インプレスR&D［Next Publishing］
Cloud シリーズ
E-Book / Print Book

Next Publishing プレビュー

impress R&D
An Impress Group Company

目次

はじめに

2013年にDockerがリリースされ、Linuxにおけるコンテナが脚光を浴びるようになりました。そのLinuxコンテナは一時的な流行とはならず、最近では話題はコンテナそのものよりもコンテナをどう運用していくかというステージに入っています。それに伴い、話題の中心はオーケストレーションに移っています。

つまり、コンテナがプロダクション環境で使われるステージに入り、さまざまなシステムがコンテナ上で稼働するようになってきたわけです。活用が進むにつれ、色々な要件がコンテナに求められるようになるでしょう。その要件を満たす機能がLinuxコンテナにあるのか、それ以前に開発するシステムはコンテナ上で動かすのが適当なのかどうかを判断するには、Linuxコンテナの基本的な機能を理解することが必要になるでしょう。

Linuxにおけるコンテナは、Linuxカーネルに実装されています。しかし、「コンテナ」というひとつの機能として実装されているわけではなく、多数の機能を組み合わせてコンテナが作られています。これらの機能はコンテナ専用の機能というわけではなく、単一の機能として使える機能がたくさんあります。

本書では、このLinuxカーネルに多数実装されている機能のうち、コンテナ利用環境でよく使われるファイルシステムについて紹介します。

技術の泉シリーズからは、本シリーズの第1巻としてNamespaceとネットワーク機能を紹介した『Linux Container Book』を、第2巻としてcgroup v1を紹介した『Linux Container Book 2』を、第3巻としてコンテナに関係するセキュリティー機能を紹介した『Linux Container Book 3』を出版しました。本書でも第1章でコンテナとは何か？について説明しますが、この部分はこの第1〜3巻と同じ内容です。初めて本書から読み始めた方のためにコンテナの基本を説明しますが、第1〜3巻のいずれかをすでにご覧いただいている方は、飛ばしてもらっても問題ありません。

また、ファイルシステムという性格上、第1巻で紹介したマウント関係の機能が密接に関係します。そこで、Mount Namespaceなど、マウントまわりの機能については、第1巻と同じ説明を入れました。この部分も、第1巻をすでにご覧いただいている方は、飛ばしても問題ありません。

本書では、Linuxカーネルに実装されているファイルシステムについて説明しますが、Linuxカーネルに実装されているファイルシステムの多くは、コンテナに特化した機能を持っているわけではありません。そこで、本書では、特にコンテナで利用されることが多いOverlayFSの機能を中心に説明しました。それ以外にも、現在コンテナに求められるファイルシステムの要件とともに、コンテナを起動する環境で利用すると便利なBtrfsとLVMについて、簡単に紹介しています[1]。

紹介する機能については、実行例を入れるなど、なるべく丁寧に説明するようにしました。また、機能を紹介する際には、特定のコンテナランタイムを使わず、シェル上で確認しながら機能を確認していけるようにしています。

なお、この本の実行例は、一部をのぞいてUbuntu 22.04上で動作確認をしています。

1.LVMはファイルシステムではありませんが…

凡例

コマンド名や実行例などで出てきた具体的なパスは、本文中でも等幅フォントを使用します。

本文中で、説明のためにカーネルのバージョンを表記することがあります。文中で使うカーネルのバージョンは、パッチなどが当たっていないカーネル（バニラカーネル）のバージョンです。ディストリビューションのカーネルは、機能がバックポートされていることがあるため、あるバージョン以前は使えなかったという説明をしていても、機能が使える場合があります。

第1章　コンテナとは

コンテナに関するさまざまな機能を説明する前に、まずは「コンテナ」とは何か？を説明しましょう。

一言でいうとコンテナとは、**隔離した空間でプロセスを実行する**ことです。

短く一言で説明しましたが、ここに重要なキーワードがふたつ含まれています。

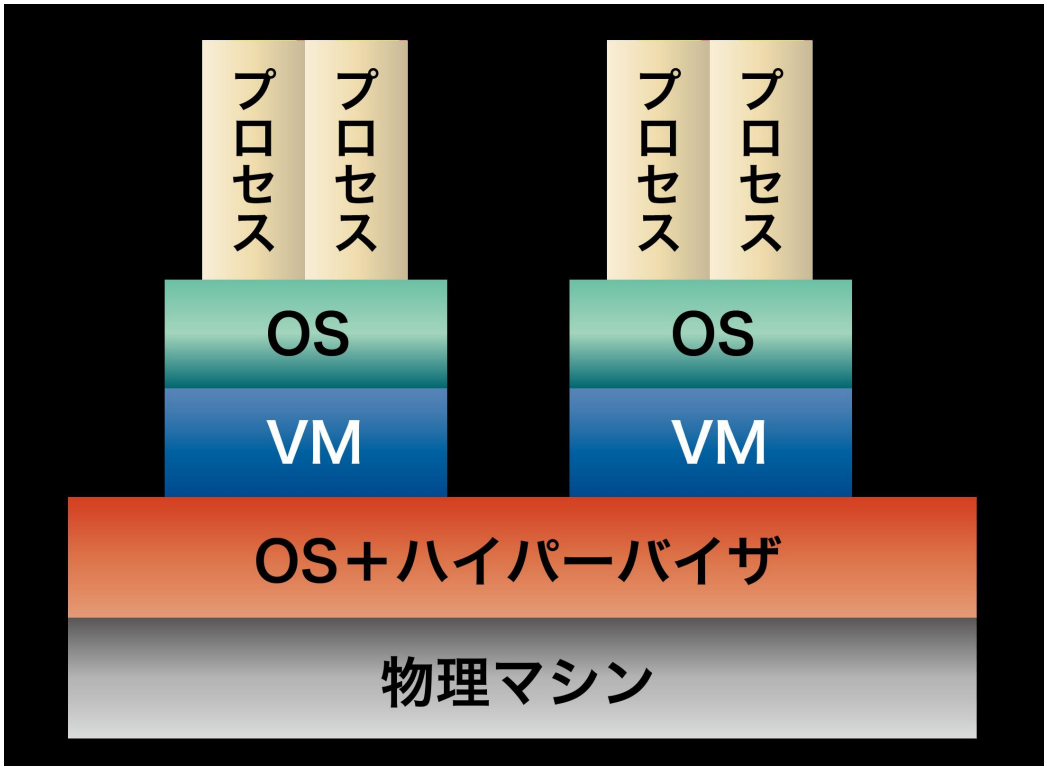
コンテナを説明する際、「仮想化」や「仮想環境」といったような「仮想」という単語を使うことがあります。たしかに、コンテナを起動すると、ある種の「仮想環境」が作られます。しかし筆者は、「仮想」よりは「隔離」という言葉を使ったほうがよりコンテナを的確に表していると考えます。あとでもう少し詳しく説明しますので、まずはこの**隔離**が重要であることを覚えておいてください。

そして、もうひとつが「プロセス」です。**コンテナはプロセスである**ということが、もうひとつ忘れてはいけない重要な点です。

もう少し詳細に説明していきましょう。

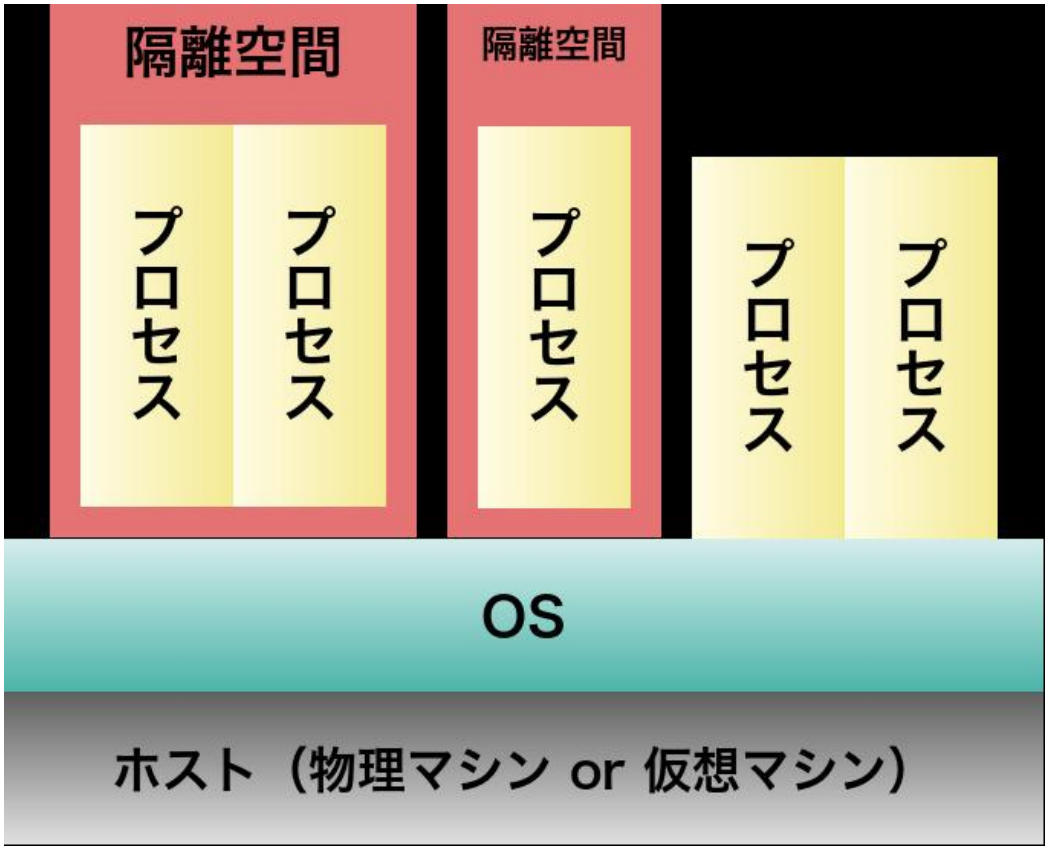
「仮想環境」を作り出すための仕組みとして、「仮想マシン」（VM）を使ったことがある方は多いのではないでしょうか。ここでは説明のために、VMと比較をしてみます。

図1.1: 仮想マシン



VMでは図1.1のように、コンピューター上で動くOSやVMを実現するためのハイパーバイザ上で、実際のハードウェアをエミュレートするVMが動きます。実際の物理的なコンピューターと同じような動作がソフトウェアによって実現されているので、このVMを使うにはOSが必要になります。つまり、図1.1のようにOSが物理的なコンピューター上で動き、その物理的なコンピューター上で動くOS上でさらにOSが動きます。また、このVM自体は物理マシン上で実行されているOS上のプロセスですが、物理マシン上のOSからはVMのプロセスが見えているだけで、VM内で動作しているプロセスは見えません。まさに「仮想環境」と呼べる仕組みです。

図1.2: コンテナ



一方コンテナは、図1.2のように、起動するすべてのプロセスはコンピューター上にインストールされたOS（ホストOS）上で直接起動します。このOSは図1.2のように物理マシン上で実行されていても、仮想マシン上で実行されていても構いません。

通常のプロセスの動作と異なる点は、そのプロセスの一部をグループ化し、他のグループやグループに属していないプロセスから隔離した空間で動作させる点です。貨物輸送で使うコンテナのように、隔離された空間にプロセスが入っているので、この空間を『コンテナ』と呼ぶわけです。貨物輸送のコンテナのように、あるコンテナの内部から他のコンテナの内部を見ることはできません。

図1.2で「普通に起動したプロセス」と書いたプロセスが、コンテナ化しないで起動したプロセス

です。隔離されていない空間で動いているように見えますが、実際は**デフォルトの空間**で動いています。つまりデフォルトではすべてのプロセスが同じ空間で動いているので、隔離されていないように見えるだけです。

コンテナ内で実行するプロセスは、図1.2のようにひとつでも構いませんし、複数のプロセスを実行しても構いません。

この隔離された空間を作り出すのは、OSのカーネルに実装された機能です。OSを通して使用できるコンピューターが持つリソースを各コンテナごとに隔離して、ホストOS上で直接動作するプロセスや他のコンテナから独立した空間を作り出し、リソースを分割、分配、制限するわけです。

実際にコンテナ化したプロセスを実行する際には、隔離された空間で動作させるだけではなく、さまざまなセキュリティー設定を行ったり、通常起動するプロセスとは違う属性を持たせたりします。つまり、コンテナとはOS上で起動するプロセスに、ある種の属性を持たせたものと考えることもできます。

ここでいう属性とはたとえば、

・他のプロセスから見えなくする（隔離）
・実行できる操作に制限を加える or 特権を与える
・使えるリソースに制限を加える

といった属性です。普通に起動したプロセスとは、ちょっと違うプロセスといったところです。

Linuxカーネルには、この隔離した環境を作り出したり、さまざまな属性を付与するといった機能がたくさん備わっています。

Linuxでコンテナを実行する際は、目的に応じてLinuxカーネルに備わる色々な機能を組み合わせます。決して単一の「コンテナ」という機能が備わっているわけではありません。すべての機能がコンテナ向けの機能というわけではなく、他の用途で使う機能をコンテナでも使用したりします。すなわち、**使いたい機能のみを使ってコンテナを作成できる**ということです。

世の中に存在するDockerやLXCなどのコンテナを実行するための実装は、その実装が目指す動きをするように、あらかじめ必要そうな色々な機能を組み合わせた状態でコンテナを作成します。つまり、Linuxカーネルに備わっているさまざまな機能から、必要な機能を選択し、それらを組み合わせてコンテナを作るということです。

1.1 コンテナのメリット

ここまで説明したことだけでも、コンテナのメリットが見えてきます。

起動が早い

先に説明したとおり、コンテナは**プロセス**です。つまり、起動の速度はプロセスとほぼ等しいということになります。

仮想マシンの場合は、仮想的にエミュレートしたハードウェアの起動からOSの起動を経ないといけませんので、使える状態になるまでには少し時間がかかります。

それに対して、プロセスが起動すると使える状態になるコンテナは、起動の速度が求められる場

面では適したソリューションになるはずです。

オーバーヘッドがない

これは仮想マシンと比べたメリットになります。仮想マシンの場合はハードウェアを仮想化しなければいけませんので、そのためにリソースを割かなければなりません。

さらに、仮想的に実現したハードウェア上でOSが稼働します。ベースとなるホスト環境でもOSが動作していますので、同じように動作するOSがいくつも起動していることになります。なので当然オーバーヘッドとなります。

それと比べると、コンテナはプロセスが起動するだけです。通常のプロセスに比べると、コンテナ化するために内部的には余分な処理は必要です。そうはいってもプロセスが起動するだけですので、通常の環境で起動させるプロセスと比べても大きな差がない程度のオーバーヘッドで起動するはずです。

高密度化が可能

コンテナを起動させない通常のOS上でも、OS上では多数のプロセスが起動しています。コンテナはプロセスですから、コンテナ化したとしてもコンテナ化しない場合と同じ程度の数のプロセス、つまりコンテナを起動させることができるはずです。

アプリケーションのみをコンテナ化できる

仮想マシンの場合、仮想的に実現したハードウェア上で必ずOSを動作させる必要があります。

それに対してコンテナはプロセスですので、要件に必要なアプリケーションのみをコンテナ内で動作させることができます。つまり、必要なプロセスのみを仮想的な環境であるコンテナ内で動作させることができます。

特に、システムとして必要なプロセスが起動しなければならない仮想マシンと比べると、大きなメリットとなります。

固定的なリソース割当が不要

これも仮想マシンと比べたメリットになります。仮想マシンの場合、必ず「仮想CPUをいくつ、メモリーをいくつ」という風に固定的な値を割り当てる必要があります。仮想マシン内のプロセスが、特にリソースを必要とするタスクを実行していなくても、割り当てたメモリーなどのリソースは占有したままです[1]。

通常、プロセスには固定的にリソースを割り当てる必要はありませんので、当然コンテナにも必ず割り当てなければならないリソースはありません。ですので、使われていないリソースは最大限利用できます。

---

1. もちろん仮想マシンにも仮想マシン間でメモリーを共有したりといったリソース消費を抑える機能はあります。

1.2 コンテナのデメリット

前述のようにコンテナとはプロセスですので、コンテナが実現できることは通常のプロセスと同じ範囲に留まります。そういう意味では、コンテナとして実行するタスクは、その範囲に留まります。

その範囲内でコンテナを実行することに関して、特にデメリットとして感じることは筆者としてはないのですが、その仕組み上どうしてもできないことはありますので、それをデメリットとすると、コンテナのデメリットとしていえることはいくつかあります。

異なるアーキテクチャ、OS上のアプリケーションは実行できない

コンテナはLinuxカーネル上で動くプロセスですので、当然異なるOS用のバイナリを動かすことはできません[2]。仮想マシンであれば、Linux上でWindowsを動かせますが、当然Linuxコンテナとして Windows用のプログラムを動かすことはできません。また、異なるアーキテクチャ向けのプログラムを動かすこともできません。

コンテナは単なるLinux上で動くプログラムですので、当然のことです。

カーネルに関わる操作をコンテナごとに行えない

コンテナは単一のカーネル上で動くプロセスに特別な属性を与えたプロセスです。つまり、複数のコンテナがひとつのOS環境で動いているとすると、その複数のコンテナは共通のLinuxカーネル上で動作していることになります。

ということは、コンテナごとにカーネルの動きを変えるような操作はできないということです。

たとえば、Linuxではモジュールという仕組みでドライバや色々な機能を動的に組み込むことができますが、コンテナごとに異なるモジュールを読み込むようなことができません。カーネルは単一ですので、当然のことです。

Linuxカーネルに機能として実装されている、コンテナごとに異なった属性を与えられるようになっている機能以外は、コンテナ間ではカーネルが提供する機能は共通となります。

これはコンテナとはどのようなものであるかを理解すれば当たり前の話であって、特にデメリットではないと筆者は考えます。コンテナとはどのようなものであるかをきちんと理解した上で、コンテナの機能によくフィットするユースケースでコンテナを使えば、特にデメリットといえるようなものではないでしょう。

これ以降ではここまでで述べたような、Linuxカーネルに備わるコンテナに関係するさまざまな機能を紹介していきます。

1.3 Linuxにおけるコンテナ

先に書いたように、コンテナはカーネルが持っている機能を使いますので、Linuxでコンテナを実行するにはLinuxカーネルに備わっている機能を使います。

---

2. もちろんコンテナのプロセスとして仮想マシンやエミュレーターを動かせば動きます。

ところが、Linuxカーネルには「コンテナ」という機能があるわけではありません。Linuxカーネルには多数の機能があり、その中にコンテナ向けの機能やコンテナで使える機能をたくさん持っています。

そのたくさんの機能の中から、使いたい機能を使ってコンテナを実現します。したがって、要件に合った機能だけを選択して自分用のコンテナを実装できます。ただ、要件ごとにコンテナ実行用のプログラムを開発することは容易ではありませんし、コストもかかります。そのため、一般的に使われる機能を色々組み合わせてコンテナが実行できるように作られているのが、DockerやLXC/LXDといったコンテナを起動するためのプログラムやライブラリーです。

そして、Linuxカーネルが持っている多数の機能には、コンテナ専用といえる機能もありますが、ほとんどの機能はコンテナ専用というわけではなく、その機能単体でも使えます。

つまりLinuxでは、カーネルに実装されている**多数の機能を組み合わせて**コンテナを作ります。

このLinuxカーネルが持っている多数の機能の中で、Linuxでコンテナを作る場合の主要機能といえる機能がふたつありますので、まず簡単に紹介していきましょう。

Namespace（名前空間）

Namespaceは**隔離空間**を作成する機能です。つまり、このNamespace機能こそが「コンテナ」そのものといっても過言ではないでしょう。

OS起動後にカーネルが作成する、色々なOSリソースに対してNamespace機能が実装されており、そのNamespaceを使うことで、そのNamespaceが対象とするリソースを隔離した空間が作られます。カーネルに実装されているリソースごとに存在するNamespace機能は、単独で使うことも、組み合わせて使うこともできます。

つまりコンテナごとに、Linuxカーネルで実装されているNamespace機能が対象としているリソースを隔離した空間が持てるということです。どのようなリソースに対してNamespace機能が実装されているのかについては、本シリーズの第1巻で詳しく紹介しています。

cgroup

プロセスやプロセスのグループを、隔離された空間に入れてコンテナを作る機能が、先に紹介したNamespaceです。このNamespace内のプロセスに、まとめて物理的なリソース制限をかけたいというケースは多いでしょう。

仮想マシンの場合は、仮想マシンを作成・起動する際に仮想マシンにCPUやメモリーを与えますが、それと同様のことをコンテナに対しても設定したいようなケースです。

このような機能を提供するのがcgroupです。コンテナが使えるCPUやメモリー、ネットワークやディスクI/O帯域を制限できます。

cgroupはコンテナ、Namespaceとは関係なく使えます。つまりコンテナと関係なく、普通にOS上で起動したWebサーバーやブラウザーなどが使えるCPUやメモリーも制限できます。コンテナ内の一部のプロセスにだけ、独立して制限をかけることもできます。

cgroupには、2.6.24カーネル以来のcgroup v1と、v1の問題点を改良した4.5カーネルでstableと

なったcgroup v2があります。cgroup v1については、本シリーズの第2巻で詳しく紹介しています。

ここまでで、コンテナの概要について説明しました。このあとは、いよいよ、コンテナのファイルシステムのお話をしていきます。

第2章　コンテナのファイルシステム

Linuxにおいては、ほとんどのリソースが**ファイル**で表されます。そのファイルを開く、ファイルを書き込む、ファイルの情報を取得するといったような、ファイルを操作するための共通のインターフェースを提供するのがファイルシステムです。

この章では、ファイルシステムをコンテナから使う場合、コンテナに対してコンテナ専用のファイルシステムを提供するために、ファイルシステムに求められる機能を考えていきましょう。

コンテナのファイルシステムに求められる条件は何でしょう?

一般的なコンテナランタイムを使ってコンテナを起動した際の見え方から、コンテナのファイルシステムを考えてみましょう。

2.1　コンテナ用の独立したファイルシステム

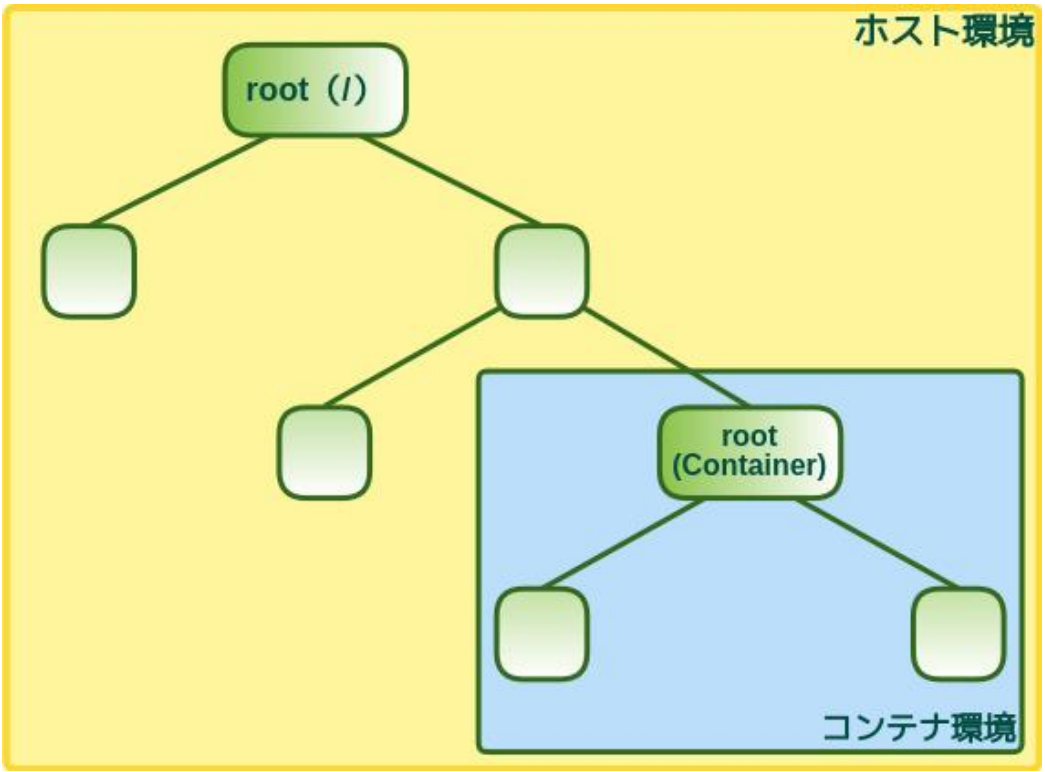
コンテナを起動する場合、ホストとは別に、そのコンテナ専用ファイルシステムを持ちたいことがほとんどではないでしょうか[1]。

コンテナ専用のファイルシステムを持ちたい理由は、ホスト環境にインストールされたパッケージとは別のバージョンのパッケージを使ったり、ホスト環境とは別のディストリビューションを使ったり、ホスト環境のパッケージ更新に左右されない固定的なバージョンのパッケージやライブラリーを使うといった理由でしょう。

コンテナで、ホスト環境と別のファイルシステムを使うためには、一般的には図2.1のように、ホスト上の任意のディレクトリーをコンテナのルートディレクトリーとし、そのディレクトリー以下に、コンテナ用にルート（/）以下のツリーを展開します。

1. もちろん別のファイルシステムは不要で、たとえばファイルシステムはホストのファイルシステムを使い、ネットワークだけは別の独立した環境がほしいという場合もあるでしょう。

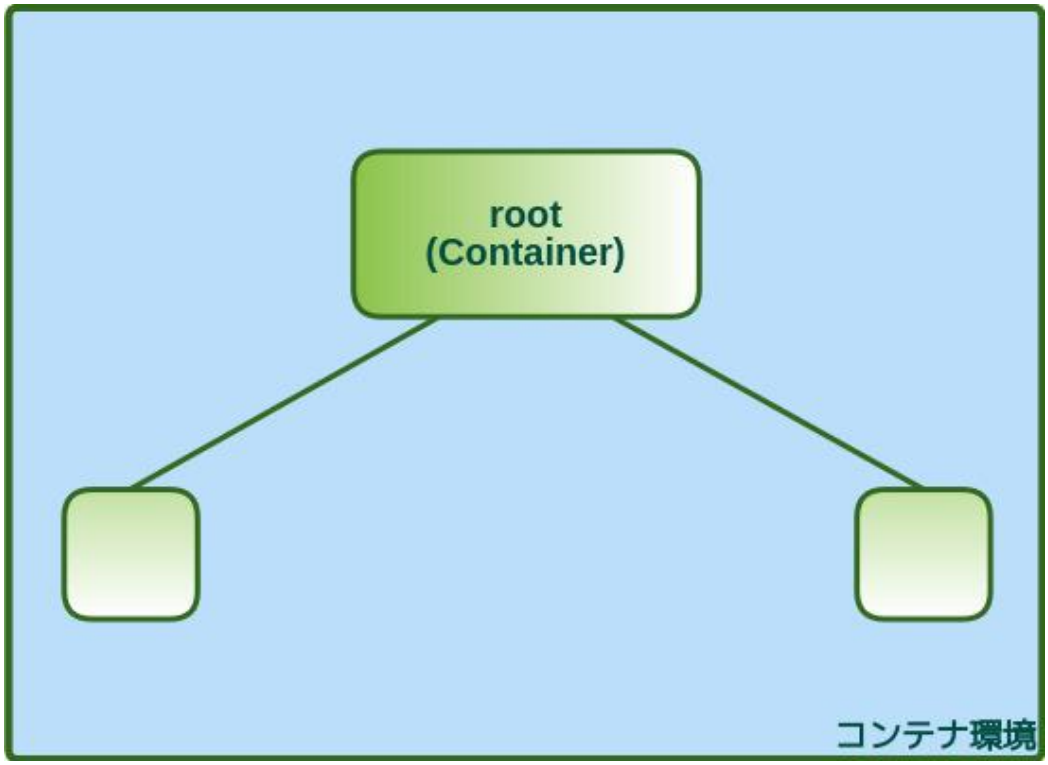
図2.1: ホスト上に構築したコンテナイメージ



そして、コンテナからは図2.1の「コンテナ環境」の部分だけが見えれば、ファイルシステム的には、ホストや他のコンテナから独立した隔離空間、つまりコンテナに見えます。

ホストからは図2.1のように見え、コンテナ内からは、図2.1の「コンテナ環境」部分だけが、図2.2のように見えるということです。

図2.2: コンテナ内で見えるファイルシステム



このように、ホストと別のファイルシステムだけが見えると、いかにもコンテナ内の仮想環境にいるような気分になりますね。

ここまでで説明したように、コンテナを起動したとき、コンテナ内から見たファイルシステムは、他のコンテナとは独立したツリーが見えることがほとんどでしょう。

2.2 pivot_root

図2.1のように、コンテナホスト上にあるディレクトリーツリーの一部を、図2.2のように、コンテナ専用の独立したファイルシステムであるかのように見せるために使われている機能がpivot_rootです。

似たような機能に、chrootという機能があります。この機能はプロセスから見えるルート（/）を、指定するディレクトリーに変更する機能です。chrootは、chrootした環境内でさらにchrootすると、chrootしたルートから抜け出せるため、コンテナランタイムでは一般的には使われません[2]。

このような動きをするchrootに対して、pivot_rootは実行する前後でルートファイルシステム自体が変更されてしまいますので、そもそもchrootのように「抜ける」という概念がありません。

pivot_rootはchrootよりは新しい機能ですが、比較的古くからLinuxカーネルに実装されていた

2.chrootについての詳しい話は本シリーズ第1巻をご覧ください。

機能で、2000年の2.3.41で実装されています。

図2.3: pivot_root前のファイルシステム

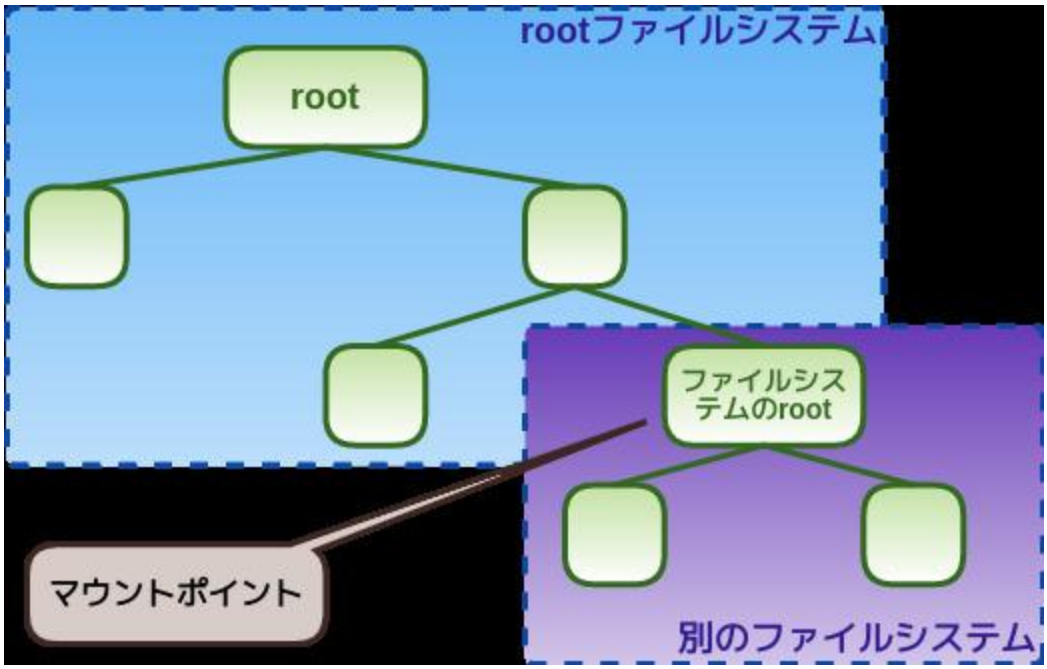
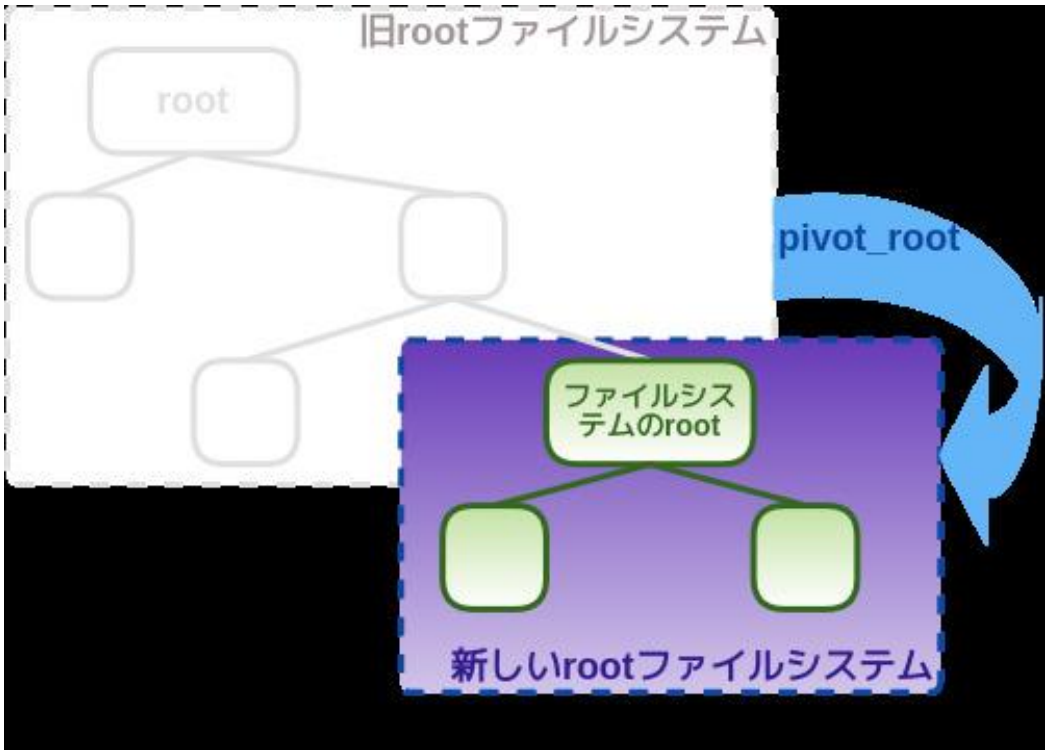


図2.3のように、**「rootファイルシステム」**と、コンテナ用に準備した**「別のファイルシステム」**があるとします。

図2.3の状態でpivot_rootを実行して、ルートファイルシステムを「別のファイルシステム」に切り替えます。すると図2.4のようになり、コンテナ内では**「新しいrootファイルシステム」**しか見えなくなります。図2.4のように、元のファイルシステムはpivot_root後に見えなくなっていますので、chrootのように元のファイルシステムへは抜け出せません。

図2.4: pivot_root後のファイルシステム



ただし、図2.3、図2.4のように、pivot_rootは**ファイルシステム**を切り替えますので、コンテナホスト上の任意のディレクトリーへpivot_rootできません。pivot_rootで取り替える先として指定するのは、**ファイルシステム**である必要があります。

取り替える先はファイルシステムである必要があると書きました。しかし実際にpivot_rootできる条件は、pivot_rootでルートとして指定するディレクトリーが、マウントポイントであることです[3]。マウントポイントに対して、マウント操作を行う対象はファイルシステムですので、pivot_rootで取り替える先はファイルシステムとなるわけです。

ただ、コンテナを起動する際に、常にコンテナ専用にファイルシステムを準備するわけではありません。同一ファイルシステム内の特定のディレクトリー以下をコンテナのファイルシステムにする場合もあります。

そういう場合には、**バインドマウント**という操作を行うことで、ディレクトリーをマウントポイントに見せてpivot_rootを行えます。

pivot_rootとバインドマウントについては、本シリーズの第1巻である『Linux Container Book』で詳しく説明していますので、詳しくはそちらをご参照ください。

3.「マウントポイント」については次のセクションで説明します。

2.3 Mount Namespace

pivot_rootによって、コンテナ専用の独立したファイルシステムを、コンテナ内から使えるようになりました。

このpivot_rootと並んで、コンテナ専用のファイルシステムを提供するのに重要な機能がMount Namespaceです。

Linuxが起動すると、色々なファイルシステムをシステム上で使用するために特定のディレクトリーに結びつける処理を行います。これを**マウント**といいます。このマウントが行われるディレクトリーを**マウントポイント**といいます。

たとえば、ディスク上のパーティションにファイルシステムを作成し、ルート（/）としてマウントした場合は、この/がマウントポイントです。その下に存在するディレクトリーである/homeというディレクトリーに、別のファイルシステムをマウントした場合、/homeもマウントポイントとなります。

システムで利用するさまざまなファイルシステムが、特定のディレクトリーにマウントされます。

Mount Namespaceを使うと、Namespace内で行ったマウント操作を、他のNamespaceには反映させないようにできます。このようにすると、コンテナ内で行うマウントが、コンテナ専用の独立したマウントになりますので、ファイルシステムとしてもコンテナ専用になったといえます。

Mount Namespaceは、厳密にいうとNamespace内のプロセスから見えるマウントポイントの一覧を分離します。つまり、Namespaceごとに異なるマウントポイントの一覧を持てるということです。

この機能を使うと、コンテナ内でマウント操作を行った場合でも、そのマウントはホストOSや他のコンテナからは見えないようにできます。通常はコンテナごとに独立したファイルシステムを使いたいことがほとんどでしょうから、あるコンテナ内で行ったマウント処理が他のコンテナで見えないというのは重要な機能です[4]。

システムが起動する際には、デフォルトでMount Namespaceがひとつ作られます。システムが起動したあとに、起動時に作られたデフォルトのNamespaceとは別のNamespaceをシステム上に作成できます。コンテナ起動時は、通常はデフォルトとは別のNamespaceを作ります。

新たにMount Namespaceを作った場合、新しく作られたNamespaceは、元のNamespaceのコピーとなります。新しく作ったNamespace内で、マウント操作やマウントを解除するアンマウント操作を行い、コンテナ専用のファイルシステムを作成します。

実際に起動したコンテナで、Mount Namespaceを使ってマウントが独立している例を見てみましょう。

まずは、コンテナホスト上のルート（/）にマウントされるルートファイルシステムの情報を`mount -l`コマンドで見てみましょう。

```
$ mount -l | grep ' / '
/dev/sda1 on / type ext4 (rw,relatime) [rootfs]
```

4.Mount Namespace 内のマウントを、他の Namespace に反映させるという設定もできます。

コンテナホストは、/dev/sda1にあるファイルシステムをルート（/）にマウントしており、typeのあとを見ると、このファイルシステムはext4であることがわかります。

それではコンテナを起動して、コンテナ内でファイルシステムを見てみましょう。次のようにDockerコンテナを起動します。

```
$ docker run -ti ubuntu:22.04 /bin/bash （Ubuntu 22.04コンテナを起動）
```

コンテナ内でmount -lコマンドを使い、コンテナ内のルート（/）にマウントされるルートファイルシステムを見てみます。

```
root@a66e3b2c7c6e:/# mount -l | grep ' / '
overlay on / type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/4ALMMR
5ESBCKH23E4OJNZDJCGL:/var/lib/docker/overlay2/l/MFDN45GO4IPZCJUWXTYYUB7SVL,upperd
ir=/var/lib/docker/overlay2/18b99aee62bb1ccfc689311de9e88228eee6ac597f62ff8e60d41
cd2157c170d/diff,workdir=/var/lib/docker/overlay2/18b99aee62bb1ccfc689311de9e8822
8eee6ac597f62ff8e60d41cd2157c170d/work,index=off)
```

この例では、overlayという名前を持つファイルシステムをルート（/）にマウントしていることと、typeがoverlayとなっており、ファイルシステムがOverlayFSであることがわかります。OverlayFSに関しては、あとの第4章で説明します。

ここまでで、ホストとコンテナのファイルシステムが独立していることを紹介しました。

説明だけではわかりづらいかもしれませんので、Mount Namespaceを使うと、コンテナ内で行ったマウントがホストに反映されないことを見ておきましょう。

まずは、Namespaceを作成するためのコマンドであるunshareコマンドでMount Namespaceを作り、シェルを実行します。新しく作ったMount Namespace上で動くシェル上で、バインドマウントを実行します。

Mount Namespaceを作成するには、unshareコマンドに-mまたは--mountというオプションを与えます。

バインドマウントはmountコマンドに--bindというオプションを与えて実行します。次の例では、/etc/hostsファイルを、/root/hostsファイルにバインドマウントしています。

```
$ sudo unshare --mount /bin/bash （Mount Namespaceを作成しbashを実行）
# touch /root/hosts （バインドマウント先として/root/hostsファイルを作成）
# mount --bind /etc/hosts /root/hosts （/etc/hostsをバインドマウント）
# ls -l /root/hosts
-rw-r--r-- 1 root root 220 Apr  8 09:49 /root/hosts
# cat /root/hosts （/etc/hostsが/root/hostsにマウントされていることを確認）
127.0.0.1   localhost
```

バインドマウントのあとで/root/hostsを確認すると、/etc/hostsと同じ内容であり、確かにバインドマウントされていることが確認できます。

このバインドマウントを、マウントの情報を見られる/proc/self/mountinfoファイルで確かめてみましょう。

```
# grep hosts /proc/self/mountinfo
477 423 252:2 /etc/hosts /root/hosts rw,relatime - ext4 /dev/vda2 rw
```

このように/etc/hostsが/root/hostsにバインドマウントされていることがわかります。

さて、ここでホスト上で別のシェルを起動してみましょう。この起動したシェルは、OS起動時に作られたデフォルトのNamespaceで動作しており、先ほどの例で作成したMount Namespaceとは別のNamespaceです。

このシェル上で/root/hosts/ファイルと/proc/self/mountinfoファイルを確認します。

```
# ls -l /root/hosts
-rw-r--r-- 1 root root 0 May 13 13:32 /root/hosts
（/root/hostsファイルはあるがサイズが0）
# cat /root/hosts （中身も空）
# grep hosts /proc/self/mountinfo
（バインドマウントされていない）
```

unshareで作成したNamespace内のシェルではバインドマウントされていましたが、ホストOS上で直接実行しているシェル、つまり別のMount Namespaceから見ると/root/hostsファイルの中身は空で、/prof/self/mountinfoファイルにもバインドマウントされている様子はありません。

つまりマウントは、unshareコマンドで作成した**Mount Namespace内だけで有効**であることがわかります。

Mount Namespaceは、ここで紹介した以外にも色々な機能を持っています。詳しくは、本シリーズの第1巻である『Linux Container Book』をご覧ください。

2.4　レイヤー構造のファイルシステムとユニオンファイルシステム

先に説明したように、pivot_rootを使い、ホストや他のコンテナと独立したコンテナ用のファイルシステムを作れます。

実はコンテナとしては、ファイルシステムの視点からだとpivot_rootで独立したファイルシステムを作ることができれば、他に必要な条件はありません[5]。

もちろん、pivot_rootを使うには独立したマウントが必要です。しかし、先にも紹介したとおり、バインドマウントを使えば普通のディレクトリーをマウントポイントにできますので、コンテナ用

5. コンテナそのものとしては、独立したファイルシステムが不要であれば、たとえばネットワークだけ独立しているコンテナというケースもありえます。

の独立したファイルシステムは必ずしも不要です。

実際Docker登場以前のコンテナでは、コンテナ専用のファイルシステムを準備することなく、特定のディレクトリー以下をコンテナのファイルシステムとして使用することが多かったです。

ところがDockerの登場により、コンテナのファイルシステムとして、すっかり当たり前になった構造があります。その構造を取るために、今ではコンテナ専用のファイルシステムをマウントして利用することがほとんどです。

Dockerリリース直後の話を紹介しながら、今では当たり前となったコンテナのファイルシステム構造について見ていきましょう。

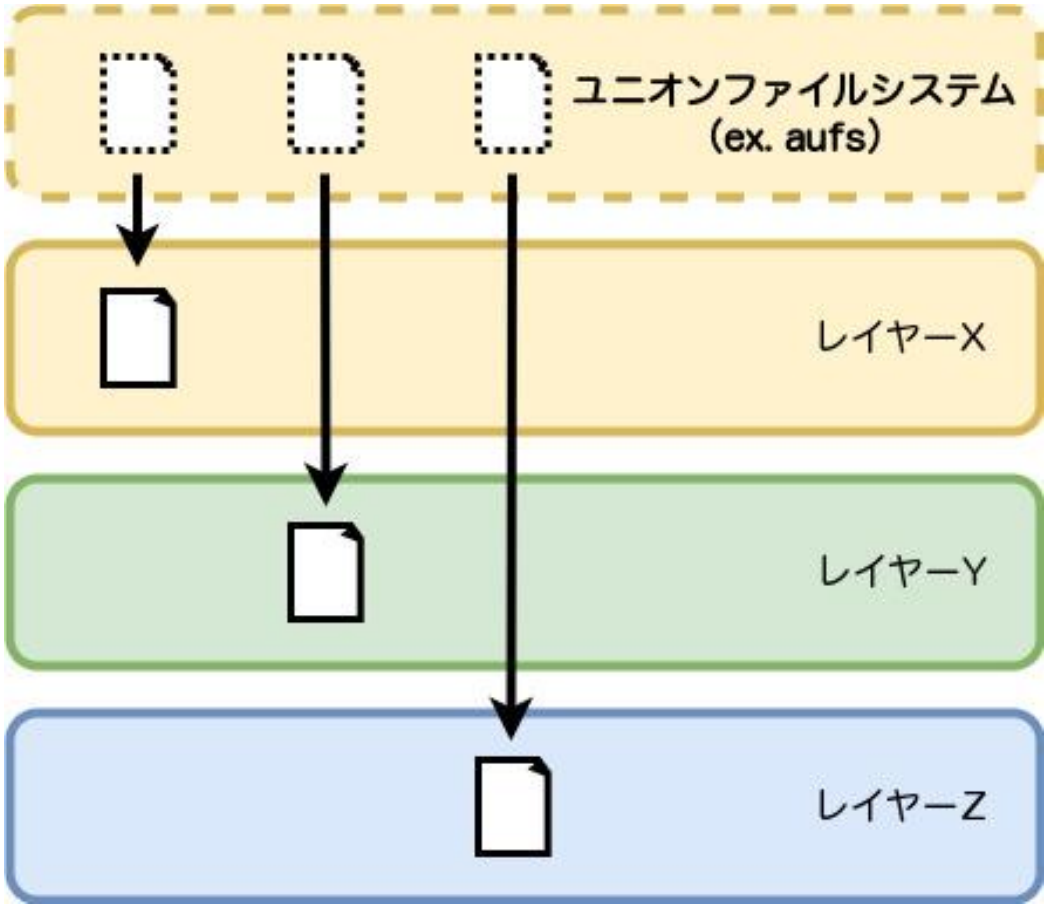
Dockerとユニオンファイルシステム

2013年に登場したDockerには、それまでのコンテナにはなかった革新的な特徴が色々とあります。そのひとつが、レイヤー構造のイメージでしょう。

当時、Ubuntuのカーネルは、aufs[6]という重ね合わせができるユニオンファイルシステムの機能を持っていました。ユニオンファイルシステムとは、図2.5のように、複数のレイヤーを重ね合わせてひとつに見せることができるファイルシステムです。

6.https://aufs.sourceforge.net/

図2.5: ユニオンファイルシステム



ユニオンファイルシステムとしてファイルシステムをマウントすると、図2.5の一番上の点線の部分が見えるので、この部分を操作します。この点線の部分は、実際はレイヤーX、Y、Zを重ね合わせて透過的に見せており、実際のファイルは各レイヤーにあります。

ファイルに対して変更したり、新たにファイルを作成すると、いずれかのレイヤーに対して変更が加えられます。このようなファイルシステムがユニオンファイルシステムです。

このレイヤーは、具体的にはファイルシステム上のディレクトリーです。つまり、複数のディレクトリーの中身を重ね合わせて、ひとつのディレクトリーのように見せています。ext4やXFSといった、システム上でマウントされているファイルシステム上の複数のディレクトリー以下が、重ね合わされて仮想的に見えます。

リリース直後のDockerは、LXCを使ってコンテナを操作していました。このLXCが持つaufsを扱う機能を使い、ベースのイメージと、そこからの差分のみを記録したレイヤーを重ね合わせて、コンテナイメージの軽量化と効率的な管理を実現しました。

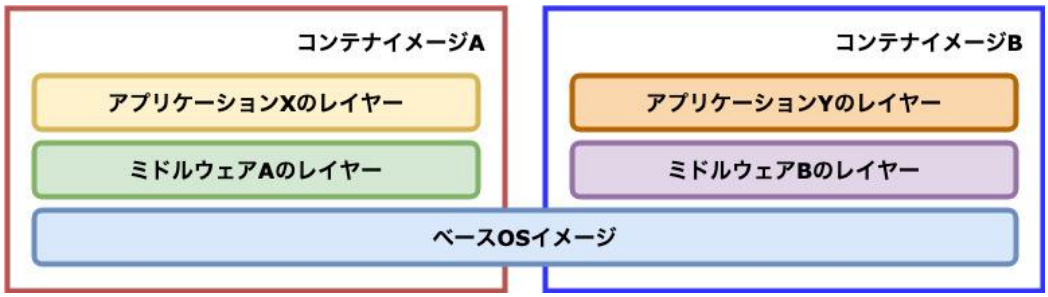
ユニオンファイルシステム自体は、Docker登場よりかなり前から存在していたファイルシステムですので、それ自身は革新的というわけではありません。しかし、そのユニオンファイルシステム

をコンテナイメージに使ったことが、Dockerの革新的だったところでしょう。

このユニオンファイルシステムの機能を使うと、図2.6のように、ベースOSのイメージがあるところに新たにミドルウェアをインストールし、さらにアプリケーションをインストールしたときでも、ベースOSイメージはひとつで済みます。

このベースOSのイメージとは別に、ミドルウェアをインストールしたときにインストールされたファイルだけを管理するレイヤーと、アプリケーションをインストールしたときにインストールされたファイルだけを管理するレイヤーという、ふたつの差分だけを管理し、図2.6のように共通のイメージ（レイヤー）を複数のコンテナで共用できます。

図2.6: ベースOSイメージを共用するコンテナイメージの例



システム上にベースOSイメージはひとつだけインストールすればいいため、容量が節約できます。また、ミドルウェアやアプリケーションを更新するためだけに、ベースOSイメージから作成しなおす必要はありません。差分となる各レイヤーの部分のみを更新すればよくなり、イメージ作成時にかかる負荷も軽減できます。

運用中のコンテナを更新する際、動作中のコンテナの中身を更新するのではなく、新たなイメージを作成して入れ替えるという考え方で運用する場合、更新があるレイヤーだけを取得すればいいので、イメージ更新の負荷が軽減されました。

今では当たり前となった、コンテナを更新する場合は新たなコンテナに入れ替えるという考え方は、このレイヤー構造の考え方により実現されたともいえるでしょう。

aufsは執筆時点で最新である6.8カーネルでも、カーネルにはマージされていません。今ではその代わりとなるOverlayFSがカーネルに標準装備されていますので、コンテナランタイムでユニオンファイルシステムを使う場合は、OverlayFSを使うことがほとんどです。

ユニオンファイルシステムが動作する仕組みとOverlayFSについては、第4章で詳しく説明します。

2.5 コピーオンライトファイルシステム

先に紹介したユニオンファイルシステムは、ファイルシステム上の特定のディレクトリーを重ね合わせてひとつのファイルシステムに見せています。

つまり、操作の単位はファイルです。ベースとなるファイルシステムの、特定の領域（ディレクトリー）に存在するファイルに対して操作が行われます。

これに対して、ファイルシステム自身に、レイヤー構造が取れる構造を持ったファイルシステム

があります。このようなファイルシステムが、コピーオンライトファイルシステムです。

一般的にファイルシステムでは、ファイルやディレクトリーの情報はメタデータとして記録され、実際にデータが記録されている領域であるデータブロックは、メタデータからたどれるようになっています。

ユニオンファイルシステムは、ファイル単位で更新を操作します。それに対して、コピーオンライトファイルシステムでは、ファイルのデータを保存するデータブロック単位で操作します。

コピーオンライトファイルシステムの動きを簡単に見てみましょう。

図2.7のように、「データブロック1」と「データブロック2」からなる「ファイルA」があります。

図2.7: コピーオンライトファイルシステム（1）

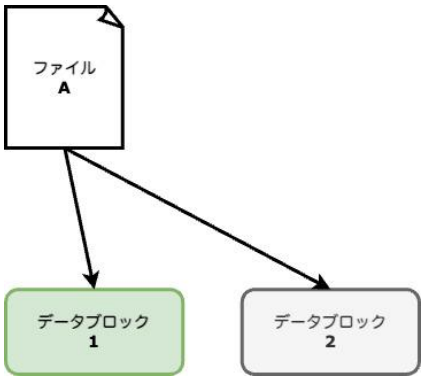
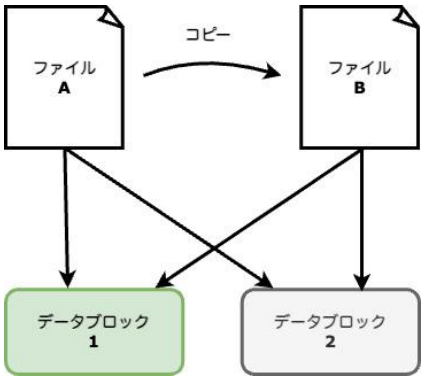


図2.8のように、「ファイルA」を「ファイルB」にコピーします。

図2.8: コピーオンライトファイルシステム（2）

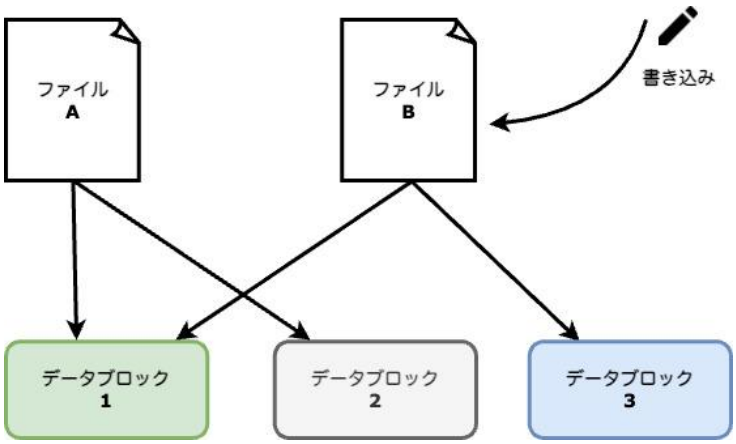


すると、データブロックすべてが新しい領域にコピーされるのではなく、「ファイルA」のメタデータを「ファイルB」用としてコピーします。コピーしただけで、このふたつのファイルのメタデータの中身は同じですので、メタデータは図2.8のように同じ領域を指しています。つまり、実際のデータはふたつのファイルともに共通の「データブロック1」と「データブロック2」に記録されたままです。新たにファイルシステムが使用する領域もメタデータ分以外は増えません。

ここで図2.9のように「ファイルB」を更新し、「データブロック2」に記録されていた部分が変更

されたとします。

図2.9: コピーオンライトファイルシステム（3）



すると図2.9のように、更新されたデータが新たに「データブロック3」に記録されます。そして、「ファイルB」のメタデータも更新され、「データブロック3」を指すようになります。更新がなかった「データブロック1」については変更がなく、「ファイルB」のメタデータも変わらず「データブロック1」を指したままです。

変更がない部分に関しては新たな領域は作られず、変更があった部分だけ新たな領域を確保しますので、容量が節約できます。

このように、コピーオンライトファイルシステムでは、効率的にデータが保存できることが特徴です。

容量の面で効率的であること以外にも、コピーオンライトファイルシステムにはコンテナから使うと便利な機能を持っていることが多く、それがコンテナ環境で使われる理由です。

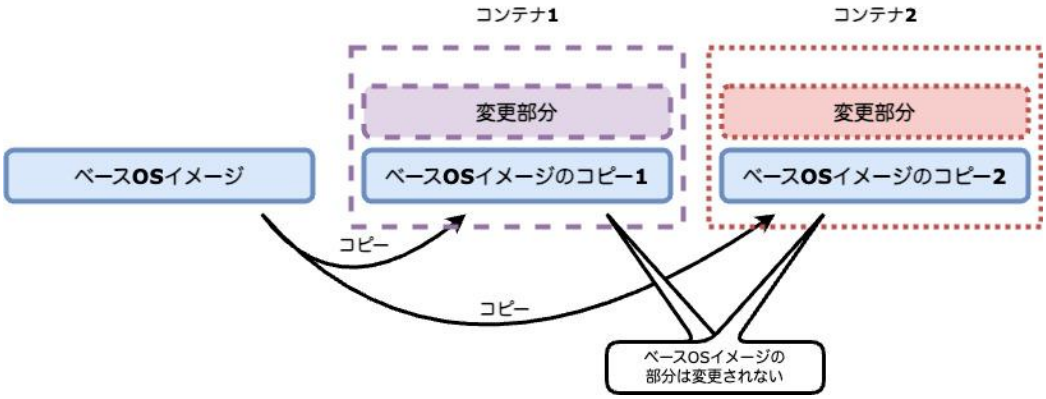
Linuxでよく使われるext4やXFSでは、パーティションにファイルシステムを作成します。しかし、コピーオンライトファイルシステムでは、複数のストレージデバイスやパーティションからストレージプールといった領域を作成し、その領域の中にファイルシステム領域を作成できます。

ファイルシステム領域を作成する際には、スナップショットと呼ばれる機能で、すでに存在するファイルシステムをコピーできます。このファイルシステムをコピーする際に、変化した部分だけを記録するといったコピーオンライト機能を使いますので、ファイルシステム単位で効率的にデータの管理ができます。

このような、コピーオンライトファイルシステムが持つ高度なファイルシステムの機能を、図2.6のようなベースOSイメージを共用する構造に使用します。

たとえば、図2.10のようにベースとなるOSのイメージがあるとします。この「ベースOSイメージ」をコピーしてコンテナをふたつ作ったとします。

図2.10: コピーオンライトファイルシステムで作成するコンテナイメージ



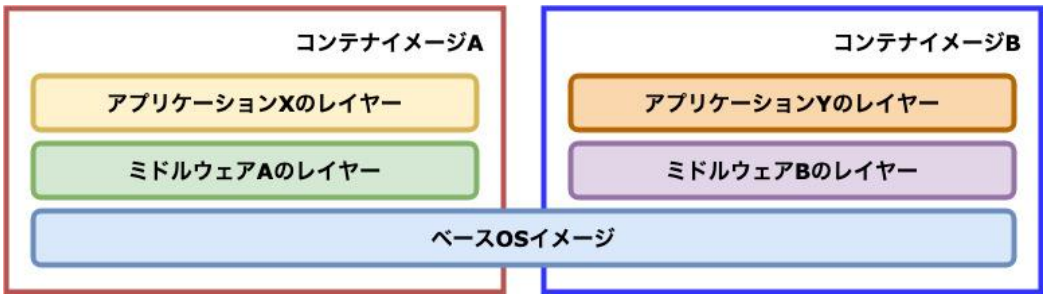
この場合、図2.10の「ベースOSイメージのコピー1」、「ベースOSイメージのコピー2」の部分は、「ベースOSイメージ」から**メタデータのみ**をコピーします。そして、データは「ベースOSイメージ」から変更した部分だけを、それぞれのコンテナの領域で更新していきます。

つまり、図2.11のように、ベースOSイメージ部分のデータは共有で、それぞれのコンテナのファイルシステムでは、コンテナ内で変更された部分のみが記録されていきます。

図2.11: コピーオンライトファイルシステムで作成するコンテナイメージ

元の「ベースOSイメージ」自体をコピーオンライトファイルシステムのストレージプール領域内に、ファイルシステムとして構成しておき、そのコピーを取ってコンテナや新たなコンテナイメージを作成するようにすれば、結局のところは図2.12のように、先にユニオンファイルシステムのところで示した図2.6と同様の構造が取れます。

図2.12: ベースOSイメージを共用するコンテナイメージの例（再掲）



ここまでで、コンテナのファイルシステムに求められる機能や、コンテナのファイルシステムとしてよく使われるファイルシステムが持つ特徴を説明しました。

それでは、具体的なファイルシステムの説明に移りましょう。

第3章　コンテナで使われるファイルシステム

Linuxで使えるファイルシステムは、カーネルソースでfs/以下に実装されています。確認すると、多数のファイルシステムが実装されていることがわかります。

一般的に使われるファイルシステムには、ext4、xfs、btrfsなどがあります。その他、Linuxを起動すると必ず利用する疑似ファイルシステムであるprocfs、cgroupfsや、メモリー上に存在する一時的なファイルシステムであるtmpfsなどがあります。

第2章で説明したとおり、コンテナから使うファイルシステムは、レイヤー構造であることを求められることが多いため、レイヤー構造に対応できるファイルシステムを使うことが多いです。

この章では、コンテナのファイルシステム領域で使われることが多いファイルシステムの概要を紹介します。

3.1　aufs

「2.4 レイヤー構造のファイルシステムとユニオンファイルシステム」で、異なるディレクトリーの内容を重ね合わせて透過的に表示できる、ユニオンファイルシステムについて説明しました。そして、ユニオンファイルシステムのうち、初期のDockerで使われていたaufsを紹介しました。

aufsは2006年ごろから開発されている、ユニオンファイルシステムの実装のひとつです。カーネルにはマージされていませんが、2024年時点でも開発は継続しており、6.xカーネルにも対応しています。

しかし、aufsはカーネルソースに対するパッチという形で提供されており、現在では普通のディストリビューションをインストールした環境では使えません。

aufs以前にも、Unionfsという同様のファイルシステムが存在しました[1]。Unionfsは、CD-ROMブートができるLinuxディストリビューションなどで、読み込み専用のCD-ROM領域と、起動後に書き込まれる領域のファイルシステムを重ね合わせて、透過的に見せる際に使われていました。

aufsは"AnotherUnionFS"の略で、Unionfsを再実装し、機能は維持したまま独自のアイデアや改善を取り入れ実装されています。

「2.4 レイヤー構造のファイルシステムとユニオンファイルシステム」でも紹介したように、Dockerリリース当時（2013年）は、Ubuntuのカーネルにaufsのパッチが適用されていました。そして、DockerはコンテナのLXCを使っており、Dockerの特徴であるレイヤー構造による差分管理を、LXCのaufsドライバーを使って実現していました。

しかしその後、このあと紹介するOverlayFSがカーネルにマージされると、aufsはUbuntuカーネルでサポートされなくなり、今ではコンテナのファイルシステムとしてaufsが使われることはほ

1.https://unionfs.filesystems.org/

とんどなくなりました。

Dockerでは、Docker Engine v24.0でaufsドライバーは削除され、LXCでも3.0.0でaufsドライバーが削除されました。

3.2 OverlayFS

OverlayFSは、2014年、3.18カーネルで導入されたユニオンファイルシステムの実装のひとつです。

先に「3.1 aufs」で、以前はUbuntuではパッチが適用され、aufsが使えるようになっていたという話をしました。実は正式にカーネルに導入される以前から、Ubuntu 12.04や14.04では、aufsだけでなくOverlayFSもパッチが適用されていました。その他、当時のopenSUSEやSUSEでもOverlayFSが使えました。

その後OverlayFSは積極的に開発が進み、カーネルにマージされました。開発の進行につれて、カーネルマージ以前のOverlayFSと、カーネルマージ後のOverlayFSでは仕様が変わりました[2]。

Ubuntuカーネルで使えたため、LXCでは当時からOverlayFSドライバーが実装されており、コンテナのファイルシステムとして使えました。Dockerでは1.4.0からOverlayFSが使えるようになりました[3]。

3.18でマージされたOverlayFSのパッチは非常にシンプルで、たったふたつのパッチで成り立っていました。そのうちひとつはドキュメントですので、実装自体はひとつのパッチのみで成り立っていました[4]。

そのため、3.18時点では機能的には非常にシンプルで、重ね合わせられるレイヤーは2層のみでした。3層以上の複数のレイヤーを重ね合わせられるようになったのは、4.0カーネルからです[5]。

現在Dockerで使われているoverlay2ドライバーは、この4.0でサポートされた多数のレイヤーを重ね合わせられる機能に対応したドライバーです[6]。

なお、LXCではOverlayFSがサポートされている一方で、LXDやIncusではOverlayFSはサポートされていません。

OverlayFSについては、第4章で詳しく説明します。

3.3 Btrfs

Btrfsは、「2.5 コピーオンライトファイルシステム」で紹介したコピーオンライトファイルシステムのひとつです。

もともとはOracle社で開発が始まったファイルシステムです。今では、多くの企業が開発に参加しています。

---

2. これは自慢ですが :-p、Ubuntu などで使えた OverlayFS から仕様が変わった時点では、仕様が変わった OverlayFS に対応するために、LXC の OverlayFS ドライバーに変更が必要であることを誰も認識していませんでした。筆者がそれに気づき、仕様変更に対応するパッチを作成し、無事マージされました。

3.https://github.com/moby/moby/blob/1.12.x/CHANGELOG.md

4.https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e9be9d5e76e34872f0c37d72e25bc27fe9e2c54c

5.https://kernelnewbies.org/Linux_4.0

6. それ以前に overlay というドライバーがありました

Btrfsという名前は、"b-tree filesystem"の略です。Btrfsは、耐障害性、修復、簡単な管理に重点を置きながら、高度な機能を実装することを目的にしています。

Btrfsが持つ特徴は、ここまで紹介したようなコピーオンライトであるということ以外にもたくさんあります。

プール

ひとつ以上のストレージデバイスやパーティションから、ファイルシステムとしてマウントできる領域を切り出す元となるストレージの「**プール**」を作成できます。

プールは、文献によって「ストレージプール」や「Btrfsプール」などと表現されていることもあります。

サブボリューム

前述のプールから、マウント可能な領域としてサブボリュームを切り出せます。

サブボリュームは、プールから論理的に領域を切り出します。プールから複数のサブボリュームを切り出した場合、サブボリュームは容量としてはプールの容量を共有します。

スナップショット

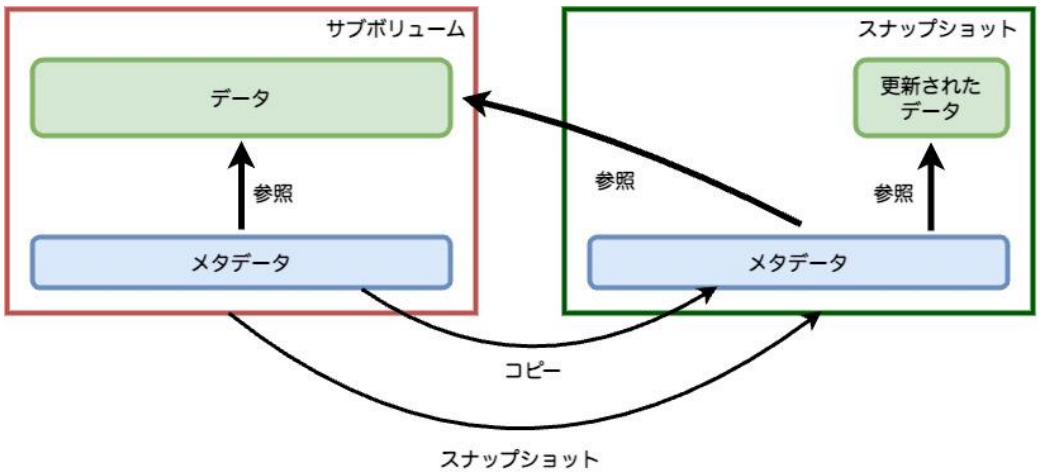
ある時点のサブボリュームの内容からスナップショットが取れます。スナップショットとは、ある時点に存在したファイルやディレクトリーの状態を保存しておく仕組みです。

Btrfsにおけるスナップショットは、図3.1のように取得時点でファイルやディレクトリーをすべてコピーするのではなく、コピーオンライトの仕組みを使って、メタデータのみをコピーし、データそのものは元のサブボリュームの内容を共有します。

したがって、高速に取得でき、容量を大量に消費することはありません[7]。

7. 厳密には、メタデータをコピーするだけでなく、スナップショットを取得した瞬間、ディスクに書き込まれずにメモリー上のキャッシュに乗っているデータを書き戻す処理が必要でしょう。

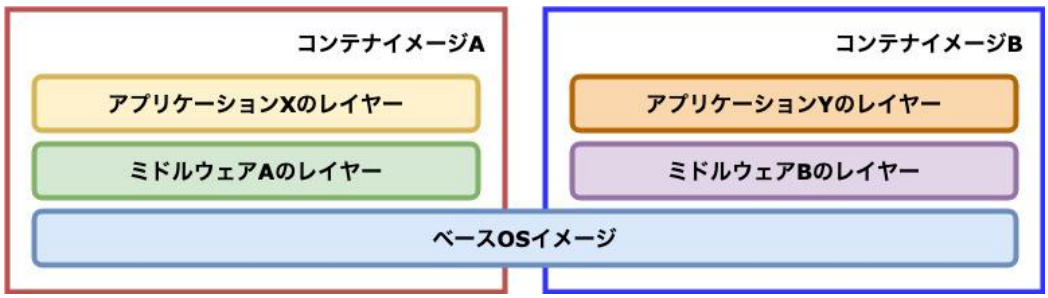
図3.1: Btrfsのスナップショット



このスナップショットは別の領域にマウントでき、そこからスナップショットに対して更新がかけられます。

つまり、コンテナイメージを取得したあと、そのコンテナイメージ自身はそのまま保存しておき、コンテナイメージのスナップショットを取り、そのスナップショットをコンテナのルートファイルシステムとして更新していくことで、図3.2のような構造を取れることが容易に想像できるのではないでしょうか（この図は図2.6の再掲です）。

図3.2: ベースOSイメージを共用するコンテナイメージの例



イメージベースでコンテナを扱い、レイヤー構造を取るにはいちばん重要な機能であるといってもいいでしょう。

その他の機能

ここまでで、コンテナのレイヤー構造で重要なBtrfsの機能は紹介しました。

Btrfsには、これ以外にもファイルシステムとして使う場合に便利な機能が色々と備わっています。

サブボリュームの送受信

サブボリュームの内容をシリアライズして標準出力やファイルに送信できます。たとえばネット

ワーク越しのホストで送信した内容を受信して、サブボリュームとして復元できます。また、親子関係のあるスナップショット間の差分だけを送信できますので、差分バックアップに使えます。

クラスター環境のコンテナホストで、コンテナをホスト間移動するようなケースでも活用できるでしょう。

RAID

RAID構成を組めます。RAID 0、1、10、5、6がサポートされています。その他、データブロックを同じデバイス上の2ヶ所に書き込む"dup"というモードがあります。メタデータとデータそれぞれに別々にRAIDレベルを設定できます。

RAIDはサブボリューム単位ではなく、Btrfsファイルシステム全体に対して設定します。

自己修復機能

データとメタデータにチェックサムを持っています。これにより、データが壊れた場合に検出できます。また、RAID構成の場合、もう一方の正しいデータをもとに修復できます。

透過的圧縮

データを圧縮してストレージ使用量やI/O帯域を節約できます。これは、書き込み時にデータを圧縮してストレージに書き込みます。また、読み出し時は、圧縮されたデータを自動的に展開して元のデータに復元します。ただし、当然ながら読み書きすると、常にCPUパワーを使うことになります。

3.4 LVM

LVMは、ファイルシステムではありません。物理ストレージを抽象化し、論理的にストレージやパーティションを管理します。直接、物理的なデバイスを操作・使用するよりも柔軟性が高くなります。

LVMによって作成した論理ボリューム上に、ファイルシステムを作成し、OSから使用します。

LVMは次のような構成要素から構成されます。

物理ボリューム（PV）

LVMで管理する物理的なストレージやパーティションです。

ボリュームグループ（VG）

PVの集合で構成します。論理ボリュームを切り出す元になるプールです。

論理ボリューム（LV）

VGから切り出した仮想的・論理的なパーティションです。ディスクから作成する、パーティションのようなブロックデバイスです。LVをフォーマットして、ファイルシステムを作成します。

「3.3 Btrfs」で紹介したBtrfsと似た機能があることに気づいた方もいらっしゃるでしょう。Btrfsは普通のファイルシステムの機能に、LVMが持つボリューム管理の仕組みまで含めたファイルシステムであると考えることもできます。

ここまで説明すると、ピンとくる方もいらっしゃるでしょう。LVMにもBtrfsと同様の特徴があ

りますので、コンテナ用のストレージとして利用する際に同じような機能が実現できます。

LVMの特徴をいくつか紹介します。

柔軟性

VGは複数の物理的なストレージやパーティションをひとつのプールにまとめられます。VGにストレージを追加したり削除したりできますので、その上に作成するLVは、柔軟に領域を拡張したり縮小したりできます。拡張や縮小はオンラインでも操作できます。もちろん、その上に作成するファイルシステムがオンラインでのリサイズをサポートしている必要があります。しかし、現在よく使われるファイルシステムであればサポートしているはずです。

オンラインでのデータ移動

システムがアクティブな状態で、データを他のディスクへオンライン移動できます。新しいディスク、より高速なディスク、耐障害性の高いディスクへのデータを移動するといったメンテナンスの際にも役立つでしょう。

ストライピングやRAID構成

LVMを使って、複数のディスクにデータを配置するストライピングや、ミラーなどのRAID構成を取れます。

スナップショット

LVMは、コピーオンライトによるスナップショットをサポートしています。

シンプロビジョニング

実際に利用可能なストレージ容量以上のLVを作成できます。これは、事前にLV内にブロックを割り当てず、実際に書き込むときにブロックを割り当てることで実現しています。LV作成時には仮想的なサイズを設定しますので、実際の容量以上に設定できます。あとで物理的な容量が足りなくなったときにストレージを足します。

コンテナでは、LVMのコピーオンライトを使った機能であるスナップショットとシンプロビジョニングを使って、レイヤー構造のコンテナイメージやファイルシステムを作ります。

ここまで、第2章とこの章で、コンテナのファイルシステムに求められる要件を満たすために使われる機能や、コンテナでよく使われるファイルシステムの概要を紹介しました。

このあとの章では、コンテナでよく使われるOverlayFSについて詳しく見ていきましょう。

第4章　OverlayFS

ここでは、第2章で説明した、コンテナのファイルシステムとして求められる、レイヤー構造のファイルシステムを構成する場合によく使われるOverlayFSについて詳しく紹介していきましょう。

OverlayFSは、3.18カーネルでカーネルに導入されました。しかし、それ以前に、カーネルに正式に導入される前のOverlayFSが、UbuntuやSUSEといったディストリビューションで利用できました。

当然ですが、OverlayFSはLinuxカーネルにファイルシステムとして実装されていますので、コンテナ専用で使う機能ではありません。しかし現在では、もっともコンテナで広く使われているファイルシステムではないかと思います。

4.1　OverlayFSとは

OverlayFSは、ユニオンファイルシステムの実装のひとつです。ディレクトリーを重ね合わせて、ひとつのディレクトリーツリーが構成できます。

Dockerは登場直後には、「3.1 aufs」で紹介した、ユニオンファイルシステムであるaufsをコンテナのファイルシステムとして使っていました。しかし先に紹介したように、aufsはカーネルに対するパッチという形で提供されていますので、すべてのディストリビューションで使えるわけではありません。

一方、OverlayFSはカーネルにマージされていますので、今ではどのディストリビューションでも標準で使えるようになっており、コンテナ環境では一般的に利用されています。

Ubuntuでは12.04 LTSや14.04 LTSの時点で、ユニオンファイルシステムとして、aufsだけではなくパッチを適用し、カーネルにマージされる前のOverlayFSも使えるようになっていました。他にSUSEやopenSUSEでも以前からOverlayFSが利用できていたようです。

Ubuntuで対応していたためか、LXCではかなり前から、aufsだけでなくOverlayFSにも対応していました。

3.18カーネルでマージされた時点のOverlayFSは、比較的シンプルな機能でシンプルに実装されていました。カーネルにマージされたパッチはふたつだけで、そのうちひとつはドキュメントでしたので、実質ひとつのパッチで実現されていました[1]。

ここからは、実際にOverlayFSの動きを見ていきましょう。

4.2　OverlayFSのマウント

OverlayFSは、**複数のディレクトリー**を重ね合わせ、ひとつのディレクトリーに見せる形で、ユ

1.https://github.com/torvalds/linux/commit/e9be9d5e76e34872f0c37d72e25bc27fe9e2c54c

ニオンファイルシステムを構成します。

図 4.1: OverlayFS マウントに使うディレクトリー

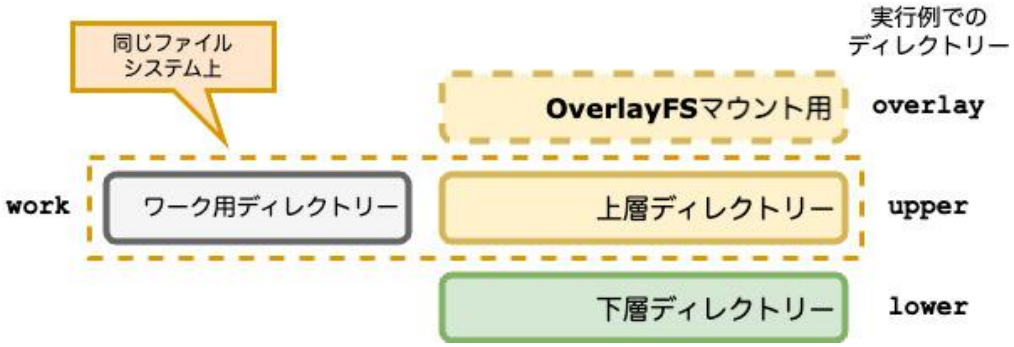


図4.1のように、2層のディレクトリーを重ね合わせます。2層のうちの下層を lower、上層を upper という名前のディレクトリーとします。このふたつを重ね合わせて、overlay というディレクトリーにマウントします。

まずはマウントする方法から説明していきます。

まずは必要なディレクトリーを作成します。重ね合わせの下層にするディレクトリー、上層にするディレクトリー、重ね合わせてマウントするためのディレクトリーを作成します。

```
$ mkdir lower upper overlay work
$ ls -F
lower/  overlay/  upper/  work/
```

OverlayFSでマウントするには、表4.1のようなオプションが必要です。

表 4.1: OverlayFS のマウント関連オプション

オプション	説明	必須	実行例
lowerdir	重ね合わせの下層側ディレクトリー	必須	lower
upperdir	重ね合わせの上層側ディレクトリー	省略可	upper
workdir	ワーク用の空のディレクトリー。upperdir で指定するディレクトリーと同じファイルシステムに存在する空のディレクトリーである必要がある。upperdir を指定しない場合は省略可で、指定しても無視される	省略可	work

upperdir を省略する話は、この後「4.8 複数レイヤー」で行いますので、ここでは省略しない形でマウントします。

それではマウントしてみましょう。

```
$ sudo mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay
overlay
```

何事もなくmountコマンドが終了し、マウントできるはずです。

次に、ここで与えているmountコマンドに与えているオプションを説明します。

-t overlay

マウントするファイルシステムとして、OverlayFSを表すoverlayを指定しています

-o lowerdir=lower,upperdir=upper,workdir=work

-oはマウントオプションを与えるオプションです。それぞれの指定は表4.1のとおりです

overlay

ひとつ目のoverlayという文字列です。実際のデバイスやパーティションの場合は、ここはデバイス名を与えます。しかし、OverlayFSの場合はデバイスはありませんので、ここは適当なキーワードを与えます。わかりやすい文字列にするといいですが、何でも構いません。ここではoverlayというキーワードを指定しました

overlay

ふたつ目のoverlayという文字列です。ここはマウント先のディレクトリーを指定しますので、今回は先に作成したoverlayディレクトリーを指定しています

マウントされているか確認してみましょう。マウントを確認するには、mount -lコマンドを使うか、/proc/self/mountinfoや/proc/self/mountsファイルを確認します。ここではmount -lコマンドで確認しましょう。

```
$ mount -l | grep overlay
overlay on /home/tenforward/tmp/overlay type overlay (rw,relatime,lowerdir=lower,
upperdir=upper,workdir=work)
```

確かにマウントされています。オプションも与えたとおりになっていることを確認してください。

4.3 OverlayFSへの書き込み

それでは、先にマウントしたoverlayディレクトリーに、ファイルとディレクトリーを作成してみましょう。

```
$ touch overlay/testfile_overlay
$ mkdir overlay/testdir_overlay
$ ls -F overlay/
testdir_overlay/  testfile_overlay
```

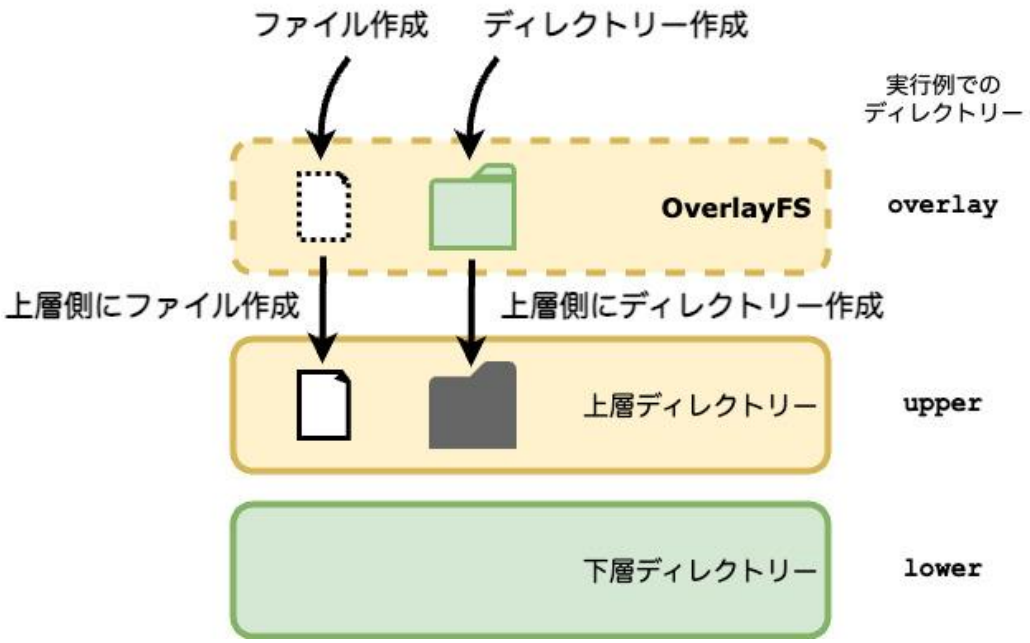
当たり前ですが、普通にディレクトリーとファイルができましたね。ではこのとき、重ね合わせの元となっているlowerとupperはどうなっているか確認してみましょう。

```
$ ls -F lower/ (下層側ディレクトリーの確認)
$ ls -F upper/ (上層側ディレクトリーの確認)
testdir_overlay/  testfile_overlay
```

lowerには何もありません。一方で、upperには先ほど作成したファイルとディレクトリーがあります。

このことから、OverlayFSでは、図4.2のように重ね合わせた**上層側ディレクトリーに変更が加えられる**ということがわかります。

図4.2: ファイルとディレクトリーを作成したときの動き



ここで更に色々試すために、一旦アンマウントします。

```
$ sudo umount overlay
$ mount -l | grep overlay
$
```

4.4　元から上層と下層にファイルが存在する状態でマウント

それではマウントされていない状態で下層であるlowerと、上層であるupperにファイルとディレクトリーを置いてみましょう。

```
$ touch lower/testfile_lower （下層にファイルを作成）
$ mkdir lower/testdir_lower  （下層にディレクトリーを作成）
$ touch upper/testfile_upper （上層にファイルを作成）
$ mkdir upper/testdir_upper  （上層にディレクトリーを作成）
$ ls -F lower/ （下層側ディレクトリーの確認）
testdir_lower/  testfile_lower
$ ls -F upper/ （上層側ディレクトリーの確認）
testdir_overlay/  testdir_upper/  testfile_overlay  testfile_upper
```

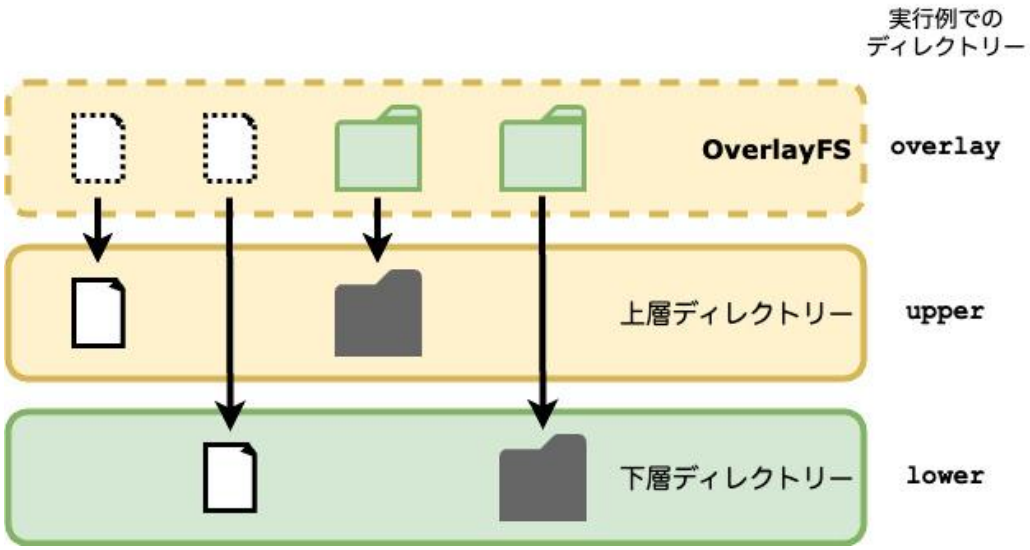
lowerにはファイルとディレクトリーがひとつずつ、upperには先ほどOverlayFSをマウントした状態で作ったファイルとディレクトリーに加えて、今作ったファイルとディレクトリーが存在する状態です。

それでは、再度マウントしてみましょう。

```
$ sudo mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay
overlay
$ ls -F overlay/ （マウントしたディレクトリーの確認）
testdir_lower/    testdir_upper/  testfile_overlay
testdir_overlay/  testfile_lower  testfile_upper
```

「4.2 OverlayFSのマウント」でマウントした状態で作ったファイル、ディレクトリーと、マウントしない状態で作ったファイル、ディレクトリーが全て見えます。

図4.3: 上下層にファイルやディレクトリーが存在する状態でマウント



つまり、図4.3のように、下層と上層それぞれのディレクトリーの内容が重ね合わされた状態で見

えます。

4.5　上層側への変更

それでは、OverlayFSでマウントしたディレクトリーから、上層側に存在するファイルに変更を加えてみましょう。

```
$ echo "test" > overlay/testfile_upper（上層側ファイルを変更）
$ cat overlay/testfile_upper（マウントしたディレクトリーでファイルを確認）
test
$ cat upper/testfile_upper（上層側ディレクトリーでファイルを確認）
test
```

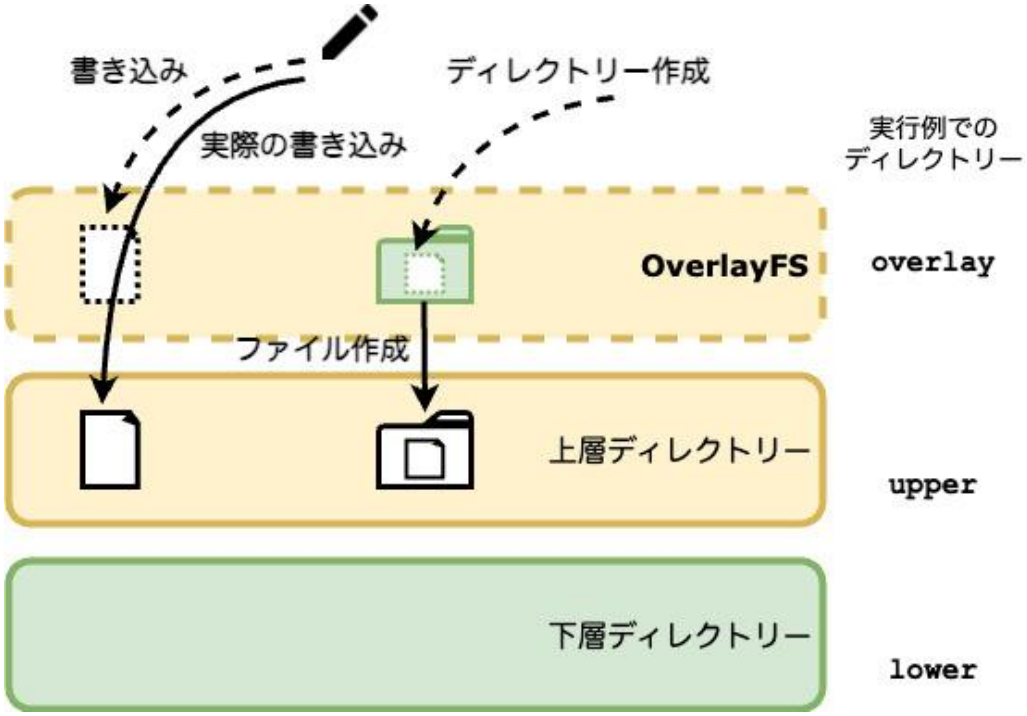
上層側にあるファイルに変更を加えると、そのまま上層にあるファイルが変更されています。

次に、上層側にあるディレクトリー内にファイルを作成してみます。

```
$ touch overlay/testdir_upper/testfile（上層側ディレクトリーにファイルを作成）
$ ls -F overlay/testdir_upper/  （マウントしたディレクトリーでファイルを確認）
testfile
$ ls -F upper/testdir_upper/（上層側ディレクトリーでファイルを確認）
testfile
```

上層側に存在するディレクトリー内にファイルを作ると、図4.4のようにそのまま上層であるディレクトリーに変更が反映され、upper/testdir_upper以下にファイルが作成されました。

図 4.4: 上層側のファイルやディレクトリーへの変更



4.6 下層側への変更

それでは、下層側に存在するファイルやディレクトリーに変更を加えると、どうなるでしょうか? 変更して下層側を確認する前に、下層側のファイルは中身が空なので、あらかじめ何か文字列を書いておきましょう。

```
$ echo test1 > lower/testfile_lower
（下層側ディレクトリーのファイルに直接文字列を書き込む）
$ cat lower/testfile_lower
test1
$ cat overlay/testfile_lower
test1
（マウントしたディレクトリー内で確認しても文字列が書き込まれている）
```

下層側ディレクトリー内のファイルに対して文字列を書き込みました[2]。当然、OverlayFSでマウントされたディレクトリー内にあるファイルの中身も同じです。

それでは、OverlayFSでマウントされたディレクトリー内のファイルを書き換えてみましょう。

2. 実際は、OverlayFS でマウントされた状態で、元となる下層・上層ディレクトリーのデータを書き換えてはいけません。

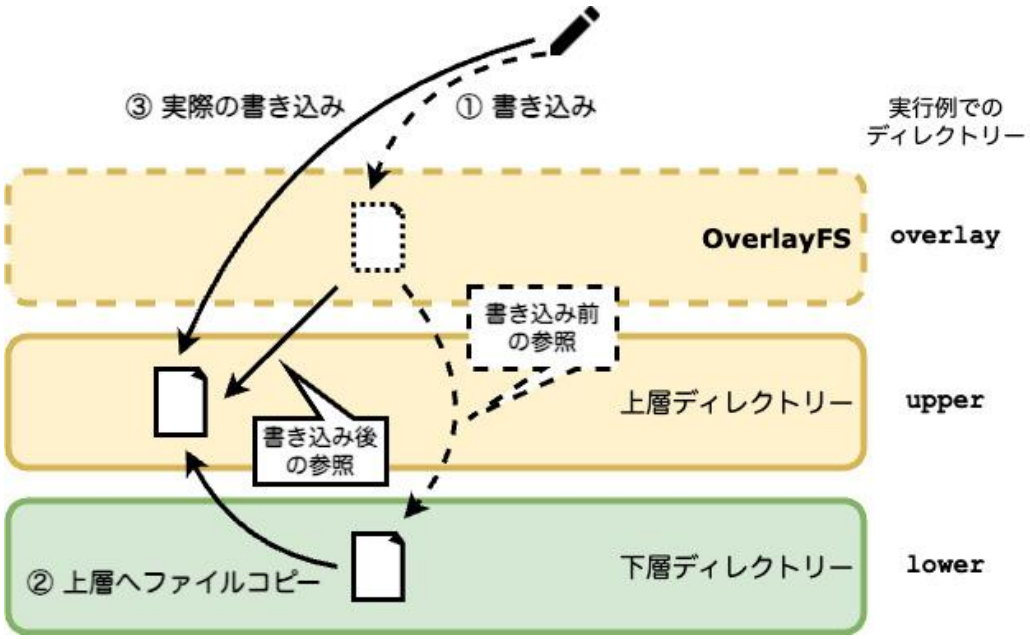
```
$ echo "test2" >> overlay/testfile_lower
（マウントしたディレクトリーから下層側ファイルに追記）
$ cat overlay/testfile_lower
test1
test2
（追記された）
$ cat lower/testfile_lower
test1
（下層側は変更されていない）
```

OverlayFSでマウントされたディレクトリーから見ると、ファイルには確かに変更が加えられています。しかし、下層側に存在するファイルは元通りで変更は加えられていません。ここで、上層側のディレクトリー内を確認してみましょう。

```
$ ls -F upper/
testdir_overlay/  testfile_lower    testfile_upper
testdir_upper/    testfile_overlay
（上層側にtestfile_lowerファイルができている）
$ cat upper/testfile_lower
test1
test2
（中身は追記したあとの状態になっている）
```

上層ディレクトリーには、先ほどまでは存在しなかったtestfile_lowerが存在しています。ファイルの中身は、追記したあとの内容になっています。

図 4.5: 下層側のファイルへの変更



下層側にだけ存在していたファイルに変更を加えると、図4.5のように、

1．下層側のファイルが上層側にコピーされる

2．上層側にコピーされたファイルに変更内容が書き込まれる

という手順でファイルが更新されます。

次に、OverlayFSでマウントされたディレクトリーから、下層側に存在するディレクトリーにファイルを作成してみましょう。

```
$ touch overlay/testdir_lower/testfile（下層側に存在するディレクトリーにファイルを作成）
$ ls -F lower/testdir_lower/  （下層側ディレクトリーでディレクトリーを確認）
$ （下層側ディレクトリーにはファイルは存在しない）
```

ディレクトリーにファイルは存在しません。上層側を確認してみましょう。

```
$ ls -F upper/
testdir_lower/    testdir_upper/  testfile_overlay
testdir_overlay/  testfile_lower  testfile_upper
（上層側にtestdir_lowerができている）
$ ls -F upper/testdir_lower/
testfile  （上層側にできたディレクトリー内にファイルができている）
```

上層ディレクトリーには、先ほどまでは存在しなかった`testdir_lower`と`testfile_lower`が存在しています。ファイルの中身は書き込んだ中身になっていますし、出現したディレクトリー内に

は作成したファイルが存在します。

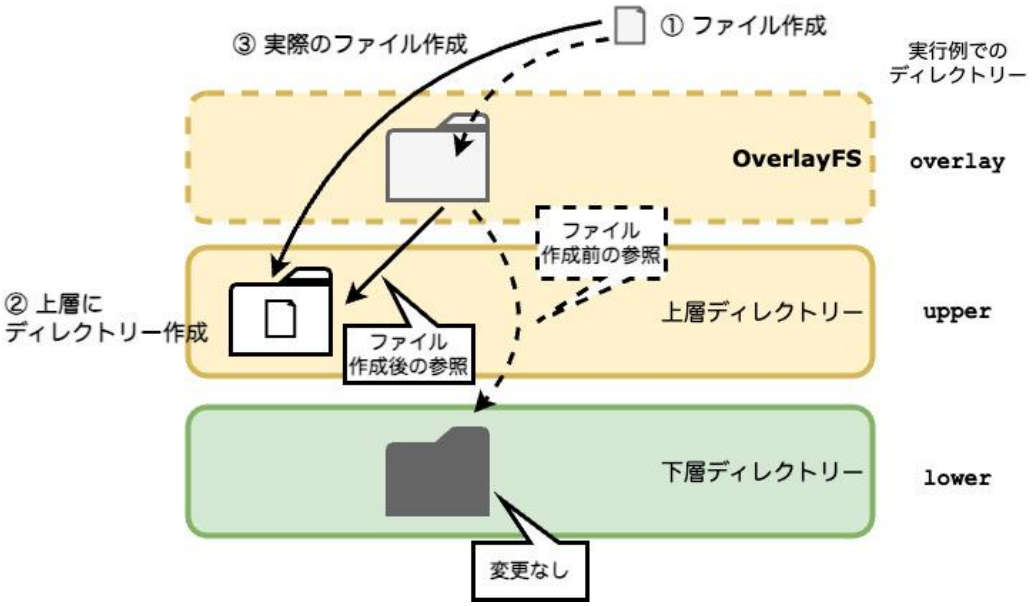
つまり、図4.5のように、下層側にだけ存在するディレクトリーにファイルを作成すると、

1．上層側に下層側と同じ名前のディレクトリーが作成される

2．作成したディレクトリーにファイルが作成される

という動きになります。

図4.6: 下層側ディレクトリーへのファイル作成



ここまでで紹介した動きから、常にupperdirオプションで指定した上層側ディレクトリー以下に変更が加えられることがわかります。

ディレクトリーやファイルが下層にも上層にも存在する場合は、上層側のディレクトリーやファイルが見えます。

4.7　ファイルやディレクトリーの消去

ここまでは、ファイルの作成や変更を見てきました。それでは、ファイルやディレクトリーを削除した場合はどうなるでしょう？

ここまで紹介したように、変更は常に上層側に対して行われたことを考えると、上層側にだけ存在するファイルやディレクトリーを削除した場合はわかりやすいでしょう。

そのまま、上層側に存在するファイルが消去されます。

```
$ rm overlay/testfile_overlay    （マウントしたディレクトリーでファイルを削除）
$ rmdir overlay/testdir_overlay/ （マウントしたディレクトリーでディレクトリーを削除）
$ ls -F overlay/ （マウントしているディレクトリーで確認）
```

```
testdir_lower/  testdir_upper/  testfile_lower  testfile_upper
$ ls -F upper/ (上層側ディレクトリーで確認)
testdir_lower/  testdir_upper/  testfile_lower  testfile_upper
```

先ほどまで上層側に存在していたtestdir_overlayとtestfile_overlayが、マウントされたディレクトリーでも上層側のディレクトリーでも消去されています。

では、下層側のファイルやディレクトリーを消去した場合はどうなるでしょう？

```
$ ls -F overlay/
testdir_lower/  testdir_upper/  testfile_lower  testfile_upper
$ rm overlay/testfile_lower (下層側ファイルをマウントしたディレクトリーで削除)
$ ls -F overlay/
testdir_lower/  testdir_upper/  testfile_upper
(マウントしたディレクトリーではファイルは消えている)
$ ls -F lower/
testdir_lower/  testfile_lower (下層側ディレクトリーに消したファイルが存在している)
$ ls -F upper/
testdir_lower/  testdir_upper/  testfile_lower  testfile_upper
(上層側ディレクトリーにも消したファイルは存在している)
```

下層にも上層にも存在していたtestfile_lowerファイルを消してみました。すると、下層側にはそのままの状態でファイルが残っています。上層側でもtestfile_lowerが残っています。

しかし、グラフィカルな端末で確認している場合、lsコマンド出力のtestfile_lowerファイルのフォントの色や太さが変わっていることに気づくかもしれません。

ls -lコマンドでもう少し詳しく見てみると、

```
$ ls -l upper/testfile_lower
c--------- 2 root root 0, 0 Mar  1 13:42 upper/testfile_lower
```

ご覧のように、普通のファイルではありません。ls -lコマンドの出力で、ファイルの一番最初のカラムはファイルの種類を表します。ここではcとなっていますので、**キャラクター型のデバイスファイル**であることがわかります。

ls -lコマンドの出力で、"0，0"となっている部分は、デバイスファイルの「メジャー番号」、「マイナー番号」を表しており、それぞれどのようなデバイスであるかを示します。ここが両方とも0になっています。下層側に存在するオブジェクトが削除された場合は、上層側にこのような特殊なファイルが作成されます。

ディレクトリーも消去してみましょう。

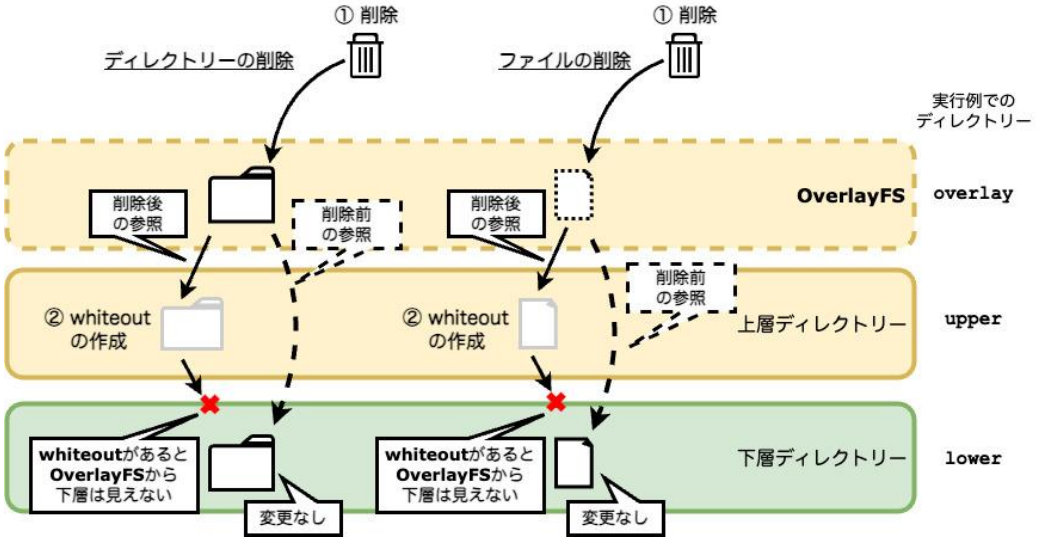
```
$ rm -rf overlay/testdir_lower/ (下層側に存在するディレクトリーをマウントしたディレクトリーから削除)
$ ls -l overlay/testdir_lower
ls: cannot access 'overlay/testdir_lower': No such file or directory
(確かに消えていてマウントされたディレクトリーからは見えない)
$ ls -F lower/
testdir_lower/  testfile_lower
(下層側ディレクトリーにはtestdir_lowerディレクトリーは残ったまま)
$ ls -lF upper/testdir_lower
c--------- 3 root root 0, 0 Mar  1 13:42 upper/testdir_lower
(上層側ディレクトリーでは0, 0のキャラクター型デバイスファイルになっている)
```

削除しましたので、OverlayFSとしてマウントされているoverlayディレクトリー内では、testdir_lowerディレクトリーはなくなりました。

そして上層側ディレクトリーを確認すると、ディレクトリーの場合でも、ファイルのときと同様に、メジャー番号、マイナー番号ともに0のキャラクター型デバイスファイルになりました。

このように、下層に存在するファイルやディレクトリーなどのオブジェクトを削除すると上層に記録される、メジャー番号、マイナー番号がいずれも0の、特殊なキャラクター型デバイスファイルを**whiteout**と呼びます。

図4.7: 下層側ファイルやディレクトリーの削除



whiteoutは、3.18でカーネルにマージされる前のOverlayFSでは、サイズが0で拡張属性trusted.overlay.whiteoutに"y"が設定されたファイルでした。互換性のために、拡張属性が設

定されたファイルやディレクトリーも認識されます[3]。

ここまで紹介したOverlayFSの動作をまとめると、次のようになります[4]。

・変更は、常にupperdirオプションで指定した上層側ディレクトリー以下に対して行われる
・lowerdirオプションで指定した、下層側に存在するファイルに加えられた変更は、上層側に存在するファイルに対して行われる。変更対象であるファイルやディレクトリーが上層側にない場合は、下層側からコピーされる
・下層側に存在するファイルやディレクトリーを削除すると、メジャー番号、マイナー番号ともに0である、特殊なキャラクター型のデバイスファイルが上層側に作成される

4.8 複数レイヤー

ここまでの機能は、3.18カーネルの時点で実装されていたOverlayFSが持つ基本的な機能です[5]。ここまで紹介したOverlayFSの機能だけでは、現在のコンテナに求められる要件には対応できないことに気づいた方もいらっしゃるのではないでしょうか（笑）。

3.18カーネルの時点ではここまでの実行例のように、下層がひとつ、上層がひとつだけを指定する、2層でしか重ね合わせできませんでした。2層だけの重ね合わせでは、一般的に使われる、何層にも構成されたコンテナイメージを作成するには色々と工夫が必要です。

2層しか構成できなかった機能が拡張され、複数のレイヤーが構成できるようになったのは4.0カーネルからです。

複数レイヤーが構成できるようになったのと同時に、上層を指定するマウントオプションであるupperdirは省略できるようになりました。upperdirを指定しない場合は、同時にworkdirも省略でき、指定しても無視されます。

複数レイヤーを構成するには、lowerdirを複数指定します。3.18時点のOverlayFSと変わらず、下層は読み取り専用です。下層を複数指定した場合でも、下層のいずれにも書き込みはされず、上層にだけ変更が書き込まれます。

つまりupperdirを指定しない場合は、読み取り専用のファイルシステムとしてマウントされます。

それでは、複数レイヤーの構成を試してみましょう。

次のようにlower1、lower2、lower3という3つの下層用ディレクトリーと、これまでの例と同様のディレクトリーを作成しました。

```
$ mkdir lower1 lower2 lower3 overlay upper work
（ディレクトリーの作成）
$ touch lower1/testfile_lower1 lower2/testfile_lower2 lower3/testfile_lower3
（各下層ディレクトリーにファイルを作成）
```

3. 拡張属性はこのあと説明します。拡張属性が設定されると同時に、特殊なシンボリックリンクでもありました。なお、デバイスファイル形式と拡張属性 trusted.overlay.whiteoutが設定されたオブジェクトは同居できません

4. 3.18 カーネルで正式導入されるより前の OverlayFS は、マウント時のオプションや、ファイルやディレクトリーを削除した際の動きが異なりました。どのような動きだったか興味がある方は、私の連載の第 18 回（https://gihyo.jp/admin/serial/01/linux_containers/0018）をご覧ください。

5. このあと紹介する opaque ディレクトリー機能も 3.18 時点で実装されていました。

複数の下層を指定するには、lowerdirオプションに、コロン（:）区切りでディレクトリーを複数指定します。

```
$ sudo mount -t overlay -o lowerdir=lower1:lower2:lower3,upperdir=upper,workdir=work
overlay overlay
（コロン区切りで複数の下層側ディレクトリーを指定）
$ ls overlay/
testfile_lower1  testfile_lower2  testfile_lower3
（3つの下層側ディレクトリーの内容が重ね合わされて出現している）
```

マウントすると、上の例のように3つの下層側ディレクトリーの内容がマージされ、表示されていることがわかります。

このように、複数のレイヤーを指定できるようになり、レイヤー構造のコンテナファイルシステムが自然な形で構成できるようになりました。

ここまで紹介した機能が、ユニオンファイルシステムとしてのOverlayFSが持つ基本的な機能です。このあとは、OverlayFSの動きを変えるオプション機能について紹介します。コンテナの起動には直接は関係ありませんので、コンテナに直接関係する機能を知りたい場合は、「4.14 User Namespaceとマウント操作」までは読み飛ばしても問題ありません。

しかし、コンテナのユースケースによっては、オプション機能を使う検討をするとコンテナ運用における問題が解決するかもしれません。

4.9 拡張属性

ここで説明する「拡張属性」はコンテナとは直接関係ない機能であり、ファイルシステムに一般的に実装される機能です。OverlayFSに特化した機能でもありません。

このあと紹介するOverlayFSのオプション機能は、拡張属性を使います。そこで、オプション機能を説明する前に、まずは簡単に拡張属性について説明します。拡張属性の詳細については`man 7 xattr`などの参考文献をご覧ください。

一般的に、ファイルやディレクトリーには所有者やタイムスタンプ、パーミッションといった属性があります。このようなファイルシステムが一般的に持つ属性以外に、ファイルやディレクトリーに任意の情報を保存するために属性を追加できます。

このように、ファイルやディレクトリーに追加される任意の属性を、**拡張属性**といいます。拡張属性を使い、ファイルシステムに追加的な機能を実装したり、カーネルやアプリケーションから拡張属性を参照したりします。

拡張属性は、通常のパーミッションなどの属性と同様に、ファイルやディレクトリーを管理するデータであるiノードに関連付けられます。

拡張属性には、使用用途に応じて**Namespace（名前空間）**があります。ここでいうNamespaceは、第1章で紹介した、コンテナを作るために使用するNamespaceとは別で、拡張属性のNamespace

です。

拡張属性は、"key = value"というようなキーと値のペアで構成されます。このキーの部分が拡張属性名です。拡張属性名は「(拡張属性クラス).(属性名)」というように、「拡張属性クラス」と「属性名」を"."(ドット)で連結した文字列です。拡張属性を、この「拡張属性クラス」で名前空間化して、クラス分けします。

「拡張属性クラス」は用途に応じて使用します。拡張属性クラスのトップレベルは、あらかじめ決まった文字列だけが使えるようになっており、表4.2のような拡張属性クラスがあります。

表4.2: 拡張属性クラスの種類

拡張属性クラス	代表的な用途
system	カーネルが使用。ACL(アクセス制御リスト)で使用
security	SELinuxなどのカーネルのセキュリティーモジュールやケーパビリティが使用
trusted	CAP_SYS_ADMINケーパビリティを持つプロセスが、通常のプロセスからアクセスできない情報を保存するために使用
user	ユーザー定義(自由に定義できる)

拡張属性名は、たとえばSELinuxの場合、拡張属性クラスにsecurityを使いますので、security.selinuxという拡張属性名になります。

拡張属性を扱うには、Ubuntuの場合attrパッケージをインストールします。

```
$ sudo apt install attr
```

拡張属性クラスがuserであれば任意に設定できますので、簡単に試してみましょう。テスト用のファイルを作成し、user.myattrという拡張属性にtestvalueという値を設定します。

拡張属性の設定にはsetfattrコマンドを、拡張属性を参照するにはgetfattrコマンドを使用します。

```
$ touch testfile （ファイルの作成）
$ setfattr -n user.myattr -v testvalue testfile
（testfileの拡張属性名user.myattrにtestvalueという値を設定）
$ getfattr -n user.myattr testfile
（設定した値の確認）
# file: testfile
user.myattr="testvalue" （設定できている）
```

SELinuxが有効になっているRHEL系のディストリビューションでは、ファイルを作成すると、SELinuxの拡張属性が設定されます。Rocky Linux 8の環境で次のようにファイルを作成すると、自動的に拡張属性が設定されました。

```
$ touch testfile （ファイルの作成）
$ sudo getfattr -n security.selinux testfile
（testfileのsecurity.selinuxの値を参照）
# file: testfile
security.selinux="unconfined_u:object_r:user_home_t:s0"
```

ここまでで、拡張属性について簡単に紹介しました。

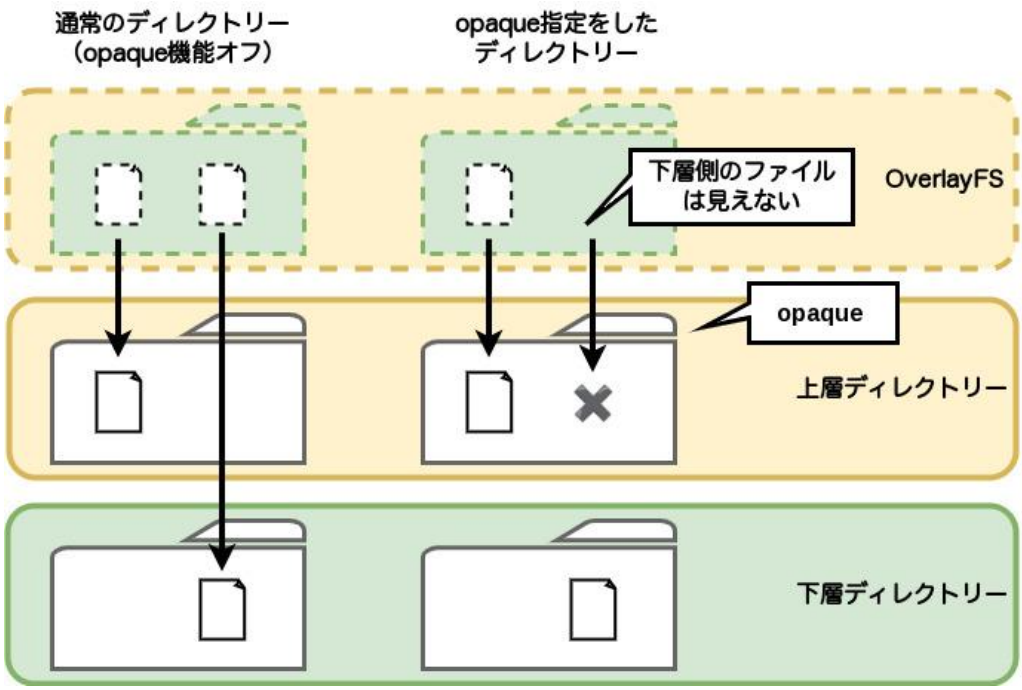
OverlayFSは、オプション機能でこの拡張属性を使います。それでは、オプション機能について紹介していきましょう。

4.10　opaque（不透明）ディレクトリー機能

OverlayFSには、opaque（不透明）ディレクトリーという機能があります。この機能は、OverlayFSがカーネルにマージされた時点の3.18で実装されていた機能です。

通常のOverlayFSでは、上層側と下層側が透過的に重ね合わせられます。しかし、上層側に存在するディレクトリーで、図4.8のようにopaqueという指定がなされている場合は、上層側に存在するオブジェクトだけが見えて、下層側に存在するオブジェクトは見えません。

図4.8: opaque（不透明）ディレクトリー機能



OverlayFSは、この機能に拡張属性を使います。opaqueを指定する場合、上層側ディレクトリー

の拡張属性trusted.overlay.opaqueに"y"を設定します。

試してみましょう。

```
$ mkdir lower upper overlay work
$ mkdir lower/opaquetest upper/opaquetest（上層、下層側ディレクトリーにディレクトリーを作成）
$ touch lower/opaquetest/testfile_lower upper/opaquetest/testfile_upper
（作成したディレクトリーにファイルを作成）
```

上のように、opaquetestというディレクトリーを上層側と下層側のディレクトリーに作成します。その中にそれぞれファイルを作成します。

このままovarlayディレクトリーにマウントすると、overlay/opaquetestの下にはふたつのファイルが見えるはずです。

```
$ sudo mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay
overlay
（マウント）
$ ls overlay/opaquetest/
testfile_lower  testfile_upper （下層・上層両方のファイルが見えている）
$ sudo umount overlay
```

期待通りふたつファイルが存在します。確認したら一度アンマウントします。

そして、上層側ディレクトリーupperに、拡張ファイル属性trusted.overlay.opaqueとして"y"を設定します。

```
$ sudo setfattr -n "trusted.overlay.opaque" -v "y" upper/opaquetest/
$ sudo getfattr -n "trusted.overlay.opaque"  upper/opaquetest/
# file: upper/opaquetest/
trusted.overlay.opaque="y"
```

拡張ファイル属性が設定されていることが確認できましたので、マウントします。

```
$ sudo mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay
overlay
$ ls overlay/opaquetest/
testfile_upper（上層側に存在するファイルしか見えない）
```

先ほどと違って、上層側であるディレクトリーに存在するファイルしか見えなくなっています。

このように下層側に存在する同名のディレクトリーを無視する機能が、opaqueディレクトリー機能です。

4.11 redirect_dir機能

次に、4.10カーネルで導入されたredirect_dir機能を紹介します。

この機能は、**下層側のディレクトリーをリネーム（移動）する場合**に関係する機能です。

redirect_dir機能の制御

redirect_dir機能を使うには、いくつか方法があります。

・カーネルビルド時のオプション

・カーネルモジュールロード時のオプション

・マウント時のオプション

カーネルビルド時に指定できるredirect_dir機能関連のオプションは、表4.3のとおりです。

表4.3: redirect_dir機能に関連するカーネルオプション

カーネルオプション	機能
OVERLAY_FS_REDIRECT_DIR	Y（有効）の場合、redirect_dir機能はデフォルトで有効になる
OVERLAY_FS_REDIRECT_ALWAYS_FOLLOW	Y（有効）の場合、redirect_dir機能が無効の場合でも、redirect_dir機能を使ってリネームされたディレクトリーを読み取る場合はredirect_dir機能を使う

本書の実行例で使用しているUbuntu 22.04のカーネルでは、OVERLAY_FS_REDIRECT_DIRは無効、OVERLAY_FS_REDIRECT_ALWAYS_FOLLOWは有効に設定されています。つまり、デフォルトではredirect_dir機能は無効ですが、過去に機能を有効にしてリネームしたディレクトリーがある場合には、redirect_dir機能を使います。過去に有効なカーネルを使ったり、機能を有効にして使ったことがある場合との互換性を考慮した設定がされていると思われます[6]。

カーネルモジュールをロードする際に指定できるオプションは、表4.4のとおりです。

表4.4: redirect_dir機能を使うためのモジュールオプション

オプション	機能	指定できる値
redirect_dir	redirect_dir機能の有効・無効の切り替え	on/off
redirect_always_follow	onを指定すると、redirect_dir機能が無効の場合でも、redirect_dir機能を使ってリネームされたディレクトリーを読み取る場合はredirect_dir機能を使う	on/off
redirect_max	リダイレクトの最大バイト数（デフォルト256）	バイト数

6. この設定はディストリビューションによって異なりますので、お使いのディストリビューションのカーネルconfigを確認してください。

マウント時に指定できるオプションは、表4.5のとおりです。

表4.5: redirect_dir機能を使うためのマウントオプション

オプション	機能
redirect_dir	redirect_dir機能の制御。指定できる値は表4.6を参照

表4.6: redirect_dirオプションに指定する値

redirect_dirの値	処理
on	機能が有効になる
follow	機能は無効だが、redirect_dir機能を使ってリネームされたディレクトリーを読み取る際はredirect_dir機能を使う
nofollow	機能は無効になる
off	redirect_always_followが有効な場合はfollowと同じ。無効な場合はnofollowと同じ

それでは、redirect_dir機能の説明をするために、下層側ディレクトリーに存在するディレクトリーをリネームしたときのデフォルト（redirect_dir無効）の動きを詳しく見ていきましょう。

redirect_dir機能が無効の場合

OverlayFSのデフォルトでは、redirect_dir機能は無効です。

機能が無効の場合、下層側に存在するディレクトリーがリネームされたとき、つまりmvなどのコマンドからrename(2)システムコールが呼ばれた場合は、OverlayFSはEXDEVというエラーを返します。

通常、ext4やxfsなどの普通のファイルシステムを使っているときのリネーム（移動）では、ファイルやディレクトリーのフルパスをファイルシステムをまたぐように変更した場合、つまり異なるファイルシステムをまたがって移動した場合、リネーム先のファイルシステムにディレクトリーが作成され、ファイルシステムをまたいでファイルがコピーされます。

このようなとき、内部的にはコマンドに対してEXDEVというエラーが返っています[7]。

EXDEVというエラーが返ったときは、rename(2)を呼び出したコマンドが対処します。たとえばmvコマンドでは、ファイルなどのオブジェクトを移動先にコピーし、元のオブジェクトを削除します。

OverlayFSではこれまで説明したように、下層側のオブジェクトに変更があると、下層側のオブジェクトが上層部にコピーされます。

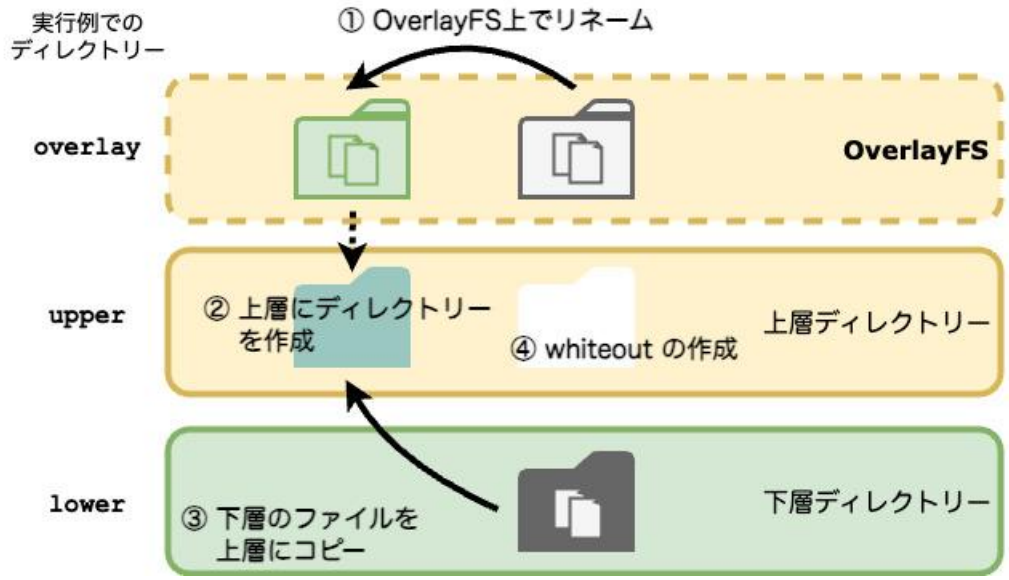
つまり下層側のディレクトリーをリネームしたときは、元のディレクトリーのデータがコピーされます。コピーにより新しいオブジェクトが生成されると、OverlayFSは上層側に新たなオブジェクトを作成します。よって、リネーム先のディレクトリーが上層部に作成され、ディレクトリーに含まれるオブジェクトは上層部のディレクトリーにコピーされます。

7. 詳しくはman 2 renameをご覧ください。

上層部への変更が完了すると、リネーム元のディレクトリーは削除されますので、下層側のリネーム元のディレクトリーが見えないように、上層側にwhiteoutが作成されます。

このように、OverlayFSからEXDEVというというエラーを返すことで、OverlayFS的にも矛盾しない動きになるように考えられて実装されていることがわかります。

図4.9: redirect_dir機能無効（デフォルト動作）のときの動作



それでは、この動きを実際に確認しましょう。

まずは従来の動きにするように、redirect_dir=offというオプションを与えます。Ubuntu 22.04の場合、redirect_dirは無効がデフォルトですので、指定しないでマウントするのと同じです。

```
$ mkdir lower upper work overlay （overlayfs用のディレクトリーの作成）
$ mkdir lower/lowerdir upper/upperdir （下層、上層それぞれにディレクトリー作成）
$ touch lower/lowerdir/lowfile upper/upperdir/upfile （ディレクトリー内にファイル作成）
$ sudo mount -t overlay \
> -o lowerdir=lower,upperdir=upper,workdir=work,redirect_dir=off \
> overlay overlay/ （redirect_dir=offを指定してマウント）
```

重ね合わされたoverlayディレクトリーの中身は次のようになります。普通の動きですね。

```
$ tree -n overlay/
overlay/
├── lowerdir
│   └── lowfile
└── upperdir
```

```
    └──── upfile

2 directories, 2 files
```

lowerdirをリネームします。

```
$ cd overlay
$ mv lowerdir lowerdir2 （リネーム）
$ ls -F
lowerdir2/  upperdir/
```

ここで下層側のlowerdirの中を覗いてみます。

```
$ tree -n lower/
lower/
└──── lowerdir
    └──── lowfile

1 directory, 1 file
```

下層側に変化はありません。OverlayFSでは下層側は変化しませんので、これは当然の結果です。
次に上層側のupperを確認します。

```
$ ls -l upper/
total 8
c--------- 2 root       root       0, 0 Mar 18 13:45 lowerdir
drwxrwxr-x 2 tenforward tenforward 4096 Mar 18 13:37 lowerdir2
drwxrwxr-x 2 tenforward tenforward 4096 Mar 18 13:37 upperdir
```

OverlayFSでは、削除されたファイルやディレクトリーは、すでに説明したようにwhiteoutになります。lowerdirは削除された状態になっており、新たに移動先のlowerdir2が上層側に作成されています。lowerdir2内には、元々下層側のlowerdir内に存在していたファイルがコピーされています。

```
$ ls -l upper/lowerdir2/
total 0
-rw-rw-r-- 1 tenforward tenforward 0 Mar 18 13:37 lowfile
```

ディレクトリー内に多数のファイルやディレクトリーがある場合は、それらをすべてコピーしますので時間がかかりそうです。

redirect_dir機能が有効の場合

それでは、redirect_dir機能を有効にしてみましょう。
先の例と同様にディレクトリーやファイルを作成したあと、機能を有効にしてマウントします。

```
$ mkdir lower upper work overlay
$ mkdir lower/lowerdir upper/upperdir
$ touch lower/lowerdir/lowfile upper/upperdir/upfile
$ sudo mount -t overlay \
> -o lowerdir=lower,upperdir=upper,workdir=work,redirect_dir=on \
> overlay overlay
```

マウントができたら、redirect_dir機能が有効になっているかを確認します。

```
$ mount -l | grep overlay
overlay on /home/tenforward/tmp/overlay type overlay (rw,relatime,lowerdir=lower,
upperdir=upper,workdir=work,redirect_dir=on)
```

このようにredirect_dir=onという表示がありますので、機能は有効です。

```
$ tree -n overlay
overlay
├── lowerdir
│   └── lowfile
└── upperdir
    └── upfile

2 directories, 2 files
```

redirect_dir機能が無効のときと同じように、下層と上層が重ね合わされた状態になっています。この状態で、先ほどと同じようにディレクトリをリネームしてみましょう。

```
$ cd overlay/
$ ls
lowerdir/  upperdir/
$ mv lowerdir lowerdir2
$ ls
lowerdir2/  upperdir/
```

リネームされました。それでは、下層と上層のディレクトリー内を覗いてみましょう。

```
$ ls -l upper/
total 12
c--------- 1 root       root       0, 0 Mar 18 13:55 lowerdir
drwxrwxr-x 2 tenforward tenforward 4096 Mar 18 13:49 lowerdir2
drwxrwxr-x 2 tenforward tenforward 4096 Mar 18 13:49 upperdir
```

上層側のディレクトリー内は、機能が無効のときと同じですね。移動前のlowerdirがwhiteoutにより削除された状態になっており、新たにlowerdir2が作成されています。このlowerdir2内を覗いてみましょう。

```
$ ls -l upper/lowerdir2
total 0
（上層側ディレクトリー内にはファイルがない）
$ ls -l overlay/lowerdir2
total 0
-rw-rw-r-- 1 tenforward tenforward 0 Mar 18 13:49 lowfile
（マウントしたディレクトリーから見るとファイルがある）
```

機能が無効のときは、ディレクトリー以下のファイルが上層側ディレクトリーにコピーされていました。しかし、今回は上層側ディレクトリー配下にはlowfileファイルが存在しません。にもかかわらず、マウントして重ね合わせたディレクトリーにはちゃんとファイルが存在します。下層側ディレクトリーを見てみると、

```
$ ls -l lower/lowerdir/
total 0
-rw-rw-r-- 1 tenforward tenforward 0 Mar 18 13:49 lowfile
```

ファイルが存在します。こちらは従来と変わりありません。

つまり、redirect_dir機能を有効にすると、

・上層側にリネームしたディレクトリーは作成される

・もともと下層側にあったファイルはコピーされない

・マウント先から見るとファイルは存在する

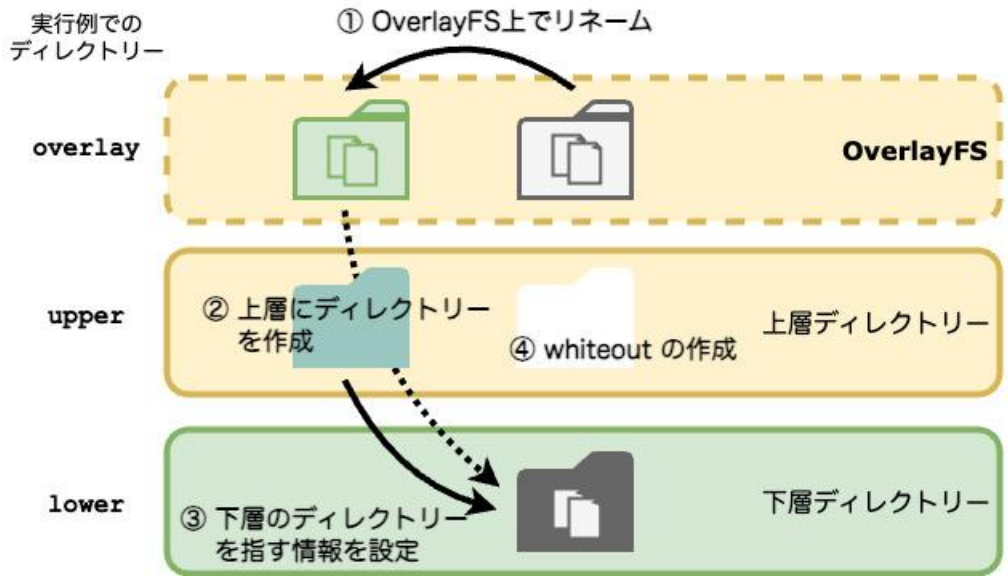
という動きになることがわかります。

このように見える理由は、redirect_dir機能を有効にしてディレクトリーをリネームした場合、上層側にリネームしたディレクトリーが作成されるだけで、ディレクトリーに存在するファイルは元の下層側のファイルを参照しているためです。

このような動きをどうやって実現しているかというと、図4.10のように、上層側に作成されたディレクトリーに下層側を参照する情報を設定します。この情報は**拡張属性**に設定されます。設定される拡張属性名はtrusted.overlay.redirectです。

そして、OverlayFSとしてマウントされたディレクトリーでリネームされたディレクトリーが参照されると、拡張属性の値を見て、実際にファイルが存在するディレクトリーの中身を参照します。

図 4.10: redirect_dir 機能が有効なときの動作



先ほどの実行例で、リネームしたディレクトリーの拡張属性に設定されている値を見てみましょう。

```
$ sudo getfattr -n trusted.overlay.redirect upper/lowerdir2
# file: upper/lowerdir2
trusted.overlay.redirect="lowerdir"

（拡張属性に参照先のディレクトリー名が保存されている）
```

このように参照先が保存されています。

ここで、redirect_dir 機能を使っている場合、下層側に存在するファイルに変更が加わるとどうなるかも見ておきましょう。

```
$ echo test >> overlay/lowerdir2/lowfile
（マウントしたディレクトリー上でファイルを変更）
$ ls -l upper/lowerdir2/
total 4
-rw-rw-r-- 1 tenforward tenforward 5 Mar 18 14:47 lowfile
（上層側にファイルができた）
$ cat upper/lowerdir2/lowfile
test
（上層側のファイルには行った変更が反映されている）
```

```
$ cat lower/lowerdir/lowfile
（下層側のファイルには変更は反映されていない）
$
```

このように、redirect_dir機能を使っていても、下層側に存在するファイルに変更が加わると、上層側にファイルがコピーされて変更が加えられます。つまり、redirect_dir機能は上層側に存在しないファイルのみ、下層側を参照するということがわかります。

次に、redirect_dir機能を使っている場合で、ディレクトリーを親ディレクトリーに移動したりして階層が変わったりした場合に、どのような情報が拡張属性に設定されるのかを見ておきましょう。

```
$ mkdir lower upper work overlay
$ mkdir -p lower/lowerdir/lowerdir2
$ sudo mount -t overlay \
> -o lowerdir=lower,upperdir=upper,workdir=work,redirect_dir=on \
> overlay overlay
$ mv overlay/lowerdir/lowerdir2 overlay/lowerdir3
（ディレクトリーをひとつ上の階層に移動）
$ ls -l overlay/
total 16
drwxrwxr-x 1 tenforward tenforward 4096 Mar 18 14:18 lowerdir
drwxrwxr-x 1 tenforward tenforward 4096 Mar 18 14:18 lowerdir3
（移動した）
$ ls -l upper/lowerdir
total 0
c--------- 1 root root 0, 0 Mar 18 14:18 lowerdir2
（移動元はwhiteoutが設定されている）
$ sudo getfattr -n trusted.overlay.redirect upper/lowerdir3
# file: upper/lowerdir3
trusted.overlay.redirect="/lowerdir/lowerdir2"

（下層側ディレクトリーで指定したディレクトリーからのパスが設定されている）
```

このように階層が変わると、拡張属性にはパスを含めた情報が入ります。

パス名を含んだtrusted.overlay.redirectの値の場合は、マウント時にlowerdirで指定したディレクトリーを/（ルート）としたパス名で値が設定されます。

ちなみに、redirect_dir機能をサポートしていないカーネルでは、redirect_dir機能を使って操作されたOverlayFSの下層・上層ディレクトリーを使ったマウントはできません。

4.12　xino機能

xino機能は4.17カーネルで導入された機能です。

ファイルシステムが、ファイルやディレクトリーといったオブジェクトを管理する際には、オブジェクトのサイズやアクセス権限、データ領域の情報をまとめたメタデータと、実際のデータが保存されている領域を分けて管理しています。

このうち、オブジェクトのメタデータが保存されている領域を**iノード**といいます。このiノードを識別するために、iノードには**iノード番号**という番号が振られており、ファイルシステムはiノード番号とオブジェクト名を対応させて、オブジェクトを管理しています。

iノードは、ファイルシステムごとに持つ情報ですので、ファイルシステムが異なれば、同じiノード番号が存在する可能性があります。

このため、ファイルシステムとiノードの組で、システム上のオブジェクトを一意に特定します。

ファイルシステムやiノードなどのオブジェクトの情報を取得するには、Linuxではstatコマンドや、stat(2)システムコールを使います。

statコマンドを実行すると、次のような結果が得られます。

```
$ stat /usr/bin/stat
  File: /usr/bin/stat
  Size: 93040       Blocks: 184        IO Block: 4096   通常ファイル
Device: 259,5   Inode: 4981212     Links: 1
Access: (0755/-rwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2024-04-07 22:49:20.536646833 +0900
Modify: 2024-01-23 14:44:28.000000000 +0900
Change: 2024-01-23 14:50:59.984729766 +0900
 Birth: 2024-01-23 14:50:59.984729766 +0900
```

上の実行例でファイルシステムは"Device"として表示されている"259,5"、iノードは"Inode"として表示されている"4981212"です。

これらの情報は、stat(2)システムコールを実行した結果として返されるstat構造体に含まれています。このうち、Deviceに表示されている情報は構造体のメンバーst_devの値、Inodeに表示されている情報はst_inoの値です。

```
struct stat {
    dev_t     st_dev;     /* ファイルがあるデバイスの ID */
    ino_t     st_ino;     /* inode 番号 */
    :（略）
```

通常のファイルシステムであれば、st_devの値はファイルシステム上のオブジェクトすべてで共

通です。そして、st_inoの値は同じオブジェクトであれば不変です[8]。

さて、OverlayFSでは複数のファイルシステムを重ね合わせている関係上、同一ファイルシステム上にあるオブジェクトであってもst_devが違っていたり、st_inodeの値が永続的ではなく、マウント中であってもst_inodeが変化する可能性があるようです[9]。

OverlayFSは、他のファイルシステム上のオブジェクトを重ね合わせているために、普通のファイルシステムと異なる動きをする可能性があります。inode番号とデバイスIDが、永続的ではない可能性があることはそのひとつです。

このように一般的なファイルシステムと異なる動きを、一般的なファイルシステムに近づける機能のひとつが、このxino機能です。OverlayFSで、inode番号（st_inode）とデバイスID（st_dev）を永続的にします。

xino機能の制御

xino機能を使うには、いくつか方法があります。

・カーネルビルド時のオプション
・カーネルモジュールロード時のオプション
・マウント時のオプション

カーネルビルド時に指定できる、xino機能関連のオプションは表4.7のとおりです。

表4.7: xino機能に関連するカーネルオプション

カーネルオプション	機能
CONFIG_OVERLAY_FS_XINO_AUTO	Y（有効）の場合、xino機能はデフォルトで有効になる

このカーネルオプションが有効か無効かは、ディストリビューション次第です。本書で実行例として使っているUbuntu 22.04では、この設定が有効になっています。マウントオプションを指定しなくても、常にxino機能が有効です。無効化したい場合は、このあと紹介するモジュールオプション、またはマウントオプションで無効化する必要があります。

カーネルモジュールをロードするときのオプションは、表4.8のとおりです。

表4.8: xino機能に関連するモジュールオプション

モジュールオプション	機能	指定できる値
xino_auto	xino機能の有効・無効を切り替える	on/off

マウントするときのオプションは、表4.9のとおりです。

表4.10のautoにある「要件」についてはあとで説明します。

それでは、実際にxino機能を有効にした場合と無効にした場合の違いを説明していきましょう。

8.mv コマンドで、ファイルに別ファイルを上書きしたようなケースで、同一ファイル名なのに inode 番号が変わることはあります。ファイルが別オブジェクトに置き換わるためです。

9. 筆者は少し試してみたのですが、同一 OverlayFS で st_dev の値が異なったり、st_ino の値が変わるようなケースには出会えませんでした。どういう場合にこのようなことになるのかがわかりません。

表4.9: xino機能を使うためのマウントオプション

オプション	機能
xino	xino機能の制御。指定できる値は表4.10を参照

表4.10: xinoオプションに指定する値

xinoの値	処理
on	機能が有効になる
auto	永続的なst_inoとなる要件が満たされているときのみ機能が有効になる
off	機能が無効になる

その前に、特殊ケースとして、ひとつのファイルシステム上のディレクトリーを重ね合わせて、OverlayFSをマウントした場合について説明します。

すべて同じファイルシステム上にあるOverlayFSの場合

xino機能では特殊ケースが存在します。特殊ケースといっても、ここまでの実行例はすべて特殊ケースで実行していました。

OverlayFSの上層、下層が同じファイルシステム上にある場合、xino機能は**暗黙で有効**になります。

この場合、OverlayFSは次のように動きます。

・すべてのオブジェクトは、OverlayFSからst_devを報告する

・すべてのオブジェクトは、基礎となるファイルシステムからst_inoを報告する

st_devについては、マウントしたOverlayFSから値を返すのは自然な動きです。

st_inoは、基礎となるファイルシステムからst_inoを報告します。これは、下層もしくは上層のオブジェクトのどちらかのst_inoが報告されます。これは、オブジェクトが上層・下層両方に存在する場合、下層にだけ存在する場合、上層にだけ存在する場合、ファイルが変更された場合などで、どのオブジェクトのst_inoが返るかが異なります。

下層側または上層側からst_inoを返すと、同じ下層・上層を使っている限りは、st_inoが永続的になります。

```
$ mkdir -p lower upper overlay work
$ touch lower/testfile upper/testfile lower/lowerfile upper/upperfile
$ sudo mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay
overlay
```

上のように、オブジェクトが下層・上層の両方、下層のみ、上層のみにある状態で、OverlayFSとしてマウントします。

この状態で、OverlayFS、下層、上層それぞれで、ファイルのst_inoを調べてみましょう。

次の実行例でstat -c %iと実行しているのは、iノード番号のみを得るためです。

```
$ stat -c %i overlay/lowerfile lower/lowerfile （下層のみにファイルが存在する場合）
393326 （overlayのst_ino）
393326 （lowerdirのst_ino）
$ stat -c %i overlay/testfile lower/testfile upper/testfile
（下層・上層両方にファイルが存在する場合）
393325 （overlayのst_ino）
393324 （lowerdirのst_ino）
393325 （upperdirのst_ino）
$ stat -c %i overlay/upperfile upper/upperfile （上層にのみファイルが存在する場合）
393328 （overlayのst_ino）
393328 （upperdirのst_ino）
```

この結果を見ると、下層側のファイルのst_inoが返っている場合と、上層側のファイルのst_inoが返っている場合があることがわかります。いずれにせよ、OverlayFS上から問い合わせても、上層、下層ディレクトリーが存在する、基礎となるファイルシステムの値がそのまま返っていることがわかります。

st_devの値を見てみましょう。statコマンドでst_devだけを得るにはstat -c %dと実行します。上層と下層のディレクトリーは同じファイルシステム上にあるので、同じ値が返っています。そして、OverlayFSはいずれも、上層、下層のディレクトリーとは異なる、47という同じ値が返っています。

```
$ stat -c %d overlay/lowerfile lower/lowerfile
47
64768
$ stat -c %d overlay/testfile lower/testfile upper/testfile
47
64768
64768
$ stat -c %d overlay/upperfile upper/upperfile
47
64768
```

xino無効の場合（上層と下層が異なるファイルシステムにある場合）

xino機能が無効な場合で、上層と下層が異なるファイルシステム上にある場合、OverlayFSから返るst_inoは永続的ではない可能性があり、st_devが同じOverlayFS上のオブジェクトなのに異なる可能性があります。また、readdir(3)が返す値も通常のファイルシステムとは異なる可能性があります。

xino有効の場合（上層と下層が異なるファイルシステムにある場合）

xino機能を有効にすると、上層と下層が異なるファイルシステムにあっても、上層と下層が同じファイルシステム上にあるときと同じように動作します。

ただし、まったく同じように動作するわけではありません。st_inoについては、ファイルシステムごとに一意となるID（FSID）と、基礎となるファイルシステムのst_inoから、一意となる識別子を生成します。これにより、st_inoが永続的な値になります。

このような一意な値を得るには条件があります。64ビットシステムでは、iノード番号（st_ino）は64ビットです。しかし、通常ファイルシステムは、64ビットあるiノード番号の上位ビットを使用しません。この通常使用されない領域に、先ほど紹介したFSIDを格納します。下位のビットには通常のst_inoの値を格納します。

もし、基礎となるファイルシステムが、st_inoでxinoがFSIDを格納する上位のビットを使っている場合は、OverlayFSはxino機能を使えません。

表4.9のxinoオプションの値として"auto"を指定した場合、マウント時に、ここまでで説明したような、xino機能を有効にできる条件が満たされているかをチェックします。そして、満たされている場合、有効にします。

また、xino機能が有効なときに、xino機能で使う上位ビットを使うiノードが現れた場合は、そのiノードについてはxino機能が無効になります。

このように上位ビットを使いますので、次の実行例のようにxino機能を有効にした場合、st_inoとして大きな値が返ってきます。

```
$ mkdir -p lower /data/upper overlay /data/work
（/dataはlowerと別のファイルシステムにある）
$ touch lower/lowerfile
$ sudo mount -t overlay -o lowerdir=lower,upperdir=/data/upper,workdir=/data/wor
k,xino=on overlay overlay
$ stat -c %i overlay/lowerfile lower/lowerfile
8590327874
393282
```

4.13　metacopy機能

OverlayFSではここまでで説明したとおり、下層側ディレクトリーにだけ存在するオブジェクトに変更が加えられたとき、上層側ディレクトリーへオブジェクトがコピーされます。

これは、ファイルの中身に変更がなく、パーミッションや所有者などのオブジェクトのメタデータのみを変更した場合でも同様です。

巨大なサイズのファイルの、パーミッションのみを変更したような場合でも、下層側ディレクトリーから上層側ディレクトリーへファイルがコピーされます。このようなケースでは、パフォーマ

ンスの問題を起こす可能性があります。

このような問題を解決するために、4.19カーネルで**metacopy機能**が実装されました。

metacopy機能は、メタデータのみが変更された場合、メタデータのみを上層側ディレクトリーにコピーします。そして、実際にファイルへ書き込みされるときに、上層へファイルデータがコピーされます。

metacopy機能の制御

この機能を使うには、いくつか方法があります。

・カーネルビルド時のオプション
・カーネルモジュールロード時のオプション
・マウント時のオプション

カーネルビルド時のオプションは、表4.11のとおりです。

表4.11: metacopy機能に関連するカーネルオプション

カーネルオプション	機能
CONFIG_OVERLAY_FS_METACOPY	metacopy機能をデフォルトで有効にする

カーネルビルド時のオプションで有効にしても、カーネルモジュールロード時やマウント時のオプションで無効にできます。

カーネルモジュールをロードするときのオプションとマウントするときのオプションは、表4.12のように同じです。

表4.12: metacopy機能を使うためのモジュール・マウントオプション

オプション	機能	指定できる値
metacopy	metacopy機能の有効・無効を切り替える	on/off

metacopy機能を有効にする場合、機能が競合するため、redirect_dirオプションの指定にoff、nofollow、followを指定できません[10]。また、nfs_export機能は有効にできません[11]。

metacopy機能はセキュリティー上の理由から、信頼できない上層、下層ディレクトリーでは有効にしないようにと、カーネル付属文書には書かれています[12]。

metacopy機能が無効の場合

デフォルトの動きである、metacopy機能が無効の場合に、メタデータのみを変更した場合の動きを確認してみましょう。

10."redirect_dir=follow"の指定は、同時に upperdir が指定されている場合のみ、"metacopy=on"と競合します。
11.nfs_export機能は本書では説明しません。
12.https://docs.kernel.org/filesystems/overlayfs.html

```
$ mkdir -p lower upper overlay work
$ echo test > lower/testfile
（下層側に文字列が書かれたファイルを置く）
$ sudo mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay
overlay
（OverlayFSマウント）
$ tree .
.
├── lower
│   └── testfile
├── overlay
│   └── testfile
├── upper
└── work
    └── work  [error opening dir]

5 directories, 2 files
```

上のようにディレクトリーを作成し、下層側ディレクトリーにだけ文字列が書き込まれたファイルを置き、OverlayFSでマウントします。

マウントしたディレクトリーで、このファイルのパーミッションを変更します。

```
$ chmod 600 overlay/testfile
（chmodでパーミッションを変更）
$ ls -l upper/testfile
-rw------- 1 tenforward tenforward 5 Apr 14 13:49 upper/testfile
$ cat upper/testfile
test
```

下層側にだけ存在するファイルに変更が加わりましたので、上層側ディレクトリーへファイルがコピーされます。ファイルの中身も当然同じです。

metacopy機能が有効の場合

metacopy機能を有効にした場合の動きを確認してみます。ディレクトリーやファイルの構成は同じです。

```
$ tree .
.
├── lower
│   └── testfile
```

```
├── overlay
├── upper
└── work

4 directories, 1 file
$ ls -l lower/testfile
-rw-rw-r-- 1 tenforward tenforward 5 Apr 14 13:38 lower/testfile
```

metacopy=onを指定して、OverlayFSをマウントします。OverlayFSマウントされたディレクトリー上で、ファイルのパーミッションを変更します。

```
$ sudo mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work,metacopy=on
overlay overlay
（metacopy=onでOverlayFSマウント）
$ chmod 600 overlay/testfile
（パーミッションを変更）
```

これで、下層側から上層側ディレクトリーへ、ファイルがコピーされます。

```
$ ls -l overlay/testfile
-rw------- 1 tenforward tenforward 5 Apr 14 13:38 overlay/testfile
$ ls -l lower/testfile
-rw-rw-r-- 1 tenforward tenforward 5 Apr 14 13:38 lower/testfile
$ ls -l upper/testfile
-rw------- 1 tenforward tenforward 5 Apr 14 13:38 upper/testfile
```

上のように、下層側のファイルは元のパーミッションのまま、上層側とOverlayFS上では、変更後のパーミッションで見えます。ファイルサイズはいずれも"5"で変わっていません。

ところが、上層側ディレクトリーでファイルの中身を確認してみると、次のように中身は空です。

```
$ cat upper/testfile
$
（上層ディレクトリーのファイルの中身は空）
```

ここで、OverlayFS上から、ファイルの中身を確認してみましょう。

```
$ cat overlay/testfile
test
```

上のように、下層側のファイルの中身と同じです。

ここでもう一度、上層側からファイルの中身を確認してみます。

```
$ cat upper/testfile
$
（OverlayFSのファイルを読んでも上層側ファイルの中身は空）
```

先ほどの確認と変わらず、上層側のファイルの中身は空です。つまり、**ファイルを読んだだけでは、ファイルの中身が上層側にコピーされることはない**ということがわかります。

それでは、OverlayFS上からこのファイルの中身を書き換えてみます。

```
$ echo test >> overlay/testfile
（OverlayFS上のファイルの中身を書き換える）
$ cat overlay/testfile
test
test
$ cat upper/testfile
test
test
（OverlayFS上のファイルの中身とともに、上層側のファイルの中身も書き換わっている）
```

上記のように、OverlayFS上のファイルの中身とともに、上層側のファイルの中身も書き換え後の内容に変わっています。

つまり、metacopy機能を使うと、OverlayFS上から**ファイルに書き込んだときに、下層側ファイルの中身が上層側ファイルにコピーされる**ということがわかります。

このように、ファイルのメタデータのみを変更したときは、メタデータのみを上層側にコピーし、ファイルにデータが書き込まれるときにデータをコピーする機能がmetacopy機能です。

4.14　User Namespaceとマウント操作

User Namespaceは、Linuxに実装されているNamespace機能のひとつで、Namespace内では特権ユーザー（root）、Namespace外では一般ユーザーというUID/GIDのマッピングができる機能です。

このようにマッピングすることにより、一般ユーザー権限でコンテナを起動でき、起動したコンテナ内ではroot権限でプロセスを起動できます。最近では、このUser Namespaceを使って起動されたコンテナをrootlessコンテナという風に呼んだりします。

User Namespaceでは、Namespace内では特権を持つユーザーであっても、Namespaceの外では特権を持たないユーザーとして処理されています。このとき、コンテナ内でroot権限が与えられているといっても、コンテナ外のrootとまったく同じ権限が与えられているわけではありません。User Namespace内のrootが実行できる操作は、Namespace外のrootが実行できる操作よりも少ないです。

User Namespace内のrootが制限されている操作のひとつに、ファイルシステムのマウントがあります。ファイルシステムのマウントについては、コンテナ内で自由にマウント操作ができるとホストや他のコンテナに影響を与える可能性があるため、**一般的には**User Namespace内（非特権）ではマウント操作はできません[13]。

User Namespaceについての詳細は、本シリーズ第1巻「Linux Container Book」、第3巻「Linux Container Book 3」をご覧ください。

非特権マウントのための定義

先に書いたように、User Namespace内からはファイルシステムをマウントできません。しかし、一部のファイルシステムは、User Namespace内でマウントできます。たとえば、コンテナ内で/procやtmpfsなどをマウントする操作は普通に行われている操作です。

カーネルコードで、User Namespace内でファイルシステムをマウントできるようにするには、ファイルシステムの定義（struct file_system_typeの定義）でフラグを指定します。このためのファイルシステムに定義する定数が、6.8カーネルではinclude/linux/fs.hの2444行目付近に定義されています[14]。

```
 2444 struct file_system_type {
 2445     const char *name;
 2446     int fs_flags;
 2447 #define FS_REQUIRES_DEV        1
 2448 #define FS_BINARY_MOUNTDATA    2
 2449 #define FS_HAS_SUBTYPE         4
 2450 #define FS_USERNS_MOUNT        8    /* Can be mounted by userns root */
   :（略）
```

このFS_USERNS_MOUNTが、User Namespace内からファイルシステムをマウントできるようにするためのフラグ値です。ファイルシステムを実装する際に、この値をfile_system_type構造体のfs_flagsに設定すると、コメントにあるようにUser Namespace内のrootが、そのファイルシステムをマウントできるようになります。

OverlayFSは5.11カーネルで、次のようなパッチでFS_USERNS_MOUNTがfs_flagsに設定され、User Namespace内でマウントできるようになりました。

```
--- a/fs/overlayfs/super.c
+++ b/fs/overlayfs/super.c
@@ -2096,6 +2096,7 @@ static struct dentry *ovl_mount（struct file_system_type
*fs_type, int flags,
```

13.「一般的には」と書いたのは、第3巻で紹介したSeccomp notify機能を使い、コンテナマネージャーでファイルシステムのマウントを許可する実装が可能になっているためです。詳しくは第3巻「セキュリティ編」をご覧ください。

14.https://elixir.bootlin.com/linux/v6.8/source/include/linux/fs.h

```
 static struct file_system_type ovl_fs_type = {
    .owner      = THIS_MODULE,
    .name       = "overlay",
+   .fs_flags   = FS_USERNS_MOUNT,
    .mount      = ovl_mount,
    .kill_sb    = kill_anon_super,
 };
```

もちろん、上記のようにフラグを設定しただけで、OverlayFS全体の機能が安全にUser Namespace内から利用できるわけではありません。OverlayFSを非特権でマウントできるようにするために、さまざまな部分に変更が加えられています[15]

4.15　非特権OverlayFSマウント

それでは、非特権でのOverlayFSを試してみましょう。

「非特権」といっても、先に説明したとおり「User Namespace 内のrootがマウントできる」ということですので、unshareコマンドでUser Namespaceを作成して試します（いずれにせよmountコマンドはrootでないと実行が失敗するようになっています）。

ただ、ここでUser Namespaceだけを作っても、マウントは失敗します。

```
$ unshare --user --map-root-user
（User Namespaceのみを作成）
# mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay overlay
mount: /home/karma/tmp/overlay: permission denied. （失敗した）
```

これは、User Namespaceを作るだけでなく、Mount Namespaceも元のNamespaceと独立している必要があるためです。

そこで次の例では、unshareコマンドに--mountも指定してUser NamespaceとMount Namespaceを作成しています。--map-root-userは、unshareを実行するユーザーと、User Namespace内のrootをマッピングするオプションです。次の例では、元のUser Namespace内のUID:1000のユーザーと、作成するUser Namespace内のUID:0をマッピングするということです。

テスト前に準備として、これまでのように下層側と上層側のディレクトリー配下に、ファイルやディレクトリーを作成します。

```
$ tree -n lower upper
lower
├── testdir_lower
```

15.5.11 カーネルで User Namespace 内でマウントできるようにするパッチは 10 個ほどのパッチから構成されていました。https://lwn.net/Articles/839210/ それ以前のバージョンのカーネルでも必要な変更が行われているようです。

```
|    └── testfile
└── testfile_lower
upper
├── testdir_upper
|    └── testfile
└── testfile_upper

1 directory, 2 files
```

それではマウントしてみましょう。

```
$ id -u （現在のユーザーは UID:1000）
1000
$ unshare --user --map-root-user --mount （User NamespaceとMount Namespaceを作成する）
# id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
（User Namespaceが作られ、その中のrootユーザーになっている）
# mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay overlay
（マウントする）
# mount -l | grep overlay
overlay on /home/tenforward/tmp/overlay type overlay (rw,relatime,lowerdir=lower,upperdir=upper,workdir=work)
（マウントの確認）
# tree overlay/
overlay/
├── testdir_lower
|    └── testfile
├── testdir_upper
|    └── testfile
├── testfile_lower
└── testfile_upper

2 directories, 4 files
（重ね合わせた状態でマウントできている）
```

マウントが成功し、きちんと重ね合わされています。マウントしたディレクトリーで所有権を確認します。

```
# ls -l overlay/
total 8
drwxrwxr-x 2 root root 4096 Mar 21 13:21 testdir_lower
drwxrwxr-x 2 root root 4096 Mar 21 13:22 testdir_upper
-rw-rw-r-- 1 root root    0 Mar 21 13:16 testfile_lower
-rw-rw-r-- 1 root root    0 Mar 21 13:17 testfile_upper
```

これらのファイルは、元のUser NamespaceのUIDが1000のユーザー権限で作成しましたので、User Namespace内でマウントしてもマッピング先のユーザー（UID: 0=root）の所有権になっています[16]。

5.11より前のカーネルでの実行例

比較のために、5.11より前のバージョンのカーネルでは、非特権マウントができないことも確認しておきましょう。次の実行例は、5.2カーネルの環境で試しています。

```
$ uname -r
5.2.1-plamo64
$ id -u
1000
$ unshare --user --map-root-user --mount
# mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay overlay
mount: /home/karma/tmp/overlay: permission denied.
```

同様に実行すると、失敗しました。

4.16　User NamespaceとOverlayFSの機能

ここまでで、User Namespace内でOverlayFSがマウントできることを紹介しました。

ところでOverlayFSでは、拡張属性を設定してオプション機能を使う場合、「4.10 opaque（不透明）ディレクトリー機能」などで紹介したとおり、拡張属性クラスとしてtrustedを使います。

「4.9 拡張属性」で紹介したように、trustedという拡張属性クラスがついた拡張属性は、CAP_SYS_ADMINという、かなり広い特権を持っているプロセスからしかアクセスできません。

また、User Namespace内のrootでは、拡張属性クラスtrustedを使用できません。一般ユーザー権限で作成できるUser Namespaceから、trustedで始まる拡張属性にアクセスできると、セキュリティー上の問題となるためです。

先に「4.9 拡張属性」で説明したように、拡張属性には、自由に定義して使える拡張属性クラスとしてuserが存在します。userであれば、一般ユーザーから自由に設定、参照できます。

16. このあたりの仕組みは本シリーズ第１巻、第３巻をご覧ください

そこで、OverlayFSでは非特権でマウントする際に、拡張属性クラスとしてuserを使って機能の設定ができるようになっています。

OverlayFSでuserで始まる拡張属性を使いたい場合、OverlayFSのマウントオプションとしてuserxattrを指定してマウントします。userxattrが指定されてマウントされている場合、trusted.overlayではなくuser.overlayを使うようになります。

非特権の場合の拡張ファイル属性

この拡張ファイル属性を設定した場合の動きを、「4.10 opaque（不透明）ディレクトリー機能」で説明したopaqueディレクトリー機能を使って説明しましょう。

opaqueディレクトリーを使いたい場合、拡張ファイル属性trusted.overlay.opaqueに"y"という値を設定しました。非特権OverlayFSの場合は、この代わりにuser.overlay.opaqueを使います。

比較のために、まずはopaque指定なしでマウントしてみましょう。

```
$ mkdir lower upper work overlay
$ mkdir {lower,upper}/opaquetest
$ touch lower/opaquetest/testfile_lower upper/opaquetest/testfile_upper
$ unshare --user --map-root-user --mount -- /bin/bash
# mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay overlay
# tree overlay/
overlay/
└── opaquetest
    ├── testfile_lower
    └── testfile_upper

1 directory, 2 files
```

opaque指定をしないと、これまで説明したように、上層と下層が重なって見えます。

userxattrオプションを使わない場合

まずは、必要な拡張属性user.overlay.opaqueをyに設定し、マウントオプションは指定せずにOverlayFSマウントを行い、動きを見てみましょう。当然、期待する動きはしないはずです。

```
$ unshare --user --map-root-user --mount -- /bin/bash
（User NamespaceとMount Namespaceを作成）
# setfattr -n "user.overlay.opaque" -v "y" upper/opaquetest/
（拡張属性user.overlay.opaqueをyに設定）
# getfattr -n "user.overlay.opaque" upper/opaquetest/
（設定の確認）
# file: upper/opaquetest/
```

```
user.overlay.opaque="y"
（設定されている）
# mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay overlay
# tree overlay/ （拡張ファイル属性を設定したものの、特にOverlayFSの動きに変化はない）
overlay/
└── opaquetest
    ├── testfile_lower
    └── testfile_upper

1 directory, 2 files
# umount overlay
```

上記のように、user.overlay.opaque属性を設定しても、普通にOverlayFSマウントを行ったときと動きに変化はありません。

userxattrオプションを追加してマウントした場合

それでは、OverlayFSマウントを行う際に、userxattrオプションを追加してみましょう。

```
# getfattr -n "user.overlay.opaque"  upper/opaquetest/ （拡張属性が設定されているのを確認）
# file: upper/opaquetest/
user.overlay.opaque="y"

# mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work,userxattr
overlay overlay （userxattrオプションを指定してマウント）
# tree overlay/
overlay/
└── opaquetest
    └── testfile_upper

1 directory, 1 file
（overlayディレクトリーを見ると、上層側に置いたファイルしか見えない）
# tree upper lower/
upper
└── opaquetest
    └── testfile_upper
lower/
└── opaquetest
    └── testfile_lower
```

```
1 directory, 1 file
（念のため確認すると上層、下層ともにファイルは存在する）
```

このように、下層側ディレクトリーにファイルが存在するにも関わらず、上層側のファイルしか見えません。

拡張属性user.overlay.opaqueを設定し、マウント時にuserxattrオプションを指定したことで、OverlayFSがuserではじまる拡張属性を参照し、opaqueディレクトリー機能が働いていることがわかります。

非特権マウントの場合に使えない機能

以上のように、非特権マウントする場合はtrusted.overlayの代わりにuser.overlayを使ってOverlayFS独自の機能を実現します。

しかし非特権の場合、特権を持っているケースと同じように機能を使うと、特権の取得につながるなど危険なケースがあります。このような機能については、非特権の場合には使えないようになっています。

userxattrと同時に使えない機能は、

・redirect_dir

・metacopy

です。これらの機能を使うと、マウントは失敗します。カーネルログにもエラーが出力されます。

```
[170311.770537] overlayfs: conflicting options: userxattr,redirect_dir=on
[170341.070243] overlayfs: conflicting options: userxattr,metacopy=on
```

4.17　OverlayFSを使ったコンテナの作成とイメージファイルの作成

ここまでで、OverlayFSの機能について説明しました。ここまでご覧になれば、大体どのような感じでコンテナがレイヤー構造のファイルシステムを使うのかが理解できているかと思います。

そこで、ここではその理解を確認するために、実際に簡易的にコンテナを作成し、OverlayFSがどのように使われるのかを見ていきましょう。

ここでは、特定のコンテナエンジンのストレージドライバーの動きを追うわけではありません。実際のドライバーは、もっと複雑なことをしているでしょう。おおむねこのような感じで使っているのだということを、感覚的に理解してください[17]。

17.Dockerのoverlay2ドライバーについては公式ドキュメント（https://docs.docker.com/storage/storagedriver/overlayfs-driver/）に詳しいです。

図4.11: 作成するrsyncコンテナ

図4.11のような非常にシンプルな2層構造のコンテナを考えます。

・Ubuntu 22.04コンテナイメージが存在する

・そこにrsyncパッケージを入れた"rsyncコンテナ"を作成する

Dockerfileで書くと、次のような感じになるでしょうか。

```
FROM ubuntu:22.04
RUN apt update && apt install -y rsync && apt clean && rm -rf
/var/lib/apt/lists/*
```

このようなコンテナの例を使って、「2.4 レイヤー構造のファイルシステムとユニオンファイルシステム」で説明したレイヤー構造が、OverlayFSを使ったコンテナでどのように構成されるかを説明します。そして、そのできあがったコンテナから、レイヤーをコンテナイメージファイルとして保存し、それを再現する方法を実際に見てみましょう。

図4.11のような、Ubuntuイメージ+rsyncパッケージという構成の場合、次のような構成になります。

・下層側ディレクトリーにUbuntu 22.04をインストール

・上層側ディレクトリーにrsyncパッケージをインストール

この構成は、次のような手順で作成できます。

1．Ubuntu 22.04をインストールしたディレクトリーを下層側ディレクトリーとして、上層側ディレクトリーは空の状態でOverlayFSマウント

2．その状態で、OverlayFSでマウントされた領域からrsyncパッケージをインストール

このようにインストールすれば、下層側は読み取り専用ですので、変更はすべて上層側ディレクトリーに対して行われます。すると上層側ディレクトリーには、rsyncパッケージをインストールしたときにインストールされるオブジェクトのみが格納されます。

このように、それぞれの層に、それぞれをインストールした際にインストールされるファイルやディレクトリーが格納されます。

このようにOverlayFSをマウントし、Namespaceを作成するなどの必要な処理をすれば、「rsyncコンテナ」が完成です。これはみなさんも容易にイメージできるのではないでしょうか。

そして、コンテナのイメージファイルを作るときは、

・下層側ディレクトリー、上層側ディレクトリーをそれぞれtar+gzipなどのアーカイブファイルで固める
・上層側のアーカイブファイルに、「Ubuntu 22.04イメージが下層である」という情報を含める

とします。

このようにしておくと、次のような手順で「rsyncコンテナ」を他の環境に作成でき、最初に作成したコンテナと同じ「rsyncコンテナ」がどこでも再現できます。

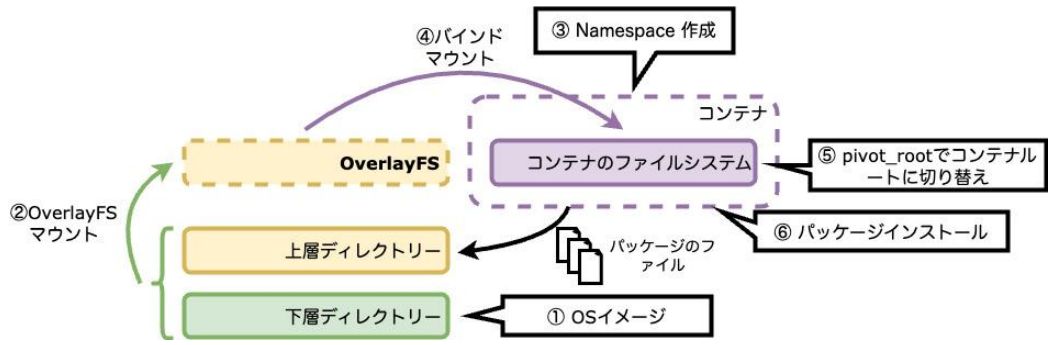
1．rsyncコンテナ（上層側ディレクトリー）のアーカイブファイルを取得し、上層側ディレクトリーに展開
2．rsyncコンテナのアーカイブに含まれる下層情報を参照し、Ubuntu 22.04（下層側ディレクトリー）のアーカイブファイルを取得し、下層側ディレクトリーに展開
3．OverlayFSで重ね合わせてコンテナを作成

それでは、実際にOverlayFSを使ったコンテナを作っていきましょう。実行しているコマンドの詳細は、本シリーズの第1巻を参照してください。

次のような手順で確認します。

1．下層側ディレクトリーに、`debootstrap`コマンドでOSイメージを作成
2．上層側ディレクトリーとして、空のディレクトリーを使いOverlayFSマウント
3．`unshare`コマンドを使い、Mount NamespaceとPID Namespaceのみを分離してシェルを起動[18]
4．OverlayFSマウントしたディレクトリーを、コンテナ用のディレクトリーにバインドマウント[19]
5．`pivot_root`コマンドを使い、ルートを、OverlayFSでマウントされたディレクトリーにバインドマウントした、コンテナ用ファイルシステムへ切り替え（コンテナの完成）
6．作成したコンテナ上でパッケージインストール
7．パッケージがインストールされたレイヤーを保存
8．別ディレクトリーにレイヤーを展開して、OverlayFSマウントし、コンテナを作成
9．レイヤーが正しくコンテナ上で再現されていることを確認

図4.12: OverlayFSを使ったコンテナの作成



18. これ以外のNamespaceを作成して分離しても問題ありません。必要最低限のNamespaceだけを作っています。

19.OverlayFSマウントしていますので、バインドマウントする必要はありません。実際のコンテナエンジンの動きに近づけるようにバインドマウントしています

ホスト側の準備

まずは、必要なディレクトリーを作成します。ここでは/var/lib/container以下に、これまでの実行例と同様にlower,upper,overlay,workディレクトリーを作成しました。ここまで使ったディレクトリーの他に、OverlayFSマウントしたoverlayディレクトリーをバインドマウントし、コンテナのルートとするためにmycontainerディレクトリーを作成します。

そして、コンテナ用のOSイメージを作成するために、debootstrapコマンドをインストールしておきます。

```
$ sudo mkdir -p /var/lib/container/{lower,upper,overlay,work,mycontainer}
$ sudo apt install debootstrap
```

ベースOSイメージのインストール

次に下層側ディレクトリーに、OSイメージをインストールします。これが「ベースOSイメージ」となります。下層側には、OSイメージのツリーがインストールされます。ここでは、Ubuntu 22.04 LTS（jammy）をインストールします。Ubuntuの必須パッケージと、aptがインストールされた状態の、minbaseを選択しています。

上層側ディレクトリーは空のままです。

```
$ cd /var/lib/container/lower
$ sudo debootstrap --variant=minbase --arch=amd64 jammy .
http://ftp.jaist.ac.jp/pub/Linux/ubuntu/
（下層側ディレクトリーにUbuntu Jammyをインストール）
$ ls /var/lib/container/upper
（上層側ディレクトリーは空のまま）
```

OverlayFSマウント

下層側、上層側の準備は整いましたので、OverlayFSマウントしましょう。このマウントは、コンテナのファイルシステムにする予定ですので、コンテナのルートがわかるようにoverlayディレクトリー直下にCONTAINERROOTというファイルを置いておきます。同時にコンテナホストのルートであることがわかるように、ホストのルートにはHOSTROOTというファイルを置いておきます。

```
$ cd /var/lib/container
$ sudo mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=work overlay
overlay
（OverlayFSマウント）
$ sudo touch overlay/CONTAINERROOT
$ sudo touch /HOSTROOT
```

```
（目印となるファイルを置いておく）
```

コンテナ（Namespace）の作成

ファイルシステムの準備はできましたので、コンテナ（Namespace）を作成します。ここでは、必要であるMount NamespaceとPID Namespaceのみを作成しました。

```
$ sudo unshare --mount --fork --pid -- /bin/bash
（Mount, PID Namespaceの作成）
# mount --make-rprivate /
（作成したNamespaceをprivateに設定）
```

バインドマウント

Namespaceができましたので、OverlayFSマウントしたoverlayを、コンテナルートとしたいmycontainerディレクトリーにバインドマウントします。

```
# mount --bind /var/lib/container/overlay /var/lib/container/mycontainer
（overlayをmycontainerにバインドマウント）
# cd /var/lib/container/mycontainer/
# ls
CONTAINERROOT  boot  etc   lib    lib64   media  opt   root  sbin  sys  usr
bin            dev   home  lib32  libx32  mnt    proc  run   srv   tmp  var
```

バインドマウントしたmycontainer以下は、コンテナのルートとしたいファイルシステムになっていることが確認できます。

このあと行う操作がきちんと動くように、mycontainer以下でprocファイルシステムをマウントしておきます。

```
# mount -t proc -o rw,nosuid,nodev,noexec,relatime proc proc
```

pivot_rootでコンテナのファイルシステムを作成

それでは、いよいよpivot_rootコマンドで、コンテナのファイルシステムに切り替えましょう。mycontainerディレクトリーがルートになります。

pivot_rootでは、pivot_root実行前のルートをマウントするディレクトリーが必要ですので、oldディレクトリーを作成します。pivot_root実行後は、/oldにマウントされた元のルートファイルシステムをumountします。

```
# mkdir old
（pivot_rootで必要な現在のルートをマウントするディレクトリーを作成）
# pivot_root . old
（pivot_rootの実行。oldに現在のルートをマウントする）
# ls /
CONTAINERROOT  boot  etc   lib    lib64   media  old  proc  run   srv  tmp  var
bin            dev   home  lib32  libx32  mnt    opt  root  sbin  sys  usr
（ルートがmycontainer直下になっている）
# ls /old
HOSTROOT  boot  dev  home  lib32  libx32      media  opt   root  sbin  srv  tmp
var
bin       data  etc  lib   lib64  lost+found  mnt    proc  run   snap  sys  usr
（/oldにはホストのルートがマウントされている）
# umount -l /old
（/oldをアンマウント）
# mount -l
overlay on / type overlay (rw,relatime,lowerdir=lower,upperdir=upper,workdir=work)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
（OverlayFSとprocのみがマウントされており、コンテナのファイルシステムが完成）
```

元のルートファイルシステムをumountしたあとにマウント一覧を確認すると、pivot_rootで変更したルート（OverlayFSマウントされている）と、事前にマウントしていた/procのみがマウントされていることが確認できます。

コンテナ内でパッケージインストール

この時点では、コンテナのファイルシステムに行った変更としては、oldディレクトリーを作成しただけです。ここで、コンテナにパッケージとしてrsyncパッケージをインストールしてみます。aptのキャッシュなどは削除します。

```
# which rsync
（rsyncコマンドはない）
# apt update && apt install -y rsync && apt clean && rm -rf /var/lib/apt/lists/*
（rsyncをインストール）
# which rsync
/usr/bin/rsync
（インストールされた）
```

これで、Ubuntu Jammy環境にrsyncをインストールした「rsyncコンテナ」が完成しました。

上層側ディレクトリーの確認

さて、下層側にベースOSのイメージを作成し、OverlayFSをマウントした状態で、rsyncパッケージをインストールしました。

ホスト側から上層側ディレクトリーであるupperを確認してみましょう。

```
$ cd /var/lib/container/upper/
$ sudo find . -type f -o -type d
./CONTAINERROOT
  :（略）
./usr/bin
./usr/bin/rsync-ssl
./usr/bin/rsync
./usr/lib
./usr/lib/systemd
./usr/lib/systemd/system
./usr/lib/systemd/system/rsync.service
./usr/lib/x86_64-linux-gnu
./usr/lib/x86_64-linux-gnu/libpopt.so.0.0.1
./etc
./etc/ld.so.cache
./etc/init.d
./etc/init.d/rsync
./etc/default
./etc/default/rsync
  :（略）
```

確認のために作成したファイル（CONTAINERROOT）や、作業のために作成したディレクトリー（/old）のほかは、rsyncパッケージを構成するファイルやディレクトリーのみが存在していることがわかるでしょう[20]。

ここで、キャラクターデバイスファイルだけをfindコマンドで検索してみると、upperには、apt installのあとに、"apt clean && rm -rf /var/lib/apt/lists/*"を実行して削除したオブジェクトのwhiteoutが存在していることが確認できます。

```
$ find . -type c | less
（キャラクターデバイスファイルのみを検索）
./var/lib/apt/lists/ftp.jaist.ac.jp_pub_Linux_ubuntu_dists_jammy_Release.gpg
./var/lib/apt/lists/ftp.jaist.ac.jp_pub_Linux_ubuntu_dists_jammy_Release
./var/lib/apt/lists/ftp.jaist.ac.jp_pub_Linux_ubuntu_dists_jammy_main_binary-amd
```

20.apt などに関連して作成されたファイルも存在します

```
64_Packages
./var/lib/apt/lists/partial
./var/lib/apt/lists/ftp.jaist.ac.jp_pub_Linux_ubuntu_dists_jammy_InRelease
./var/cache/apt/archives/libext2fs2_1.46.5-2ubuntu1_amd64.deb
./var/cache/apt/archives/hostname_3.23ubuntu2_amd64.deb
./var/cache/apt/archives/libssl3_3.0.2-0ubuntu1_amd64.deb
./var/cache/apt/archives/libkrb5support0_1.19.2-2_amd64.deb
./var/cache/apt/archives/libncursesw6_6.3-2_amd64.deb
  :（略）
（aptのキャッシュなどで削除したファイルのwhiteoutの存在を確認）
```

rsyncパッケージのレイヤーを作成

それでは、Ubuntu 22.04のベースOSイメージを使って、あとで同様のrsyncコンテナが作成できるようにしましょう。上層ディレクトリーにはrsync関連のファイルのみが存在するはずですので、upper以下のファイルだけをtarでアーカイブします。

```
$ cd /var/lib/container/upper
$ sudo tar zcvf ../rsync.tar.gz $(find . -type f)
（upper以下のファイルのみをtarアーカイブ）
```

これでrsync関連のファイルだけが含まれた、上層側ディレクトリー用のアーカイブが作成されたはずです。

作成したレイヤーイメージを作成しrsyncコンテナを作成

それでは、先ほど使ったUbuntu 22.04がインストールされた下層側ディレクトリーはそのまま使い、上層用には別のディレクトリー（/var/lib/container/rsync）を使って、OverlayFSマウントを行い、先ほどと同様の手順でコンテナを作成してみます。

```
$ sudo mkdir /var/lib/container/rsync
（新たな上層側ディレクトリーrsyncの作成）
$ cd /var/lib/container/rsync
$ sudo tar xvf ../rsync.tar.gz
（先に作成したアーカイブファイルをrsyncディレクトリー以下で展開）
$ sudo mount -t overlay -o lowerdir=lower,upperdir=rsync,workdir=work overlay
overlay
$ sudo unshare --mount --pid --fork -- /bin/bash
# mount --make-rprivate /
# mount --bind /var/lib/container/overlay /var/lib/container/mycontainer
```

```
# cd mycontainer/
# mount -t proc -o rw,nosuid,nodev,noexec,relatime proc proc
# mkdir old
# pivot_root . old
# umount -l /old
（先の手順と同じようにコンテナを作成）
```

これで、OverlayFSマウントを行い、コンテナが作成できたはずですので、コンテナ内でrsyncパッケージがインストールされていることを確認してみましょう。

```
# which rsync
/usr/bin/rsync
# dpkg -l | grep rsync
ii  rsync                  3.2.3-8ubuntu3                          amd64
fast, versatile, remote (and local) file-copying tool
```

確かに、rsyncパッケージがインストールされた状態が再現されています。

ここまでで説明したように、OverlayFSでマウントする際に指定した各層をそれぞれ保存してアーカイブファイルを作成し、コンテナのイメージファイルとします。

そして、コンテナのイメージファイルとして、それぞれのアーカイブファイルを取得し、下層側ディレクトリー、上層側ディレクトリーそれぞれで展開し、OverlayFSで重ね合わせることで、レイヤー構造のファイルシステムを使ったコンテナが作成できます。

下層側のベースOSイメージがすでにホスト上に存在するときは、上層用のrsyncレイヤーのアーカイブファイルのみ取得すればいいので、コンテナ作成にかかる時間やネットワーク帯域が削減できます。

ここまでで、OverlayFSを使ったコンテナの作成と、コンテナアーカイブ（コンテナイメージ）の作成、コンテナアーカイブからのコンテナの作成の例を紹介しました。

あとがき

本書は、技術評論社のオンラインメディアである"gihyo.jp"（https://gihyo.jp/）で、2014年から始めた連載「LXCで学ぶコンテナ入門 －軽量仮想化環境を実現する技術」から、OverlayFSをテーマで書いた回と、私の技術ブログのOverlayFSについて書いたエントリーをベースに、加筆や最新の情報への更新を行い作成した本です。gihyo.jp編集部と担当の小坂さんには、本書の出版について快諾いただきました。ありがとうございました。

そして今回、インプレスの山城様には「技術の泉シリーズ」からこの本を出版できる機会をいただきました。ありがとうございます。

本書は、技術の泉シリーズで出版されている『Linux Container Book』、『Linux Container Book 2』、『Linux Container Book 3』に続く第4巻です。

本書の元になった連載は2014年に始まっています。そして、本書の内容は、連載で2015年に書いた記事、ブログで2017年～2021年に書いた記事をベースにしています。連載やブログでの実行例は、執筆時点の環境で実行していました。そこで本書執筆にあたって、実行例は現在使われているディストリビューションの新しいバージョンでの実行例に更新しました。

また、連載やブログでは、OverlayFSの機能に絞った説明しかしていませんので、現在、コンテナのファイルシステムとして求められる機能や、それに対して、OverlayFSやその他のファイルシステムをどう当てていくのかについても説明しました。その部分や、OverlayFSの標準機能以外の機能については、今回の本に合わせて書き下ろしています。

本シリーズの内容自体はすべて、連載や私のブログなどですでに公開したコンテンツを元にしています。連載当時の情報からの更新をするとともに、色々なところに分散している情報をひとまとめにして、私が調べた成果を集大成としてまとめたいと思ったのが、このシリーズを執筆したきっかけです。

連載開始のころとは違い、今ではコンテナについての書籍やブログなどの記事による解説が多数存在しています。この本と同様の情報が色々なところで公開されており、そのようなさまざまな情報とこの本をあわせて、みなさまがコンテナの知識をつける助けになればうれしいです。

本書をきっかけに、Linuxカーネルに実装されているコンテナの基本的な機能に興味を持つ方が増えることを期待しています。

著者紹介

加藤 泰文（かとう やすふみ）

2009年頃にLinuxカーネルのcgroup機能に興味を持ち、以来Linuxのコンテナ関連の最新情報を追う。2013年から続く勉強会「コンテナ型仮想化の情報交換会」の開催や、lxc-jpプロジェクトでLXC/LXD方面の翻訳を行う。日本発のLinuxディストリビューション「Plamo Linux」のメンテナ。

◎本書スタッフ
アートディレクター/装丁：岡田章志＋GY
編集協力：深水央
ディレクター：栗原 翔
〈表紙イラスト〉
α（あるふぁ）
アニメーター出身の駆け出しイラストレーター。現在は、主に格闘ゲームのファンイベントのイラストなどを描いています。

技術の泉シリーズ・刊行によせて

技術者の知見のアウトプットである技術同人誌は、急速に認知度を高めています。インプレス NextPublishingは国内最大級の即売会「技術書典」（https://techbookfest.org/）で頒布された技術同人誌を底本とした商業書籍を2016年より刊行し、これらを中心とした『技術書典シリーズ』を展開してきました。2019年4月、より幅広い技術同人誌を対象とし、最新の知見を発信するために『技術の泉シリーズ』へリニューアルしました。今後は「技術書典」をはじめとした各種即売会や、勉強会・LT会などで頒布された技術同人誌を底本とした商業書籍を刊行し、技術同人誌の普及と発展に貢献することを目指します。エンジニアの“知の結晶”である技術同人誌の世界に、より多くの方が触れていただくきっかけになれば幸いです。

インプレス NextPublishing
技術の泉シリーズ　編集長　山城 敬

技術の泉シリーズ

Linux Container Book 4

2025年3月28日　初版発行Ver.1.0（PDF版）

著　者　　加藤 泰文
編集人　　山城 敬
企画・編集　合同会社技術の泉出版
発行人　　高橋 隆志
発　行　　インプレス NextPublishing
〒101-0051
東京都千代田区神田神保町一丁目105番地
https://nextpublishing.jp/
販　売　　株式会社インプレス
〒101-0051　東京都千代田区神田神保町一丁目105番地

ISBN978-4-295-60372-6