

Nigel Brown

Search



Subscribe via RSS

About Docker Training

GitHub LinkedIn Twitter Contact

© 2017 Nigel Brown • All rights reserved.

Proudly published with Ghost

The Overlay Filesystem

2 years ago

Docker · Overlay · Overlayfs · Union Mount · Aufs · Btrfs · Devicemapper · ThinP · rkt · Graphdriver

The overlay filesystem (formally known as overlayfs) was merged into the mainline Linux kernel at version 3.18 in December 2014. Whilst other, similar union mount filesystems have been around for many years (notably, [aufs](#)), overlay is the first to become [integrated into the Linux kernel](#).

An overlay sits on top of an existing filesystem, and combines an upper and a lower directory tree (which can be from different filesystems), in order to present a unified representation of both directory trees. Where objects with the same name exist in both directory trees, then their treatment depends on the object type:

- **File:** the object in the upper directory tree appears in the overlay, whilst the object in the lower directory tree is hidden

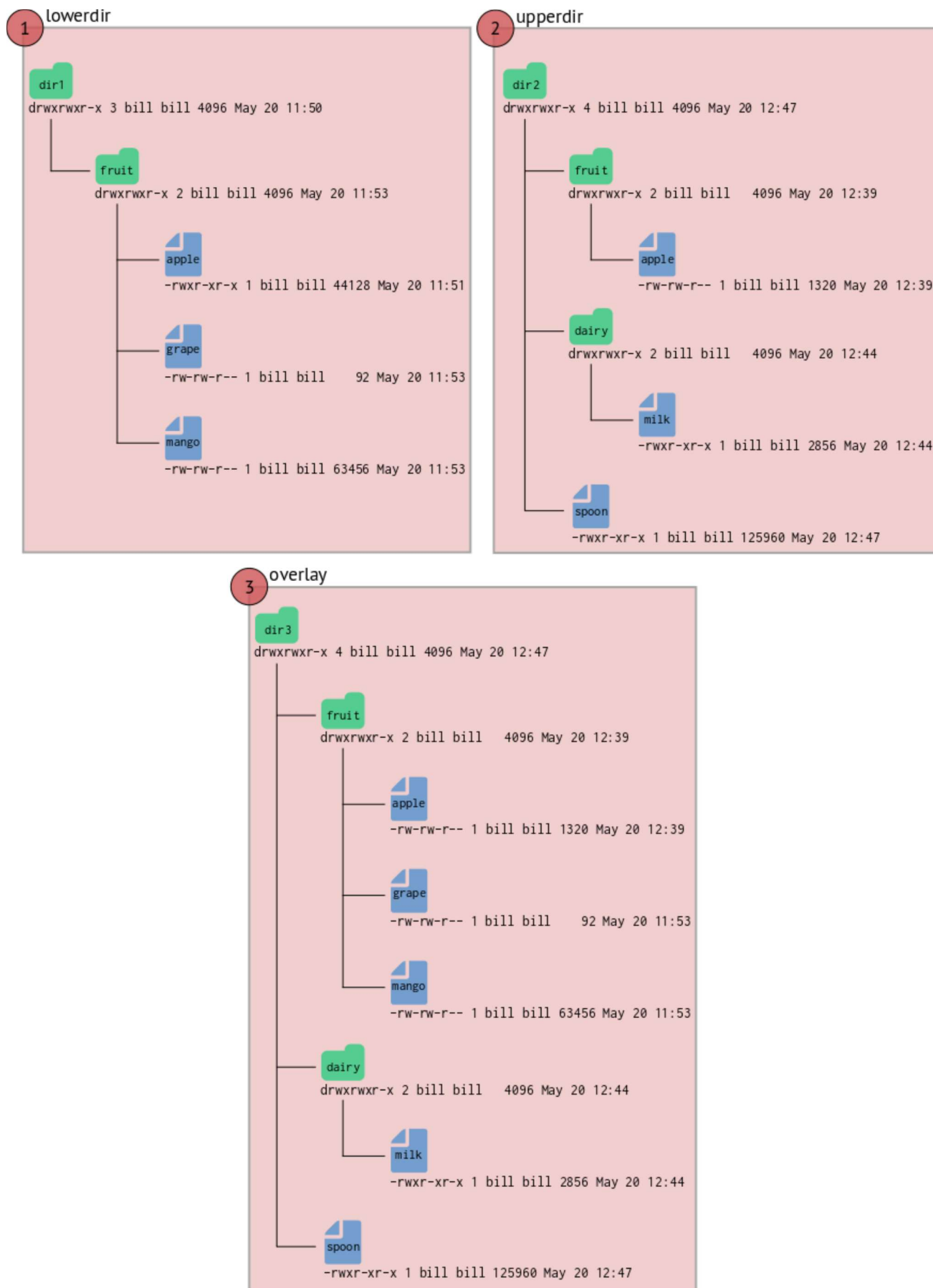
- **Directory:** the contents of each directory object are merged to create a combined directory object in the overlay

The lower directory can be read-only, and could be an overlay itself, whilst the upper directory is normally writeable. In order to create an overlay of two directories, `dir1` and `dir2`, we can use the following mount command:

```
mount -t overlay -o lowerdir=./dir1,upperdir=./dir2,workd
```



A union of the two directories is created as an overlay in the `dir3` directory. The `workdir` option is required, and used to prepare files before they are switched to the overlay destination in an atomic action (the `workdir` needs to be on the same filesystem as the `upperdir`). The following illustrates a simple example of the overlay mount above:



When a file or directory that originates in the upper directory is removed from the overlay, it's also removed from the upper directory. If a file or directory that originates in the lower directory is removed from the overlay, it remains in the lower directory, but a 'whiteout' is created in the upper directory. A whiteout takes the form of a character device with device number 0/0, and a name identical to the removed object. The result of the whiteout creation means that the object in the lower directory is ignored, whilst the whiteout itself is not visible in the overlay. The following illustrates the creation of a whiteout in the `upperdir` on removal of the file `mango`:

```
$ ls -l ./dir3/fruit
total 72
-rw-rw-r-- 1 bill bill 1320 May 20 12:39 apple
-rw-rw-r-- 1 bill bill  92 May 20 11:53 grape
-rw-rw-r-- 1 bill bill 63456 May 20 11:53 mango
$ rm ./dir3/fruit/mango
$ ls -l ./dir3/fruit
total 8
-rw-rw-r-- 1 bill bill 1320 May 20 12:39 apple
-rw-rw-r-- 1 bill bill  92 May 20 11:53 grape
$ ls -l ./dir2/fruit
total 4
-rw-rw-r-- 1 bill bill 1320 May 20 12:39 apple
c----- 1 bill bill 0, 0 May 20 17:38 mango
$
```

Linux kernel 4.0 further extends the overlay capabilities, to enable multiple lower directories to be specified, separated by a `:`, with the rightmost lower directory on the bottom, and

the leftmost lower directory on the top of the union. For example:

```
mount -t overlay -o lowerdir=./dir3:./dir2:./dir1 overlay
```



In this extended version, the `upperdir` is optional, and if it is omitted, then the `workdir` option is also optional, and will be ignored in any case. In this scenario, the overlay will be read-only.

At the time of writing, Linux kernel version 4.0 is very new, and will not have found its way into many Linux distributions.

Use Cases

Union filesystems are often used for Live CD creation, where a read-only image is augmented with a writeable layer in tmpfs, thereby enabling a dynamic, but ephemeral session.

Effectively, this is 'copy-on-write', where read-only data is used until such time as the data requires changing, whereupon it is copied and altered in the read-write layer. This copy-on-write mechanism is used in the creation of filesystems for Linux containers, used by container runtime environments like [Docker](#) or [rkt](#). It's not the only option for assembling container filesystems, but it is one of the more performant, because it allows pages in the kernel's page cache to be shared between containers - an option which is not available with block device copy-on-write mechanisms, such as the device mapper framework with the [thinp](#) target, or the [btrfs](#) filesystem. Docker's overlay graphdriver is currently the [last in the queue](#) for automatic selection (vfs is used for testing), behind aufs, btrfs and the devicemapper

graphdrivers, but as the remaining issues are closed out, I expect it to become the default.

Nigel Brown

IT Professional

📍 *Woodbridge, United Kingdom*

Share this post

Use the OverlayFS storage driver

Estimated reading time: 17 minutes

OverlayFS is a modern *union filesystem* that is similar to AUFS, but faster and with a simpler implementation. Docker provides a storage driver for OverlayFS.

This topic refers to the Linux kernel driver as `OverlayFS` and to the Docker storage driver as `overlay` or `overlay2`.

Note: Note: If you use OverlayFS, use the `overlay2` driver rather than the `overlay` driver, because it is more efficient in terms of inode utilization. To use the new driver, you need version 4.0 or higher of the Linux kernel.

For more information about differences between `overlay` vs `overlay2`, refer to [Select a storage driver](https://docs.docker.com/engine/userguide/storagedriver/selectadriver/#overlay-vs-overlay2) (<https://docs.docker.com/engine/userguide/storagedriver/selectadriver/#overlay-vs-overlay2>).

Prerequisites

OverlayFS is supported if you meet the following prerequisites:

- **Docker CE only.** OverlayFS is not supported on Docker EE. See [Product compatibility matrix](https://success.docker.com/Policies/Compatibility_Matrix) (https://success.docker.com/Policies/Compatibility_Matrix) for a list of supported storage drivers for each Docker EE platform.
- Version 4.0 or higher of the Linux kernel. If you use an older kernel, you will need to use the `overlay` driver, which is not recommended.
- The following backing filesystems are supported:
 - `ext4` (RHEL 7.1 only)
 - `xfs` (RHEL 7.2 and higher), but only with `d_type=true` enabled. Use `xfs_info` to verify that the `ftype` option is set to `1`. To format an `xfs` filesystem correctly, use the flag `-n ftype=1`.
- Changing the storage driver will make any containers you have already created inaccessible on the local system. Use `docker save` to save containers, and push existing images to Docker Hub or a private repository, so that you not need to re-create them later.

Configure Docker with the `overlay` or `overlay2` storage driver

To configure Docker to use the `overlay` storage driver your Docker host must be running version 3.18 of the Linux kernel (preferably newer) with the `overlay` kernel module loaded. For the `overlay2` driver, the version of your kernel must be 4.0 or newer. It is highly recommended that you use `overlay2` if possible.

Before following this procedure, you must first meet all the prerequisites ([/engine/userguide/storagedriver/overlayfs-driver/#prerequisites](https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/#prerequisites)).

1. Stop Docker.

```
$ sudo systemctl stop docker
```

2. Copy the contents of `/var/lib/docker` to a temporary location.

```
$ cp -au /var/lib/docker /var/lib/docker.bk
```

3. If you want to use a separate backing filesystem from the one used by `/var/lib/`, format the filesystem and mount it into `/var/lib/docker`. Make sure add this mount to `/etc/fstab` to make it permanent.
4. Edit `/etc/docker/daemon.json`. If it does not yet exist, create it. Assuming that the file was empty, add the following contents.

```
{
  "storage-driver": "overlay2"
}
```

If you need to use the legacy `overlay` driver, specify it instead.

To use `overlay2` on CentOS (Docker CE only), you must also set the storage option `overlay2.override_kernel_check`. In this case the `daemon.json` would look like this:

```
{
  "storage-driver": "overlay2",
  "storage-opts": [
    "overlay2.override_kernel_check=true"
  ]
}
```

Docker will not start if the `daemon.json` file contains badly-formed JSON.

5. Start Docker.

```
$ sudo systemctl start docker
```

6. Verify that the daemon is using the `overlay` / `overlay2` storage driver. Use the `docker info` command and look for `Storage Driver` and `Backing filesystem`.

```
$ docker info

Containers: 0
Images: 0
Storage Driver: overlay
  Backing Filesystem: extfs
<output truncated>
```

Docker is now using the `overlay2` storage driver. Docker has automatically created the `overlay` mount with the required `lowerdir`, `upperdir`, `merged`, and `workdir` constructs.

Continue reading for details about how OverlayFS works within your Docker containers, as well as performance advice and information about limitations of its compatibility with different backing filesystems.

How the `overlay2` driver works

If you are still using the `overlay` driver rather than `overlay2`, see [How the overlay driver works \(/engine/userguide/storagedriver/overlayfs-driver/#how-the-overlay-driver-works\)](#) instead.

OverlayFS layers two directories on a single Linux host and presents them as a single directory. These directories are called *layers* and the unification process is referred to as a *union mount*. OverlayFS refers to the lower directory as `lowerdir` and the upper directory as `upperdir`. The unified view is exposed through its own directory called `merged`.

While the `overlay` driver only works with a single lower OverlayFS layer and hence requires hard links for implementation of multi-layered images, the `overlay2` driver natively supports up to 128 lower OverlayFS layers. This capability provides better performance for layer-related Docker commands such as `docker build` and `docker commit`, and consumes fewer inodes on the backing filesystem.

Image and container layers on-disk

After downloading a five-layer image using `docker pull ubuntu`, you can see six directories under `/var/lib/docker/overlay2`.

Warning: Do not directly manipulate any files or directories within `/var/lib/docker/`. These files and directories are managed by Docker.

```
$ ls -l /var/lib/docker/overlay2

total 24
drwx----- 5 root root 4096 Jun 20 07:36 223c2864175491657d238e2664251df13b63adb8d050924fd1bfcdb278b866f7
drwx----- 3 root root 4096 Jun 20 07:36 3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b
drwx----- 5 root root 4096 Jun 20 07:36 4e9fa83caff3e8f4cc83693fa07a4a9fac9573deaf481506c102d484dd1e6a1
drwx----- 5 root root 4096 Jun 20 07:36 e8876a226237217ec61c4baf238a32992291d059fdac95ed6303bdf3f59cff5
drwx----- 5 root root 4096 Jun 20 07:36 eca1e4e1694283e001f200a667bb3cb40853cf2d1b12c29feda7422fed78afed
drwx----- 2 root root 4096 Jun 20 07:36 1
```


The new `l` (lowercase `l`) directory contains shortened layer identifiers as symbolic links. These identifiers are used to avoid hitting the page size limitation on arguments to the `mount` command.

```
$ ls -l /var/lib/docker/overlay2/l

total 20
lrwxrwxrwx 1 root root 72 Jun 20 07:36 6Y5IM2XC7TSNIJZZFLJCS6I4I4 -> ../3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b/
lrwxrwxrwx 1 root root 72 Jun 20 07:36 B3WWEFKBG3PLL737KZFIASSW7 -> ../4e9fa83caff3e8f4cc83693fa407a4a9fac9573deaf481506c102d4
lrwxrwxrwx 1 root root 72 Jun 20 07:36 JEYMODZYFCZFYSDABYXD5MF6YO -> ../eca1e4e1694283e001f200a667bb3cb40853cf2d1b12c29feda742d
lrwxrwxrwx 1 root root 72 Jun 20 07:36 NFYKDW6APBCCUCTOUSYDH4DXAT -> ../223c2864175491657d238e2664251df13b63adb8d050924fd1bfcd1
lrwxrwxrwx 1 root root 72 Jun 20 07:36 UL2MW33MSE3Q5VYIKBRN4ZAGQP -> ../e8876a226237217ec61c4baf238a32992291d059fdac95ed6303bd1
```

The lowest layer contains a file called `link`, which contains the name of the shortened identifier, and a directory called `diff` which contains the layer's contents.

```
$ ls /var/lib/docker/overlay2/3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b/

diff link

$ cat /var/lib/docker/overlay2/3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b/link

6Y5IM2XC7TSNIJZZFLJCS6I4I4

$ ls /var/lib/docker/overlay2/3a36935c9df35472229c57f4a27105a136f5e4dbef0f87905b2e506e494e348b/diff

bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
```

The second-lowest layer, and each higher layer, contain a file called `lower`, which denotes its parent, and a directory called `diff` which contains its contents. It also contains a `merged` directory, which contains the unified contents of its parent layer and itself, and a `work` directory which is used internally by OverlayFS.

```
$ ls /var/lib/docker/overlay2/223c2864175491657d238e2664251df13b63adb8d050924fd1bfcd1b278b866f7

diff link lower merged work

$ cat /var/lib/docker/overlay2/223c2864175491657d238e2664251df13b63adb8d050924fd1bfcd1b278b866f7/lower

1/6Y5IM2XC7TSNIJZZFLJCS6I4I4

$ ls /var/lib/docker/overlay2/223c2864175491657d238e2664251df13b63adb8d050924fd1bfcd1b278b866f7/diff/

etc sbin usr var
```

To view the mounts which exist when you use the `overlay` storage driver with Docker, use the `mount` command. The output below is truncated for readability.

```
$ mount | grep overlay

overlay on /var/lib/docker/overlay2/9186877cdf386d0a3b016149cf30c208f326dca307529e646afce5b3f83f5304/merged
type overlay (rw,relatime,
lowerdir=1/DJA75GUWHWG7EWICFYX54FIOVT:1/B3WWEFKBG3PLL737KZFIASSW7:1/JEYMODZYFCZFYSDABYXD5MF6YO:1/UL2MW33MSE3Q5VYIKBRN4ZAGQP:1,
upperdir=9186877cdf386d0a3b016149cf30c208f326dca307529e646afce5b3f83f5304/diff,
workdir=9186877cdf386d0a3b016149cf30c208f326dca307529e646afce5b3f83f5304/work)
```

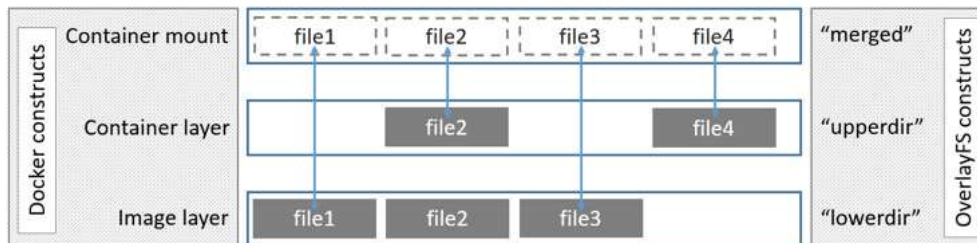
The `rw` on the second line shows that the `overlay` mount is read-write.

How the `overlay` driver works

This content applies to the `overlay` driver only. Docker recommends using the `overlay2` driver, which works differently. See [How the overlay2 driver works \(/engine/userguide/storagedriver/overlayfs-driver/#how-the-overlay2-driver-works\)](#) for `overlay2`.

OverlayFS layers two directories on a single Linux host and presents them as a single directory. These directories are called *layers* and the unification process is referred to as a *union mount*. OverlayFS refers to the lower directory as `lowerdir` and the upper directory as `upperdir`. The unified view is exposed through its own directory called `merged`.

The diagram below shows how a Docker image and a Docker container are layered. The image layer is the `lowerdir` and the container layer is the `upperdir`. The unified view is exposed through a directory called `merged` which is effectively the containers mount point. The diagram shows how Docker constructs map to OverlayFS constructs.



Where the image layer and the container layer contain the same files, the container layer “wins” and obscures the existence of the same files in the image layer.

The `overlay` driver only works with two layers. This means that multi-layered images cannot be implemented as multiple OverlayFS layers. Instead, each image layer is implemented as its own directory under `/var/lib/docker/overlay`. Hard links are then used as a space-efficient way to reference data shared with lower layers. As of Docker 1.10, image layer IDs no longer correspond to directory names in `/var/lib/docker/`.

To create a container, the `overlay` driver combines the directory representing the image’s top layer plus a new directory for the container. The image’s top layer is the `lowerdir` in the overlay and is read-only. The new directory for the container is the `upperdir` and is writable.

Image and container layers on-disk

The following `docker pull` command shows a Docker host downloading a Docker image comprising five layers.

```
$ docker pull ubuntu

Using default tag: latest
latest: Pulling from library/ubuntu

5ba4f30e5bea: Pull complete
9d7d19c9dc56: Pull complete
ac6ad7efd0f9: Pull complete
e7491a747824: Pull complete
a3ed95cae02: Pull complete
Digest: sha256:46fb5d01b88ad904c5c732b086b596b92cfb4a4840a3abd0e35dbb6870585e4
Status: Downloaded newer image for ubuntu:latest
```

THE IMAGE LAYERS

Each image layer has its own directory within `/var/lib/docker/overlay/`, which contains its contents, as shown below. The image layer IDs do not correspond to the directory IDs.

Warning: Do not directly manipulate any files or directories within `/var/lib/docker/`. These files and directories are managed by Docker.

```
$ ls -l /var/lib/docker/overlay/

total 20
drwx----- 3 root root 4096 Jun 20 16:11 38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72e3e0a15baa8
drwx----- 3 root root 4096 Jun 20 16:11 55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358
drwx----- 3 root root 4096 Jun 20 16:11 824c8a961a4f5e8fe4f4243dab57c5be798e7fd195f6d88ab06aea92ba931654
drwx----- 3 root root 4096 Jun 20 16:11 ad0fe55125ebf599da124da175174a4b8c1878afe6907bf7c78570341f308461
drwx----- 3 root root 4096 Jun 20 16:11 edab9b5e5bf73f2997524eebeac1de4cf9c8b904fa8ad3ec43b3504196aa3801
```

The image layer directories contain the files unique to that layer as well as hard links to the data that is shared with lower layers. This allows for efficient use of disk space.

```
$ ls -l /var/lib/docker/overlay/38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72e3e0a15baa8/root/bin/ls
19793696 /var/lib/docker/overlay/38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72e3e0a15baa8/root/bin/ls

$ ls -l /var/lib/docker/overlay/55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358/root/bin/ls
19793696 /var/lib/docker/overlay/55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358/root/bin/ls
```

THE CONTAINER LAYER

Containers also exist on-disk in the Docker host's filesystem under `/var/lib/docker/overlay/`. If you list a running container's subdirectory using the `ls -l` command, three directories and one file exist:

```
$ ls -l /var/lib/docker/overlay/<directory-of-running-container>

total 16
-rw-r--r-- 1 root root 64 Jun 20 16:39 lower-id
drwxr-xr-x 1 root root 4096 Jun 20 16:39 merged
drwxr-xr-x 4 root root 4096 Jun 20 16:39 upper
drwx----- 3 root root 4096 Jun 20 16:39 work
```

The `lower-id` file contains the ID of the top layer of the image the container is based on, which is the OverlayFS `lowerdir`.

```
$ cat /var/lib/docker/overlay/ec444863a55a9f1ca2df72223d459c5d940a721b2288ff86a3f27be28b53be6c/lower-id
55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a5358
```

The `upper` directory contains the contents of the container's read-write layer, which corresponds to the OverlayFS `upperdir`.

The `merged` directory is the union mount of the `lowerdir` and `upperdir`, which comprises the view of the filesystem from within the running container.

The `work` directory is internal to OverlayFS.

To view the mounts which exist when you use the `overlay` storage driver with Docker, use the `mount` command. The output below is truncated for readability.

```
$ mount | grep overlay

overlay on /var/lib/docker/overlay/ec444863a55a.../merged
type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay/55f1e14c361b.../root,
upperdir=/var/lib/docker/overlay/ec444863a55a.../upper,
workdir=/var/lib/docker/overlay/ec444863a55a.../work)
```

The `rw` on the second line shows that the `overlay` mount is read-write.

How container reads and writes work with `overlay` or `overlay2`

Reading files

Consider three scenarios where a container opens a file for read access with overlay.

- **The file does not exist in the container layer:** If a container opens a file for read access and the file does not already exist in the container (`upperdir`) it is read from the image (`lowerdir`). This incurs very little performance overhead.
- **The file only exists in the container layer:** If a container opens a file for read access and the file exists in the container (`upperdir`) and not in the image (`lowerdir`), it is read directly from the container.
- **The file exists in both the container layer and the image layer:** If a container opens a file for read access and the file exists in the image layer and the container layer, the file's version in the container layer is read. Files in the container layer (`upperdir`) obscure files with the same name in the image layer (`lowerdir`).

Modifying files or directories

Consider some scenarios where files in a container are modified.

- **Writing to a file for the first time:** The first time a container writes to an existing file, that file does not exist in the container (`upperdir`). The `overlay` / `overlay2` driver performs a `copy_up` operation to copy the file from the image (`lowerdir`) to the container (`upperdir`). The container then writes the changes to the new copy of the file in the container layer.

However, OverlayFS works at the file level rather than the block level. This means that all OverlayFS `copy_up` operations copy the entire file, even if the file is very large and only a small part of it is being modified. This can have a noticeable impact on container write performance. However, two things are worth noting:

- The `copy_up` operation only occurs the first time a given file is written to. Subsequent writes to the same file operate against the copy of the file already copied up to the container.
 - OverlayFS only works with two layers. This means that performance should be better than AUFS, which can suffer noticeable latencies when searching for files in images with many layers. This advantage applies to both `overlay` and `overlay2` drivers. `overlayfs2` will be slightly less performant than `overlayfs` on initial read, because it has to look through more layers, but it caches the results so this is only a small penalty.
- **Deleting files and directories:**
 - When a *file* is deleted within a container, a *whiteout* file is created in the container (`upperdir`). The version of the file in the image layer (`lowerdir`) is not deleted (because the `lowerdir` is read-only). However, the whiteout file prevents it from being available to the container.
 - When a *directory* is deleted within a container, an *opaque directory* is created within the container (`upperdir`). This works in the same way as a whiteout file and effectively prevents the directory from being accessed, even though it still exists in the image (`lowerdir`).
 - **Renaming directories:** Calling `rename(2)` for a directory is allowed only when both the source and the destination path are on the top layer. Otherwise, it returns `EXDEV` error (“cross-device link not permitted”). Your application needs to be designed to handle `EXDEV` and fall back to a “copy and unlink” strategy.

OverlayFS and Docker Performance

Both `overlay2` and `overlay` drivers are more performant than `aufs` and `devicemapper` . In certain circumstances, `overlay2` may perform better than `btrfs` as well. However, be aware of the following details.

- **Page Caching.** OverlayFS supports page cache sharing. Multiple containers accessing the same file share a single page cache entry for that file. This makes the `overlay` and `overlay2` drivers efficient with memory and a good option for high-density use cases such as PaaS.
- **copy_up.** As with AUFS, OverlayFS has to perform copy-up operations whenever a container writes to a file for the first time. This can add latency into the write operation, especially for large files. However, once the file has been copied up, all subsequent writes to that file occur in the upper layer, without the need for further copy-up operations.

The OverlayFS `copy_up` operation is faster than the same operation with AUFS, because AUFS supports more layers than OverlayFS and it is possible to incur far larger latencies if searching through many AUFS layers. `overlay2` supports multiple layers as well, but mitigates any performance hit with caching.

- **Inode limits.** Use of the `overlay` storage driver can cause excessive inode consumption. This is especially true in the presence of a large number of images and containers on the Docker host. The only way to increase the number of inodes available to a filesystem is to reformat it. To avoid running into this issue, it is highly recommended that you use `overlay2` if at all possible.

Performance best practices

The following generic performance best practices also apply to OverlayFS.

- **Use fast storage:** Solid-state drives (SSDs) provide faster reads and writes than spinning disks.
- **Use volumes for write-heavy workloads:** Volumes provide the best and most predictable performance for write-heavy workloads. This is because they bypass the storage driver and do not incur any of the potential overheads introduced by thin provisioning and copy-on-write. Volumes have other benefits, such as allowing you to share data among containers and persisting even when no running container is using them.

Limitations on OverlayFS compatibility

To summarize the OverlayFS's aspect which is incompatible with other filesystems:

- **open(2):** OverlayFS only implements a subset of the POSIX standards. This can result in certain OverlayFS operations breaking POSIX standards. One such operation is the *copy-up* operation. Suppose that your application calls `fd1=open("foo", O_RDONLY)` and then `fd2=open("foo", O_RDWR)`. In this case, your application expects `fd1` and `fd2` to refer to the same file. However, due to a copy-up operation that occurs after the first calling to `open(2)`, the descriptors refer to different files.

`yum` is known to be affected unless the `yum-plugin-ovl` package is installed. If the `yum-plugin-ovl` package is not available in your distribution (e.g. RHEL/CentOS prior to 6.8 or 7.2), you may need to run `touch /var/lib/rpm/*` before running `yum install`.

- **rename(2):** OverlayFS does not fully support the `rename(2)` system call. Your application needs to detect its failure and fall back to a "copy and unlink" strategy.

🔗 [container](https://docs.docker.com/glossary/?term=container) (<https://docs.docker.com/glossary/?term=container>), [storage](https://docs.docker.com/glossary/?term=storage) (<https://docs.docker.com/glossary/?term=storage>), [driver](https://docs.docker.com/glossary/?term=driver) (<https://docs.docker.com/glossary/?term=driver>), [OverlayFS](https://docs.docker.com/glossary/?term=OverlayFS) (<https://docs.docker.com/glossary/?term=OverlayFS>)

Rate this page:

17 1

What is Docker (<https://www.docker.com/what-docker>)

What is a Container (<https://www.docker.com/what-container>)

Use Cases (<https://www.docker.com/use-cases>)

Customers (<https://www.docker.com/customers>)

Partners (<https://www.docker.com/partners/partner-program>)

For Government (<https://www.docker.com/industry-government>)

About Docker (<https://www.docker.com/company>)

Management (<https://www.docker.com/company/management>)

Press & News (<https://www.docker.com/company/news-and-press>)

Careers (<https://www.docker.com/careers>)

Product (<https://www.docker.com/products/overview>)

Pricing (<https://www.docker.com/pricing>)

Community Edition (<https://www.docker.com/docker-community>)

Enterprise Edition (<https://www.docker.com/enterprise>)

Docker Datacenter (<https://www.docker.com/products/docker-datacenter>)

Docker Cloud (<https://cloud.docker.com/>)

Docker Store (<https://store.docker.com/>)

Docker for Mac (<https://www.docker.com/docker-mac>)

Docker for Windows (PC) (<https://www.docker.com/docker-windows>)