EX_NOUSER`	67	addressee unknown
`#define EX_NOHOST`	68	host name unknown
`#define EX_UNAVAILABLE`	69	service unavailable
`#define EX_SOFTWARE`	70	internal software error
`#define EX_OSERR`	71	system error (e.g., can't fork)
`#define EX_OSFILE`	72	critical OS file missing
`#define EX_CANTCREAT`	73	can't create (user) output file
`#define EX_IOERR`	74	input/output error
`#define EX_TEMPFAIL`	75	temp failure; user is invited to retry
`#define EX_PROTOCOL`	76	remote error in protocol
`#define EX_NOPERM`	77	permission denied
`#define EX_CONFIG`	78	configuration error
`#define EX__MAX 78`	78	maximum listed value

状态码参考

更多状态码含义可参考以下表格：

状态码	含义	示例	描述
1	Catchall for general errors	let "var1 = 1/0"	Miscellaneous errors, such as

			"divide by zero" and other impermissible operations
2	Misuse of shell builtins (according to Bash documentation)	empty_function() {}	Missing keyword or command
126	Command invoked cannot execute	/dev/null	Permission problem or command is not an executable
127	"command not found"	illegal_command	Possible problem with `$PATH` or a typo
128	Invalid argument to exit	exit 3.14159	**exit** takes only integer args in therange0 - 255 (seefirst footnote)
128+n	Fatal error signal "n"	*kill -9* `$PPID` of script	**$?** returns137 (128 + 9)
130	Script terminated by Control-C	*Ctl-C*	Control-C is fatal error signal 2, (130 = 128 + 2, see above)
255*	Exit status out of range	exit -1	**exit** takes only integer args in the range0 - 255

Pod 排错指南

概述

最近更新时间：2020-04-30 14:47:03

为满足一定的业务需求，用户往往需要对容器服务集群进行一系列复杂的自定义配置。而当集群中的 Pod 出现某种异常时，可能一时无法直接通过异常状态准确定位异常原因。

基于以上现象，您可参考 Pod 异常问题 系列文档进行问题排查、定位及解决。

常用命令

排查过程的常用命名如下：

- 查看 Pod 状态：

```
kubectl get pod <pod-name> -o wide
```

- 查看 Pod 的 yaml 配置：

```
kubectl get pod <pod-name> -o yaml
```

- 查看 Pod 事件：

```
kubectl describe pod <pod-name>
```

- 查看容器日志：

```
kubectl logs <pod-name> [-c <container-name>]
```

Pod 状态

下表中列举了 Pod 的状态信息：

状态	描述
`Error`	Pod 启动过程中发生错误。
`NodeLost`	Pod 所在节点失联。
`Unkown`	Pod 所在节点失联或其他未知异常。

`Waiting`	Pod 等待启动。
`Pending`	Pod 等待被调度。
`ContainerCreating`	Pod 容器正在被创建。
`Terminating`	Pod 正在被销毁。
`CrashLoopBackOff`	容器退出，Kubelet 正在将它重启。
`InvalidImageName`	无法解析镜像名称。
`ImageInspectError`	无法校验镜像。
`ErrImageNeverPull`	策略禁止拉取镜像。
`ImagePullBackOff`	正在重试拉取。
`RegistryUnavailable`	连接不到镜像中心。
`ErrImagePull`	通用的拉取镜像出错。
`CreateContainerConfigError`	不能创建 Kubelet 使用的容器配置。
`CreateContainerError`	创建容器失败。
`RunContainerError`	启动容器失败。
`PreStartHookError`	执行 preStart hook 报错。
`PostStartHookError`	执行 postStart hook 报错。
`ContainersNotInitialized`	容器没有初始化完毕。
`ContainersNotReady`	容器没有准备完毕。
`ContainerCreating`	容器创建中。
`PodInitializing`	Pod 初始化中。
`DockerDaemonNotReady`	Docker 还没有完全启动。
`NetworkPluginNotReady`	网络插件还没有完全启动。

问题定位

您可根据 Pod 的异常状态，选择对应参考文档进一步定位异常原因：

- Pod 一直处于 ContainerCreating 或 Waiting 状态
- Pod 一直处于 ImagePullBackOff 状态
- Pod 一直处于 Pending 状态
- Pod 一直处于 Terminating 状态
- Pod 健康检查失败
- Pod 处于 CrashLoopBackOff 状态
- 容器进程主动退出

Pod 一直处于 ContainerCreating 或 Waiting 状态

最近更新时间：2020-10-10 14:20:01

本文档介绍可能导致 Pod 一直处于 ContainerCreating 或 Waiting 状态的几种情形，以及如何通过排查步骤定位异常原因。请按照以下步骤依次进行排查，定位问题后恢复正确配置即可。

可能原因

- Pod 配置错误
- 挂载 Volume 失败
- 磁盘空间不足
- 节点内存碎片化
- Limit 设置过小或单位错误
- 拉取镜像失败
- CNI 网络错误
- controller-manager 异常
- 安装 docker 时未完全删除旧版本
- 存在同名容器

排查方法

检查 Pod 配置

1. 检查是否打包了正确的镜像。
2. 检查是否配置了正确的容器参数。

检查 Volume 挂载情况

通过检查 Volume 挂载情况，定位引发异常的真正原因。以下列出两种已知原因：

1. Pod 漂移导致未正常解挂磁盘

问题分析

在云托管的 K8S 服务环境下，默认挂载的 Volume 通常为一块存储类型的云硬盘。如果某个节点出现故障，导致 kubelet 无法正常运行或无法与 apiserver 通信，则到达时间阈值后就会触发该节点驱逐，自动在其他节点上启动相同的 Pod（Pod 漂移），被驱逐的节点无法正常运行及确定自身状态，故该节点的 Volume 也未正确执行解挂操

作。

由于 cloud-controller-manager 需等待 Volume 正确解挂后，才会调用云厂商接口将磁盘从节点上解挂。因此，Pod 漂移会导致 cloud-controller-manager 在达到一个时间阈值后，强制解挂 Volume 并挂载到 Pod 最新调度的节点上。

造成影响

最终导致 ContainerCreating 的时间相对较长，但通常是可以启动成功的。

2. 命中 K8S 挂载 configmap/secret 时 subpath 的 bug

实践发现，当修改了 Pod 已挂载的 configmap 或 secret 的内容，Pod 中容器又进行了原地重启操作（例如，存活检查失败被 kill 后重启拉起），就会触发该 bug。

如果出现了以上情况，容器将会持续启动不成功。报错示例如下：

```
$ kubectl -n prod get pod -o yaml manage-5bd487cf9d-bqmvm
...
lastState: terminated
containerID: containerd://e6746201faa1dfe7f3251b8c30d59ebf613d99715f3b800740e587e681d2a903
exitCode: 128
finishedAt: 2019-09-15T00:47:22Z
message: 'failed to create containerd task: OCI runtime create failed: container_linux.go:345:
starting container process caused "process_linux.go:424: container init
caused \"rootfs_linux.go:58: mounting \\\"/var/lib/kubelet/pods/211d53f4-d08c-11e9-b0a7-b6655eaf0
2a6/volume-subpaths/manage-config-volume/manage/0\\\"
to rootfs \\\"/run/containerd/io.containerd.runtime.v1.linux/k8s.io/e6746201faa1dfe7f3251b8c30d59
ebf613d99715f3b800740e587e681d2a903/rootfs\\\"
at \\\"/run/containerd/io.containerd.runtime.v1.linux/k8s.io/e6746201faa1dfe7f3251b8c30d59ebf613d
99715f3b800740e587e681d2a903/rootfs/app/resources/application.properties\\\"
caused \\\"no such file or directory\\\"\"": unknown'
```

解决方法及更多信息请参见 pr82784。

检查磁盘空间是否不足

启动 Pod 时会调 CRI 接口创建容器，容器运行时组件创建容器时通常会在数据目录下为新建的容器创建一些目录和文件。如果数据目录所在的磁盘空间不足，将会导致容器创建失败。通常会显示以下报错信息：

```
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Warning FailedCreatePodSandBox 2m (x4307 over 16h) kubelet, 10.179.80.31 (combined from similar e
vents): Failed create pod sandbox: rpc error: code = Unknown desc = failed to create a sandbox fo
r pod "apigateway-6dc48bf8b6-l8xrw": Error response from daemon: mkdir /var/lib/docker/aufs/mnt/1
f09d6c1c9f24e8daaea5bf33a4230de7dbc758e3b22785e8ee21e3e3d921214-init: no space left on device
```

解决步骤及更多信息请参见 磁盘爆满。

检查节点内存是否碎片化

如果节点出现内存碎片化严重、缺少大页内存问题，即使总体剩余内存较多，但仍会出现申请内存失败的情况。请参考 内存碎片化 进行异常定位及解决。

检查 limit 设置

现象描述

- 执行 `kubectl describe pod` 命令，查看 event 报错信息如下：

```
Pod sandbox changed, it will be killed and re-created。
```

- Kubelet 报错如下：

```
to start sandbox container for pod ... Error response from daemon: OCI runtime create failed: container_linux.go:348: starting container process caused "process_linux.go:301: running exec setns process for init caused \"signal: killed\"": unknown
```

解决思路

当 limit 设置过小以至于不足以成功运行 Sandbox 时，也会导致 Pod 一直处于 ContainerCreating 或 Waiting 状态，通常是由于 memory limit 单位设置错误引起的。

例如，误将 memory limit 单位设置为小写字母 `m`，则该单位将会被 K8S 识别成 Byte。**您需要将 memory limit 单位设置应为 `Mi` 或 `M`**。若 memory limit 设置为1024m，则表示其大小将会被限制在1.024Byte以下。较小的内存环境下，pause 容器一启动就会被 cgroup-oom kill 掉，导致 Pod 状态一直处于 ContainerCreating。

检查拉取镜像是否失败

拉取镜像失败也会引起 Pod 发生该问题，镜像拉取失败原因较多，常见原因如下：

- 配置了错误的镜像。
- Kubelet 无法访问镜像仓库。例如，默认 pause 镜像在 gcr.io 上，而国内环境访问需要特殊处理。
- 拉取私有镜像的 imagePullSecret 没有配置或配置有误。
- 镜像太大，拉取超时。可在拉取时适当调整 kubelet 的 `--image-pull-progress-deadline` 和 `--runtime-request-timeout` 选项。

针对上述情况调整配置后，请再次拉取镜像并查看 Pod 状态。

检查 CNI 网络是否错误

检查网络插件的配置是否正确，以及运行状态是否正常。当网络插件配置错误或无法正常运行时，会出现以下提示：

- 无法配置 Pod 网络。
- 无法分配 Pod IP。

检查 controller-manager 是否异常

查看 Master 上 kube-controller-manager 状态，异常时尝试重启。

检查节点已有 docker

若在节点已有 docker 或旧版本 docker 未完全卸载的情况下，又安装了 docker，也可能引发 Pod 出现此问题。例如，在 CentOS 上，执行以下命令重复安装了 docker：

```
yum install -y docker
```

由于重复安装的 docker 版本不一致，组件间不完全兼容，导致 dockerd 持续无法成功创建容器，使 Pod 状态一直 ContainerCreating。执行 `kubectl describe pod` 命令，查看 event 报错信息如下：

```
Type Reason Age From Message
---- ------ ---- ---- -------
Warning FailedCreatePodSandBox 18m (x3583 over 83m) kubelet, 192.168.4.5 (combined from similar events): Failed create pod sandbox: rpc error: code = Unknown desc = failed to start sandbox container for pod "nginx-7db9fccd9b-2j6dh": Error response from daemon: ttrpc: client shutting down: read unix @->@/containerd-shim/moby/de2bfeefc999af42783115acca62745e6798981dff75f4148fae8c086668f667/shim.sock: read: connection reset by peer: unknown
Normal SandboxChanged 3m12s (x4420 over 83m) kubelet, 192.168.4.5 Pod sandbox changed, it will be killed and re-created.
```

请选择保留 docker 版本，并完全卸载其余版本 docker。

检查是否存在同名容器

节点上存在同名容器会导致创建 sandbox 时失败，也会导致 Pod 一直处于 ContainerCreating 或 Waiting 状态。

执行 `kubectl describe pod` 命令，查看 event 报错信息如下：

```
Warning FailedCreatePodSandBox 2m kubelet, 10.205.8.91 Failed create pod sandbox: rpc error: code = Unknown desc = failed to create a sandbox for pod "lomp-ext-d8c8b8c46-4v8tl": operation timeout: context deadline exceeded
Warning FailedCreatePodSandBox 3s (x12 over 2m) kubelet, 10.205.8.91 Failed create pod sandbox: rpc error: code = Unknown desc = failed to create a sandbox for pod "lomp-ext-d8c8b8c46-4v8tl": Error response from daemon: Conflict. The container name "/k8s_POD_lomp-ext-d8c8b8c46-4v8tl_default_65046a06-f795-11e9-9bb6-b67fb7a70bad_0" is already in use by container "30aa3f5847e0ce89e9d411e7
```

```
6783ba14accba7eb7743e605a10a9a862a72c1e2". You have to remove (or rename) that container to be ab
le to reuse that name.
```

请修改容器名，确保节点上不存在同名容器。

Pod 一直处于 ImagePullBackOff 状态

最近更新时间：2020-04-24 16:17:13

本文档介绍可能导致 Pod 一直处于 ImagePullBackOff 状态的几种情形，以及如何通过排查步骤定位异常原因。请按照以下步骤依次进行排查，定位问题后恢复正确配置即可。

可能原因

- HTTP 类型 Registry 地址未加入 insecure-registry
- HTTPS 自签发类型 Registry CA 证书未添加至节点
- 私有镜像仓库认证失败
- 镜像文件损坏
- 镜像拉取超时
- 镜像不存在

排查方法

检查 HTTP 类型 Registry 地址是否加入 insecure-registry

Dockerd 默认从 HTTPS 类型的 Registry 拉取镜像。当您使用 HTTP 类型的 Registry 时，请确保已将其地址添加到 insecure-registry 参数中，并重启或 reload Dockerd 使其生效。

检查 HTTPS 自签发类型 Registry CA 证书是否未添加至节点

当您使用 HTTPS 类型 Registry 且其证书属于自签发证书时，Dockerd 将会校验该证书，只有校验成功才可以正常使用镜像仓库。

为确保校验成功，需要将 Registry 的 CA 证书放置到以下位置：

```
/etc/docker/certs.d/<Registry:port>/ca.crt
```

检查私有镜像仓库配置

若 Pod 未配置 imagePullSecret、配置的 Secret 不存在或者有误都会造成 Registry 认证失败，使 Pod 一直处于 ImagePullBackOff 状态。

检查镜像文件是否损坏

若 Push 的镜像文件损坏，下载成功后也不能正常使用，则需要重新 push 镜像文件。

检查镜像是否拉取超时

现象描述

当节点上同时启动大量 Pod 时，可能会导致容器镜像下载需要排队。假设下载队列靠前位置已有许多大容量镜像且需较长的下载时间，则会导致排在队列靠后的 Pod 拉取超时。

默认情况下，kubelet 支持串行下载镜像。如下所示：

```
--serialize-image-pulls Pull images one at a time. We recommend *not* changing the default value on nodes that run docker daemon with version < 1.9 or an Aufs storage backend. Issue #10959 has more details. (default true)
```

解决思路

必要情况下，为避免 Pod 拉取超时，可开启并行下载及控制并发。示例如下：

```
--Registry-qps int32 If > 0, limit Registry pull QPS to this value. If 0, unlimited. (default 5)
--Registry-burst int32 Maximum size of a bursty pulls, temporarily allows pulls to burst to this number, while still not exceeding Registry-qps. Only used if --Registry-qps > 0 (default 10)
```

检查镜像是否存在

镜像本身不存在也会导致 Pod 一直处于 ImagePullBackOff 状态，可以通过 kubelet 日志进行确认。如下所示：

```
PullImage "imroc/test:v0.2" from image service failed: rpc error: code = Unknown desc = Error response from daemon: manifest for imroc/test:v0.2 not found
```

Pod 一直处于 Pending 状态

最近更新时间：2020-09-04 16:45:31

本文档介绍可能导致 Pod 一直处于 Pending 状态的几种情形，以及如何通过排查步骤定位异常原因。请按照以下步骤依次进行排查，定位问题后恢复正确配置即可。

现象描述

当 Pod 一直处于 Pending 状态时，说明该 Pod 还未被调度到某个节点上，需查看 Pod 分析问题原因。例如执行 `kubectl describe pod <pod-name>` 命令，则获取到的事件信息如下：

```
$ kubectl describe pod tikv-0
...
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Warning FailedScheduling 3m (x106 over 33m) default-scheduler 0/4 nodes are available: 1 node(s) had no available volume zone, 2 Insufficient cpu, 3 Insufficient memory.
```

可能原因

- 节点资源不足
- 不满足 nodeSelector 与 affinity
- Node 存在 Pod 没有容忍的污点
- 低版本 kube-scheduler 的 bug
- kube-scheduler 未正常运行
- 驱逐后其他可用节点与当前节点的有状态应用不在相同可用区

排查方法

检查节点是否资源不足

问题分析

节点资源不足有以下几种情况：

- CPU 负载过高。

- 剩余可以被分配的内存不足。
- 剩余可用 GPU 数量不足（通常在机器学习场景、GPU 集群环境）。

为进一步判断某个 Node 资源是否足够，可执行以下命令获取资源分配信息：

```
kubectl describe node <node-name>
```

在返回信息中，请注意关注以下内容：

- `Allocatable` ：表示此节点能够申请的资源总和。
- `Allocated resources` ：表示此节点已分配的资源（Allocatable 减去节点上所有 Pod 总的 Request）。

造成影响

前者与后者相减，即可得出剩余可申请的资源大小。如果该值小于 Pod 的 Request，则不满足 Pod 的资源要求，Scheduler 在 Predicates（预选）阶段就会剔除掉该 Node，不会调度 Pod 到该 Node。

检查 nodeSelector 及 affinity 的配置

假设 Pod 中 nodeSelector 指定了节点 Label，则调度器将只考虑调度 Pod 到包含该 Label 的 Node 上。当不存在符合该条件的 Node 时，Pod 将无法被调度。更多相关信息可前往 Kubernetes 官方网站 进行查看。

此外，如果 Pod 配置了 affinity（亲和性），则调度器根据调度算法可能无法发现符合条件的 Node，从而无法调度。affinity 包括以下几类：

- `nodeAffinity` ：节点亲和性，可以看作增强版的 nodeSelector，用于限制 Pod 只允许被调度到某一部分符合条件的 Node。
- `podAffinity` ：Pod 亲和性，用于将一系列有关联的 Pod 调度到同一个地方，即同一个节点或同一个可用区的节点等。
- `podAntiAffinity` ：Pod 反亲和性，防止某一类 Pod 调度到同一个地方，可以有效避免单点故障。例如，将集群 DNS 服务的 Pod 副本分别调度到不同节点，可避免因一个节点出现故障而造成整个集群 DNS 解析失败，甚至使业务中断。

检查 Node 是否存在 Pod 没有容忍的污点

问题分析

假如节点上存在污点（Taints），而 Pod 上没有相应的容忍（Tolerations），Pod 将不会调度到该 Node。在调度之前，可以先通过 `kubectl describe node <node-name>` 命令查看 Node 已设置污点。示例如下：

```
$ kubectl describe nodes host1
...
Taints: special=true:NoSchedule
...
```

Node 上已设置的污点可通过手动或自动的方式添加，详情请参见 添加污点。

解决方法

本文提供以下两种方法，通常选择方法2解决该问题：

- 方法1：删除污点
  执行以下命令，删除污点 `special` 。

```
kubectl taint nodes host1 special-
```

- 方法2：在 Pod 上增加污点容忍

> **说明：**
> 本文以向 Deployment 中已创建的 Pod（名称为 `nginx` ）添加容忍为例。

i. 参考 使用标准登录方式登录 Linux 实例（推荐），登录 `nginx` 所在的云服务器。
ii. 执行以下命令，编辑 Yaml。

```
kubectl edit deployment nginx
```

iii. 在 Yaml 文件 `template` 中的 `spec` 处添加容忍。例如，增加已存在 `special` 污点所对应的容忍：

```
tolerations:
- key: "special"
operator: "Equal"
value: "true"
effect: "NoSchedule"
```

添加完成后如下图所示：

```
template:
  metadata:
    creationTimestamp: null
    labels:
      k8s-app: nginx
      qcloud-app: nginx
  spec:
    containers:
    - image: nginx:latest
      imagePullPolicy: Always
      name: test
      resources:
        limits:
          cpu: 500m
          memory: 1Gi
        requests:
          cpu: 250m
          memory: 256Mi
      securityContext:
        privileged: false
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    imagePullSecrets:
    - name: qcloudregistrykey
    - name: tencenthubkey
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
    tolerations:
    - effect: NoSchedule
      key: special
      operator: Equal
      value: "true"
```

iv. 保存并退出编辑即可成功创建容忍。

检查是否存在低版本 kube-scheduler 的 bug

Pod 一直处于 Pending 状态可能是低版本 `kube-scheduler` 的 bug 导致的，该情况可以通过升级调度器版本进行解决。

检查 kube-scheduler 是否正常运行

请注意时检查 Master 上的 `kube-scheduler` 是否运行正常，如异常可尝试重启临时恢复。

检查驱逐后其他可用节点与当前节点的有状态应用是否不在相同可用区

服务部署成功且正在运行时，若此时节点突发故障，就会触发 Pod 驱逐，并创建新的 Pod 副本调度到其他节点上。对于已挂载了磁盘的 Pod，通常需要被调度到与当前故障节点和挂载磁盘所处同一个可用区的新的节点上。若集群中同一个可用区内不具备满足可调度条件的节点时，即使其他可用区内具有满足条件的节点，此类 Pod 仍不会调度。

限制已挂载磁盘的 Pod 不能调度到其他可用区的节点的原因如下：
云上磁盘允许被动态挂载到同一个数据中心上的不同机器，为了有效避免网络时延极大地降低 IO 速率，通常不允许跨数据中心挂载磁盘设备。

相关操作

添加污点

手动添加污点

通过以下或类似方式，可以手动为节点添加指定污点：

```
$ kubectl taint node host1 special=true:NoSchedule
node "host1" tainted
```

> **说明：**
>
> 在某些场景下，可能期望新加入的节点在调整好某些配置之前默认不允许调度 Pod。此时，可以给该新节点添加 `node.kubernetes.io/unschedulable` 污点。

自动添加污点

从 v1.12 开始，Beta 默认开启 `TaintNodesByCondition` 特性，controller manager 将会检查 Node 的 Condition。Node 运行状态异常时，当检查的 Condition 符合如下条件（即符合 Condition 与 Taints 的对应关系），将自动给 Node 加上相应的污点。
例如，检查 Condition 为 `OutOfDisk` 且 Value 为 `True`，则 Node 会自动添加 `node.kubernetes.io/out-of-disk` 污点。
Condition 与污点的对应关系如下：

```
Conditon Value Taints
-------- ----- ------
OutOfDisk True node.kubernetes.io/out-of-disk
Ready False node.kubernetes.io/not-ready
Ready Unknown node.kubernetes.io/unreachable
MemoryPressure True node.kubernetes.io/memory-pressure
PIDPressure True node.kubernetes.io/pid-pressure
```

```
DiskPressure True node.kubernetes.io/disk-pressure
NetworkUnavailable True node.kubernetes.io/network-unavailable
```

当每种 Condition 取特定的值时，将表示以下含义：

- `OutOfDisk` 为 True，表示节点磁盘空间不足。
- `Ready` 为 False，表示节点不健康。
- `Ready` 为 Unknown，表示节点失联。在 `node-monitor-grace-period` 所确定的时间周期内（默认40s）若节点没有上报状态，controller-manager 就会将 Node 状态置为 Unknown。
- `MemoryPressure` 为 True，表示节点内存压力大，实际可用内存很少。
- `PIDPressure` 为 True，表示节点上运行了太多进程，PID 数量不足。
- `DiskPressure` 为 True，表示节点上的磁盘可用空间不足。
- `NetworkUnavailable` 为 True，表示节点上的网络没有正确配置，无法跟其他 Pod 正常通信。

> **说明：**
>
> 上述情况一般属于被动添加污点，但在容器服务中，存在一个主动添加/移出污点的过程：
> 在新增节点时，首先为该节点添加 `node.cloudprovider.kubernetes.io/uninitialized` 污点，待节点初始化成功后再自动移除此污点，以避免 Pod 被调度到未初始化好的节点。

Pod 一直处于 Terminating 状态

最近更新时间：2020-04-24 16:17:13

本文档将为您展示可能导致 Pod 一直处于 Terminating 状态的几种情形，以及如何通过排查步骤定位异常原因。请按照以下步骤依次进行排查，定位问题后恢复正确配置即可。

可能原因

- 磁盘空间不足
- 存在 “i” 文件属性
- Docker 17 版本 bug
- 存在 Finalizers
- 低版本 kubelet list-watch 的 bug
- Dockerd 与 containerd 状态不同步
- Daemonset Controller Bug

排查方法

检查磁盘空间是否不足

当 Docker 的数据目录所在磁盘被写满时，Docker 将无法正常运行，甚至无法进行删除和创建操作。kubelet 调用 Docker 删除容器时将无响应，执行 `kubectl describe pod <pod-name>` 命令，查看 event 通常返回信息如下：

```
Normal Killing 39s (x735 over 15h) kubelet, 10.179.80.31 Killing container with id docker://apiga
teway:Need to kill Pod
```

解决方法及更多信息请参考 磁盘爆满。

检查是否存在 “i” 文件属性

现象描述

“i” 文件属性描述可通过 `man chattr` 进行查看，描述示例如下：

```
A file with the 'i' attribute cannot be modified: it cannot be deleted or renamed, no link can be
created to this file and no data can be written to the file. Only the superuser or a process poss
essing the CAP_LINUX_IMMUTABLE capability can set or clear this attribute.
```

如果容器镜像本身或者容器启动后写入的文件存在“i”文件属性，此文件将无法被修改或删除。

在进行 Pod 删除操作时，会清理容器目录，若该目录中存在不可删除的文件，会导致容器目录无法删除，Pod 状态也将一直保持 Terminating。此种情况下，kubelet 将会出现以下报错：

```
Sep 27 14:37:21 VM_0_7_centos kubelet[14109]: E0927 14:37:21.922965 14109 remote_runtime.go:250] RemoveContainer "19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257" from runtime service failed: rpc error: code = Unknown desc = failed to remove container "19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257": Error response from daemon: container 19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257: driver "overlay2" failed to remove root filesystem: remove /data/docker/overlay2/b1aea29c590aa9abda79f7cf3976422073fb3652757f0391db88534027546868/diff/usr/bin/bash: operation not permitted
Sep 27 14:37:21 VM_0_7_centos kubelet[14109]: E0927 14:37:21.923027 14109 kuberuntime_gc.go:126] Failed to remove container "19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257": rpc error: code = Unknown desc = failed to remove container "19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257": Error response from daemon: container 19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257: driver "overlay2" failed to remove root filesystem: remove /data/docker/overlay2/b1aea29c590aa9abda79f7cf3976422073fb3652757f0391db88534027546868/diff/usr/bin/bash: operation not permitted
```

解决方法

- 彻底解决方法：不在容器镜像中或启动后的容器设置“i”文件属性，彻底杜绝此问题发生。
- 临时恢复方法：
  i. 针对 kubelet 日志报错提示的文件路径，执行 `chattr -i <file>` 命令。示例如下：

  ```
  chattr -i /data/docker/overlay2/b1aea29c590aa9abda79f7cf3976422073fb3652757f0391db88534027546868/diff/usr/bin/bash
  ```

  ii. 等待 kubelet 自动重试，Pod 即可自动删除。

检查是否存在 Docker 17 版本 bug

现象描述

Docker hang 住，没有任何响应。执行 `kubectl describe pod <pod-name>` 命令查看 event 显示如下：

```
Warning FailedSync 3m (x408 over 1h) kubelet, 10.179.80.31 error determining status: rpc error: code = DeadlineExceeded desc = context deadline exceeded
```

造成该问题原因可能为17版本 dockerd 的 bug，虽然可以通过 `kubectl -n cn-staging delete pod apigateway-6dc48bf8b6-clcwk --force --grace-period=0` 强制删除 Pod，但执行 `docker ps` 命令后，仍然看得到该容器。

解决方法

升级 Docker 版本至18，该版本使用了新的 containerd，针对很多已有 bug 进行了修复。
若 Pod 仍出现 Terminating 状态，请 提交工单 联系工程师进行排查。**不建议直接强行删除**，可能会导致业务出现问题。

检查是否存在 Finalizers

现象描述

K8S 资源的 metadata 中如果存在 `finalizers`，通常说明该资源是由某个程序创建的，`finalizers` 中也会添加一个专属于该程序的标识。例如，Rancher 创建的一些资源就会写入 `finalizers` 标识。

若想要删除该程序所创建的资源时，则需要由创建该资源的程序进行删除前的清理，且只有清理完成并将标识从该资源的 `finalizers` 中移除，才可以彻底删除资源。

解决方法

使用 `kubectl edit` 命令手动编辑资源定义，删除 `finalizers`，删除资源便不会再受阻。

检查是否存在低版本 kubelet list-watch 的 bug

历史排查异常过程中发现，使用 v1.8.13 版本的 K8S 时，kubelet 会出现 list-watch 异常的情况。该问题会导致在删除 Pod 后，kubelet 未获取相关事件，并未真正删除，使 Pod 一直处 Terminating 状态。

请参考文档 升级集群 步骤进行集群 Kubernetes 版本升级。

Dockerd 与 containerd 状态不同步

现象描述

docker 在 aufs 存储驱动下如果磁盘爆满，则可能发生内核 panic，报错信息如下：

```
aufs au_opts_verify:1597:dockerd[5347]: dirperm1 breaks the protection by the permission bits on the lower branch
```

若磁盘曾爆满过，dockerd 日志通常会有以下类似记录，且后续可能发生状态不同步问题。

```
Sep 18 10:19:49 VM-1-33-ubuntu dockerd[4822]: time="2019-09-18T10:19:49.903943652+08:00" level=error msg="Failed to log msg ¥"¥" for logger json-file: write /opt/docker/containers/54922ec8b1863bcc504f6dac41e40139047f7a84ff09175d2800100aaccbad1f/54922ec8b1863bcc504f6dac41e40139047f7a84ff09175d2800100aaccbad1f-json.log: no space left on device"
```

问题分析

判断 dockerd 与 containerd 的某个容器状态是否同步，可采用以下几种方法：

- 首先通过 `describe pod` 获取容器 ID，再通过 `docker ps` 查看容器状态是否为 dockerd 中所保存的状态。

- 通过 `docker-container-ctr` 查看容器在 containerd 中的状态。示例如下：

```
$ docker-container-ctr --namespace moby --address /var/run/docker/containerd/docker-container
d.sock task ls |grep a9a1785b81343c3ad2093ad973f4f4f8e52dbf54823b8bb089886c8356d4036fe0
a9a1785b81343c3ad2093ad973f4f4f8e52dbf54823b8bb089886c8356d4036fe0 30639 STOPPED
```

若 containerd 中容器状态是 stopped 或者已经无记录，而 docker 中容器状态却是 running，则说明 dockerd 与 containerd 之间容器状态同步存在问题。

解决方法

- 临时解决方法：执行 `docker container prune` 命令或重启 dockerd。
- 彻底解决方法：运行时推荐直接使用 containerd，绕过 dockerd 避免 Docker 本身的 Bug。

Daemonset Controller Bug

K8S 中存在的 Bug 会导致 Daemonset Pod 持续 Terminating，Kubernetes 1.10 和 1.11 版本受此影响。由于 Daemonset Controller 复用 scheduler 的 predicates 逻辑，将 nodeAffinity 的 nodeSelector 数组进行排序（传递的指针参数），导致 spec 与 apiserver 中的值不一致。为了版本控制，Daemonset Controller 又使用了 spec 为 rollingUpdate 类型的 Daemonset 计算 hash。
上述传递过程造成的前后参数不一致问题，导致了 Pod 陷入持续启动和停止的循环。

解决方法

- 临时解决方法：确保 rollingUpdate 类型 Daemonset 使用 nodeSelector 而不使用 nodeAffinity。
- 彻底解决方法：参考文档 升级集群 步骤将集群 Kubernetes 版本升级至 1.12。

Pod 健康检查失败

最近更新时间：2020-04-24 16:17:14

本文档介绍可能导致 Pod 健康检查失败的几种情形，以及如何通过排查步骤定位异常原因。请按照以下步骤依次进行排查，定位问题后恢复正确配置即可。

现象描述

Kubernetes 健康检查包含就绪检查（readinessProbe）和存活检查（livenessProbe），不同阶段的检查失败将会分别出现以下现象：

- Pod IP 从 Service 中摘除。通过 Service 访问时，流量将不会被转发给就绪检查失败的 Pod。
- kubelet 将会停止容器并尝试重启。

引发健康检查失败的诱因种类较多。例如，业务程序存在某个 Bug 导致其不能响应健康检查，使 Pod 处于 Unhealthy 状态。接下来您可按照以下方式进行排查。

可能原因

- 健康检查配置不合理
- 节点负载过高
- 容器进程被木马进程停止
- 容器内进程端口监听故障
- SYN backlog 设置过小

排查方法

检查健康检查配置

如果健康检查配置不合理，会导致 Pod 健康检查失败。例如，容器启动完成后首次探测的时间 `initialDelaySeconds` 设置过短。容器启动较慢时，导致容器还没完全启动就开始探测。同时，若 `successThreshold` 默认值设置为1，则 Pod 健康检查失败一次就会被停止，那么 Pod 将会持续被停止并重启。

检查节点是否负载过高

CPU 占用高（例如跑满）将导致进程无法正常发包收包，通常会出现 timeout，导致 Pod 健康检查失败。请参考高负载 进行异常问题定位及解决。

检查容器进程是否被木马进程停止

请参考 使用 Systemtap 定位 Pod 异常退出原因 异常问题定位及解决。

检查容器内进程端口是否监听故障

使用 `netstat -tunlp` 检查端口监听是否还存在。分析该命令的返回结果可知：当端口监听不存在时，健康检查探测的连接将会被直接 reset 掉。示例如下：

```
20:15:17.890996 IP 172.16.2.1.38074 > 172.16.2.23.8888: Flags [S], seq 96880261, win 14600, optio
ns [mss 1424,nop,nop,sackOK,nop,wscale 7], length 0
20:15:17.891021 IP 172.16.2.23.8888 > 172.16.2.1.38074: Flags [R.], seq 0, ack 96880262, win 0, l
ength 0
20:15:17.906744 IP 10.0.0.16.54132 > 172.16.2.23.8888: Flags [S], seq 1207014342, win 14600, opti
ons [mss 1424,nop,nop,sackOK,nop,wscale 7], length 0
20:15:17.906766 IP 172.16.2.23.8888 > 10.0.0.16.54132: Flags [R.], seq 0, ack 1207014343, win 0,
length 0
```

分析以上结果可知，健康检查探测连接异常，将会导致健康检查失败。可能原因为：
节点上同时启动多个使用 `hostNetwork` 监听相同宿主机端口的 Pod，但只有一个Pod 会监听成功，其余 Pod 监听失败且不会退出，继续进行适配健康检查。从而使得 kubelet 发送健康检查探测报文给 Pod 时，发送给监听失败（即没有监听端口）的 Pod，导致健康检查失败。

检查 SYN backlog 设置是否过小

现象描述

SYN backlog 大小即 SYN 队列大小，如果短时间内新建连接比较多，而 SYN backlog 设置过小，就会导致新建连接失败。使用 `netstat -s | grep TCPBacklogDrop` 可查看由于 backlog 满了导致丢弃的新连接数量。

解决方法

如果已确认由于 backlog 满了导致的丢包，则建议调高 backlog 值，相关内核参数为 `net.ipv4.tcp_max_syn_backlog` 。

Pod 处于 CrashLoopBackOff 状态

最近更新时间：2020-04-24 16:17:14

本文档介绍可能导致 Pod 处于 CrashLoopBackOff 状态的几种情形，以及如何通过排查步骤定位异常原因。请按照以下步骤依次进行排查，定位问题后恢复正确配置即可。

现象描述

Pod 处于 `CrashLoopBackOff` 状态，说明该 Pod 在正常启动过后异常退出过，此状态下 Pod 的 restartPolicy 如果不是 Never 就可能会被重启拉起，且 Pod 的 `RestartCounts` 通常大于0。可首先参考 通过 Exit Code 定位 Pod 异常退出原因 查看对应容器进程的退出状态码，缩小异常问题范围。

可能原因

- 容器进程主动退出
- 系统 OOM
- cgroup OOM
- 节点内存碎片化
- 健康检查失败

排查步骤

检查容器进程是否主动退出

容器进程主动退出时，退出状态码通常在0 - 128之间，导致异常的原因可能是业务程序 Bug，也可能是原因。请参考 容器进程主动退出 进一步定位异常问题。

检查是否发生系统 OOM

问题分析

如果发生系统 OOM，Pod 中容器退出状态码为137，表示其被 `SIGKILL` 信号停止，同时内核将会出现以下报错信息。

```
Out of memory: Kill process ...
```

该异常是由于节点上部署了其他非 K8S 管理的进程消耗了较多的内存，或是 kubelet 的 `--kube-reserved` 和 `--system-reserved` 所分配的内存太小，没有足够的空间运行其他非容器进程。

节点上所有 Pod 的实际内存占用总量不会超过 `/sys/fs/cgroup/memory/kubepods` 中的 cgroup 值（`cgroup = capacity - "kube-reserved" - "system-reserved"`）。通常情况下，如果预留空间设置合理，且节点上其他非容器进程（例如 kubelet、dockerd、kube-proxy 及 sshd 等）内存占用没有超过 kubelet 配置的预留空间，是不会发生系统 OOM 的。

解决方法

为确保不再发生此类问题，您可以根据实际需求对预留空间进行合理的调整。

检查是否发生 cgroup OOM

现象描述

如果是因 cgroup OOM 而停止的进程，可看到 Pod 事件下 `Reason` 为 `OOMKilled`，说明容器实际占用的内存已超过 limit，同时内核日志会报 `Memory cgroup out of memory` 错误信息。

解决方法

请根据需求调整 limit。

节点内存碎片化

如果节点出现内存碎片化严重、缺少大页内存问题，即使总体剩余内存较多，但仍会出现申请内存失败的情况。请参考 内存碎片化 进行异常定位及解决。

健康检查失败

请参考 Pod 健康检查失败 进一步定位异常问题。

容器进程主动退出

最近更新时间：2020-04-24 16:17:14

本文档介绍可能导致容器进程主动退出的几种场景，以及如何通过排查步骤定位异常原因。请按照以下步骤依次进行排查，定位问题后恢复正确配置即可。

现象描述

容器进程主动退出（不是被外界中断停止）时，退出状态码通常在0 - 128之间。根据规定，正常退出时状态码为0，状态码为1 - 127则说明为程序发生异常导致其主动退出。例如，当检测到程序启动参数和条件不满足要求，或者程序运行过程中发生 panic 但没有捕获处理就会导致程序主动退出。
可首先参考 通过 Exit Code 定位 Pod 异常退出原因 查看对应容器进程的退出状态码，缩小异常问题范围。

可能原因

- DNS 无法解析
- 程序配置有误

排查方法

检查 DNS 是否无法解析

若程序依赖集群 DNS 服务，则解析失败将导致程序报错并主动退出。例如，程序启动时需要连接数据库，且数据库使用 service 名称或外部域名都需要 DNS 解析，若解析失败将引发程序报错并主动退出，导致容器进程主动退出。解析失败的可能原因如下：

- 集群网络存在异常，Pod 无法连接集群 DNS 服务。
- 集群 DNS 服务故障，无法响应解析请求。
- Service 或域名地址配置有误，致使其无法解析。

检查程序配置

程序配置有错误也可能引起容器进程主动退出，可能原因如下：

- 配置文件格式错误，程序启动时解析配置失败从而报错退出。
- 配置内容不符合规范。例如，配置中某个必选字段未填写，导致配置校验不通过，程序报错主动退出。

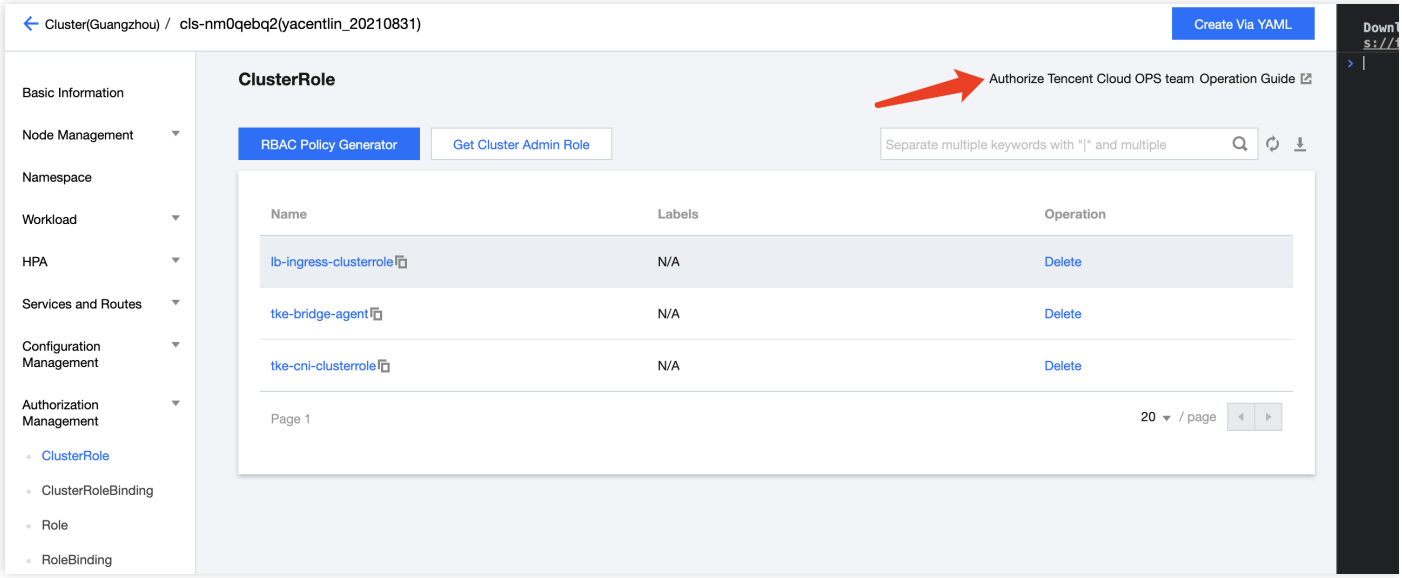
授权腾讯云运维

最近更新时间：2021-11-12 14:59:34

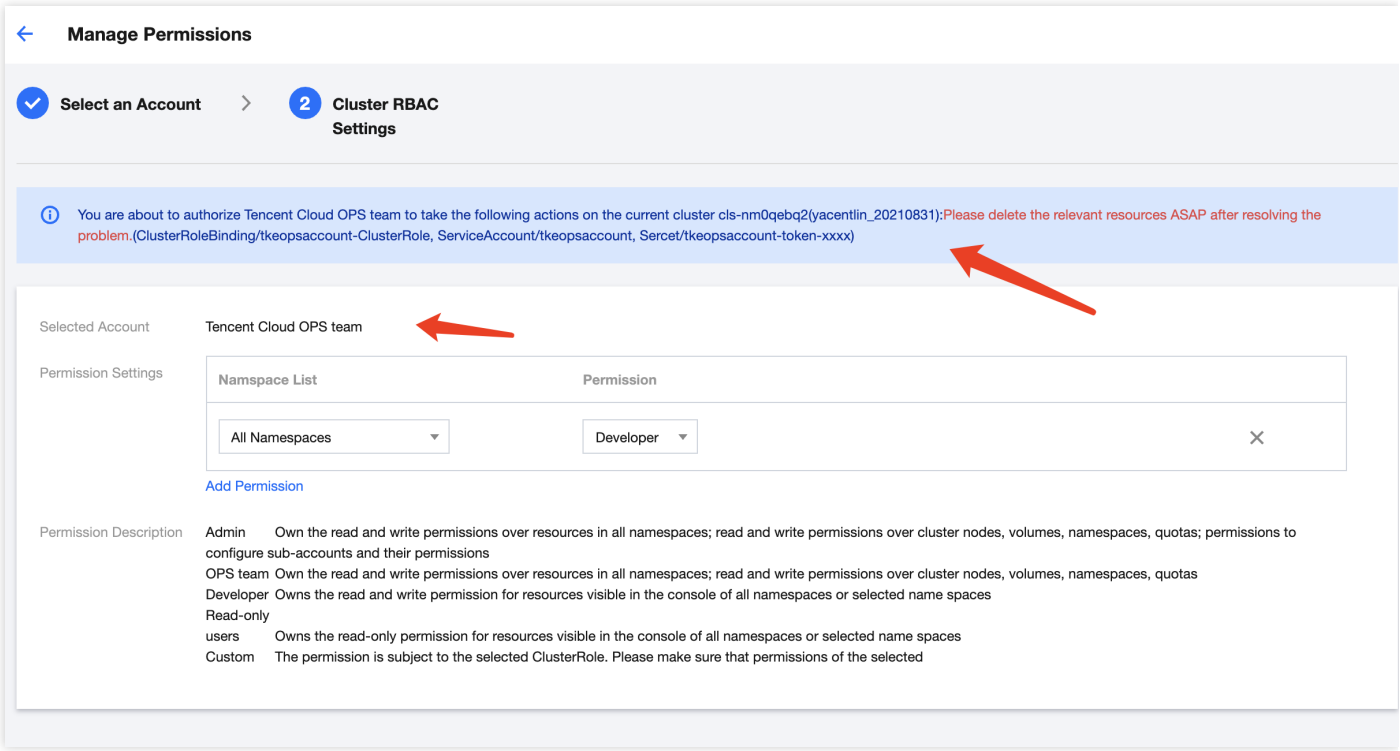
默认情况下腾讯云无法登录集群进行问题排障，如果您需要腾讯云售后协助进行运维排障，请参考以下步骤授予腾讯云运维权限。您有权随时吊销回收授予腾讯云的运维排障权限。

通过控制台授予腾讯云权限

1. 登录 容器服务控制台。
2. 在集群管理中选择需要腾讯云协助的集群。
3. 在集群详情页，选择**授权管理 > 授权腾讯云运维**。如下图所示：



4. 在**管理权限**页中，选择赋予腾讯云的操作权限。如下图所示：



5. 单击**完成**。您可在 我的工单 中查看问题处理进度。

> 注意：
> 腾讯云仅能够登录您授权的集群，您可以随时吊销腾讯云运维权限。您可以通过删除相关资源（ClusterRoleBinding/tkeopsaccount-ClusterRole、ServiceAccount/tkeopsaccount、Sercet/tkeopsaccount-token-xxxx）吊销腾讯云运维权限。

通过 Kubernetes API 授予腾讯云权限

您可以通过创建以下 Kubernetes 资源授予腾讯云指定权限。

ServiceAccount 授予腾讯云访问集群凭证

```
kind: ServiceAccount
apiVersion: v1
metadata:
name: tkeopsaccount
namespace: kube-system
labels:
cloud.tencent.com/tke-ops-account: tkeops
```

ClusterRoleBinding/RoleBing 授予腾讯云的操作权限规则

说明：

1. 名称和 label 需按如下规则创建。
2. roleRef 可替换为您期望授权腾讯云的权限。

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
annotations:
cloud.tencent.com/tke-ops-account: tkeops
labels:
cloud.tencent.com/tke-ops-account: tkeops
name: tkeopsaccount-ClusterRole
roleRef:
apiGroup: rbac.authorization.k8s.io
kind: ClusterRole
name: tke:admin
subjects:
- kind: ServiceAccount
name: tkeopsaccount
namespace: kube-system
```

（可选）ClusterRole/Role 授予腾讯云的操作权限

如集群内有相关 ClusterRole/Role 可直接使用 ClusterRoleBinding/RoleBinding 关联。通过控制台授权，将自动创建策略，无需单独创建。

- 管理员权限
- 只读权限

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
labels:
cloud.tencent.com/tke-rbac-generated: "true"
name: tke:admin
```

```
rules:
- apiGroups:
- '*'
resources:
- '*'
verbs:
- '*'
- nonResourceURLs:
- '*'
verbs:
- '*'
```

CLB 回环问题

最近更新时间：2021-11-09 10:47:46

问题描述

用户使用容器服务 TKE 时，会遇到因 CLB 回环问题导致服务访问不通或访问 Ingress 存在几秒延时的现象，本文档介绍该问题的相关现象、原因，并提供解决方法。

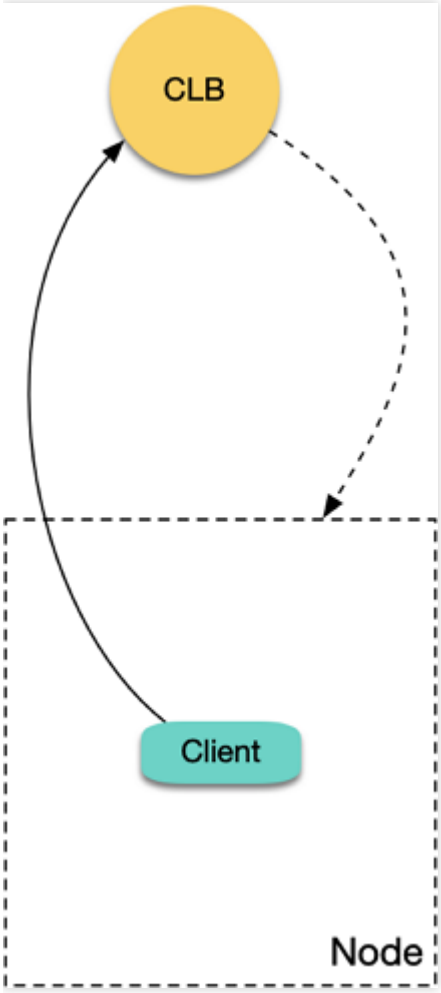
现象描述

CLB 回环问题可能导致存在以下现象：

- 无论是 iptables 或是 ipvs 模式，访问本集群内网 Ingress 会出现4秒延时或不通。
- ipvs 模式下，集群内访问本集群 LoadBanacer 类型的内网 Service 出现完全不通，或者联通不稳定。

问题原因

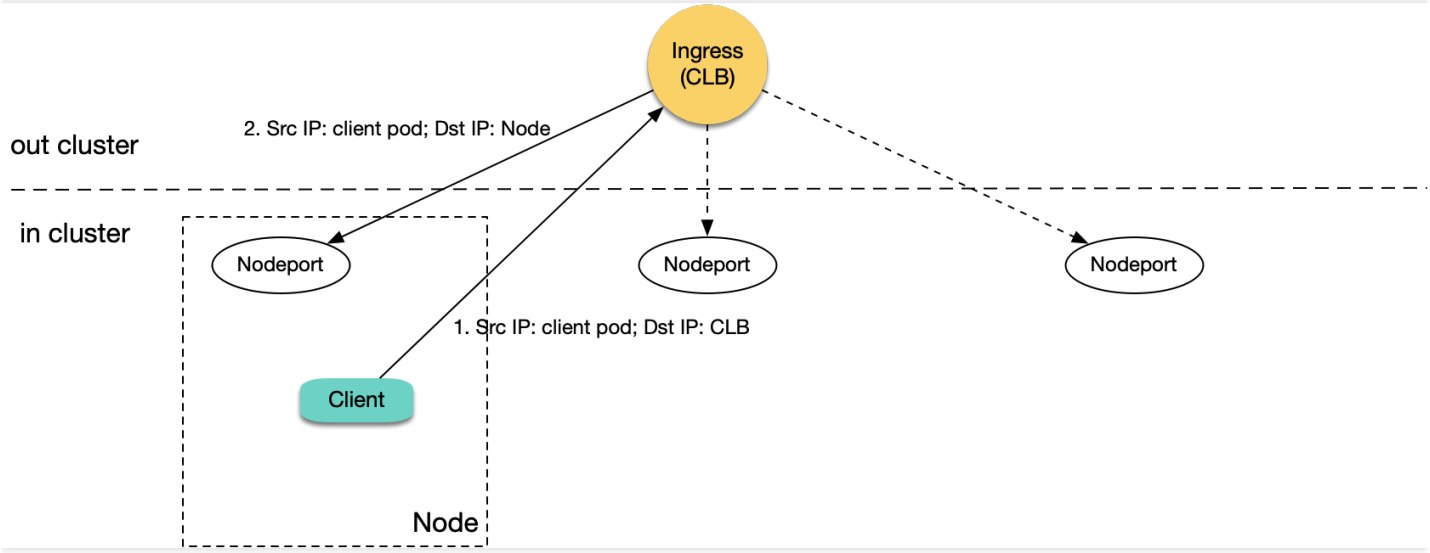
根本原因在于 CLB 将请求转发到后端 RS 时，报文的源目的 IP 都在同一节点内，而 Linux 默认忽略收到源 IP 是本机 IP 的报文，导致数据包在子机内部回环，如下图所示：



分析 Ingress 回环

使用 TKE CLB 类型的 Ingress，会为每个 Ingress 资源创建一个 CLB 以及80、443的7层监听器规则（HTTP/HTTPS），并为 Ingress 每个 location 绑定对应 TKE 各个节点某个相同的 NodePort 作为 RS（每个 location 对应一个 Service，每个 Service 都通过各个节点的某个相同 NodePort 暴露流量），CLB 根据请求匹配 location 转发到相应的 RS（即 NodePort），流量经过 NodePort 后会再经过 Kubernetes 的 iptables 或 ipvs 转发给对应的后端 Pod。集群中的 Pod 访问本集群的内网 Ingress，CLB 将请求转发给其中一台节点的对应

NodePort：



如上图，当被转发的这台节点恰好也是发请求的 Client 所在节点时：

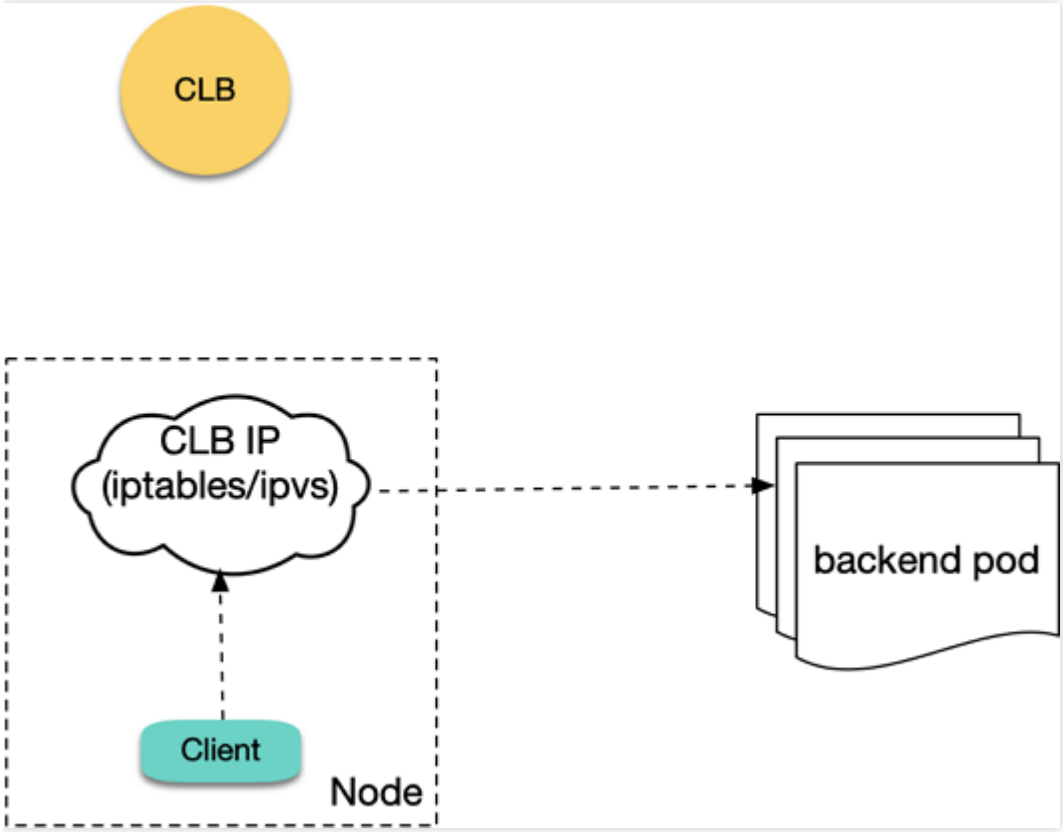
1. 集群中的 Pod 访问 CLB，CLB 将请求转发到任意一台节点的对应 NodePort。
2. 报文到达 NodePort 时，目的 IP 是节点 IP，源 IP 是 Client Pod 的真实 IP， 因为 CLB 不进行 SNAT，会将真实源 IP 透传过去。
3. 由于源 IP 与目的 IP 都在这台机器内，因此将导致回环，CLB 将收不到来自 RS 的响应。

访问集群内 Ingress 的故障现象大多为几秒延时，原因是7层 CLB 如果请求 RS 后端超时（大概4s），会重试下一个 RS，所以如果 Client 设置的超时时间较长，出现回环问题的现象就是请求响应慢，有几秒的延时。如果集群只有一个节点，CLB 没有可以重试的 RS，则现象为访问不通。

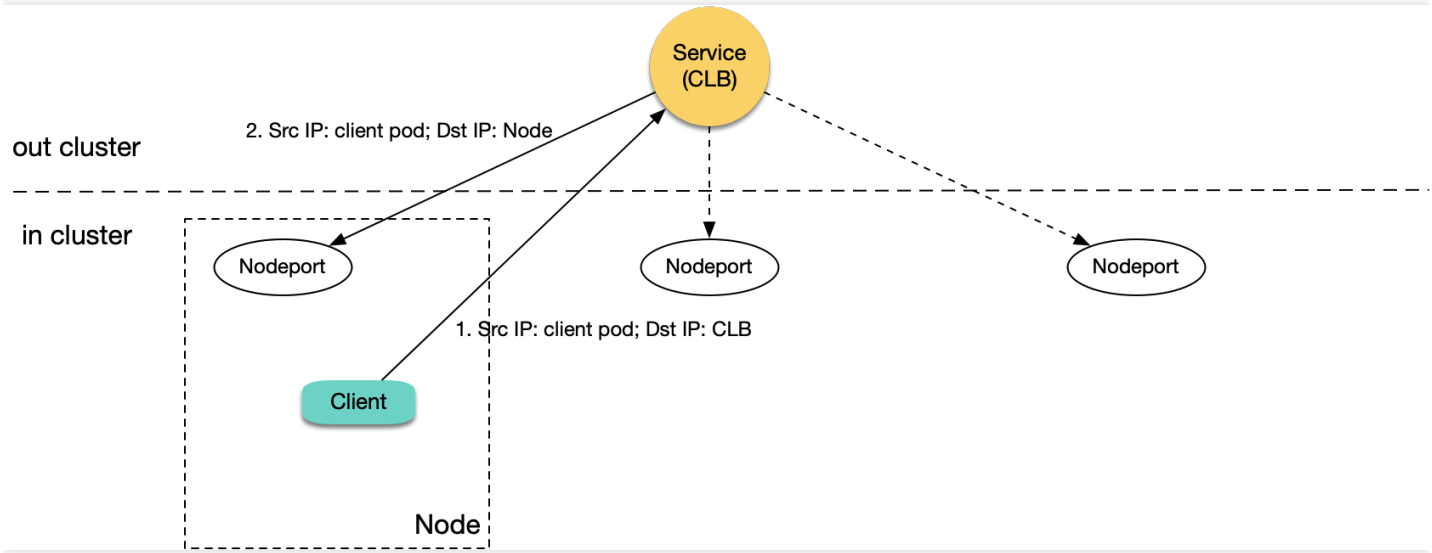
分析 LoadBalancer Service 回环

当使用 LoadBalancer 类型的内网 Service 时暴露服务时，会创建内网 CLB 并创建对应的4层监听器（TCP/UDP）。当集群内 Pod 访问 LoadBalancer 类型 Service 的 `EXTERNAL-IP` 时（即 CLB IP），原生 Kubernetes 实际上不会去真正访问 LB，而是直接通过 iptables 或 ipvs 转发到后端 Pod （不经过 CLB），如下

图所示：



因此原生 Kubernetes 的逻辑不存在回环问题。但在 TKE 的 ipvs 模式下，Client 访问 CLB IP 的包会真正到 CLB，如果在 ipvs 模式下，Pod 访问本集群 LoadBalancer 类型 Service 的 CLB IP 将存在回环问题，情况跟上述内网 Ingress 回环类似，如下图所示：



不同的是，四层 CLB 不会重试下一个 RS，当遇到回环时，现象通常是联通不稳定。如果集群只有一个节点，那将导致完全不通。

为什么 TKE 的 ipvs 模式不用原生 Kubernetes 类似的转发逻辑（不经过 LB，直接转发到后端 Pod）？

背景是因为以前 TKE 的 ipvs 模式集群使用 LoadBalancer 内网 Service 暴露服务，内网 CLB 对后端 NodePort 的健康探测会全部失败，原因如下：

- ipvs 主要工作在 INPUT 链，需要将要转发的 VIP（Service 的 Cluster IP 和 EXTERNAL-IP）当成本机 IP，才好让报文进入 INPUT 链交给 ipvs 处理。
- kube-proxy 的做法是将 Cluster IP 和 EXTERNAL-IP 都绑到名称为 `kube-ipvs0` 的 dummy 网卡，该网卡仅仅用来绑定 VIP (内核自动为其生成 local 路由)，不用于接收流量。
- 内网 CLB 对 NodePort 的探测报文源 IP 是 CLB 自身的 VIP，目的 IP 是 Node IP。当探测报文到达节点时，节点发现源 IP 为本机 IP （因为其被绑定到了 `kube-ipvs0` ），就将其丢弃掉。所以 CLB 的探测报文永远无法收到响应，也就全部探测失败，虽然 CLB 有全死全活逻辑（全部探测失败视为全部可以被转发），但也相当于探测未起到任何作用，在某些情况下将造成一些异常。

为解决上述问题，TKE 的修复策略是：ipvs 模式不绑定 EXTERNAL-IP 到 `kube-ipvs0` 。也就是集群内 Pod 访问 CLB IP 的报文不会进入 INPUT 链，而是直接出节点网卡，真正到达 CLB，这样健康探测的报文进入节点时将不会被当成本机 IP 而丢弃，同时探测响应报文也不会进入 INPUT 链导致出不去。

虽然这种方法修复了 CLB 健康探测失败的问题，但也导致集群内 Pod 访问 CLB 的包真正到了 CLB，由于访问集群内的服务，报文又会被转发回其中一台节点，也就存在了回环的可能性。

> 说明：
> 相关问题已提交至 社区，但目前还未有效解决。

相关答疑

为什么公网 CLB 不存在回环问题？

使用公网 Ingress 和 LoadBalancer 类型公网 Service 不存在回环问题，主要是公网 CLB 收到的报文源 IP 是子机的出口公网 IP，而子机内部无法感知自己的公网 IP，当报文转发回子机时，不认为公网源 IP 是本机 IP，也就不存在回环。

CLB 是否有避免回环机制？

有。CLB 会判断源 IP，如果发现后端 RS 也有相同 IP，就不考虑转发给这个 RS，而是选择其他 RS。但是源 Pod IP 跟后端 RS IP 并不相同，CLB 也不知道这两个 IP 是在同一节点，所以同样可能会转发过去，也就可能发生回环。

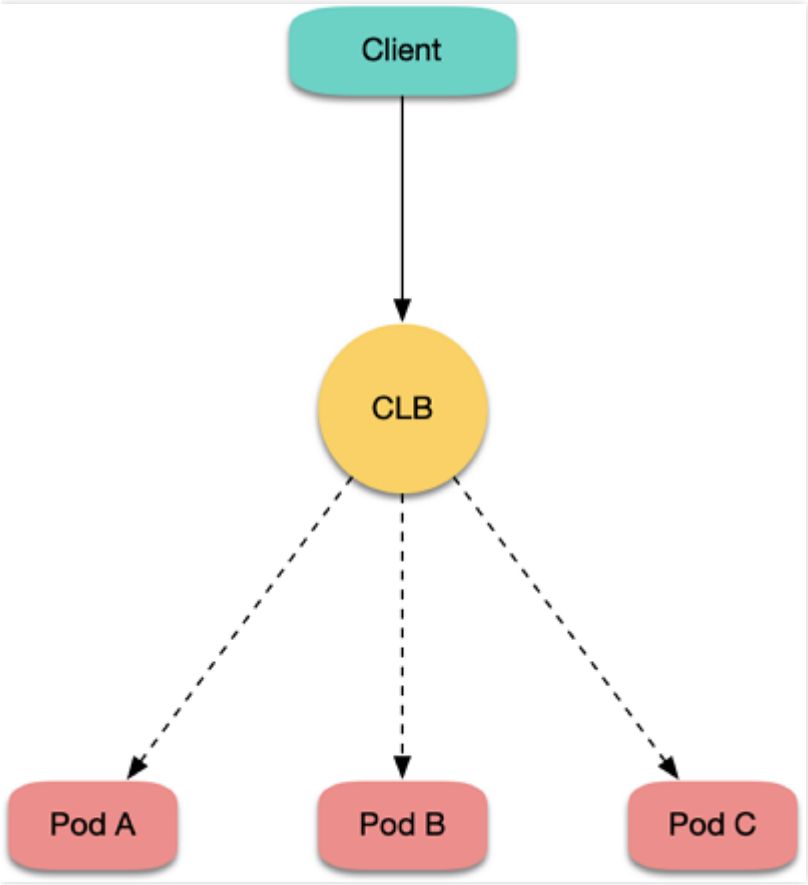
Client 与 Server 反亲和部署能否规避？

如果将 Client 与 Server 通过反亲和性部署，避免 Client 跟 Server 部署在同一节点，能否规避 CLB 回环问题？

默认情况下，LB 通过节点 NodePort 绑定 RS，可能转发给任意节点 NodePort，此时 Client 与 Server 是否在同一节点都可能发生回环。但如果为 Service 设置 `externalTrafficPolicy: Local`， LB 就只会转发到有 Server Pod 的节点，如果 Client 与 Server 通过反亲和调度在不同节点，则此时不会发生回环，所以反亲和 + `externalTrafficPolicy: Local` 可以规避回环问题（包括内网 Ingress 和 LoadBalancer 类型内网 Service）。

VPC-CNI 的 LB 直通 Pod 是否也存在 CLB 回环问题？

TKE 通常用的 Global Router 网络模式（网桥方案），还有一种是 VPC-CNI（弹性网卡方案）。目前 LB 直通 Pod 只支持 VPC-CNI 的 Pod，即 LB 不绑 NodePort 作为 RS，而是直接绑定后端 Pod 作为 RS，如下图所示：



这样即可绕过 NodePort，不再像之前一样可能会转发给任意节点。但如果 Client 与 Server 在同一节点，同样可能会发生回环问题，通过反亲和可以规避。

有什么建议？

反亲和与 `externalTrafficPolicy: Local` 的规避方式不太优雅。一般来讲，访问集群内的服务避免访问本集群的 CLB，因为服务本身在集群内部，从 CLB 绕一圈不仅会增加网络链路的长度，还会引发回环问题。

访问集群内服务尽量使用 Service 名称，例如 `server.prod.svc.cluster.local` ，通过这样的配置将不会经过 CLB，也不会导致回环问题。

如果业务有耦合域名，不能使用 Service 名称，可以使用 coredns 的 rewirte 插件，将域名指向集群内的 Service，coredns 配置示例如下：

```
apiVersion: v1
kind: ConfigMap
metadata:
name: coredns
namespace: kube-system
data:
Corefile: |2-
.:53 {
rewrite name roc.oa.com server.prod.svc.cluster.local
...
```

如果多个 Service 共用一个域名，可以自行部署 Ingress Controller (例如 nginx-ingress)：

1. 用上述 rewrite 的方法将域名指向自建的 Ingress Controller。
2. 将自建的 Ingress 根据请求 location (域名+路径) 匹配 Service，再转发给后端 Pod。整段链路不经过 CLB，同意能规避回环问题。