



下载APP



11 | 容器文件系统：我在容器中读写文件怎么变慢了？

2020-12-09 李程远

容器实战高手课

[进入课程 >](#)**讲述：李程远**

时长 16:07 大小 14.77M



你好，我是程远。从这一讲开始，我们进入容器存储这个模块。

这一模块我们所讲的内容，都和容器里的文件读写密切相关。因为所有的容器的运行都需要一个容器文件系统，那么我们就从容器文件系统先开始讲起。

那我们还是和以前一样，先来看看我之前碰到了什么问题。

这个问题具体是我们在宿主机上，把 Linux 从 ubuntu18.04 升级到 ubuntu20.04 之后发现的。



在我们做了宿主机的升级后，启动了一个容器，在容器里用 fio 这个磁盘性能测试工具，想看一下容器里文件的读写性能。结果我们很惊讶地发现，在 ubuntu 20.04 宿主机上的

容器中文件读写的性能只有 ubuntu18.04 宿主机上的 1/8 左右了，那这是怎么回事呢？

问题再现

这里我提醒一下你，因为涉及到两个 Linux 的虚拟机，问题再现这里我为你列出了关键的结果输出截图，不方便操作的同学可以重点看其中的思路。

我们可以先启动一个 ubuntu18.04 的虚拟机，它的 Linux 内核版本是 4.15 的，然后在虚拟机上用命令 `docker run -it ubuntu:18.04 bash` 启动一个容器，接着在容器里运行 `fio` 这条命令，看一下在容器中读取文件的性能。

 复制代码

```
1 # fio -direct=1 -iodepth=64 -rw=read -ioengine=libaio -bs=4k -size=10G -numjob
```

这里我给你解释一下 `fio` 命令中的几个主要参数：

第一个参数是 `-direct=1`，代表采用非 buffered I/O 文件读写的方式，避免文件读写过程中内存缓冲对性能的影响。

接着我们来看这 `-iodepth=64` 和 `-ioengine=libaio` 这两个参数，这里指文件读写采用异步 I/O (Async I/O) 的方式，也就是进程可以发起多个 I/O 请求，并且不用阻塞地等待 I/O 的完成。稍后等 I/O 完成之后，进程会收到通知。

这种异步 I/O 很重要，因为它可以极大地提高文件读写的性能。在这里我们设置了同时发出 64 个 I/O 请求。

然后是 `-rw=read`，`-bs=4k`，`-size=10G`，这几个参数指这个测试是个读文件测试，每次读 4KB 大小数块，总共读 10GB 的数据。

最后一个参数是 `-numjobs=1`，指只有一个进程 / 线程在运行。

所以，这条 `fio` 命令表示我们通过异步方式读取了 10GB 的磁盘文件，用来计算文件的读取性能。

那我们看到在 ubuntu 18.04，内核 4.15 上的容器 I/O 性能是 584MB/s 的带宽，IOPS (I/O per second) 是 150K 左右。

```
root@b035dd918695:/tmp# uname -r
4.15.0-118-generic
root@b035dd918695:/tmp# fio -direct=1 -iodepth=64 -rw=read -ioengine=libaio -bs=4k -size=10G -numjobs=1 -name=./fio.test
./fio.test: (g=0): rw=read, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=libaio, iodepth=64
fio-3.16
Starting 1 process
Jobs: 1 (f=1): [R(1)][100.0%][r=584MiB/s,w=0KiB/s][r=150k,w=0 IOPS][eta 00m:00s]
```

同样我们再启动一个 ubuntu 20.04，内核 5.4 的虚拟机，然后在它的上面也启动一个容器。

我们运行 `docker run -it ubuntu:20.04 bash`，接着在容器中使用同样的 `fio` 命令，可以看到它的 I/O 性能是 70MB 带宽，IOPS 是 18K 左右。实践证明，这的确比老版本的 ubuntu 18.04 差了很多。

```
root@f908d74df063:/tmp# uname -r
5.4.0-48-generic
root@f908d74df063:/tmp# fio -direct=1 -iodepth=64 -rw=read -ioengine=libaio -bs=4k -size=10G -numjobs=1 -name=./fio.test
./fio.test: (g=0): rw=read, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=libaio, iodepth=64
fio-3.16
Starting 1 process
Jobs: 1 (f=1): [R(1)][16.1%][r=70.7MiB/s][r=18.1k IOPS][eta 02m:10s]
```

知识详解

如何理解容器文件系统？

刚才我们对比了升级前后的容器读写性能差异，那想要分析刚刚说的这个性能差异，我们需要先理解容器的文件系统。

我们在容器里，运行 `df` 命令，你可以看到在容器中根目录 (/) 的文件系统类型是 "overlay"，它不是我们在普通 Linux 节点上看到的 Ext4 或者 XFS 之类常见的文件系统。

那么看到这里你肯定想问，Overlay 是一个什么样的文件系统呢，容器为什么要用这种文件系统？别急，我会一步一步带你分析。

```
root@fd09d29e5386:/# df
Filesystem      1K-blocks    Used Available Use% Mounted on
overlay          29423440 24559308   3558172   88% /
```

在说容器文件系统前，我们先来想象一下如果没有文件系统管理的话会怎样。假设有这么一个场景，在一个宿主机上需要运行 100 个容器。

在我们这个课程的 [🔗 第一讲](#)里，我们就说过每个容器都需要一个镜像，这个镜像就把容器中程序需要运行的二进制文件，库文件，配置文件，其他的依赖文件等全部都打包成一个镜像文件。

如果没有特别的容器文件系统，只是普通的 Ext4 或者 XFS 文件系统，那么每次启动一个容器，就需要把一个镜像文件下载并且存储在宿主机上。

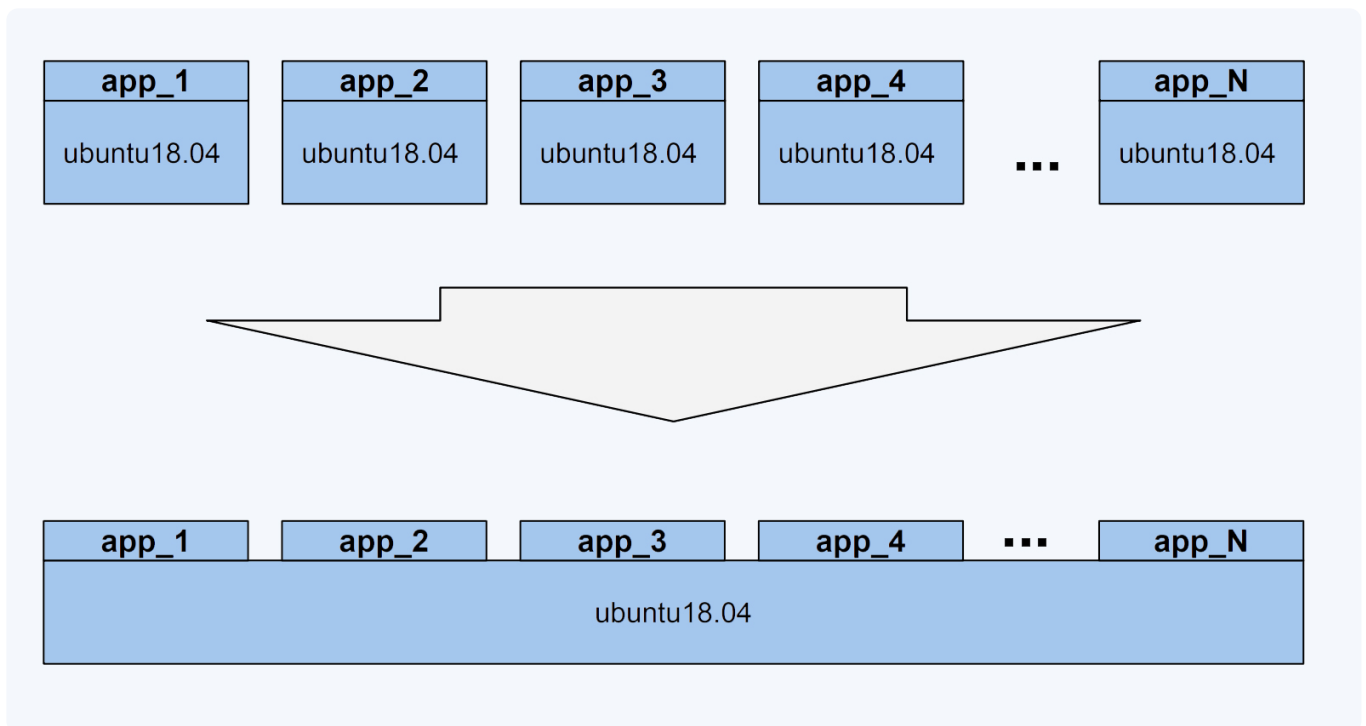
我举个例子帮你理解，比如说，假设一个镜像文件的大小是 500MB，那么 100 个容器的话，就需要下载 $500\text{MB} \times 100 = 50\text{GB}$ 的文件，并且占用 50GB 的磁盘空间。

如果你再分析一下这 50GB 里的内容，你会发现，在绝大部分的操作系统里，库文件都是差不多的。而且，在容器运行的时候，这类文件也不会被改动，基本上都是只读的。

特别是这样的情况：假如这 100 个容器镜像都是基于 "ubuntu:18.04" 的，每个容器镜像只是额外复制了 50MB 左右自己的应用程序到 "ubuntu: 18.04" 里，那么就是说在总共 50GB 的数据里，有 90% 的数据是冗余的。

讲到这里，你不难推测出理想的情况应该是什么样的？

没错，当然是在一个宿主机上只要下载并且存储一份 "ubuntu:18.04"，所有基于 "ubuntu:18.04" 镜像的容器都可以共享这一份通用的部分。这样设置的话，不同容器启动的时候，只需要下载自己独特的程序部分就可以。就像下面这张图展示的这样。



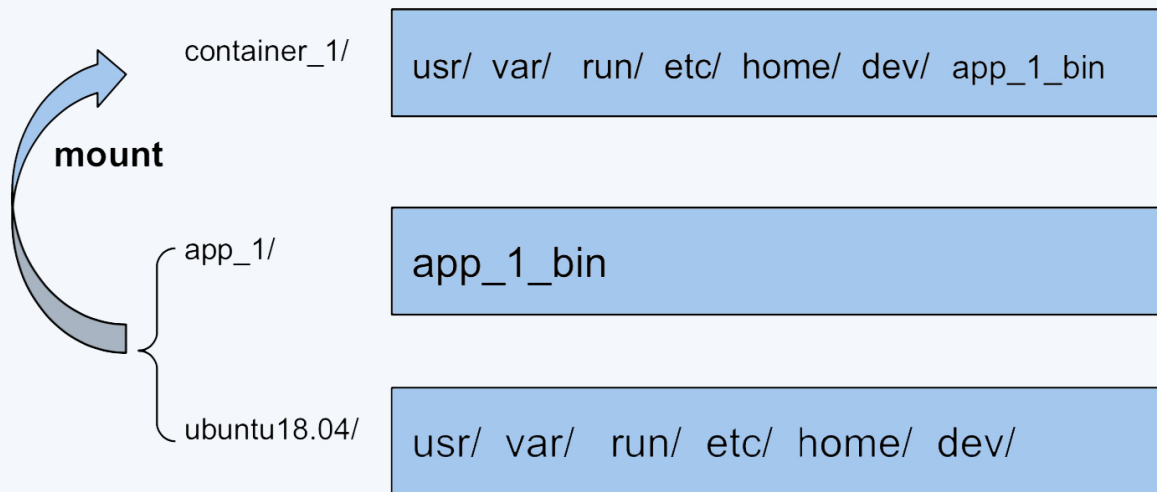
正是为了有效地减少磁盘上冗余的镜像数据，同时减少冗余的镜像数据在网络上的传输，选择一种针对于容器的文件系统是很有必要的，而这类的文件系统被称为 UnionFS。

UnionFS 这类文件系统实现的主要功能是把多个目录（处于不同的分区）一起挂载（mount）在一个目录下。这种多目录挂载的方式，正好可以解决我们刚才说的容器镜像的问题。

比如，我们可以把 `ubuntu18.04` 这个基础镜像的文件放在一个目录 `ubuntu18.04/` 下，容器自己额外的程序文件 `app_1_bin` 放在 `app_1/` 目录下。

然后，我们把这两个目录挂载到 `container_1/` 这个目录下，作为容器 1 看到的文件系统；对于容器 2，就可以把 `ubuntu18.04/` 和 `app_2/` 两个目录一起挂载到 `container_2` 的目录下。

这样在节点上我们只要保留一份 `ubuntu18.04` 的文件就可以了。




OverlayFS

UnionFS 类似的有很多种实现，包括在 Docker 里最早使用的 AUFS，还有目前我们使用的 OverlayFS。前面我们在运行df的时候，看到的文件系统类型"overlay"指的就是 OverlayFS。

在 Linux 内核 3.18 版本中，OverlayFS 代码正式合入 Linux 内核的主分支。在这之后，OverlayFS 也就逐渐成为各个主流 Linux 发行版本里缺省使用的容器文件系统了。

网上 Julia Evans 有个 [blog](#)，里面有个的 OverlayFS 使用的例子，很简单，我们也拿这个例子来理解一下 OverlayFS 的一些基本概念。

你可以先执行一下这一组命令。

 复制代码

```
1 #!/bin/bash
2
3 umount ./merged
4 rm upper lower merged work -r
5
6 mkdir upper lower merged work
7 echo "I'm from lower!" > lower/in_lower.txt
8 echo "I'm from upper!" > upper/in_upper.txt
9 # `in_both` is in both directories
10 echo "I'm from lower!" > lower/in_both.txt
11 echo "I'm from upper!" > upper/in_both.txt
12
```



```
13 sudo mount -t overlay overlay \  
14 -o lowerdir=./lower,upperdir=./upper,workdir=./work \  
15 ./merged
```

我们可以看到，OverlayFS 的一个 mount 命令牵涉到四类目录，分别是 lower，upper，merged 和 work，那它们是什么关系呢？

我们看下面这张图，这和前面 UnionFS 的工作示意图很像，也不奇怪，OverlayFS 就是 UnionFS 的一种实现。接下来，我们从下往上依次看看每一层的功能。

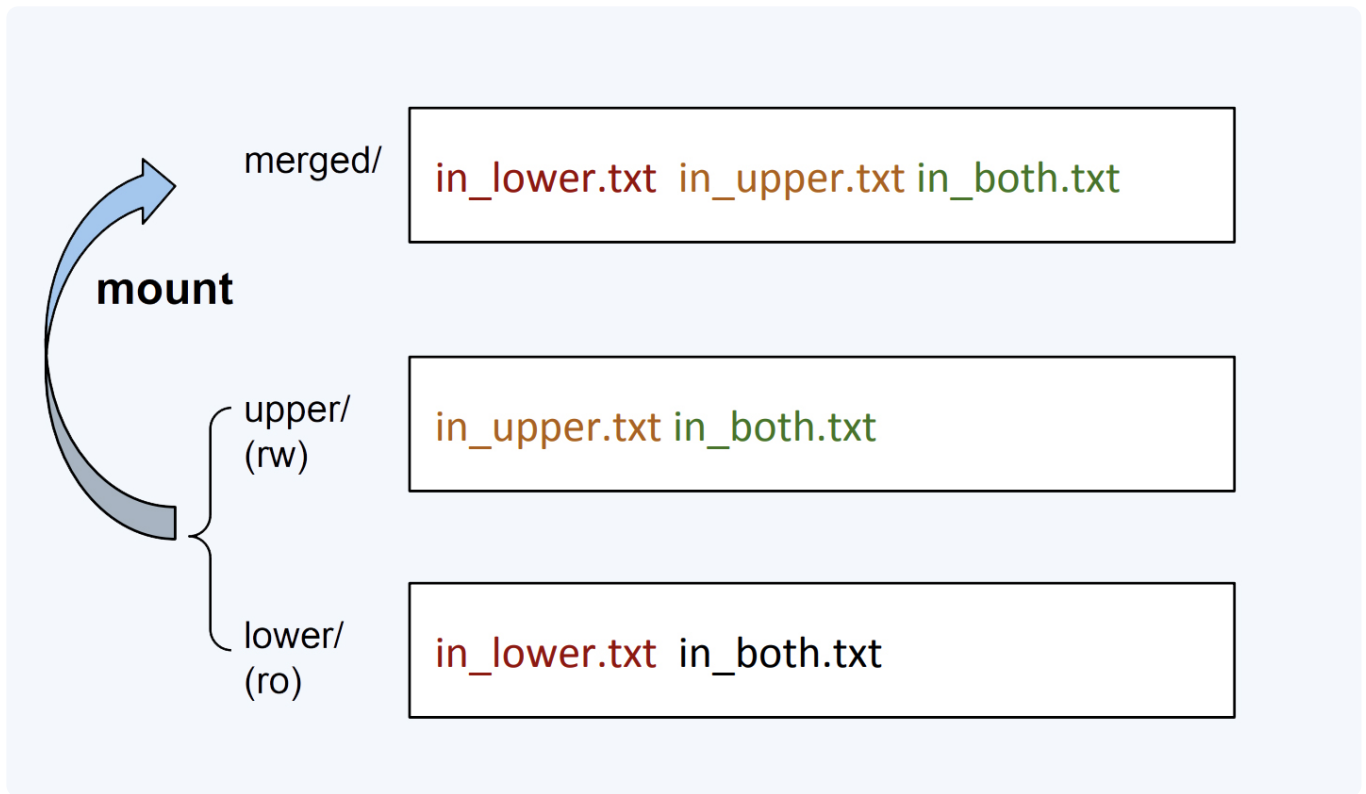
首先，最下面的"lower/"，也就是被 mount 两层目录中底下的这层（lowerdir）。

在 OverlayFS 中，最底下这一层里的文件是不会被修改的，你可以认为它是只读的。我还想提醒你一点，在这个例子里我们只有一个 lower/ 目录，不过 OverlayFS 是支持多个 lowerdir 的。

然后我们看"upper/"，它是被 mount 两层目录中上面的这层（upperdir）。在 OverlayFS 中，如果有文件的创建，修改，删除操作，那么都会在这一层反映出来，它是可读写的。

接着是最上面的"merged"，它是挂载点（mount point）目录，也是用户看到的目录，用户的实际文件操作在这里进行。

其实还有一个"work/"，这个目录没有在这个图里，它只是一个存放临时文件的目录，OverlayFS 中如果有文件修改，就会在中间过程中临时存放文件到这里。



从这个例子我们可以看到，OverlayFS 会 mount 两层目录，分别是 lower 层和 upper 层，这两层目录中的文件都会映射到挂载点上。

从挂载点的视角看，upper 层的文件会覆盖 lower 层的文件，比如"in_both.txt"这个文件，在 lower 层和 upper 层都有，但是挂载点 merged/ 里看到的只是 upper 层里的 in_both.txt.

如果我们在 merged/ 目录里做文件操作，具体包括这三种。

第一种，新建文件，这个文件会出现在 upper/ 目录中。

第二种是删除文件，如果我们删除"in_upper.txt"，那么这个文件会在 upper/ 目录中消失。如果删除"in_lower.txt"，在 lower/ 目录里的"in_lower.txt"文件不会有变化，只是在 upper/ 目录中增加了一个特殊文件来告诉 OverlayFS，"in_lower.txt"这个文件不能出现在 merged/ 里了，这就表示它已经被删除了。


```
# cd merged/
# ls
in_both.txt  in_lower.txt  in_upper.txt
# ls ../upper/
in_both.txt  in_upper.txt
# rm in_lower.txt
# ls -l ../upper/
total 8
-rw-r--r-- 1 root root 16 Nov 29 07:13 in_both.txt
c----- 1 root root 0, 0 Nov 29 07:14 in_lower.txt
-rw-r--r-- 1 root root 16 Nov 29 07:13 in_upper.txt
```

还有一种操作是修改文件，类似如果修改"in_lower.txt"，那么就会在 upper/ 目录中新建一个"in_lower.txt"文件，包含更新的内容，而在 lower/ 中的原来的实际文件"in_lower.txt"不会改变。

通过这个例子，我们知道了 OverlayFS 是怎么工作了。那么我们可以再想一想，怎么把它运用到容器的镜像文件上？

其实也不难，从系统的 mounts 信息中，我们可以看到 Docker 是怎么用 OverlayFS 来挂载镜像文件的。容器镜像文件可以分成多个层（layer），每层可以对应 OverlayFS 里 lowerdir 的一个目录，lowerdir 支持多个目录，也就可以支持多层的镜像文件。

在容器启动后，对镜像文件中修改就会被保存在 upperdir 里了。

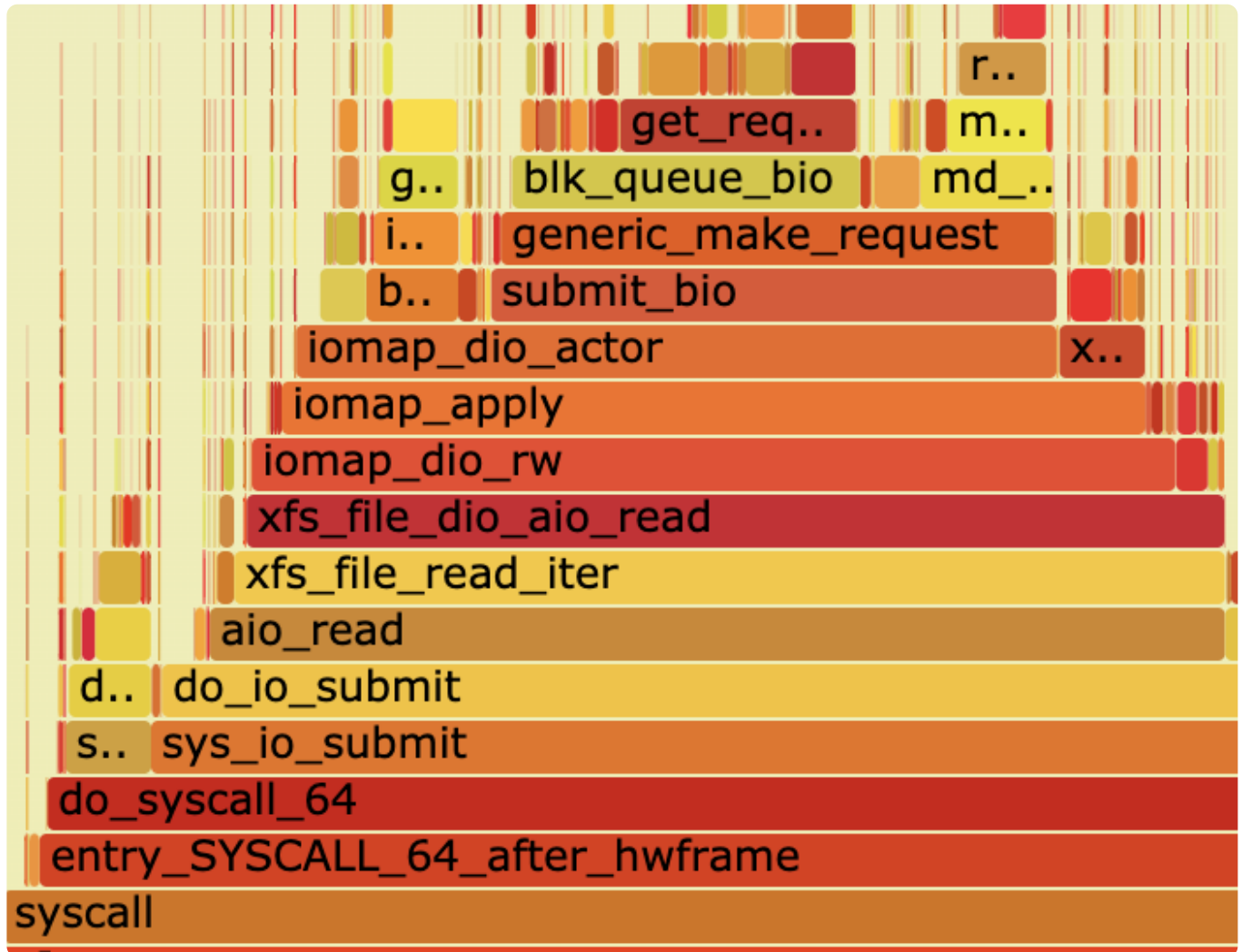
```
root@ubuntu2004-4082232:~# cat /proc/mounts | grep overlay
overlay /var/lib/docker/overlay2/5ce98fde586d534f5d33a1b77dc6a86df49a1af3cb724f7e06979dd121c8b420/merged overlay rw,relatime,lowerdir=/var/lib/docker/overlay2/l/CTVCLTHTIRQM7TMJ7QCZYBDS6:/var/lib/docker/overlay2/l/Z2YQSK653LC2YBK0TCSF6KNCVA:/var/lib/docker/overlay2/l/RWQWZMH3NYASH73ZTIHXV3RXI4:/var/lib/docker/overlay2/l/MS2TP2KVQ7BND6S70IZ03307VF,upperdir=/var/lib/docker/overlay2/5ce98fde586d534f5d33a1b77dc6a86df49a1af3cb724f7e06979dd121c8b420/diff,workdir=/var/lib/docker/overlay2/5ce98fde586d534f5d33a1b77dc6a86df49a1af3cb724f7e06979dd121c8b420/work,xino=off 0 0
```

解决问题

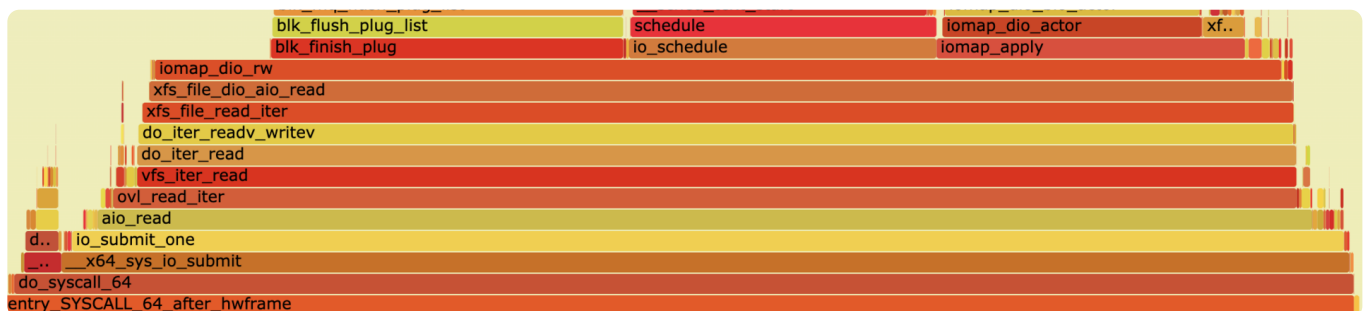
在理解了容器使用的 OverlayFS 文件系统后，我们再回到开始的问题，为什么在宿主机升级之后，在容器里读写文件的性能降低了？现在我们至少应该知道，在容器中读写文件性能降低了，那么应该是 OverlayFS 的性能在新的 ubuntu20.04 中降低了。

要找到问题的根因，我们还需要进一步的 debug。对于性能问题，我们需要使用 Linux 下的 perf 工具来查看一下，具体怎么使用 perf 来解决问题，我们会在后面讲解。

这里你只要看一下结果就可以了，自下而上是函数的一个调用顺序。通过 perf 工具，我们可以比较在容器中运行 fio 的时候，ubuntu 18.04 和 ubuntu 20.04 在内核函数调用上的不同。



ubuntu 18.04 (Linux内核4.15)环境下使用perf输出的函数调用结果



ubuntu 20.04 (Linux内核 5.4)环境下使用perf输出的函数调用结果

我们从系统调用框架之后的函数 `aio_read()` 开始比较：Linux 内核 4.15 里 `aio_read()` 之后调用的是 `xfs_file_read_iter()`，而在 Linux 内核 5.4 里，`aio_read()` 之后调用的是 `ovl_read_iter()` 这个函数，之后再调用 `xfs_file_read_iter()`。

这样我们就可以去查看一下，在内核 4.15 之后新加入的这个函数 `ovl_read_iter()` 的代码。

查看 [代码](#)后我们就能明白，Linux 为了完善 OverlayFS，增加了 OverlayFS 自己的 read/write 函数接口，从而不再直接调用 OverlayFS 后端文件系统（比如 XFS，Ext4）的读写接口。但是它只实现了同步 I/O（sync I/O），并没有实现异步 I/O。

而在 fio 做文件系统性能测试的时候使用的是异步 I/O，这样才可以得到文件系统的性能最大值。所以，在内核 5.4 上就无法对 OverlayFS 测出最高的性能指标了。

在 Linux 内核 5.6 版本中，这个问题已经通过下面的这个补丁给解决了，有兴趣的同学可以看一下。

[复制代码](#)

```
1 commit 2406a307ac7ddfd7effeeaff6947149ec6a95b4e
2 Author: Jiufei Xue <jiufei.xue@linux.alibaba.com>
3 Date:   Wed Nov 20 17:45:26 2019 +0800
4
5     ovl: implement async IO routines
6
7     A performance regression was observed since linux v4.19 with aio test usin
8     fio with iodepth 128 on overlayfs.  The queue depth of the device was
9     always 1 which is unexpected.
10
11    After investigation, it was found that commit 16914e6fc7e1 ("ovl: add
12    ovl_read_iter()") and commit 2a92e07edc5e ("ovl: add ovl_write_iter()")
13    resulted in vfs_iter_{read,write} being called on underlying filesystem,
14    which always results in synchronous IO.
15
16    Implement async IO for stacked reading and writing.  This resolves the
17    performance regresion.
18
19    This is implemented by allocating a new kiocb for submitting the AIO
20    request on the underlying filesystem.  When the request is completed, the
21    new kiocb is freed and the completion callback is called on the original
22    iocb.
23
24    Signed-off-by: Jiufei Xue <jiufei.xue@linux.alibaba.com>
25    Signed-off-by: Miklos Szeredi <mszeredi@redhat.com>
```

重点总结

这一讲，我们最主要的内容是理解容器文件系统。为什么要有容器自己的文件系统？很重要的一点是**减少相同镜像文件在同一个节点上的数据冗余，可以节省磁盘空间，也可以减少镜像文件下载占用的网络资源。**

作为容器文件系统，UnionFS 通过多个目录挂载的方式工作。OverlayFS 就是 UnionFS 的一种实现，是目前主流 Linux 发行版本中缺省使用的容器文件系统。


OverlayFS 也是把多个目录合并挂载，被挂载的目录分为两大类：lowerdir 和 upperdir。

lowerdir 允许有多个目录，在被挂载后，这些目录里的文件都是不会被修改或者删除的，也就是只读的；upperdir 只有一个，不过这个目录是可读写的，挂载点目录中的所有文件修改都会在 upperdir 中反映出来。

容器的镜像文件中各层正好作为 OverlayFS 的 lowerdir 的目录，然后加上一个空的 upperdir 一起挂载好后，就组成了容器的文件系统。

OverlayFS 在 Linux 内核中还在不断的完善，比如我们在这一讲看到的在 kernel 5.4 中对异步 I/O 操作的缺失，这也是我们在使用容器文件系统的时候需要注意的。

思考题

在这一讲 OverlayFS 的  例子的基础上，建立 2 个 lowerdir 的目录，并且在目录中建立相同文件名的文件，然后一起做一个 overlay mount，看看会发生什么？

欢迎在留言区和我分享你的思考和疑问。如果这篇文章让你有所收获，也欢迎分享给你的同事、朋友，一起学习探讨。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | Swap：容器可以使用Swap空间吗？

下一篇 12 | 容器文件Quota：容器为什么把宿主机的磁盘写满了？

精选留言 (11)

写留言



Geek3340

2020-12-10

老师提到的aufs，是完全被废弃吗？aufs的废弃是指在内核层的废弃吗？之前安装docker时，时可以配置使用aufs 还是overlay2，也就是说内核层还未完全去除对aufs的支持吗？

作者回复: aufs的代码从来就没有进入Linux内核的主干。



1



流浪地球

2020-12-09

和老师探讨一个问题，本文中描述的现象，一个重要的原因是容器镜像里只有rootfs，没有linux内核，宿主机上的所有容器是共用宿主机内核的。所以，当宿主机内核版本升级后，容器镜像并没有相应的升级，也会产生这个问题，文中并没有对这个知识要点说明。不知道我的理解是否正确

展开

作者回复: @流浪地球

容器镜像中只有rootfs没有Linux内核是对的。

在文章里，宿主机内核升级后，无论容器的镜像是否升级，都会有这个问题。文中的问题是overlayfs引起的，和镜像中的文件没有关系。

1

1



谢哈哈

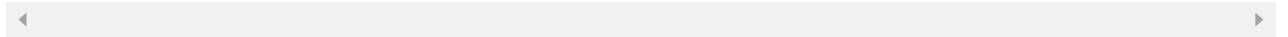
2020-12-09

经过实验确认，只会在merge即联合挂载点里生成一个文件名，也就是说overlay文件系统

为了省存储空间是做了同名文件合并优化的

展开 ▾

作者回复: 赞!



💬 1

👍 1



姜姜

2020-12-17

思考题:

我认为在多个lower之间出现同名文件，在merge中也是上层lower覆盖下层lower。

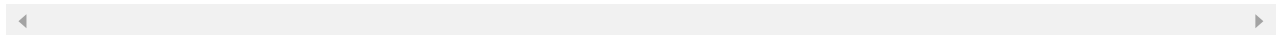
个人理解:

“merge层” 相当于提供给用户进行交互的视图层; ...

展开 ▾

作者回复: @姜姜

可以用文档中的脚本稍微修改一下，就可以验证了



💬

👍



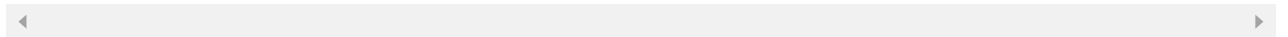
水蒸蛋

2020-12-16

老师，我没想通容器既然有内核文件为什么会依赖宿主机的内核，如果依赖宿主机内核那还要容器的内核干什么呢，这么多不同的linux版本难道内核版本都是一样的？

作者回复: @水蒸蛋

容器和宿主机是共享内核的。不通的Linux版本内核不一样。



💬

👍



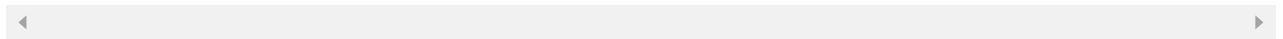
Alery

2020-12-13

老师，假如我将一个卷(宿主机上的某个目录)挂在到容器文件系统上的某个目录，我在容器中对这个卷中的数据做读写操作，因为这个挂载路径也是容器文件系统的一部分，性能是不是也是会有影响的？

作者回复: 如果以volume的方式挂载到容器中，那么它就不是以overlayfs的文件系统。

性能是否影响要看volume目录的位置在哪个物理磁盘上，和它共享物理磁盘的有哪些读写进程。

**Helios**

2020-12-12

相同文件名只会保留一个，文件内容不同应该是按照顺序了吧

**上邪忘川**

2020-12-10

实验过程如下，结果是lower1目录中的文件覆盖了lower2中同名的文件。

```
[root@localhost ~]# cat overlay.sh
```

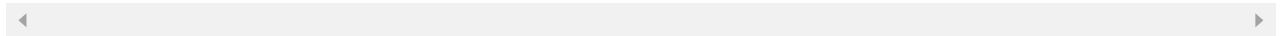
```
#!/bin/bash
```

```
umount ./merged...
```

展开 ∨

作者回复: @上邪忘川

很好的测试步骤。我再问一个问题，“merged/in_lower.txt”里的值有可能是"I'm from lower 2!"吗？

**美美**

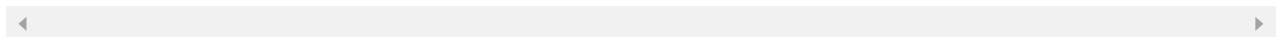
2020-12-10

想和老师探讨一个非技术问题，老师对当前k8s和docker的相爱相杀有什么看法？老师觉得未来docker市场会被podman取代吗？如果会的话，这个过程大概要多久

展开 ∨

作者回复: 容器云平台里，k8s肯定是主流，用了k8s, 基本就不需要docker了，启动容器的程序肯定是越简单越好。

我们在2019年初就不用docker了。

**closer**

2020-12-09

请教老师一个关于容器真实的生产问题，我们在fio在openstack做性能测试ioengine labiao 和

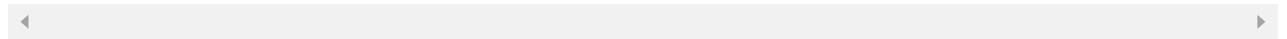
psync 的IOPS 差距巨大。但是在传统kvm架构上面labiao psync的IOPS差不多。导致在openstack上面的容器磁盘读写都很慢。这个确实是openstack的问题引起的吗？底层硬件配置都相同的条件下

展开 ∨

作者回复: @closer

我没有理解你的测试环境比较，

我的理解是openstack上应该是运行kvm的VM的，这个和你说的"在传统kvm架构上面"，是什么差别？



良凯尔

2020-12-09

思考题：在这一讲 OverlayFS 的例子的基础上，建立 2 个 lowerdir 的目录，并且在目录中建立相同文件名的文件，然后一起做一个 overlay mount，看看会发生什么？

做了 overlay mount之后，只能看到上面那一层lowerdir目录中的同名文件，是这样吗

展开 ∨

作者回复: @良凯尔

可以动手试试。

