

Windowsユーザーのための

WSL

Ubuntu
26.04 LTS
対応!

で始める Linux 環境構築術

はじめてのLinux環境を
やさしく、わかりやすく!



WSLの導入から
初期設定までを
やさしく解説



開発環境・コンテナ・
各種ツールとの
連携もカバー



コマンド&Tipsで
実務にすぐ役立つ!



Windows



WSL



Linux



Docker



DevOps



Tips

Windows ユーザーのための WSLで始めるLinux環境構築術

VirtualTech Japan Inc.

まえがき

現在、サーバーサイドのあらゆる場所で、Linux が活用されています。そのためターゲットとなる Linux に関する知識と、その上で動作するツールへの習熟は避けては通れません。ところが Linux マシンを用意して本格的に学習を始めようとしても、なかなか長続きせず、気がついたら Windows しか使わないようになっていたという初学者も多いのではないのでしょうか。

現在の Windows 11 では、本物の Linux 環境をそのまま動かすことができます。これが Windows Subsystem for Linux (WSL) です。普段使いの Windows と Linux をシームレスに運用できる WSL は、学習においても、また実際の開発作業においても、非常に便利なツールです。本書では WSL を使い、Linux の基本的な使い方、便利な Linux 環境の構築方法、様々なテクニックや Tips などを紹介します。これから Linux に入門したい Windows ユーザーの方、そして Linux をさらに便利に活用したい方にとって、役立つ情報をお届けできると考えています。

なお本書は、ThinkIT の連載「Windows ユーザーのための WSL2 ではじめる Linux 環境構築術^{*1}」を Ubuntu 26.04 向けにリライトし、再構築したものととなります。

^{*1} <https://thinkit.co.jp/corner/11545>

目次

まえがき	ii
第 1 部 基礎知識編	1
第 1 章 WSL をインストールしよう	2
1.1 WSL とは	2
1.2 なぜ WSL なのか	2
1.3 WSL のインストール	3
1.4 WSL の起動方法	7
第 2 章 Windows Terminal を使いこなそう	9
2.1 ターミナルとは	9
2.2 Windows におけるターミナル	9
2.3 Windows Terminal の機能	11
2.4 Windows Terminal のカスタマイズ	15
2.5 プロファイルとは	17
第 3 章 Ubuntu のパッケージ管理を学ぼう	18
3.1 パッケージとは	18
3.2 パッケージ管理システムとは	18
3.3 CLI の基本操作	19
3.4 引数とオプション	21
3.5 APT を使いこなす	21
3.6 「apt」と「apt-get」の関係	28
3.7 apt update を省略するには	29
第 4 章 ユニバーサルパッケージシステム「Snap」を使ってみよう	30
4.1 Ubuntu のパッケージとソフトウェアのバージョン	30
4.2 Ubuntu のパッケージが抱える問題	32
4.3 ユニバーサルパッケージとは	32
4.4 Snap を使いこなす	32
第 5 章 Windows 上で Linux の GUI アプリを動かしてみよう	37
5.1 GUI アプリのインストールと起動	37
5.2 日本語フォントのインストール	38
5.3 アプリケーションの日本語化	39
5.4 WSL で日本語を入力するには	40

第 6 章	Linux の基礎コマンド群をマスターしよう	42
6.1	Coreutils とは	42
6.2	Ubuntu と Coreutils	42
6.3	コマンドのマニュアルを読むには	43
6.4	各コマンドに共通のオプション	43
6.5	ファイルを出力するコマンド	44
6.6	ファイルを分割するコマンド	46
6.7	ファイルの中身をチェックするコマンド	46
6.8	ファイルを編集/変換/ソートするコマンド	47
6.9	ファイルやディレクトリに関するコマンド	49
6.10	ファイルの属性を変更するコマンド	53
6.11	ユーザーとシステムに関するコマンド	54
6.12	プロセスに関するコマンド	57
第 7 章	シェルをカスタマイズして、コマンドラインを便利にしよう	59
7.1	Bash の設定ファイルと読み込み順	59
7.2	Bash をカスタマイズしよう	60
第 8 章	ターミナルマルチプレクサで、コマンドラインをさらに快適にしよう	66
8.1	ターミナルマルチプレクサとは	66
8.2	Byobu のインストール	66
8.3	Byobu の設定	66
8.4	Byobu の機能	69
第 9 章	さらに現代的な Linux コマンドを活用しよう	74
9.1	より現代的な Linux コマンドの登場	74
9.2	eza	74
9.3	zoxide	76
9.4	ripgrep	77
9.5	bat	78
9.6	fd	79
9.7	delta	80
9.8	dust	82
9.9	procs	83
9.10	bottom	84
9.11	hyperfine	85
9.12	hexyl	86
9.13	lnav	87
第 10 章	標準入力と標準出力について知ろう	90
10.1	標準入力と標準出力とは	90
10.2	リダイレクト	90
10.3	パイプ	93
10.4	リダイレクトにありがちな罠	94

第 11 章	CLI でもテキストエディタを使おう	95
11.1	テキストエディタの必要性	95
11.2	Ubuntu の標準エディタ「nano」	95
11.3	Linux の定番エディタ「Vim」	96
11.4	Vim と人気を二分する「Emacs」	98
11.5	Microsoft 製の新型エディタ「Microsoft Edit」	101
11.6	Ubuntu の標準エディタを変更する	102
11.7	エディタが存在しない時の対処法	104
第 12 章	ワンライナーで作業を劇的に効率化しよう	107
12.1	ワンライナーとは	107
12.2	今日から使えるワンライナーの例	107
12.3	WSL で使えるワンライナーの例	110
第 13 章	コマンドを並列実行して効率をアップしよう	113
13.1	コマンドの並列実行とは	113
13.2	GNU Parallel とは	113
13.3	GNU Parallel のインストール	113
13.4	GNU Parallel の基本的な使い方	113
13.5	GNU Parallel の具体的な活用シーン	114
13.6	GNU Parallel が効かない例	117
第 2 部	開発環境構築編	119
第 14 章	設定ファイルを管理しよう	120
14.1	dotfiles とは	120
14.2	chezmoi とは	120
14.3	chezmoi の使い方	121
14.4	システムワイドな設定ファイルを管理するには	124
第 15 章	バージョンマネージャーを使いこなそう	126
15.1	なぜバージョンマネージャーが必要なのか	126
15.2	そもそもバージョンマネージャーとは	126
15.3	asdf とは	127
15.4	asdf を使ってみよう	127
15.5	その他のバージョンマネージャー	131
第 16 章	プロジェクト単位で環境変数を管理しよう	133
16.1	環境変数の重要性	133
16.2	direnv とは	133
16.3	direnv の特徴	134
16.4	direnv のインストール	134
16.5	基本的な使い方	134
16.6	よくある使用例	135
16.7	想定される運用	137

第 17 章	Windows でも Docker を動かそう	138
17.1	Docker とは	138
17.2	Docker Desktop とは	138
17.3	Docker Desktop のインストールと起動	138
17.4	Docker コンテナを動かす	139
17.5	コマンドラインから Docker を使う	141
17.6	WSL 内の Linux から Docker Desktop を使う	142
17.7	Docker Desktop を使わない方法	144
第 18 章	Kubernetes クラスターを構築してみよう	145
18.1	コンテナオーケストレーションツールとは	145
18.2	Windows で Kubernetes を動かす方法	145
18.3	MicroK8s のインストール	146
18.4	コマンドにエイリアスを設定する	146
18.5	クラスターの情報を確認する	147
18.6	ノードの状態を確認する	147
18.7	サービスを動かす	147
18.8	アドオンを有効化する	149
第 3 部	WSL 応用編	152
第 19 章	WSL のカスタムディストリビューションを作ってみよう	153
19.1	カスタムディストリビューションとは	153
19.2	Ubuntu のルートファイルシステムを構築する	153
19.3	WSL 構成ファイルを作成する	155
19.4	初期化スクリプトの作成	156
19.5	プロファイルの作成とアイコンファイルの配置	157
19.6	その他のファイルの配置	158
19.7	ディストリビューションのアーカイブ	158
19.8	ディストリビューションのインストール	159
19.9	カスタムディストリビューションの起動	160
第 20 章	WSL 環境の初期セットアップを自動化してみよう	162
20.1	cloud-init とは	162
20.2	user-data とは	162
20.3	cloud-init で WSL を初期化する	164
20.4	バージョンごとに初期化プロセスを使い分ける	167
第 21 章	WSL 環境をまるごとバックアップしよう	169
21.1	WSL 環境をエクスポートする	169
21.2	エクスポートしたアーカイブをインポートする	170
21.3	インポート先のディレクトリはどうするか	171
21.4	エクスポートとインポートの実践的活用例	172
21.5	タスクスケジューラで WSL 環境を自動バックアップ	173

第 22 章	Windows ファイルシステムへのアクセスを高速化しよう	179
22.1	WSL1 と WSL2 の違い	179
22.2	WSL1 への切り替え方法	180
22.3	新規インストール時に WSL1 を指定する	181
22.4	ディスクアクセス速度を計測する	181
第 23 章	WSL 上のサーバーを公開してみよう	186
23.1	WSL1 と 2 におけるネットワークアーキテクチャの違い	186
23.2	WSL1 環境の用意	186
23.3	Streamlit のインストール	186
23.4	Streamlit で Web アプリを立ち上げる	186
23.5	初回起動時にアクセスを許可しなかった場合の対処	190
23.6	WSL2 に外部からアクセスする方法	191
第 24 章	WSL におけるデータレスキュー方法を知ろう	194
24.1	WSL にログイン不能になる時とは	194
24.2	デフォルトユーザーでログインできない時は	194
24.3	ディスクイメージを別環境にマウントする	196
あとがき		201
著者紹介		202

第 1 部

基礎知識編

第 1 章 WSL をインストールしよう

まずは Windows に WSL をインストールしましょう。本書では Windows 11 のバージョン 25H2 を前提に解説します。

1.1 WSL とは

Windows 11 には、**Windows Subsystem for Linux**、略して **WSL** と呼ばれる機能が搭載されています。これは Windows 上で、完全な Linux 環境を実行できる機能です。WSL を利用すれば、Bash などのシェルをはじめ、各種 Linux コマンド、さらには Linux 向けの GUI アプリを、Windows 上で直接実行できます。

実は Windows 上で Linux や Unix のツールを実行するための環境というのは、過去にも色々な実装が存在しました。ですがそれらはあくまでサードパーティのツールであったり、複雑なインストール手順が必要であるなど、気軽にお勧めできるものではありませんでした。それに対して WSL は、Microsoft が公式に提供しているプロダクトです。ストアから簡単にインストールが可能なため、導入までのハードルも、非常に低くなっています。WSL は、普段使いのデスクトップとしては Windows を必要としている人が、Linux のツールを使いたい場合の、最適な選択肢だと言えるでしょう。

WSL には、従来の **WSL1** と、より新しい **WSL2** の、ふたつのバージョンが存在します。このふたつの大きな違いは、その内部アーキテクチャにあります。WSL1 では、**LXCore** や **LXSS** と呼ばれるサブシステムが、Linux カーネルの機能をエミュレーションしていました。簡単に言えば Linux そのものが動いていたわけではなく、API を変換する互換レイヤーであったわけです。これに対して WSL2 では、本物の Linux カーネルが Windows 上で動作しています。そのためシステムコールに完全な互換性があり、パフォーマンスも大きく向上しています。そのため特別な理由がないのであれば、WSL2 を使うべきでしょう。現在、デフォルトの状態では WSL をインストールすると、WSL2 が利用されるようになっています。もちろん本書でも、特別な場合を除いて WSL2 を動かすことを前提に解説します。そのため WSL2 のことを指して WSL と表記することもあるため、あらかじめご了承ください。

ちなみに余談ですが、WSL2 が WSL1 の完全な上位互換なのかと言えば、必ずしもそうではありません。唯一の例外と言ってもよいのが、WSL 環境内からの、Windows ファイルシステムへのアクセスです。これに関して言えば WSL1 の方が高速なため、Windows 上のファイルを WSL のコマンドで操作したいような用途に限って言えば、敢えて WSL1 を選択するというケースもアリでしょう。

1.2 なぜ WSL なのか

さて、そもそもの話になりますが、なぜ WSL が必要なのでしょう。

現在、Linux はサーバー、クラウド、コンテナ、IoT、HPC といった様々な分野において、非常に目覚ましい進出を遂げています。例えば W3Techs のサイトで 2026 年 7 月現在における Web サーバーの OS のシェアを見ると、Windows が全体の 8.3% であるのに対し、91.9% が Unix 系の OS となっています。そしてその Unix 系 OS のうち、過半数の 67.1% が Linux^{*1}なのです。また世界のスーパーコンピュータの性能ランキングである、TOP500^{*2}を見てみましょう。驚くべきことに、TOP500 にランクインしているスーパーコンピュータにおける、Linux の採用率は 100%^{*3}です。この状況は 2017 年以降、変わらず続いています。

^{*1} <https://w3techs.com/technologies/details/os-unix>

^{*2} <https://top500.org/>

^{*3} <https://top500.org/statistics/list/>

LIST STATISTICS

R_{max} and R_{peak} values are in GFlops. For more details about other fields, check the TOP500 description.

TOP500 Release

June 2026

Category

Operating system Family

Submit

Operating system Family System Share

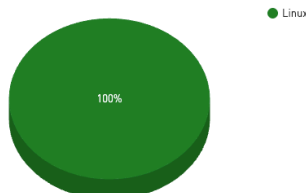


図 1.1: TOP500 で利用されている OS の種類

Linux 向けの開発作業には、当然ですがターゲットとなる Linux 環境が必要となります。対して、デスクトップ OS として最もシェアの大きい OS は Windows です。ここで開発者が使うデスクトップ OS と、ターゲットシステムの間にあるギャップが問題となります。

最近では Linux のデスクトップ環境も、非常に洗練されてきており、普段使いで困ることはほぼありません。そのため開発者が使うデスクトップ OS を Linux にしてしまうというのも、ひとつの方法でしょう。ですが業務においては、「書類の作成に MS Office が必要」「Adobe のアプリが必要」「社内管理の都合上、プリインストールの OS を入れ替えられない」などの理由で、Linux を普段使いにできないケースもあるでしょう。というより、一般的な企業であればその方が多数派でしょう。

こうした背景から、開発者が普段使いする環境として、BSD ベースの Unix である Mac を選択するケースが、2000 年代の後半頃から非常に増加しました。技術系のカンファレンスに参加して、半数以上の人が銀色の板を持っている光景を見たことがある人も多いのではないのでしょうか。ところが Mac は、Unix ではあるものの、Linux ではありません。「標準でインストールされているツールが GNU のものと挙動が異なる」「Linux カーネルの機能が使えるわけではない」など、環境の構築にあたって、Linux との差異に苦しめられるケースも多々あります。

この「OS そのものは Windows を使いたい」「でも本物の Linux 環境が手元に欲しい」という要望を、従来の仮想マシンよりも簡単に満たせるのが WSL なのです。

1.3 WSL のインストール

なにはともあれ、WSL をインストールし、土台となる環境を構築していきましょう。WSL2 は内部的に仮想マシンを使って Linux カーネルを動かしています。そのため WSL2 を利用するためには、あらかじめ CPU の仮想化支援機能 (Intel VT や AMD-V) を有効にしておく必要があります。一般的に、仮想化支援機能の有効/無効の切り替えは、PC の UEFI で行います。詳しくはお使いの PC のマニュアルを参照してください。とはいえ今時の PC であれば、デフォルトで有効となっていることが多いでしょう。

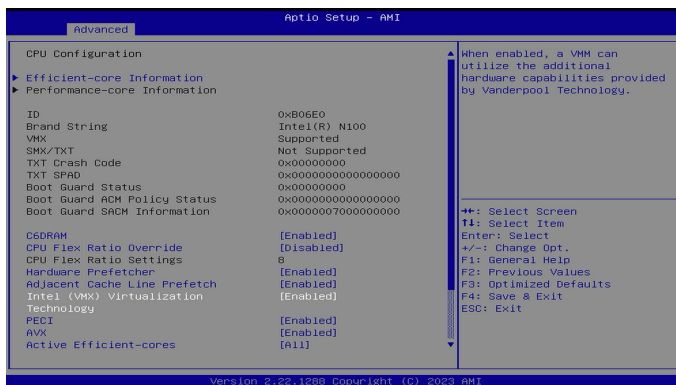


図 1.2: Intel VMX を有効にする

Windows 11 を起動したら、まずスタートメニューから Windows PowerShell を起動してください。

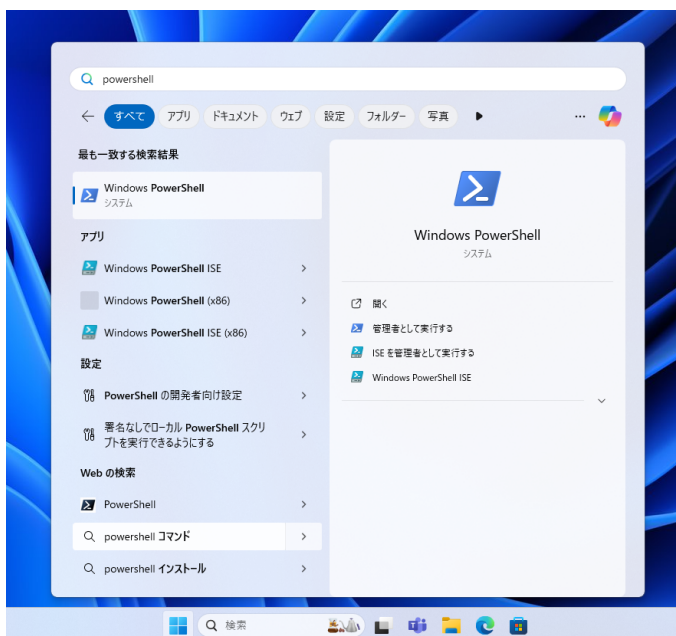


図 1.3: PowerShell を起動する

以下のコマンドを実行します。

```
$ wsl --install --no-distribution
```

WSL のインストールには、Windows の管理者特権が必要です。そのため以下のようなウィンドウが表示されますので、「はい」をクリックしてください。

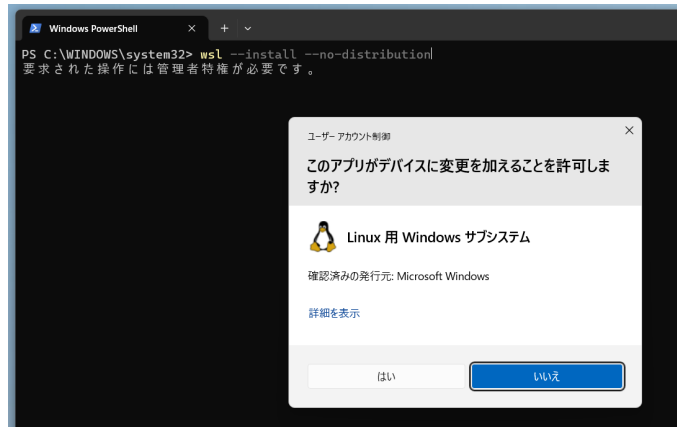


図 1.4: WSL のインストール

インストールが完了し、以下の画面が表示されたら Windows を再起動してください。

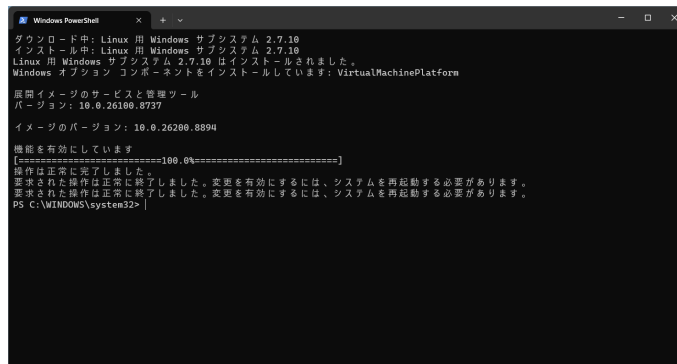
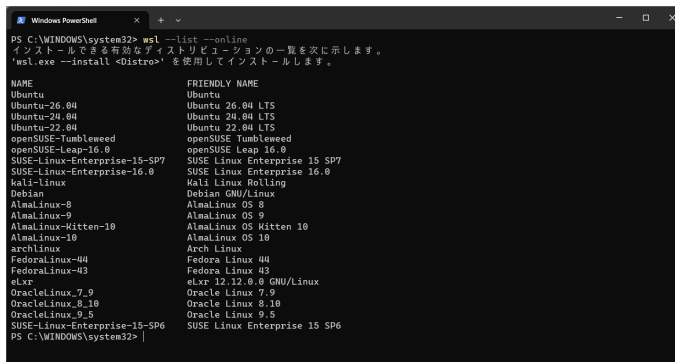


図 1.5: インストールが完了したら Windows を一旦再起動する

再起動したら、再度 PowerShell を起動してください。

WSL では複数の Linux ディストリビューションが提供されています。PowerShell に以下のコマンドを入力することで、提供されているディストリビューションの一覧を表示できます。


```
$ wsl --list --online
```



```
PS C:\WINDOWS\system32> wsl --list --online
インストールできる有効なディストリビューションの一覧を次に示します。
'wsl.exe --install <Distro>' を使用してインストールします。

NAME                                FRIENDLY NAME
-----
Ubuntu                              Ubuntu
Ubuntu-26.04                        Ubuntu 26.04 LTS
Ubuntu-24.04                        Ubuntu 24.04 LTS
Ubuntu-22.04                        Ubuntu 22.04 LTS
openSUSE-Tumbleweed                 openSUSE Tumbleweed
openSUSE-Leap-16.0                 openSUSE Leap 16.0
SUSE-Linux-Enterprise-15-SP7       SUSE Linux Enterprise 15 SP7
SUSE-Linux-Enterprise-16.0        SUSE Linux Enterprise 16.0
kali-linux                         Kali Linux Rolling
Debian                              Debian GNU/Linux
AlmaLinux-8                        AlmaLinux OS 8
AlmaLinux-9                        AlmaLinux OS 9
AlmaLinux-Kitten-10               AlmaLinux OS Kitten 10
AlmaLinux-10                      AlmaLinux OS 10
archlinux                         Arch Linux
FedoraLinux-44                    Fedora Linux 44
FedoraLinux-43                    Fedora Linux 43
elxR                               elxR 12.12.0.0 GNU/Linux
OracleLinux_7.9                   Oracle Linux 7.9
OracleLinux_8.10                  Oracle Linux 8.10
OracleLinux_9.5                   Oracle Linux 9.5
SUSE-Linux-Enterprise-15-SP6      SUSE Linux Enterprise 15 SP6
```

図 1.6: ディストリビューションの一覧を確認する

本書では世界的に人気があり、WSL の既定 Linux ディストリビューションでもある **Ubuntu** を使用することを前提に解説していきます。ここでは 2026 年 7 月現在の最新長期サポート版にあたる、**Ubuntu 26.04** をインストールします。PowerShell 上で以下のコマンドを実行してください。

```
$ wsl --install -d Ubuntu-26.04
```

しばらく待てば、インストールは完了です。続いて初回セットアップがはじまります。まずは Ubuntu 上で利用するユーザー名を入力します。一般的な Linux ディストリビューションでは、ユーザー名にはアルファベット小文字、数字、アンダースコア、ハイフンが使えます。ただし頭文字は、アルファベットの小文字にするのが一般的です。

続いてユーザーのパスワードを設定します。確認のため同じパスワードの再入力を促されますので、2 回入力してください。なおパスワードのエコーバック（●などが表示されること）は行われません。そのため正しく入力できていないように見えてしまいがちですが、気にせず入力して **Enter** を押してください。

ユーザーアカウントの作成の後に、以下のメッセージが表示されます。

Help improve Ubuntu!

Help us improve Ubuntu features and compatibility by sharing system reports with Canonical. Reports are sent anonymously and do not contain any personal data.
For legal details, please visit: <https://ubuntu.com/legal/systems-information-notice>

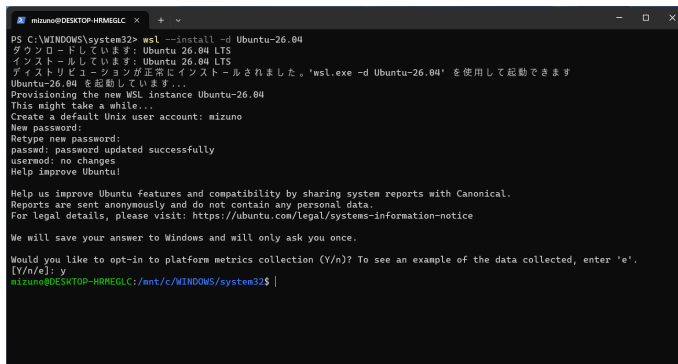
We will save your answer to Windows and will only ask you once.

Would you like to opt-in to platform metrics collection (Y/n)? To see an example of the data collected, enter 'e'.
[Y/n/e]: y

これは Ubuntu Insights と呼ばれるもので、ハードウェア情報やその他の収集されたメトリクスをレポートするためのツールです。初回起動時にシステムメトリクス収集へのオプトインを求められますので、許可する場合は **y** を、拒否する場合は **n** を入力して **Enter** を押してください。

オプトイン方式のため、ここで明示的に許可しない限り情報の送信は行いません。ですがレポートの送信は Ubuntu の改善に役立つため、可能であれば利用を検討してみてください。なお個人を特定するような情報は送信されないため、安心してください。なおこの選択は Windows 側に保持されます。そのため以降は Ubuntu 26.04 の別インスタンスをセットアップしたとしても、繰り返し確認する必要はありません。

これらの設定が完了すると、Ubuntu にログインしてシェルが操作できるようになります。これで WSL 上で、Ubuntu を使いはじめる準備は完了です。



```
mizuno@DESKTOP-HRMEGLC: ~$ wsl --install -d Ubuntu-26.04
ダウンロードしています: Ubuntu 26.04 LTS
インストールしています: Ubuntu 26.04 LTS
ディストリビューションが正常にインストールされました。'wsl.exe -d Ubuntu-26.04' を使用して起動できます
Ubuntu-26.04 を起動しています...
Provisioning the new WSL instance Ubuntu-26.04
This might take a while...
Create a default Unix user account: mizuno
New password:
Retype new password:
passwd: password updated successfully
usermod: no changes
Help improve Ubuntu!

Help us improve Ubuntu features and compatibility by sharing system reports with Canonical.
Reports are sent anonymously and do not contain any personal data.
For legal details, please visit: https://ubuntu.com/legal/systems-information-notice

We will save your answer to Windows and will only ask you once.

Would you like to opt-in to platform metrics collection (Y/n)? To see an example of the data collected, enter 'e'.
[Y/n/e]: y
mizuno@DESKTOP-HRMEGLC:/mnt/c:/WINDOWS/system32$
```

図 1.7: インストール後の初回セットアップ

また初回セットアップと同時に、以下のウィンドウが表示されます。ここには WSL を使いこなすための様々な情報が書かれていますので、一読しておくとういでしょう。またこのウィンドウは、スタートメニューから「WSL Settings」で起動できます。このウィンドウ左下の「設定」をクリックすると、GUI から WSL に割り当てるプロセッサ数やメモリ量、スワップサイズ、ネットワーク接続方式などを変更できます。



図 1.8: WSL Settings の画面

1.4 WSL の起動方法

インストールした WSL のディストリビューションは、スタートメニューに登録されます。一般的な Windows のアプリと同様に、ここから起動できます。スタートメニューを開いたら「Ubuntu」で検索するとよいでしょう。



図 1.9: スタートメニューから Ubuntu 26.04 を起動する

また PowerShell を動かしている、**Windows Terminal** 内から Ubuntu のプロファイルを指定することでも起動できます。Windows Terminal とプロファイルについては後述します。

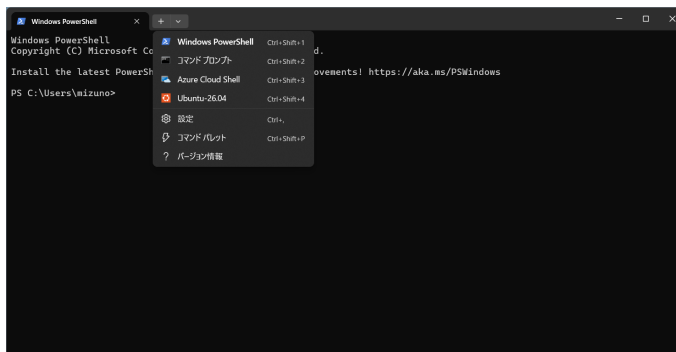


図 1.10: Windows Terminal からプロファイルを指定して起動する

前世紀から PC に触れている人であれば、Microsoft が Linux やオープンソースを嫌っていたこと、もっと直接的に言えば「敵」として見ていたことを覚えているかもしれません。その Microsoft が、Windows 上に Linux を搭載するとは、隔世の感があります。まさに Linux が IT システムにおいて、欠かせない存在になっていることの証左と言えるのではないのでしょうか。

次章以降では、本格的な環境の構築やカスタマイズ、WSL の使いこなしテクニックを紹介していきます。

第 2 章 Windows Terminal を使いこなそう

WSL は基本的に、キーボードからコマンドを入力する CLI (Command Line Interface) で操作します。そのために必要なのがターミナルです。まずは WSL への窓口となる、Windows Terminal について解説します。

2.1 ターミナルとは

ターミナル、あるいは**端末**とは、コンピューターに接続して入出力を行う専用機器の名称です。少し歴史的な部分から解説しましょう。

大昔のコンピューターは非常に大きく、専用の部屋に設置されていました。こうしたコンピューターの元まで、わざわざ出向いて操作するのは面倒で不便です。そこで入力に使うキーボードと結果を表示するディスプレイだけを備えた、専用の装置が登場しました。これがターミナルです。ターミナルはシリアルケーブルなどでコンピューターと接続され、コンピューターを別室から遠隔操作できるようになっていました。人間が操作する入出力装置と、実際に計算する CPU を分離していたわけです。

ですがパーソナルコンピューター (PC) の登場により、コンピューター本体と入出力装置の一式を、机上に設置できるようになりました。そのためコンピューターを設置するための別室や、遠隔操作の端末は、もはや必要ありません。とはいえコンピューターを操作するには、命令を与え、結果を受け取る「窓口」となる仕組みが、当時と同様に必要です。

そこで現在では、ハードウェアのターミナルをソフトウェア的に再現した**ターミナルエミュレーター**が使われます。従来のターミナルをアプリにして、OS の内部で同じことをしてしまおうというわけです。PC のキーボードはそのままターミナルのキーボードとして使われ、コンピューターに命令を送ることができます。また結果を表示するターミナルのディスプレイは、GUI のウィンドウとして、一般的なデスクトップアプリと同様に表示されます。

現在では、物理的なターミナルが使われることはほとんどないため、ターミナルとターミナルエミュレーターを混同する恐れはないでしょう。そのためターミナルエミュレーターを指して、単にターミナルや端末と呼ぶことも一般的です。本書でもターミナルと呼称することにします。

2.2 Windows におけるターミナル

もともと Windows では、**Windows Console Host (conhost.exe)** というターミナルが標準的に利用されていました。Windows で「コマンドプロンプト (cmd.exe)」を使ったことがある人であれば、このどこか野暮ったいウィンドウに見覚えがあるのではないのでしょうか。そう、映画やドラマでハッカーがキーボードから呪文を入力している、あの「黒い画面」のことです。

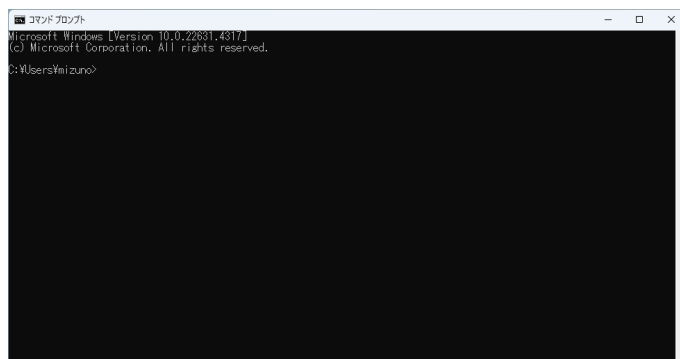


図 2.1: Windows Console Host の画面

Windows Console Host は見ての通り、単純にコマンドを実行するくらいしかできない、非常にプリミティブなターミナルです。GUI アプリではあるものの、メニューがないため、どんな操作ができるかも一目ではよくわかりません。もちろん今時のアプリでは当然と言える、タブ表示もサポートしていません。そこで Windows 11 のバージョン 22H2 以降、より現代的なターミナルである **Windows Terminal** に、標準のターミナルが変更されました。

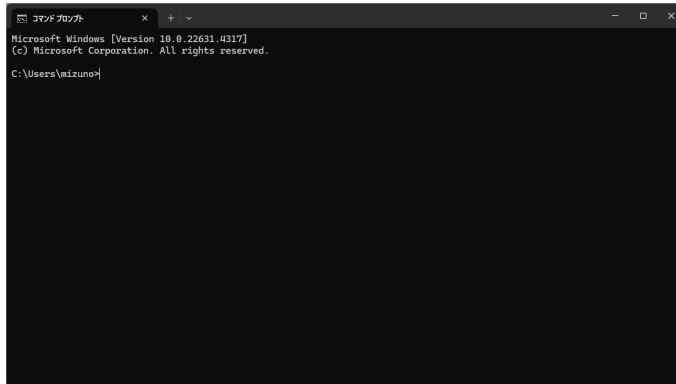


図 2.2: Windows Terminal の画面

前述の通り、ターミナルは人間がコンピューターとやり取りするための「窓口」に過ぎません。ターミナルはフォントや画面の色、タブ表示やスクロールバックといった「見た目」や「GUI 的な使い勝手」の部分のみを担っており、実際に入力されたコマンドを処理しているのは別のプログラムです。この、コマンドを解釈して処理するプログラムを**シェル**と呼びます。Windows の `cmd.exe` や PowerShell、そして Ubuntu をはじめとする Linux で標準的に使われている **Bash** はシェルの一種です。

窓口であるターミナルと、処理を行うシェルが別であるという構造上、ユーザーはターミナルとシェルの自由に組み合わせられます。例えば、ターミナルが Windows Console Host であっても Windows Terminal であっても、あるいは他のサードパーティ製のターミナルであっても、実行しているシェルが同じであれば、コマンドの実行結果はどれも同じになります。また逆に、同じターミナル上で、異なるシェルを実行することも可能です。Windows Terminal はタブ機能を備えているため、PowerShell と Bash を並べて起動もできるのです。異なるシェルであっても、同じ操作感で利用できます。

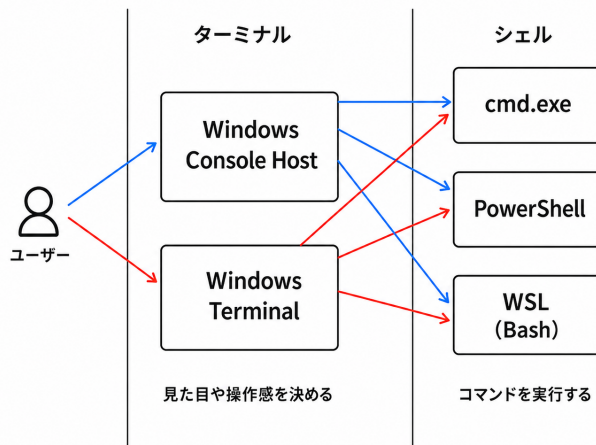


図 2.3: ターミナルとシェルの関係

Linux には複数のシェルが用意されているため、より高機能なシェルに切り替える上級者もいます。そして後からインストールした別のシェルであっても、同じターミナル内で実行できます。

2.3 Windows Terminal の機能

それでは Windows Terminal の、主な機能とその使い方を見ていきましょう。まず Windows Terminal を起動してください。スタートメニューには「ターミナル」で登録されています。



図 2.4: Windows Terminal を起動する

2.3.1 タブ

Windows Terminal はタブ機能に対応しています。タブについての説明はもはや不要でしょうが、複数のウィンドウをひとつのウィンドウ内にまとめ、タブを使って切り替えられるようにする機能です。複数のウィンドウを散らかすことなく整理できるため、Web ブラウザーのような複数のウィンドウを開くことの多いアプリには、もはや実装されていない方が珍しいくらいの定番機能です。

タブバー右側にある「+」ボタンをクリックするか、キーボードから **CTRL+SHIFT+T** キーを押すことで、新しいタブを開けます。

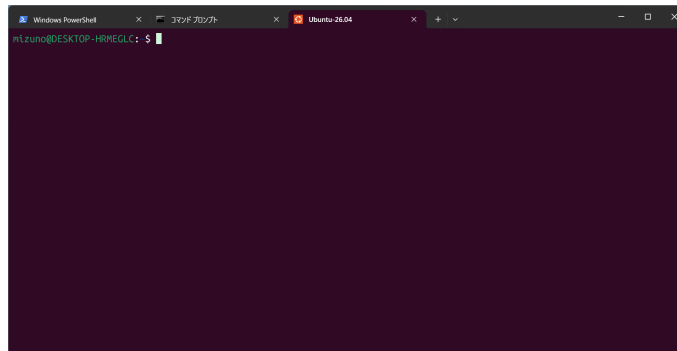


図 2.5: タブを使って複数のシェルの同時に起動した例

単に新しいタブを開くと、規定のプロファイル（後述）が起動します。そこでタブバーの一番右側にある、下向きの矢印ボタンをクリックするか、**CTRL+SHIFT+SPACE** を押してください。タブドロップダ

ウンが開き、「PowerShell」や「コマンドプロンプト」といったプロファイルの一覧が表示されます。ここからプロファイルを選択すると、そのプロファイルで新しいタブを開けます。また各プロファイルには、**CTRL+SHIFT+ 数字**というショートカットが割り当てられています。このプロファイルの番号を覚えておけば、もっと簡単に、好みのプロファイルでタブを開くことができるでしょう。

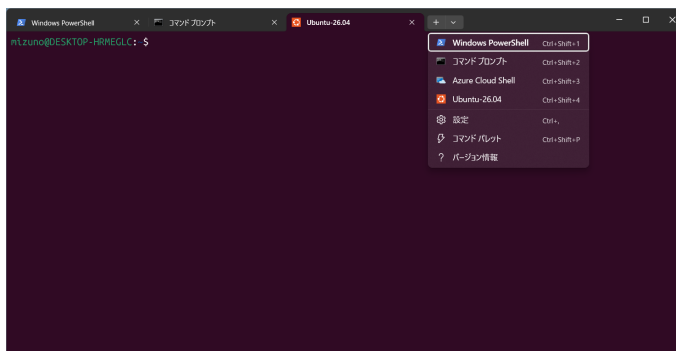


図 2.6: タブドロップダウンからプロファイルを指定して、新しいタブを開く

タブ表示はウィンドウをシンプルにまとめられて便利ですが、複数のターミナルを並べて操作したいこともあるでしょう。当然ですが、複数のウィンドウとタブは併用できます。**CTRL+SHIFT+N**を押すと、規定のプロファイルを新しいウィンドウで開きます。またタブバー右側の「+」ボタンや、タブドロップダウンのプロファイルを、**SHIFT**を押しながらクリックすることでも、新しいウィンドウを開けます。

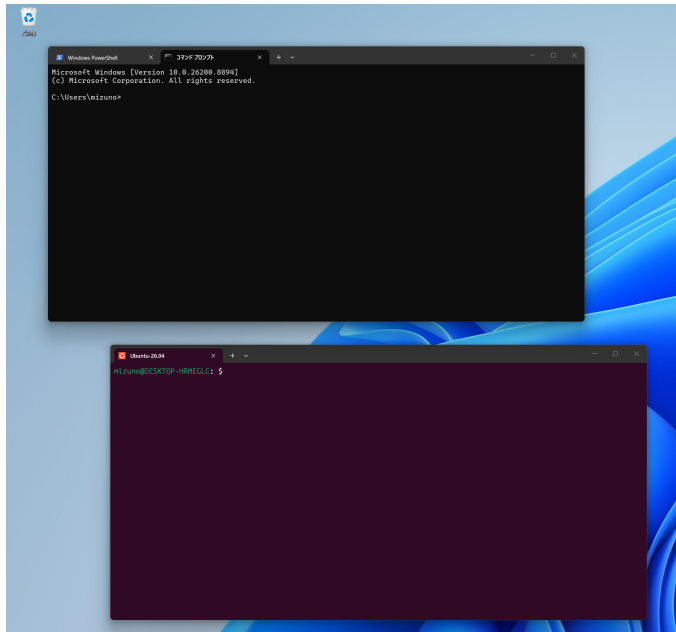


図 2.7: ウィンドウとタブを併用した例

タブはドラッグすることで、位置を入れ替えられます。またタブをウィンドウの外側へドラッグすると、そのタブをウィンドウから切り離して、新しいウィンドウを作れます。その逆に、タブを別のウィ

ンドウにドラッグすれば、ふたつのウィンドウをひとつにマージできます。こうした挙動は Firefox などの Web ブラウザーと同じですから、違和感なく操作できるのではないのでしょうか。

CTRL+SHIFT+W で現在のタブを閉じることができます。

2.3.2 ペイン

別のウィンドウを並べるのではなく、ひとつのタブの内部を、複数に分割したいという場面もよくあります。例えばサーバーのログファイルや負荷を監視しながら、コマンドを入力したいようなケースです。ペインを作成することで、タブの中身を分割し、複数のシェルを同時に動かすことができます。以下の例では、左ペインでは対話的にコマンドを入力しつつ、右上のペインではシステムの状態をモニターし、右下のペインではネットワークに ping を送信しています。

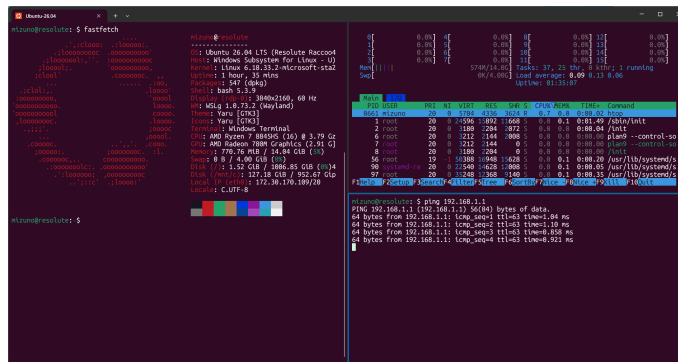


図 2.8: Ubuntu 26.04 のプロファイルで開いたタブを 3 分割した

ALT+SHIFT++ (プラス) キーを押すと、現在のペインを左右に分割します。ALT+SHIFT+- (マイナス) キーを押すと、現在のペインを上下に分割します。アクティブなペインは、対象の領域をマウスでクリックするか、ALT+カーソルキーで切り替えられます。ALT+SHIFT+カーソルキーを押すと、アクティブなペインのサイズを調整できます。

タブと同様に、CTRL+SHIFT+W で現在アクティブなペインを閉じることができます。

2.3.3 コピー&ペースト

CLI は文字を使ってコンピューターに命令を与え、その結果も文字で受け取るインターフェイスです。言いかえると、実行する処理を、文字だけで正確に表現できるわけです。そして同じコマンドを入力すれば、何度でも同じ作業を再現できます。これは GUI アプリにはないメリットだと言えるでしょう。そのため技術系の記事やブログ（もちろん本書もです！）では、敢えて手順をコマンドで解説することも珍しくありません。ここではテキストベースであるがゆえに、コピー&ペーストが大活躍します。もちろん Windows Terminal も、コピー&ペーストに対応しています。コマンドの出力をコピーして保存したり、作業手順書から実行するコマンドをペーストしたりといった使い方ができるため、非常に便利です。さらにマウスを右クリックするだけでもペーストできます。

また後述する設定画面で「選択範囲をクリップボードに自動でコピーする」をオンにすると、テキストをドラッグするだけでコピーが行えます。「ドラッグでコピー」→「右クリックでペースト」と、マウスだけでコピペ操作を行えるようになるため、これも慣れると便利です。

ちなみに選択範囲の自動コピーをオンにすると、ちょっとしたマウス操作ミスでクリップボードの内容が上書きされてしまうため、逆に不便だと感じることもあるでしょう。そういう時は、Windows のクリップボード履歴を有効にしてみましょう。「Win+V」キーを押すとクリップボードが起動しますので、「オンにする」をクリックします。



図 2.9: クリップボード履歴はデフォルトでオフになっているため、初回はオンにする必要がある。

これで以後は、コピーした内容の履歴が保存されるようになります。ペーストしたい場合は **Win+V** を押し、ペーストしたい項目を選択してください。

2.3.4 Quake モード

CLI の操作に慣れてくると、ターミナルは手放せなくなります。常に起動しておきたいアプリのひとつですが、デスクトップの一角を、常に黒い画面が占有しているのは、ちょっと邪魔に感じることもあるでしょう。そんな時に便利なのが Quake モードです。これは Windows Terminal のウィンドウ名を「_quake」と設定した時に入る特殊モードです。Windows Terminal が起動している状態で、**Win+`**（バッククオート）を押してください。Quake モードのウィンドウが開き、画面上部に表示されます。

ウィンドウはデスクトップの 1/2 のサイズに調整され、デスクトップ上部にスナップされます。このウィンドウの位置は変更できません。また上下のサイズは変更できますが、水平方向のサイズは変更できません。

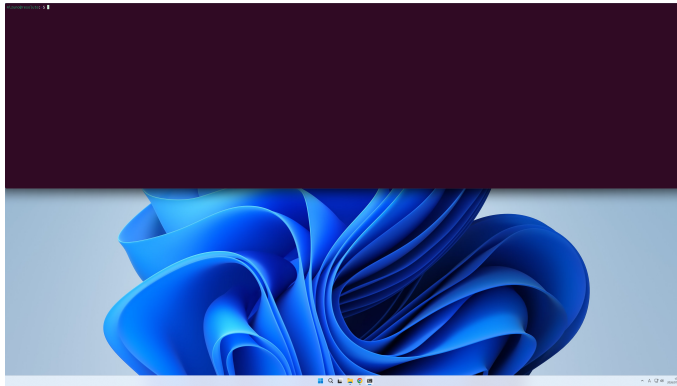


図 2.10: Quake モードの Windows Terminal

以後はWin+‘を押す度に、ウィンドウがアクティブと最小化をトグルします。Quake モードを利用すれば、普段はターミナルを隠しつつも、一瞬でターミナルを開いてコマンドを実行できます。ちなみに、このようにホットキーひとつで画面上部からウィンドウを呼び出せるインターフェイスは、名作FPS ゲーム「Quake」が元祖とされており、このモードの名前もそれに由来します。

2.4 Windows Terminal のカスタマイズ

Windows Terminal は「設定」タブから、好みに応じて見た目や挙動を細かくカスタマイズできます。タブドロップダウンをクリックして、「設定」をクリックするか、キーボードからCTRL+,を押してください。設定のための新しいタブが開きます。

設定項目は多岐に渡るため、すべてについて説明はできません。そこでオススメのカスタマイズを中心に、いくつかをピックアップして紹介します。

2.4.1 ウィンドウの大きさ

Windows Terminal はデフォルトで、120 文字 x30 行のサイズで起動します。1 行あたり 120 文字はともかく、1 画面が 30 行というのは、少し心もとないように筆者は感じています。コマンドが少し長い出力をしたら、それだけでもうスクロールバックする必要が出てきます。最近のデスクトップの解像度は、4K も珍しくなくなりました。こうした十分に広いデスクトップを使っているのであれば、ターミナルのウィンドウも大きくしておいた方が、作業効率的には有利です。

設定の「スタートアップ」を開いてください。下から 2 番目にある「起動サイズ」をクリックすると、Windows Terminal を起動した時や、新しいウィンドウを開いた時のサイズを指定できます。



図 2.11: 起動サイズを設定する

設定を変更したら、「保存」をクリックしてください。

2.4.2 背景の透明度

設定タブの左側にある「プロファイル」のうち、「既定値」を選択してください。これは、個々のプロファイルで上書きされない限り、すべてのプロファイルに適用されるデフォルト値の設定です。この中の「外観」をクリックします。ウィンドウを下方方向にスクロールしていくと、「透明度」という項目の中に「背景の不透明度」という設定があります。デフォルトでは 100% となっていますが、このスライダーを動かして数値を小さくすると、ウィンドウの背景を透明にできます。

ドキュメントを読みながらコマンドを実行したり、複数のウィンドウを並べて出力を見比べたりするような場合に、半透明にしたウィンドウを重ね合わせると便利なこともあるため、覚えておくといでしょう。

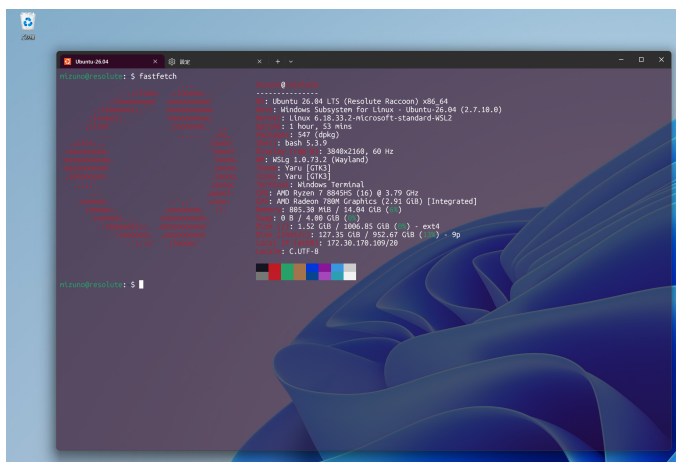


図 2.12: Windows Terminal を透明にした例

2.4.3 常に手前に表示する

Quake モードのように、必要に応じてターミナルを呼び出すのではなく、常にデスクトップ上に表示しておきたいこともあるでしょう。例えば作業中に、サーバーのログを常に見える所で流しておきたい、といったケースです。そういう場合は「外観」の「常に手前に表示」をオンにしてください。他のウィンドウがオーバーラップしても、常に Windows Terminal が最前面に表示されるようになります。

2.4.4 自動的に起動する

「スタートアップ」の「コンピューターのスタートアップ時に起動」をオンにすると、Windows の起動時に、Windows Terminal も自動的に起動するようになります。特にインフラ寄りのエンジニアには、「PC で作業する時は、必ずターミナルを使う」という人も多いでしょう。必ず使うのであれば、自動的に起動させてしまうのが便利です。

2.5 プロファイルとは

既に何度か言及していますが、Windows Terminal には**プロファイル**という概念があります。これは実行するコマンド（シェル）や、ターミナルの色、フォント、挙動といった設定のセットに名前をつけ、切り替えを簡単にしたものです。プロファイルごとに実行するコマンドが異なるため、WSL (Ubuntu) や PowerShell を同じウィンドウ内に同居させられるというわけです。また PowerShell が真っ黒な背景なのに対し、Ubuntu はやや紫がかっていることに気づいたでしょうか。これもプロファイルに指定されている、カラースキームが PowerShell と Ubuntu で異なるためです。

Windows Terminal は、新しいウィンドウやタブを開くと、デフォルトで PowerShell を起動します。これは「スタートアップ」の「規定のプロファイル」に PowerShell のプロファイルが指定されているためです。本書では Ubuntu をメインに使っていきますから、規定のプロファイルを Ubuntu に切り替えてしまいましょう。これで以後は、Windows Terminal の起動時や、新しいタブを開いた時には、Ubuntu の Bash が起動するようになります。



図 2.13: 規定のプロファイルを Ubuntu に変更する

プロファイルは WSL のディストリビューションをインストールすると自動で追加されますが、それとは別に、自分で作成もできます。設定タブの左下にある「新しいプロファイルを追加します」をクリックすると、プロファイルの作成画面に遷移します。まっさらのプロファイルを作るか、既に存在するプロファイルを複製するかを選択できます。

CLI で作業をしていると、どうしても発生してしまうミスのひとつが「ターミナルの取り違い」です。ターミナルはどれも変わりばえのしない黒い画面ですから、複数のターミナルを開いていると、どうしても取り違いミスが起きてしまうのです。手元の開発環境を操作しているつもりが、本番サーバーで破壊的なコマンドを実行してしまった、というようなミスもよく聞きます。

先ほどのカラースキームの例のように、プロファイルではターミナルの外観を大きくカスタマイズできます。そこで本番アクセス用のプロファイルを別に作成し、まったく違うカラースキームや壁紙を設定することで目立たせ、取り違いを防止するというのもよい考えです。

ターミナル内の壁紙は、設定したいプロファイルを選択した上で「外観」をクリックし、その中の「背景画像のパス」で指定できます。なお普通の画像を指定すると、明るすぎて文字が読めなくなってしまうので、「背景画像の不透明度」を 40% 程度まで下げ、視認性を上げるのがコツです。

このように、Windows Terminal は非常に高機能なターミナルです。紹介し切れなかった高度な機能も存在するため、ぜひマニュアル^{*1}を一読してみてください。

^{*1} <https://learn.microsoft.com/ja-jp/windows/terminal/>

第 3 章 Ubuntu のパッケージ管理を学ぼう

今時の Linux ディストリビューションのほとんどは、何らかのパッケージ管理システムを採用しています。この章では Ubuntu におけるパッケージ管理方法について紹介します。

3.1 パッケージとは

ソフトウェアは、プログラムのソースコードから作られます。そのため手元のコンピューター上でソフトウェアを動かすには、ソースコードを開発元のリポジトリからチェックアウトし、必要であればパッチを当ててバイナリをビルドし、システムにインストールしなくてはなりません。特定のアプリだけであればともかく、OS 全体をこのように手動で構築するのは、非常に困難な作業となります。また世界中の Ubuntu ユーザーが、個別に同じソフトウェアをビルドするのは、エコとは言えません。

そこで今時の Linux ディストリビューションの多くでは、**パッケージ**を使ってソフトウェアを管理しています。パッケージとは、事前にビルドされたソフトウェアのバイナリやライブラリ、画像やドキュメントといったリソース、インストールやアンインストール時に実行すべきスクリプト、パッケージ自身を表すメタデータなどをまとめたアーカイブです。パッケージはインターネット上にある**リポジトリ**と呼ばれるサーバーに集めて公開されており、ユーザーはここから必要なソフトウェアのパッケージをダウンロードして展開するだけで、そのソフトウェアをインストールできるというわけです。パッケージを使うことで、開発元から個別にソースコードを入手したり、調整やビルドといった面倒な作業が不要となるため、システムの運用にかかる手間を大きく削減できます。

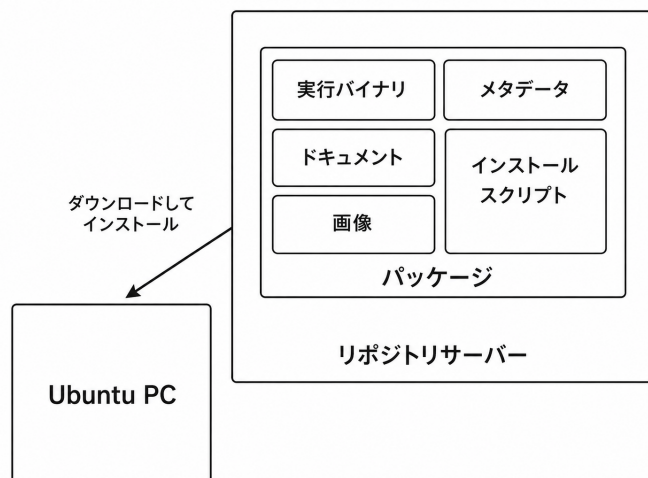


図 3.1: パッケージのしくみ

Ubuntu では、Debian GNU/Linux に由来する **Deb** と呼ばれる形式のパッケージを採用しています。

3.2 パッケージ管理システムとは

多くのソフトウェアに共通した、「ありがちな処理」というものがあります。例えば「JPEG 画像を読み込む」「ZIP ファイルを展開する」「JSON をパースする」などです。こうした処理を個々のソフトウェアが自前で再実装するのは無駄ですし、バグを作り込む原因にもなります。そこでこうした処理の

多くは、別のソフトウェアから再利用しやすいよう、機能単位で個別に実装されています。これを**ライブラリ**と呼びます。個々のソフトウェアはライブラリの機能呼び出すことで、自前での再実装を防げます。また特定の機能をライブラリに集約することで、メンテナンス性も向上するというわけです。例えば SSL/TLS に脆弱性が見つかったとしても、SSL/TLS を扱うライブラリをアップデートすれば、そのライブラリを利用しているすべてのアプリで脆弱性の対応が完了するというわけです。こうした事情から、一部のシングルバイナリなソフトウェアを除けば、今時のソフトウェアが単体で完結していることは稀です。ほとんどのソフトウェアは、なんらかのライブラリに依存しています。そして Linux では、こうしたライブラリの多くを OS 全体で共有しており、これを**共有ライブラリ**と呼びます。

こうした事情は、パッケージでも同様です。あるアプリのパッケージ A が動作するには、別のパッケージ B が必要で、パッケージ B にはまた別のパッケージ C が必要で……と、芋蔓式に複数のパッケージが必要になることも珍しくありません。GUI の高機能なアプリの場合、百を超えるパッケージを要求されることもあります。これをパッケージの**依存関係**と呼びます。つまりあるソフトウェアをインストールしようと思ったら、そのパッケージが要求する依存関係を満たせるよう、必要なすべてのパッケージをダウンロードしてインストールしなくてはならないのです。

パッケージ間の複雑な依存関係をすべて調べ上げ、要求されるすべてのパッケージを手動でインストールするのは、とても現実的ではありません。そこで依存関係を自動的に解決し、必要なすべてのパッケージのダウンロードとインストールを簡単に行えるのが**パッケージ管理システム**です。パッケージ管理システムを利用すれば、インストールされているすべてのパッケージを、最新版にアップグレードすることも簡単です。Ubuntu では **APT** と呼ばれるパッケージ管理システムを用いて、Deb パッケージのインストールやアップグレードを行っています。

3.3 CLI の基本操作

APT は同名の `apt` コマンドで操作します。ですがその前に、Ubuntu の CLI における、基本的な操作方法を覚えておきましょう。Windows Terminal で Ubuntu を起動すると、Ubuntu の標準シェルである Bash が起動し、以下の画面が表示されます。

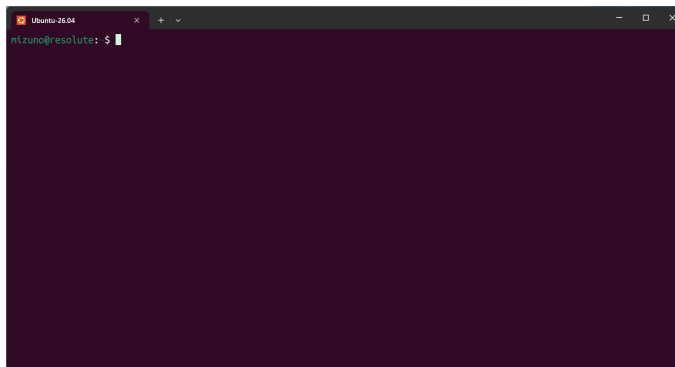


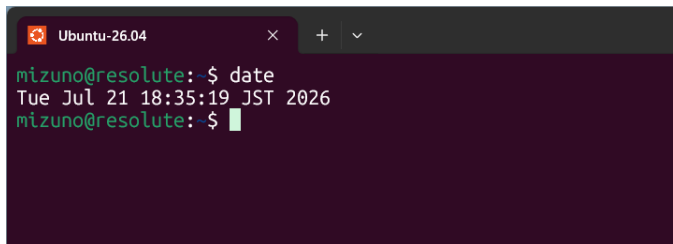
図 3.2: Windows Terminal で Ubuntu 26.04 を起動した状態

ウィンドウの左上に `ユーザー名@ホスト名: カレントディレクトリ$` という文字が表示されています。この文字をプロンプトと呼び、シェルがコマンドの入力を受け付けていることを表しています。表示される具体的な、当然文字は環境によって異なりますが、マニュアル等では慣例上、プロンプトを `$` 一文字で表します。そのため本書でも、コマンドを入力する箇所では、以下のフォーマットで示します。なおあくまで `$` はプロンプトを表す記号ですから、実際にユーザーが入力するのは、`$` より後の文字列だけです。`$` ごとコピーしないよう注意してください。

\$ 入力するコマンド

プロンプトの右側にカーソルが点滅しています。ここに実行したいコマンドを入力し、Enter キーで実行します。手はじめに、現在日時を表示する `date` コマンドを実行してみましょう。プロンプトに続いてキーボードから `date` と入力して、Enter を押してください。

```
$ date
```

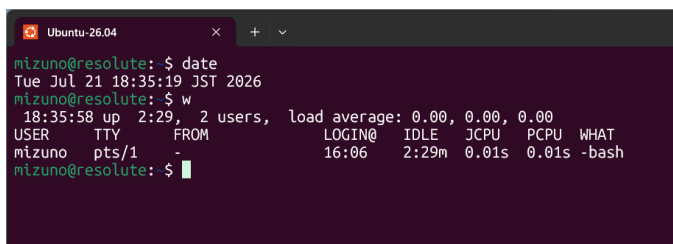
A terminal window titled 'Ubuntu-26.04' with a dark background. The prompt 'mizuno@resolute: \$' is followed by the command 'date'. The output is 'Tue Jul 21 18:35:19 JST 2026'. The prompt returns to 'mizuno@resolute: \$' with a cursor.

```
mizuno@resolute: $ date
Tue Jul 21 18:35:19 JST 2026
mizuno@resolute: $
```

図 3.3: `date` コマンドを実行した例

現在の日時が表示されました。また次の行には、新しいプロンプトが表示されていることがわかるでしょう。シェルはコマンドの実行が終わると、再びプロンプトを表示し、次のコマンドの入力を待機します。続いて現在のログイン状況を表示する `w` コマンドを実行してみましょう。

```
$ w
```

A terminal window titled 'Ubuntu-26.04' with a dark background. The prompt 'mizuno@resolute: \$' is followed by the command 'date'. The output is 'Tue Jul 21 18:35:19 JST 2026'. The prompt returns to 'mizuno@resolute: \$' with a cursor. The next command 'w' is entered, followed by its output: '18:35:58 up 2:29, 2 users, load average: 0.00, 0.00, 0.00' and a table of user sessions.

```
mizuno@resolute: $ date
Tue Jul 21 18:35:19 JST 2026
mizuno@resolute: $ w
18:35:58 up 2:29, 2 users, load average: 0.00, 0.00, 0.00
USER  TTY      FROM             LOGIN@   IDLE   JCPU   PCPU   WHAT
mizuno pts/1    -                16:06    2:29m  0.01s  0.01s  -bash
mizuno@resolute: $
```

図 3.4: 続いて `w` コマンドを実行した例

現在 Ubuntu にログインしているユーザー名や使用している端末、ログイン時刻といった情報が表示されました。このように、プロンプトに対してコマンドを入力し、コマンドが終了したら次のコマンドを入力し……を繰り返して行くのが、CLI 操作の基本です。ここでは Ubuntu の Bash の例を紹介しましたが、Windows のコマンドプロンプト (`cmd.exe`) や PowerShell も基本的な流れはまったく同じです。

コマンドラインのより詳しい操作方法は、Ubuntu のチュートリアル^{*1}も参考にしてください。

^{*1} <https://ubuntu.com/tutorials/command-line-for-beginners#3-opening-a-terminal>

3.4 引数とオプション

3.4.1 引数とは

コマンドには**引数**（ひきすう）という値を渡せます。例えば**echo** コマンドはそのまま実行すると、空の行を表示します。

```
$ echo
```

引数とはコマンドに対して与える値で、**パラメータ**とも呼びます。コマンドは受け取った引数を解釈し、それに応じて挙動を変化させます。**echo** コマンドは引数として文字列を指定できます。すると**echo** コマンドは、指定された文字列をそのまま表示します。

```
$ echo Hello  
Hello
```

コマンド名と引数は、スペースで区切ります。またコマンドによっては、複数の引数を指定できる場合もあります。具体的にどのような引数を受け取れるかは、コマンドによって異なるため、詳しくは各コマンドのマニュアルを参照してください。マニュアルの読み方については後述します。

3.4.2 オプションとは

オプションとは、簡単に言えばコマンドの挙動を変更するスイッチです。**echo** コマンドは引数に指定された文字列を表示し、最後に改行しました。ところが**-n** オプションをつけると、改行を行いません。実際に実行して違いを比べてみましょう。

```
$ echo -n Hello
```

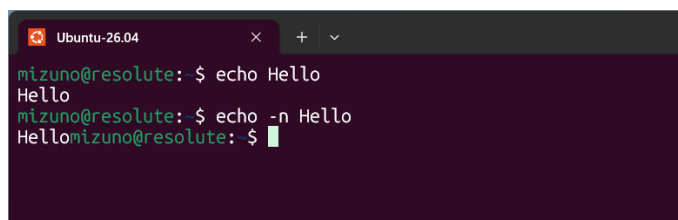


図 3.5: オプションの有無による echo コマンドの挙動の違い

オプションはスペースで区切って複数と同時に指定できます。そして複数のオプションは、個別に指定することも、ハイフンの後にまとめて書くこともできます。そして通常のオプションはハイフン1つと1文字の英数字で表されますが、ハイフン2つと英単語から構成される**ロングオプション**も存在します。ただしオプションの中には、ロングオプションでしか提供されていないものや、その逆も存在します。どんなオプションが用意されているかはコマンドによって異なるので、こちらも各コマンドのマニュアルを参照してください。

3.5 APT を使いこなす

それでは実際に、**apt** コマンドを使ってみましょう。

3.5.1 apt のサブコマンド

`apt` コマンドは、様々なサブコマンドと組み合わせて使います。よく使う代表的なサブコマンドは表 3.1 の通りです。

表 3.1: apt の主なサブコマンド

サブコマンド	用途
update	パッケージ情報を更新する。
upgrade	インストールされているすべてのパッケージを最新のバージョンに更新する。ただしパッケージの削除が発生する場合は作業を保留する。
full-upgrade	パッケージの削除が発生する場合であっても、すべてのパッケージを最新のバージョンに更新する。
install	指定したパッケージをインストールする。
remove	指定したパッケージを削除する。
purge	指定したパッケージを、設定ファイルごと削除する。
autoremove	依存関係によって自動インストールされたパッケージのうち、現在では不要になったものを削除する。
search	指定されたキーワードをもとにパッケージを検索する。
list	指定された条件に一致するパッケージの一覧を表示する。
show	パッケージの詳細な情報を表示する。

例えばパッケージ情報を更新するには `apt update`、パッケージをインストールするには `apt install` といった具合です。続いて具体的な実行手順を解説します。

3.5.2 パッケージ情報の更新

`apt` コマンドは、ローカルにキャッシュされたパッケージ情報を手がかりに、リポジトリからパッケージ本体のダウンロードやインストールを行うという仕組みになっています。そのためまず最初に、このキャッシュを最新の状態に更新しなければなりません。なぜならキャッシュが古いままだと、リポジトリサーバー上にパッケージの新しいバージョンが存在しても、`apt` コマンドはそれを見つけることができず、インストールやアップグレードが行えなくなってしまうためです。パッケージ情報のキャッシュを更新するには、`apt update` を実行します。

```
$ sudo apt update
```

ここで `sudo` という新しいコマンドが登場しました。これは後に指定したコマンドを、別のユーザー権限（通常は root ユーザー）で実行するためのコマンドです。今時のマルチユーザーの OS では、一般ユーザーと管理用のスーパーユーザーで、権限が分けられているのが一般的です。Windows でも、システムの設定を変更する時は、管理者の権限を要求されますよね^{*2}。

^{*2} Windows にも `sudo` は存在します。 <https://learn.microsoft.com/ja-jp/windows/advanced-settings/sudo/>

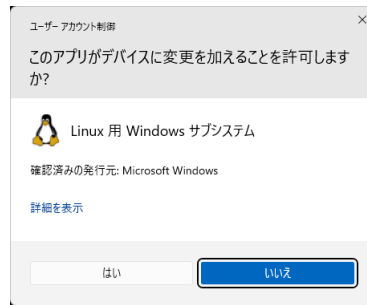


図 3.6: WSL のインストール時に表示された確認ダイアログの例

Linux もその例に漏れず、システムに対する最高の権限を持つ **root** (ルート) と、一般ユーザーに分けられています*³。WSL ディストリビューションのインストール時に作成し、現在 Ubuntu にログインしているユーザーは一般ユーザーのため、システムに対する変更権限を持ち合わせていません。パッケージ情報の更新は、システムの領域にデータを書き込むため、一般ユーザーでは実行できないのです。

従来の Linux ディストリビューションであれば、こうした作業は root ユーザーで直接ログインして行っていました。ですが Ubuntu をはじめとする現代的なディストリビューションでは、**sudo** を使って、一時的に root に「昇格」することで行うのが定石となっています。WSL で Ubuntu を起動した直後の画面をよく見てください。英語なので読み飛ばした方もいるかもしれませんが、以下のメッセージが表示されていますよね。

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

(参考訳: コマンドを管理者 (root ユーザー) として実行する場合は、"sudo コマンド" を使用してください。詳細は "man sudo_root" を参照してください。)

sudo をつけてコマンドを実行すると、ユーザーのパスワードの入力を求められます。インストール時に設定した、自分自身のパスワードを入力してください*⁴*⁵。

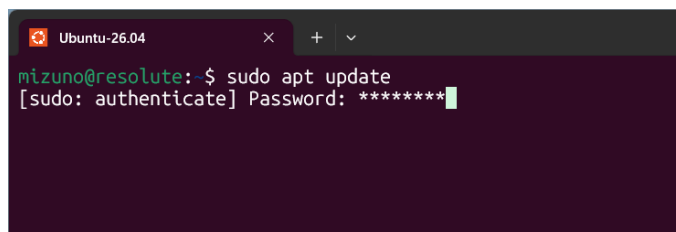


図 3.7: sudo をつけてコマンドを実行すると、パスワードを求められる

これで正しくパッケージ情報が更新されます。

*³ 厳密に言えば、現在の Linux における権限はもっと細分化されており、「なんでもできる root とそれ以外」という構造ではなくなっています。詳細は省きますので、もし興味があったら「Linux カーネル ケーパビリティ」などで検索してみてください。

*⁴ 実は以前の Ubuntu では、この際パスワードのエコーバック（※などが表示されること）が行われませんでした。そのため「パスワードが入力できない」と勘違いする人が続出したという歴史があります。Ubuntu 26.04 ではエコーバックが行われるよう仕様が変更されました。

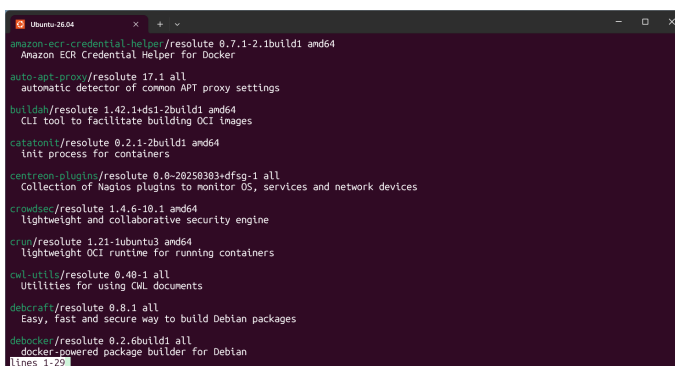
*⁵ 実は Ubuntu 26.04 では、sudo 自体が従来のものから、Rust 言語で再実装された sudo-rs に変更されています。

3.5.3 パッケージの検索

`apt` コマンドでパッケージをインストールするためには、対象のパッケージ名を知っておく必要があります。そこで、キーワードをもとにパッケージを検索するのが `apt search` です。このサブコマンドは、システムへの変更等を行わないため、`sudo` なしで実行できます。例として引数に `docker` を指定して、**Docker** に関するパッケージを探してみましょう。なお引数には、文字列だけでなく、POSIX 正規表現が使用可能です。

```
$ apt search docker
```

コマンドを実行すると、指定したキーワードを含むパッケージが一覧表示されます。緑色で表示されているのがパッケージ名で、その横にインストール元のリポジトリ、パッケージのバージョン、パッケージのアーキテクチャが表示されています。その下にはパッケージの簡単な説明が表示されます。これを手がかりに、パッケージを探してみましょう。



```
amazon-ecr-credential-helper/resolute 0.7.1-2.1build1 amd64
Amazon ECR Credential Helper for Docker

auto-apt-proxy/resolute 17.1 all
automatic detector of common APT proxy settings

buildx/resolute 1.42.1-ds1-2build1 amd64
CLI tool to facilitate building OCI images

catatini/resolute 0.2.1-2build1 amd64
init process for containers

centreon-plugins/resolute 0.8-20250303-dfsg-1 all
Collection of Nagios plugins to monitor OS, services and network devices

crowdsec/resolute 1.4.6-10.1 amd64
lightweight and collaborative security engine

crun/resolute 1.21-1ubuntu3 amd64
lightweight OCI runtime for running containers

cwl-utils/resolute 0.40-1 all
Utilities for using CWL documents

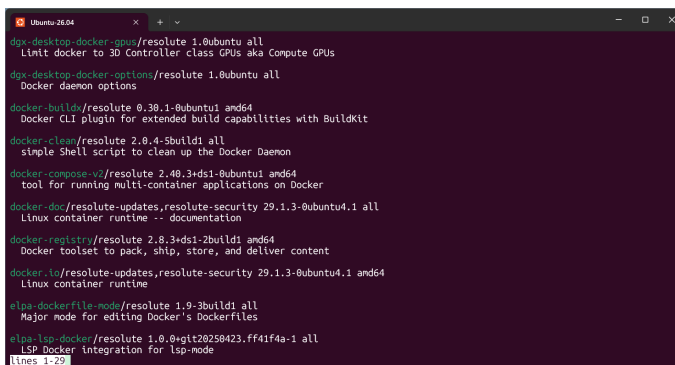
debcraft/resolute 0.8.1 all
Easy, fast and secure way to build Debian packages

docker/resolute 0.2.0build1 all
docker-powered package builder for Debian
lines 1-29
```

図 3.8: `apt search` の実行例

ただし `search` サブコマンドは、パッケージリストに対して全文検索を行うため、これだけですと膨大なパッケージがヒットしてしまいます。そこで `-n` オプションをつけると、検索の対象をパッケージ名のみに絞ることができます。やってみましょう。

```
$ apt search -n docker
```



```
dga-desktop-docker-gpus/resolute 1.0ubuntu1 all
Limit docker to 3D Controller class GPUs aka Compute GPUs

dga-desktop-docker-options/resolute 1.0ubuntu1 all
Docker daemon options

docker-buildx/resolute 0.30.1-0ubuntu1 amd64
Docker CLI plugin for extended build capabilities with BuildKit

docker-clean/resolute 2.0.4-5build1 all
simple Shell script to clean up the Docker Daemon

docker-compose-v2/resolute 2.40.3-ds1-0ubuntu1 amd64
tool for running multi-container applications on Docker

docker-doc/resolute-updates,resolute-security 29.1.3-0ubuntu4.1 all
Linux container runtime -- documentation

docker-registry/resolute 2.8.3-ds1-2build1 amd64
Docker toolset to pack, ship, store, and deliver content

docker.io/resolute-updates,resolute-security 29.1.3-0ubuntu4.1 amd64
Linux container runtime

elpa-dockerfile-mode/resolute 1.9-3build1 all
Major mode for editing Docker's Dockerfiles

elpa-lsp-docker/resolute 1.0.0+git20250423.ff41f4a-1 all
LSP Docker integration for lsp-mode
lines 1-29
```

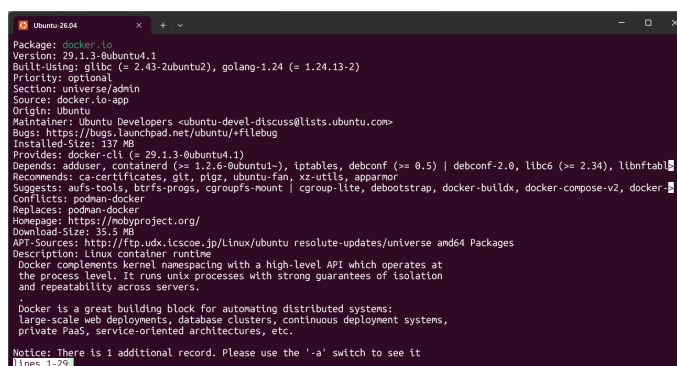
図 3.9: パッケージ名で検索した例

だいたい絞り込めたのではないのでしょうか。そしてパッケージの説明を見ると、Ubuntu における Docker ランタイムのパッケージは `docker.io` パッケージだろうと、アタリをつけることができるわけです。

3.5.4 パッケージの詳細

`search` サブコマンドでは、各パッケージの概要レベルの説明しか表示されませんでした。パッケージのより詳細な情報は `apt show` で確認できます。引数にパッケージ名を指定してください。またこのサブコマンドも、情報を参照するだけですなので、`sudo` は不要です。

```
$ apt show docker.io
```



```
Package: docker.io
Version: 20.10.11-0ubuntu0~20.04~deb10u1
Built-Using: glibc (= 2.31-0ubuntu2), golang-1.16 (= 1.16.10-1)
Priority: optional
Section: universe/admin
Source: docker.io
Origin: Ubuntu
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Installed-Size: 137 MB
Provides: docker-cli (= 20.10.11-0ubuntu0~20.04~deb10u1)
Depends: adduser, containerd (>= 1.2.6-0ubuntu1~), iptables, debconf (>= 0.5) | debconf-2.0, libc6 (>= 2.34), libnftables1
Recommends: ca-certificates, git, pigz, ubuntu-fan, xz-utils, apparmor
Suggests: aufs-tools, btrfs-progs, cgroupfs-mount | cgroup-lite, debootstrap, docker-buildx, docker-compose-v2, docker-credential-ecr-login
Conflicts: podman-docker
Replaces: podman-docker
Homepage: https://mobyproject.org/
Download-Size: 35.5 MB
APT-Sources: http://ftp.ubuntu.com/ubuntu focal-updates/universe amd64 Packages
Description: Linux container runtime
 Docker complements kernel namespaces with a high-level API which operates at the process level. It runs unix processes with strong guarantees of isolation and repeatability across servers.
 Docker is a great building block for automating distributed systems:
 large-scale web deployments, database clusters, continuous deployment systems,
 private PaaS, service-oriented architectures, etc.
Notice: There is 1 additional record. Please use the '-a' switch to see it
lines 1-29
```

図 3.10: `apt show` の実行例

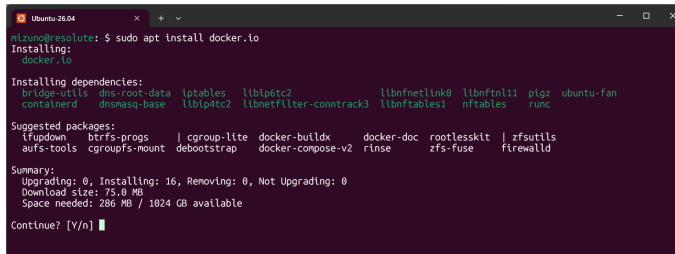
パッケージ名やバージョンだけでなく、依存しているパッケージ、同時にインストールすることが推奨されているパッケージ、開発元の Web ページ、パッケージの詳細な説明などが表示されました。Description に書かれている内容や、Homepage が Moby Project の URL であることから、このパッケージは Docker のランタイムであるということがわかります。

3.5.5 パッケージのインストール

パッケージをインストールするには、`install` サブコマンドに、パッケージ名を指定します。例えば Docker をインストールするには、先ほど特定した `docker.io` パッケージを指定します。なお当然ですが、パッケージのインストールには root 権限が必要となるため、`sudo` をつけて実行します。

```
$ sudo apt install docker.io
```

すると本当にパッケージをインストールしてよいかの確認プロンプトが表示されます。キーボードから`y`を入力してから、`Enter`を押してください。



```
Ubuntu 20.04
mizuno@resolute:~$ sudo apt install docker.io
Installing:
docker.io

Installing dependencies:
bridge-utils  dos-root-data  libtables  libipset2  libnftnl0  libnftnl1  pigz  ubuntu-fan
containerd    dnsmasq-base  libipset2  libnetfilter-contrack3  libnftables1  nftables  runc

Suggested packages:
ifupdown  nftables-progs  | cgroup-lite  docker-buildx  docker-doc  rootlesskit  | zfsutils
aufs-tools  cgroupfs-mount  debootstrap  docker-compose-v2  docker-rm  zfs-fuse  | firewallld

Summary:
Upgrading: 0, Installing: 16, Removing: 0, Not Upgrading: 0
Download size: 75.0 MB
Space needed: 286 MB / 1824 GB available

Continue? [Y/n]
```

図 3.11: インストールしてよいかの確認プロンプトが表示される

なお都度確認を求められるのが面倒な場合は、`apt install`に`-y`オプションを指定して実行してください。常に`y`を入力したものとして、確認をスキップできます。

```
$ sudo apt install -y docker.io
```

3.5.6 パッケージのアンインストール

インストール済みのパッケージをアンインストールするには、`remove`サブコマンドに、パッケージ名を指定します。こちらも当然 `root` 権限が必要ですので、`sudo`をつけて実行してください。ただし不用意にパッケージをアンインストールすると、システムに問題が発生する可能性も否定できません。アンインストールするのは、自分がインストールしたパッケージや、確実に不要であるとわかっているパッケージに限定するのが無難でしょう。

```
$ sudo apt remove docker.io
```

アンインストール時も、本当に実行してよいかの確認プロンプトが表示されます。`-y`オプションで確認をスキップできるのも同様です。

なお`remove`サブコマンドでは、設定ファイルは削除しません。そのためパッケージをインストールし直せば、アンインストール前の状態に戻すことができます。もしも設定ファイルまで綺麗に削除したい場合は、かわりに`purge`サブコマンドを使用してください。

3.5.7 パッケージのアップグレード

`upgrade`サブコマンドを使うと、インストール済みのすべてのパッケージを、最新のバージョンへとアップグレードできます。すべてのパッケージが対象となるため、パッケージ名を指定する必要はありません。

```
$ sudo apt upgrade
```

アップグレード可能なパッケージが存在する場合は、その一覧が表示され、インストール時と同様の確認プロンプトが表示されます。アップグレードを続行してよければ、`y`を入力してからEnterを押してください。`-y`オプションや、後述する`-U`オプションも同様に利用できます。`apt upgrade`を行うだけで、Ubuntu を最新の状態に保つことができます。定期的に行うようにしましょう。

ごく稀に、パッケージの依存関係が変更になった等の理由で、アップグレード時に既存パッケージが削除されることがあります。`upgrade`サブコマンドは、パッケージの削除が伴う場合は、そのパッケージのアップグレードを行わず、作業を保留します。その結果として、一部のパッケージが古いままになる可能性があります。もしも「パッケージを削除しても構わないので、とにかくすべてを最新にしたい」という場合は、かわりに`full-upgrade`サブコマンドを使用してください。

```
$ sudo apt full-upgrade
```

3.5.8 パッケージの一覧表示

`list`サブコマンドを使うと、条件を満たすパッケージの一覧を表示できます。以下のように`--installed`オプションをつけて実行すると、現在システムにインストール済みのパッケージ一覧を確認できます。

```
$ apt list --installed
```

また`--upgradable`オプションをつけて実行すると、インストール済みのパッケージのうち、アップグレードが可能なものの一覧を表示できます。

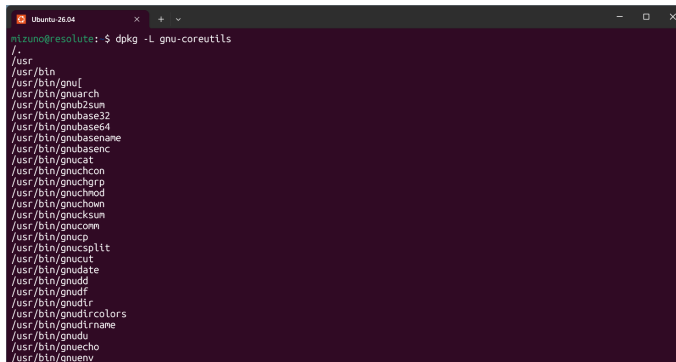
```
$ apt list --upgradable
```

Ubuntu を使っていくと、「このパッケージ、インストールされていたっけ？」となるシーンがよくあります。そのような時は`apt list`で確認してみましょう。

3.5.9 パッケージの内容の確認

Deb パッケージを操作する`dpkg`コマンドを使えば、個々のパッケージをより詳細に管理できます。冒頭で、パッケージには実行バイナリやドキュメント、画像等のリソースが含まれていると説明しました。ではあるパッケージ内には、どのようなファイルが含まれているのでしょうか。これを調べたい時は、`-L`オプションをつけて`dpkg`コマンドを実行します。引数にはインストール済みの、調査したいパッケージ名を指定してください。以下のコマンドでは、Linux で使われる基本的なコマンド群を提供している、**GNU Coreutils** のパッケージの内容を調べることができます。

```
$ dpkg -L gnu-coreutils
```

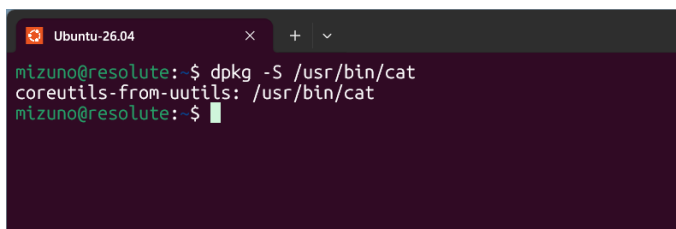


```
mizuno@resolute: $ dpkg -L gnu-coreutils
./usr
./usr/bin
./usr/bin/gnu[
./usr/bin/gnuarch
./usr/bin/gnub2sum
./usr/bin/gnubase32
./usr/bin/gnubase64
./usr/bin/gnubasename
./usr/bin/gnubasenc
./usr/bin/gnucacat
./usr/bin/gnuchcon
./usr/bin/gnuchgrp
./usr/bin/gnuchmod
./usr/bin/gnuchown
./usr/bin/gnuicksum
./usr/bin/gnuiconv
./usr/bin/gnuicp
./usr/bin/gnuicsplit
./usr/bin/gnuicut
./usr/bin/gnudeate
./usr/bin/gnudd
./usr/bin/gnuddf
./usr/bin/gnudir
./usr/bin/gnudircolors
./usr/bin/gnudirname
./usr/bin/gnudu
./usr/bin/gnuecho
./usr/bin/gnuenv
```

図 3.12: gnu-coreutils パッケージに含まれているコマンド

逆に、システム上にインストールされたあるファイルが、どのパッケージに所属しているかを知りたいこともあるでしょう。こういう時は、`-S` オプションをつけて `dpkg` コマンドを実行します。引数には調べたいファイルをフルパスで指定してください。以下のコマンドでは、ファイルの結合や表示に使う `cat` コマンドが、どのパッケージからインストールされているかを調べます*6。

```
$ dpkg -S /usr/bin/cat
```



```
mizuno@resolute: $ dpkg -S /usr/bin/cat
coreutils-from-uutils: /usr/bin/cat
mizuno@resolute: $
```

図 3.13: `/usr/bin/cat` コマンドは、`coreutils-from-uutils` パッケージによって提供されている

3.6 「apt」と「apt-get」の関係

以前から Ubuntu や、その派生元である Debian GNU/Linux を使っている人であれば、`apt-get` や `apt-cache` というコマンドをご存知かもしれません。実は以前の Ubuntu では、パッケージのインストールやアップグレード (`install` や `upgrade` など) は `apt-get`、パッケージ情報の検索 (`search` など) は `apt-cache` と、ふたつのコマンドを使い分けていました。これは面倒だということで、このふたつのコマンドを統合したのが `apt` です。そのため現在では、とりあえず `apt` コマンドだけを使っていれば問題ありません。

ただしすべてのケースで `apt` コマンドを使うべきかといえば、そうでもありません。`apt` コマンドは人間が対話的に使用することを前提としており、コマンド出力が今後変更される可能性もあります。そのためシェルスクリプトや `Dockerfile` 内で使用する場合は、現在でも `apt-get` が推奨されています。

*6 なぜ `cat` コマンドの提供元が `gnu-coreutils` パッケージではないのかは後述します。

またaptとapt-getでは、upgradeサブコマンドの挙動に違いがあります。先ほどaptでは、パッケージの削除が伴うアップグレードは保留されると紹介しました。apt-getではこれに加えて、パッケージの追加を伴うアップグレードも保留されます。Ubuntuにおいて、アップグレード時に別のパッケージが追加される代表格が、カーネルパッケージです。実はapt-get upgradeでは、OSのカーネルがアップグレードされないのです。apt-getでカーネルをアップグレードするには、dist-upgradeサブコマンドを使う必要があります。これはaptのfull-upgradeに相当するサブコマンドですが、名前が異なります。こうした違いにも注意してください。

3.7 apt update を省略するには

aptはパッケージのインストールやアップグレードといった操作をする前に、apt updateでパッケージ情報を更新しておく必要があります。そのため実行するコマンドが、updateとinstall/upgradeの二本立てになってしまうことを、面倒に思うかもしれません。特に Docker のイメージビルドでは、RUN 命令ごとにイメージレイヤーが作られてしまうため、以下のようにふたつのコマンドを連結している人も多いのではないのでしょうか^{*7}。

```
RUN apt-get update && apt-get install -y (パッケージ名)
```

Ubuntu 26.04 のaptには-Uというオプションが追加されています。例えば以下のようにコマンドを実行すると、installの実行前に、自動的にupdateを行います。

```
$ sudo apt install -U (パッケージ名)
```

apt updateを省略できるため、場合によっては便利かもしれません。活用してみてください。

^{*7} 前述の通り Dockerfile などの内部では、aptではなくapt-getの使用が推奨されているため、この例でもそのように記述しています。

第4章 ユニバーサルパッケージシステム「Snap」を使ってみよう

Ubuntu のリポジトリには、Debian GNU/Linux 譲りの豊富なパッケージが存在しますが、近年では OS が提供するパッケージの「使いづらさ」も目立ってきました。この章では従来のパッケージシステムが持っている問題と、その解決方法のひとつである「ユニバーサルパッケージシステム」について解説します。

4.1 Ubuntu のパッケージとソフトウェアのバージョン

Ubuntu は半年ごとに、新しいバージョンがリリースされます。そしてパッケージは OS を構成する一部として、各リリースごとにビルドし直され、専用のリポジトリで提供されています。Ubuntu では、通常のリリースでは 9 か月、2 年に一度リリースされる長期サポート版では 5 年のサポートが提供されています。この「OS のサポート」とは、そのリリース向けのパッケージ群に対して、セキュリティ修正やバグ修正のアップデートが提供されるかということを意味します。

ここで気をつけなければならないのは一度リリースされたパッケージは、ソフトウェアのバージョンそのものは上げないという点です。具体的な例として、4 年前にリリースされた Ubuntu 22.04 LTS の PHP のパッケージを見てみましょう。

Ubuntu 22.04 LTS は、2022 年 4 月にリリースされました。そのリリース時に提供されていた PHP のバージョンは 8.1 でした。そしてリリースから 4 年以上が経過した 2026 年 7 月現在も、Ubuntu 22.04 LTS のリポジトリにある PHP のバージョンは 8.1 のままです。

PHP 8.1 は、2021 年 11 月にリリースされたバージョンです。そして 2023 年 11 月に、開発元のアクティブサポートが終了しています。またセキュリティサポートは、2025 年 12 月に終了しています。Ubuntu 22.04 LTS は 2027 年まで、有償サポートを契約すれば 2032 年までサポートが提供される、現役の OS です。つまり PHP の開発元のサポートが先に打ち切られてしまっているわけで、どう考えても計算が合いません。

2026 年 7 月現在、PHP は最新の安定版として 8.5 がリリースされています。8.5 であれば 2029 年までセキュリティサポートが約束されていますから、PHP そのもののバージョンを都度上げていけばよいと考えるかもしれません。ですが Ubuntu では、リリース内でソフトウェアのバージョンそのものを上げるということは、原則として行いません。繰り返しになりますが、Ubuntu によって提供されているパッケージは、すべて OS を構成する一部です。そのため開発期間内に、十分なテストを行った後に提供されています。しかしリリース後に新しいバージョンを投入してしまうと、意図しない不具合を発生させてしまうことも考えられます。また迂闊に共有ライブラリのバージョンを上げてしまうと、他のアプリに異常をきたしたり、OS 全体の整合性を壊してしまう可能性もあります。こうした理由により、Ubuntu の安定性を維持するためにも、リリース後にソフトウェアの新バージョンを投入することは、原則としてできなくなっているのです*1。

そこで Ubuntu では、ソフトウェアのバージョンそのものは上げず、セキュリティ修正のみをパッチとしてパッケージに取り込む（バックポートする）という方式を採用しています。Ubuntu 22.04 LTS リリース当時の、PHP パッケージの正確なバージョンは 8.1.2-1ubuntu2 でした。そして 2026 年 7 月現在の PHP パッケージの正確なバージョンは 8.1.2-1ubuntu2.25 です。8.1.2 という部分が PHP のバージョンを表しているのですが、ここは変わらず、ハイフンより後ろのバージョンが上がっていることがわかるでしょう。これがセキュリティ修正を取り込み、Ubuntu 独自の変更バージョンを上げ

*1 いくつか例外はあります。

ているということを意味しているのです*²。ここが重要なのですが、仮に開発元のサポートが終了してしまっても、Ubuntu では独自にパッケージのメンテナンスを行います。PHP 8.1 のサポートは 2025 年に終了していますが、Ubuntu 22.04 LTS の PHP 8.1 は、2026 年になってもセキュリティサポートが、Ubuntu によって提供されているのです*³。

ちなみにこうした事情は Ubuntu に限らず、Stable リリース（バージョン X.X のような安定版を定期的にリリースするタイプのリリースモデル）を採用しているディストリビューションであれば、だいたい同じになっています。

■コラム: Ubuntu のリポジトリとサポート

長期サポート版の Ubuntu では、5 年のサポートが提供されると述べました。ですが実は、すべてのパッケージに対してセキュリティアップデートが提供されるわけではありません。Ubuntu が提供しているパッケージは、大きく **main**、**universe**、**restricted**、**multiverse** の 4 つのコンポーネントに分類されています。

表 4.1: Ubuntu のリポジトリコンポーネント

コンポーネント	内容物	ライセンス	メンテナンス	セキュリティアップデート
main	カーネルなどの主要パッケージ	OSS	Canonical	○
universe	おもに Debian GNU/Linux からインポートしたパッケージ	OSS	コミュニティ	×
restricted	グラフィックカードのドライバなど	非 OSS	Canonical	○
multiverse	利用条件や再配布条件に制限があるパッケージ	非 OSS	コミュニティ	×

セキュリティアップデートの提供が約束されているのは、Canonical 社がメンテナンスを担当している、main と restricted に限定されています。universe と multiverse のパッケージは、Ubuntu コミュニティによるベストエフォートでの対応となっています。そのため Ubuntu を長期に渡って安定運用させるためには、universe と multiverse のパッケージを採用するか、慎重に検討する必要があります。

なお有償サポートである **Ubuntu Pro***⁴を契約すると、universe と multiverse のパッケージに対しても、10 年間のセキュリティアップデートを受けられます。Ubuntu Pro は個人であれば 5 台まで無償で利用できるため、小規模なサーバー運用では、利用を検討してみるのもお勧めです。

*² 「あるソフトウェアのバージョン X.X に重大な脆弱性が発見された。修正するには X.Y にバージョンアップせよ」と言われることがあります。ですが Ubuntu の場合、「X.X」に修正パッチを当て、「X.X-2」のようなパッケージとして提供します。これを見た人が「バージョン X.X を使っている！ 脆弱性対策がなされていない！」のように勘違いするのは、Ubuntu あるあるです。修正パッチをバックポートしているケースでは、パッケージのバージョン番号を見ただけでは脆弱性の対策状況は判断できないという点を覚えておくといよいでしょう。

*³ 同様に、バージョン番号だけを見て「このソフトのバージョン X.X はもうサポートが終了している！ まだ使っているとはけしからん！」のように勘違いする人もよくいます。開発元のサポートと、ディストリビューションのサポートはまた別であることを覚えておきましょう。

*⁴ <https://ubuntu.com/pro>

4.2 Ubuntu のパッケージが抱える問題

Ubuntu が提供している公式パッケージを使えば、開発元のサポート等を気にすることなく、Ubuntu のサポートが切れるまで、特定のソフトウェアの特定のバージョンを使い続けることができます。ディストリビューターが長期間メンテナンスを継続してくれるというのは、非常に大きなメリットであるのは間違いありません。ですがこれが、昨今のアプリ開発事情と噛み合わなくなってきています。

特定のバージョンがメンテナンスされ続けるというのは、言い換えればアプリのバージョンが OS のライフサイクルに縛られるということも意味します。OS そのものは安定したものを長期間使い続けたくとも、アプリのバージョンはどんどん上げていきたいと考えるのが自然でしょう。ミドルウェアはともかく、ユーザーが直接触れるアプリであればなおさらです。ですが前述の通り、OS 全体の整合性や安定性の問題から、リリース後にアプリのバージョンを上げることはできないのです。

近年のアプリケーションの進化の早さには目を見張るものがあります。そのため APT を使えば多くのアプリが簡単にインストールできるというメリットを、望んだバージョンをインストールできないというデメリットが上回るケースも多くなってきてるのです。

4.3 ユニバーサルパッケージとは

従来の Linux のパッケージは、Ubuntu の Deb にしろ、Red Hat Enterprise Linux の RPM にしろ、そのディストリビューションの作りと密接に関連しています。そのため異なるディストリビューションはもちろん、異なるリリースのパッケージは基本的に混ざるな危険です^{*5}。

そこで、Linux カーネルが動いてさえいれば、どんなディストリビューションのどのリリースでも動く、統一的なパッケージを作ろうという試みが始まりました。これが**ユニバーサルパッケージ**です。ユニバーサルパッケージの実装には **Snap**、**Flatpak**、**AppImage** などがあり、Ubuntu ではこのうち Snap が採用されています。なお Snap は従来の APT/Deb を置き換えるものではありません。アプリの頻繁なアップデートに追従するなど、APT/Deb が苦手とする部分を補う目的で、併用することが前提となっています。

そもその話になりますが、リリース後にパッケージのバージョンを上げられない原因は、ひとえにパッケージに依存関係があるからに他なりません。仮に、あるソフトウェアが動作するために必要な要素を、すべて単一のパッケージ内に詰め込むことができれば、パッケージの依存関係をなくすることができないでしょうか。そして依存関係を持たず、自己完結しているパッケージであれば、自由に入れ替えられることにならないでしょうか。

ユニバーサルパッケージでは、環境依存をなくするため、ソフトウェアの動作に必要な環境一式を、パッケージ内に詰め込んでいます。そのためパッケージ間の依存関係を持たず、単一のパッケージをインストールするだけで、環境を問わず、同じソフトウェアを動かすことができるのです。ユニバーサルパッケージを使えば、OS の一部であった従来のパッケージを OS から切り離すことが可能となります。結果として OS のライフサイクルに縛られない、自由なバージョンアップが実現できるのです。

4.4 Snap を使いこなす

それでは実際に Snap の使い方を紹介しましょう。

4.4.1 WSL への Snap のインストール

現在の Ubuntu には、標準で Snap がインストールされています。例えばデスクトップ版の Ubuntu 26.04 LTS にプリインストールされている Firefox や Thunderbird は、従来の Deb ではなく、Snap からインストールされています。WSL の Ubuntu 26.04 LTS にも、同様に Snap がプリインストールされており、すぐに使いはじめることができます。

^{*5} 無理やりインストールすれば動いてしまうこともあります。

もしも何らかの理由で Snap がインストールされていない場合は、以下のコマンドで **snapt** パッケージをインストールしてください。

```
$ sudo apt install -U -y snapd
```

4.4.2 Snap パッケージの検索

Snap は`snap`コマンドとサブコマンドで操作します。使い勝手は`apt`コマンドとよく似ているため、難しい部分はないでしょう。よく使うであろうサブコマンドを表 4.2 にまとめました。

表 4.2: snap の主なサブコマンド

サブコマンド	用途
install	指定したパッケージをインストールする。
remove	指定したパッケージをアンインストールする。
list	インストール済みのパッケージの一覧を表示する。
find	キーワードをもとにパッケージを検索する。
info	指定したパッケージの詳細な情報を表示する。
refresh	パッケージをアップデートする。
switch	指定したパッケージのリリースチャンネルを切り替える。
disable	指定したインストール済みのパッケージを無効にする。
enable	指定した無効になっているパッケージを有効にする。

まずはどのようなパッケージがあるか検索してみましょう。パッケージの検索は`find`サブコマンドの後に、キーワードを指定して実行します。以下の例では、**kubernetes** 関連のパッケージを検索しています。

```
$ snap find kubernetes
```

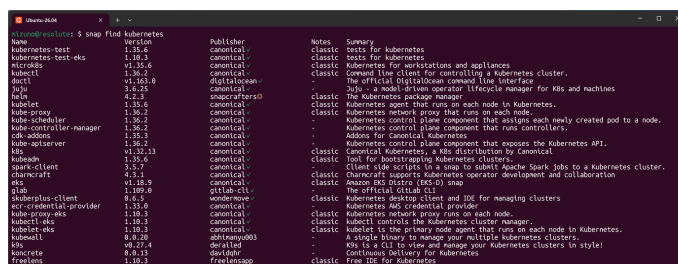


図 4.1: kubernetes 関連のパッケージを検索した例

条件にマッチしたパッケージの、パッケージ名、パッケージのバージョン、パブリッシャー名、ノート、パッケージの概要が表示されます。

Ubuntu の APT のリポジトリにあるパッケージは、すべて Ubuntu がビルドして公開している、いわゆる公式パッケージです。対して Snap のストアには、様々な開発者によるパッケージが混在しています。そのためパッケージの質も玉石混合となっています。パブリッシャー名の横に緑色のチェックマークがついているパッケージは、そのパブリッシャーが検証されていることを示しています。これらのパッケージは、信頼できるパッケージだと考えてよいでしょう。参考にしてください。

4.4.3 Snap パッケージのインストール

それでは実際に Snap パッケージをインストールしてみましょう。ここでは例として、Kubernetes のコマンドラインツールである `kubect1` をインストールします。パッケージのインストールには、`install` サブコマンドにパッケージ名を指定します。

```
$ sudo snap install kubect1 --classic
```

インストールには root 権限が必要なため、`sudo` を利用します。また `kubect1` パッケージのインストールには、`--classic` オプションが必要です。Snap ではセキュリティ的な理由から、各アプリからホストのリソースへのアクセスが制限されています。ですがアプリによっては、ホストのリソースへアクセスしなければならないものも存在します。こうしたアプリは、インストール時に `classic` オプションの指定が必要になります。`classic` オプションが必要かどうかは、`find` コマンドの結果の **Notes** の部分に、その記述があることで判断できます。

Kubernetes は進化が早いシステムのひとつで、おおよそ 4 か月ごとに新バージョンがリリースされます。それに合わせてコマンドラインツールも更新する必要がありますが、Ubuntu のリリースポリシー的に、APT のパッケージでは対応が難しいのです。ですが Snap を使えば、こうした進化の早いアプリであっても、簡単に管理できます。

4.4.4 リリースチャネルを切り替える

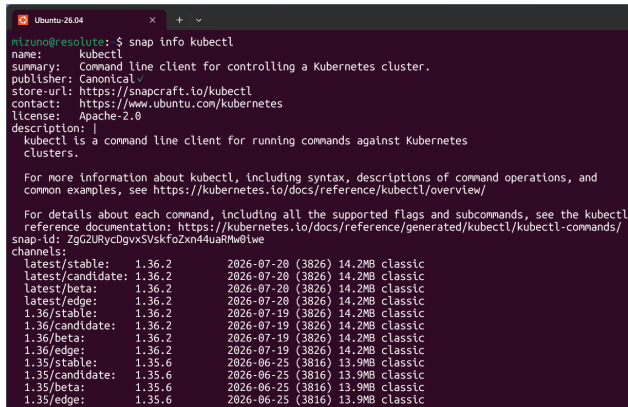
`kubect1` は、常に最新をインストールすればよいというものではなく、操作する Kubernetes クラスターのバージョンと揃える必要があります*⁶。そのため場合によっては、過去のバージョンをインストールしたいこともあるでしょう。

Snap にはリリースチャネルという概念があります。これはいわば、リポジトリを複数のチャネルに分けて、異なるバージョンを並行して配布する仕組みです。通常は `edge`、`stable`、`candidate`、`beta` の 4 つのチャネルが用意されています。これ以外にどのようなチャネルを用意し、どのように利用するかは、パブリッシャーに一任されています。特にチャネルを指定せずにインストールを行った場合は、`stable` チャネルからインストールが行われます。そのため `stable` チャネルでは、いわゆる安定版のリリースが配布され、`edge` チャネルや `beta` チャネルでは、開発中のバージョンやベータ版が配布されるのが一般的となっています。

そのパッケージにどのようなチャネルが存在するかは、`info` サブコマンドで調べることができます。

*⁶ 厳密に一致する必要はなく、クラスターのバージョンの ±1 であれば許容されます。

```
$ snap info kubectl
```



```

mizuno@resolute:~$ snap info kubectl
name:      kubectl
summary:   Command line client for controlling a Kubernetes cluster.
publisher: Canonical
store-url: https://snapcraft.io/kubectl
contact:   https://www.ubuntu.com/kubernetes
license:   Apache-2.0
description:
  kubectl is a command line client for running commands against Kubernetes
  clusters.

  For more information about kubectl, including syntax, descriptions of command operations, and
  common examples, see https://kubernetes.io/docs/reference/kubectl/overview/

  For details about each command, including all the supported flags and subcommands, see the kubectl
  reference documentation: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/
snap-id:   ZgGzUryCgVxSVskfoZKn44uabHw0Lwe
channels:
  latest/stable: 1.36.2      2026-07-20 (3826) 14.2MB classic
  latest/candidate: 1.36.2    2026-07-20 (3826) 14.2MB classic
  latest/beta: 1.36.2      2026-07-20 (3826) 14.2MB classic
  latest/edge: 1.36.2      2026-07-20 (3826) 14.2MB classic
  1.36/stable: 1.36.2      2026-07-19 (3826) 14.2MB classic
  1.36/candidate: 1.36.2    2026-07-19 (3826) 14.2MB classic
  1.36/beta: 1.36.2      2026-07-19 (3826) 14.2MB classic
  1.36/edge: 1.36.2      2026-07-19 (3826) 14.2MB classic
  1.35/stable: 1.35.6      2026-06-25 (3816) 13.9MB classic
  1.35/candidate: 1.35.6    2026-06-25 (3816) 13.9MB classic
  1.35/beta: 1.35.6      2026-06-25 (3816) 13.9MB classic
  1.35/edge: 1.35.6      2026-06-25 (3816) 13.9MB classic

```

図 4.2: kubectl パッケージのチャンネルを調べる

channels 以下が、kubectl パッケージに用意されたチャンネルです。kubectl では edge、stable、candidate、beta のチャンネルすべてで同じバージョンが配布されていること、latest（最新）では 2026 年 7 月現在の最新安定版である 1.36.2 が配布されていること、過去のバージョンはバージョン番号/stableといったチャンネルで配布されていることなどが見て取れます。

それではチャンネルを切り替えて、1 つ前のバージョンである 1.35.6 をインストールしてみましょう。switch サブコマンドで kubectl パッケージのチャンネルを切り替えます。

```
$ sudo snap switch kubectl --channel 1.35/stable
```

チャンネルを切り替えたら、refresh サブコマンドでパッケージを更新します。

```
$ sudo snap refresh
```

これでパッケージのダウングレードができます。もちろん、また latest チャンネルに切り替えて refresh すれば、バージョンの切り戻しも簡単です。

4.4.5 パッケージのアンインストール

Snap パッケージは、remove サブコマンドでアンインストールできます。kubectl パッケージをアンインストールする例は以下の通りです。

```
$ sudo snap remove kubectl
```

APT のような仕組みを使っていると、デスクトップのアプリケーションはもちろん、サーバーのミドルウェアや言語のランタイム、ライブラリなど、あらゆるソフトウェアが OS に縛られてしまいます。そこでアプリケーション実行環境をカプセル化し、OS のライフサイクルから切り離すことで、自由なバージョンアップを可能にするのがユニバーサルパッケージのメリットです。特に例として紹介した kubectl のように、進化が早く、特定のバージョンの使用が強制されるようなソフトウェアとの相性

は抜群だと言えるでしょう。Ubuntu では Snap を簡単に使い始められますので、ぜひ APT だけでなく、Snap の活用も検討してみてください。

第 5 章 Windows 上で Linux の GUI アプリを動かしてみよう

WSL は基本的にターミナルを経由したコマンドラインインターフェイス (CLI) で操作します。ですが Ubuntu にもグラフィカルなデスクトップ環境や、その上で動作する GUI アプリが数多く存在します。この章では Windows のデスクトップ上で、Linux 向けの GUI アプリを動かしてみます。

5.1 GUI アプリのインストールと起動

2026 年現在、Windows 11 上にインストールした WSL2 環境であれば、特別な設定などは必要ありません。Ubuntu に GUI アプリのパッケージをインストールしてターミナルから実行するだけで、GUI アプリを動かすことができます。それでは試してみましょう。以下のコマンドを実行してください。Linux で広く利用されているテキストエディタの **Gedit** をインストールします。

```
$ sudo apt install -U -y gedit
```

依存関係によって、かなり大量のパッケージがインストールされます。インストール完了までしばらく待ちましょう。インストールが完了し、プロンプトに復帰したら **gedit** コマンドを実行してください。

```
$ gedit
```

Windows のデスクトップ上に Gedit のウィンドウが表示されます。WSL 内で起動しているアプリケーションですが、Windows のデスクトップとシームレスに統合されているため、Windows のネイティブアプリと区別ができないほどです。

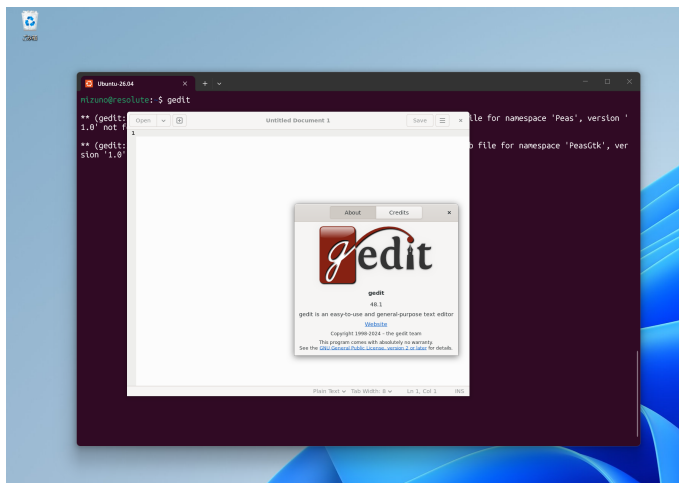


図 5.1: WSLg で起動した Gedit

またインストールした GUI アプリは、Windows のスタートメニューから直接起動できます。WSL 上で動いていることを意識せず、Windows アプリと同じように使えることがわかります。



図 5.2: スタートメニューから Ubuntu のアプリを起動できる

5.2 日本語フォントのインストール

Gedit で日本語のテキストを開くと、以下のようにフォントが文字化けし、いわゆる豆腐になってしまいます。

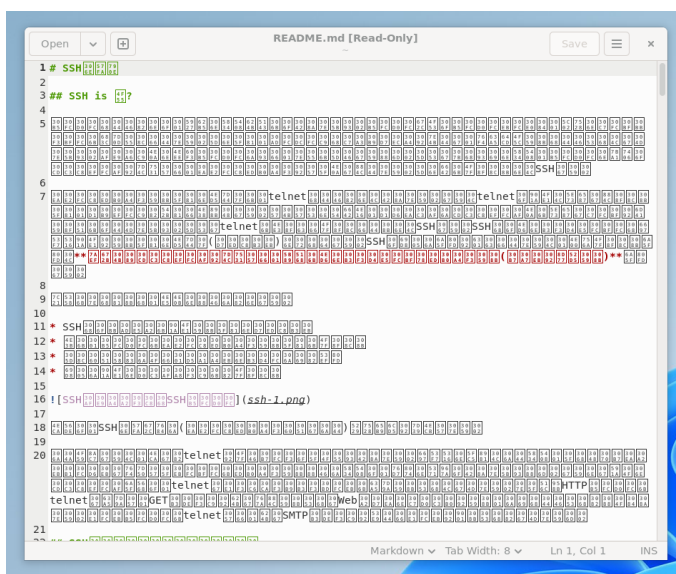


図 5.3: 日本語が豆腐になってしまう

これは WSL 版の Ubuntu では、デフォルトで日本語フォントが使われていないためです。というわけで日本語フォントを追加しましょう。当然ですが、フォントも APT で簡単にインストールできます。フォントは好みのものを使用して構いませんが、ここでは Noto フォントをインストールします。以下のコマンドを実行してください。

```
$ sudo apt install -U -y fonts-noto-cjk
```

フォントをインストールするだけで、日本語ファイル名が正しく表示されるようになります。

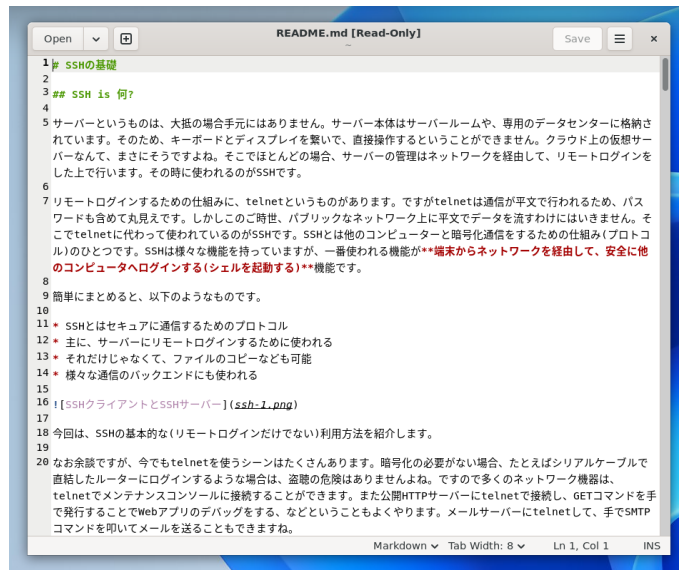


図 5.4: Noto フォントをインストールした状態

5.3 アプリケーションの日本語化

日本語は表示されるようになりましたが、アプリケーションそのものが英語のままでは、少し使いづらいでしょう。特に使い慣れていないアプリであれば、メニューなどは日本語化されているのが望ましいです。そこで以下のコマンドを実行して、翻訳データが含まれている、日本語の言語パックをインストールします。

```
$ sudo apt install -U -y language-pack-ja language-pack-gnome-ja
$ sudo update-locale LANG=ja_JP.UTF-8
```

インストールが完了したら、以下のコマンドでロケールを日本語に切り替えます。

```
$ sudo update-locale LANG=ja_JP.UTF-8
```

ここまでできたら、一度 Windows Terminal のタブを閉じ、新しくシェルを起動し直してください。新しいシェル上で `locale` コマンドを実行してみましょう。各種変数が `ja_JP.UTF-8` になっていることを確認してください。

```
$ locale
LANG=ja_JP.UTF-8
LANGUAGE=
LC_CTYPE="ja_JP.UTF-8"
LC_NUMERIC="ja_JP.UTF-8"
LC_TIME="ja_JP.UTF-8"
LC_COLLATE="ja_JP.UTF-8"
LC_MONETARY="ja_JP.UTF-8"
LC_MESSAGES="ja_JP.UTF-8"
LC_PAPER="ja_JP.UTF-8"
LC_NAME="ja_JP.UTF-8"
LC_ADDRESS="ja_JP.UTF-8"
LC_TELEPHONE="ja_JP.UTF-8"
LC_MEASUREMENT="ja_JP.UTF-8"
LC_IDENTIFICATION="ja_JP.UTF-8"
```

```
LC_ALL=
```

この状態で、再度 Gedit を起動してみましょう。メニューや各種メッセージが日本語に翻訳されているはずです。

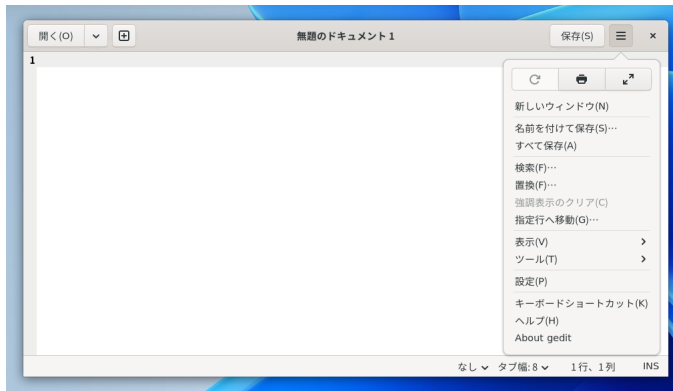


図 5.5: 日本語化された Gedit

5.4 WSL で日本語を入力するには

ここまでの作業で日本語の表示はできるようになりましたが、日本語の入力ができません。WSLg のアプリは Windows のデスクトップ上でシームレスに動作すると述べましたが、実は現時点では、Windows の IME を使って、WSLg のアプリに直接日本語の入力はできないのです。WSLg のアプリに日本語を入力するには、Linux 向けの日本語入力システムを、WSL の中で動かす必要があります。

まず以下のコマンドを実行して、インプットメソッドの **Fcitx5** と、変換エンジンの **Mozc** をインストールします。

```
$ sudo apt install -U -y fcitx5 fcitx5-mozc
```

続いて環境変数と、インプットメソッドの自動起動設定を行います。以下のコマンドを実行して、`/etc/profile.d/99-local.sh` というファイルを作成してください。

```
$ cat <<EOF | sudo tee /etc/profile.d/99-local.sh
export GTK_IM_MODULE=fcitx
export QT_IM_MODULE=fcitx
export XMODIFIERS=@im=fcitx
export DefaultIMModule=fcitx
fcitx5 --disable=wayland >/dev/null 2>&1 &
EOF
```

設定ファイルが作成できたら、以下のコマンドを実行してファイルの内容を読み込みます。あるいは Windows Terminal を一度終了してから、再起動しても構いません。

```
$ source /etc/profile.d/99-local.sh
```

日本語配列のキーボードを使っている場合は、キーボードの設定を変更してください。以下のように `fcitx5-configtool` コマンドを実行します。なお英語配列のキーボードを使っている場合、この手順は省略できます。

```
$ fcitx5-configtool
```

するとインプットメソッドの設定ウィンドウが開きます。

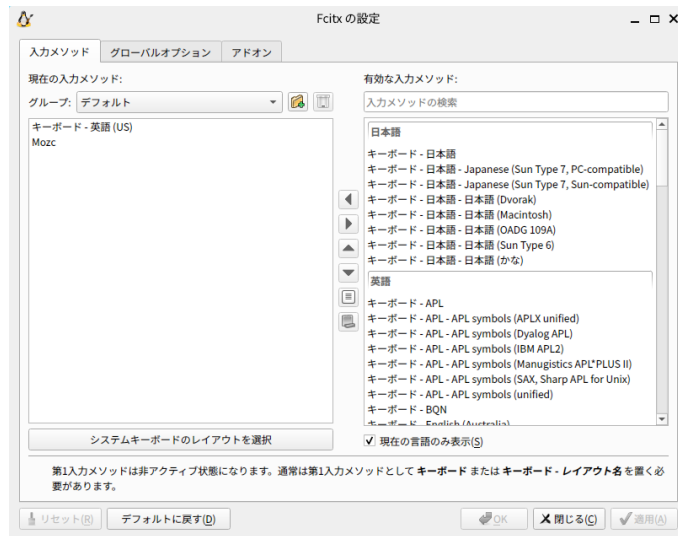


図 5.6: Fcitx の設定ウィンドウ

左側の「現在の入力メソッド」にある「キーボード - 英語 (US)」をダブルクリックして削除します。そして右側の「有効な入力メソッド」にある「キーボード - 日本語」をダブルクリックして、入力メソッドとして追加してください。最後に「OK」をクリックしてウィンドウを閉じます。

以後、WSLg の GUI アプリ上で **Ctrl+Space** キーか、**半角/全角** キーを押すと、日本語入力が ON になります。

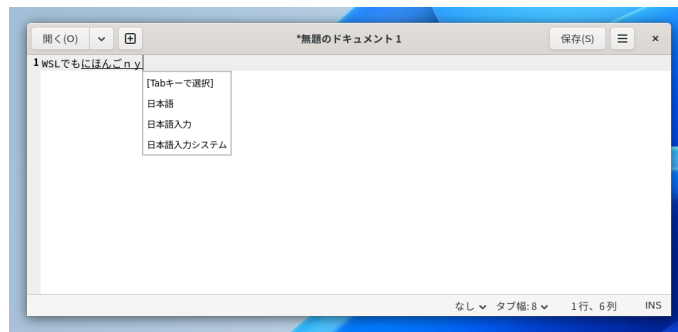


図 5.7: WSLg 上で GUI アプリに日本語入力をする

WSL に限らず、開発作業に Linux を使うのであれば CLI への習熟は必須です。とはいえ初心者の方は、すべての作業を CLI で行うのも難しいでしょう。そこで WSLg を使い、GUI を併用してみるのもお勧めです。

第 6 章 Linux の基礎コマンド群をマスターしよう

Coreutils とは、Unix ライクな OS に当然存在することが期待される、基本的なコマンドやユーティリティの詰め合わせです。この章では Linux の必修科目とも言える、Coreutils に含まれるコマンドについて解説します。

6.1 Coreutils とは

厳密な意味での Linux とは、OS のカーネルのみを指す名称です。カーネルは文字通り、OS のコアとなる非常に重要なプログラムですが、ユーザーがカーネルを直接操作するわけではありません。カーネルだけでは OS としては成立せず、ユーザーが操作するための、様々なアプリケーションやコマンドが別途必要となります。そのため一般的な Linux ディストリビューションには、最初から便利なコマンドが数多く用意されています。ファイルをコピーする `cp` コマンドや、ファイルの中身を表示する `cat` コマンドなどを使ったことがある方も多いでしょう。こうした基礎コマンド群は、**Coreutils** と呼ばれています。

Coreutils には、誰もが知っている定番から、こんな便利なものがあったのかと驚くようなものまで、多種多様なコマンドが存在します。Linux の CLI に習熟するには、なにはともあれ Coreutils を知るべきというのが筆者の考えです。もちろん WSL においても、この事情は変わりません。この章ではそうした 100 以上のコマンドのうち、よく使うものや、知っておくと便利そうだと筆者が考えたものに絞って紹介します。

6.2 Ubuntu と Coreutils

従来の Ubuntu では、Coreutils は `coreutils` というパッケージによって提供されていました。これは GNU が開発している Coreutils^{*1} をパッケージ化したもので、100 以上の基礎コマンド群が含まれていました。

ところが近年になり、これらのコマンドを現代的に作り直そうという機運が高まってきました。Coreutils に含まれるコマンドは歴史が長く成熟しているものの、C 言語ベースのためメモリ安全性の問題があります。その歴史的な背景からコードも巨大で複雑化しており、Windows をはじめとするクロスプラットフォーム対応も弱いなど、「今時のツールとしてはちょっと……」という部分が否定できないためです。

そこでより現代的な Coreutils を作るという思想のもと、Rust 言語で再実装を行っているプロジェクトが `utils coreutils`^{*2} です。そして Ubuntu 26.04 では、デフォルトで使われる Coreutils が、この `utils` 版に変更されました^{*3}。例えば `/usr/bin/cat` コマンドは `/usr/lib/cargo/bin/coreutils/cat` へのシンボリックリンクとなっています。同様に `ls` や `install` といったコマンドも、`/usr/lib/cargo/bin/coreutils` 以下の同名のコマンドへのリンクとなっています。これらの実体は `rust-coreutils` パッケージによって提供されています。

```
$ ls -l /usr/bin/cat
lrwxrwxrwx 1 root root 30 Mar 30 16:50 /usr/bin/cat -> ../lib/cargo/bin/coreutils/cat
```

`utils coreutils` は GNU Coreutils との完全互換を目標にしています。同じ名前のコマンドが同じオプションを持っており、出力までも完全に一致する前提です。もしも異なる挙動があったら、それは

^{*1} <https://www.gnu.org/software/coreutils/>

^{*2} <https://github.com/utils/coreutils>

^{*3} ただし `rm` や `cp` など、いくつかのコマンドは GNU 版を使うようになっています。

utils のバグとして扱われます。そのため実装が切り替わっていることを意識する必要はないのですが、現時点では 100% の互換性があるとはまだ言い切れません。そこで GNU 版の Coreutils も引き続きインストールされており、切り替えて使用できます。また何らかの理由で Coreutils を GNU 版に戻したい場合は、以下のコマンドを実行してください。

```
$ sudo apt install -y coreutils-from-gnu coreutils-from-utils- --allow-remove-essential
```

GNU 版のバイナリは `gnu-coreutils` パッケージによって提供されており、`/usr/bin` 以下に `gnu~` という prefix つきでインストールされています。`coreutils-from-utils` パッケージを `coreutils-from-gnu` と入れ替えることで、以下のようにシンボリックリンクの貼り替えが行われます。

```
$ ls -l /usr/bin/cat
lrwxrwxrwx 1 root root 6 Mar 30 16:50 /usr/bin/cat -> gnuccat
```

6.3 コマンドのマニュアルを読むには

Coreutils の各コマンドには、マニュアルが用意されています。マニュアルを読むには、`man` コマンドの引数に、調べたいマニュアル（コマンド）名を指定してください。例えば `cat` コマンドのマニュアルを読むには以下のように実行します。

```
$ man cat
```

なおコンテナなどの軽量な環境では、`man` コマンド自体がインストールされていないこともあります。そのような場合は `man` パッケージをインストールしてください。

```
$ sudo apt install -U -y man
```

前述の通り、Coreutils に含まれるコマンドは非常に多く、またそれぞれのコマンドの使い方も複雑なため、本記事ですべてを解説し切ることはできません。本書では概要に留めますので、ぜひ手元の WSL 環境でどのようなコマンドが存在するか調べ、`man` コマンドでマニュアルを読んでみてください。

6.4 各コマンドに共通のオプション

一般的に CLI のコマンドは、それぞれ独自のオプションを持っています。ですが Coreutils の各コマンドに、共通しているオプションもあります。以下のふたつのオプションは、覚えておくとよいでしょう。

6.4.1 --help オプション

文字通り、そのコマンドの簡易的なヘルプを呼び出すオプションです。`man` で読めるマニュアルよりも簡単な解説を、標準出力に表示できます。主要なオプションをちょっと調べたいような場合は、`man` を呼び出すより簡単で便利です。例えば `cat` コマンドのヘルプは以下のように呼び出せます。`man` と内容と比べてみましょう。

```
$ cat --help
```

6.4.2 --version オプション

そのコマンドの (Coreutils の) バージョンを表示するオプションです。トラブルに遭遇した際など、使っているコマンドのバージョンが重要な情報となります。なにか挙動がおかしいと思ったら、まずは使っているバージョンを確認してみましょう。

```
$ cat --version
```

6.5 ファイルを出力するコマンド

6.5.1 ファイルを結合して表示する「cat」

cat は Coreutils の中でも、トップクラスによく使われるコマンドのひとつでしょう。主にテキストファイルの中身を出力する目的で使用されています。以下は `example.txt` というファイルの中身を表示する例です。

```
$ cat example.txt
```

実は cat は **concatenate** の略で、連結するという意味を持っています。このコマンドの主目的は、実は複数のファイルを連結することです。引数に複数のファイルを指定すると、それらを繋げて表示します。以下のコマンドを実行すると、`example1.txt`、`example2.txt`、`example3.txt` の 3 つのファイルを繋げて表示できます。

```
$ cat example1.txt example2.txt example3.txt
```

以下のようにすると、連結したファイルを別名で保存できます。またテキストファイルだけでなく、バイナリファイルも連結できます。

```
$ cat example1.txt example2.txt example3.txt > concatenate.txt
```

6.5.2 ファイルの先頭部分だけを表示する「head」

head コマンドは、指定したファイルの先頭部分のみを表示するコマンドです。デフォルトでは先頭から 10 行を表示します。以下は `example.txt` の先頭 10 行のみを表示する例です。

```
$ head example.txt
```

`-n` オプションに続いて数値を指定することで、指定した行数のみを表示することもできます。また `-c` オプションに続いて数値を指定すると、指定したバイト数のみを表示できます。複数あるファイルの概要を確認したいような場合、`cat` ですべてを表示するのは冗長でしょう。そのような時によく利用します。

6.5.3 ファイルに行番号を付加する「nl」

`nl` コマンドは、`cat` と同じくファイルの中身を表示しますが、出力の際に行番号を付加します。以下は `example.txt` に行番号を付加して出力する例です。

```
$ nl example.txt
```

実は `cat` コマンドにも行番号を付加する `-b` オプションが存在します。そのため上記の `nl` コマンドの実行例は、以下の `cat` コマンドの実行例と同じになります。

```
$ cat -b example.txt
```

なお `nl` コマンドは、デフォルトで空行には行番号を付加しません。以下のように `-ba` オプションをつけると、空行にも行番号を付加します。この挙動は `cat` コマンドの `-n` オプションと同等です。

```
$ nl -ba example.txt  
$ cat -n example.txt
```

6.5.4 ファイルを逆順に表示する「tac」

`tac` コマンドも `cat` コマンドと同じく、ファイルの中身を出力します。ただし行単位で逆順に表示します。つまりファイルの末尾の行が 1 行目に、末尾の直前の行目が 2 行目に、となります。コマンド名はその挙動に由来し、`cat` をひっくり返したものです。

```
$ tac example.txt
```

6.5.5 ファイルの末尾部分だけを表示する「tail」

`head` コマンドと対になるのが `tail` コマンドです。このコマンドはファイルの末尾の 10 行のみを表示します。`-n` オプションや `-c` オプションも、末尾からカウントする点以外は `head` コマンドと同じです。以下は `example.txt` の末尾 10 行を表示する例です。

```
$ tail example.txt
```

`tail` コマンドがよく使われるのが、追記され続けるファイルの監視です。例えばサーバーのログファイルには、動作中のアプリの情報が書き込まれ続けています。`tail` コマンドに `-f` オプションをつけると、こうしたファイルをリアルタイムに監視し続けることができます。以下は `error.log` というログファイルを監視する例です。

```
$ tail -f error.log
```

またログファイルは、定期的にローテーションが行われるのが一般的です。監視中にローテーションされるファイルを追従するには、`--retry` オプションを併用してください。

6.6 ファイルを分割するコマンド

6.6.1 ファイルを特定の条件で分割する「csplit」

`csplit` コマンドは、行番号や正規表現を使ってファイルを分割するコマンドです。例えば以下は、`example.md` という Markdown ファイルを、レベル 2 の見出し（`##`2 つで始まる行）で分割します。またパターンの後には、分割を何回繰り返すかを指定します。ここでは「`{*}`」を指定し、ファイル末尾まで繰り返す指定をしています。

```
$ csplit example.md '/^## .*/' '{*}'
```

分割した結果は、カレントディレクトリに `xx00`、`xx01`、`xx02`……という連番のファイルとして保存されます。そのため以下のように `cat` コマンドで連結することで、元のファイルを復元できます。なおこの `xx` にあたるプレフィックスは、オプションで変更することも可能です。

```
$ cat xx*
```

6.6.2 ファイルを分割する「split」

`split` コマンドは `csplit` コマンドと同様に、ファイルを分割するコマンドです。ただし `csplit` コマンドが正規表現で柔軟に条件を指定できたのに対し、一定のサイズや行数ごとに分割します。分割単位は、デフォルトでは 1000 行となっています。また `-l` オプションで、行数を指定できます。以下は `example.txt` を 10 行ごとに分割する例です。

```
$ split -l 10 example.txt
```

分割した結果は `csplit` と多少異なり `xaa`、`xab`、`xac`……という名前のファイルとして保存されます。ですが `cat` コマンドで連結できる点は同様です。

6.7 ファイルの中身をチェックするコマンド

6.7.1 MD5 チェックサムを計算する「md5sum」

MD5 チェックサムの計算と照合を行うのが `md5sum` コマンドです。以下のコマンドを実行すると、`example.txt` ファイルのチェックサムを計算できます。

```
$ md5sum example.txt
```

`-c` オプションを指定すると、ファイルから計算したチェックサムを読み込み、ファイルとの照合が行えます。以下のコマンドでは `MD5SUMS` というファイルから、対象のファイル名とそのチェックサムを読み込み、実際のファイルと比較し、破損や改竄がないかを確認します。

```
$ md5sum -c MD5SUMS
```

6.7.2 SHA-1 チェックサムを計算する「sha1sum」

SHA-1 チェックサムの計算と照合を行うのが `sha1sum` コマンドです。オプションも含め、使い方は `md5sum` と同様です。

6.7.3 SHA-256 チェックサムを計算する「sha256sum」

SHA-256 チェックサムの計算と照合を行うのが `sha256sum` コマンドです。オプションも含め、使い方は `md5sum` と同様です。Ubuntu の ISO イメージファイルのチェックサムは、SHA-256 で提供されていますので、ここではそのチェック方法を紹介しましょう。まず以下のコマンドで、サーバー版 Ubuntu 26.04 の ISO イメージと、チェックサムが書かれたファイルをダウンロードします。

```
$ wget https://ftp.ubuntu.com/ubuntu-releases/resolute/ubuntu-26.04-live-server-amd64.iso
$ wget https://ftp.ubuntu.com/ubuntu-releases/resolute/SHA256SUMS
```

以下のように `-c` オプションと、引数に `SHA256SUMS` ファイルを指定してコマンドを実行します。なお `SHA256SUMS` ファイルには、デスクトップ版とサーバー版両方の ISO イメージファイルのチェックサムが記述されています。今回の例ではデスクトップ版の ISO イメージファイルはダウンロードしていないため、警告が表示されてしまいます。それを抑制するため `--ignore-missing` オプションもつけておくとよいでしょう。

```
$ sha256sum -c --ignore-missing SHA256SUMS
ubuntu-26.04-live-server-amd64.iso: OK
```

上記のように、ファイル名の後に OK と表示されれば、検証は完了です。インターネットからダウンロードしたファイルは、チェックサムを用いて破損や改竄がないか念のため確認するとよいでしょう。

6.7.4 ファイルの行数、文字数、バイト数を出力する「wc」

`wc` コマンドは、ファイルの行数、文字数、バイト数をカウントして出力します。以下は `example.txt` の行数、文字数、バイト数をカウントする例です。

```
$ wc example.txt
```

`-l` オプションをつけると、このうち行数のみを出力します。これを利用し、他のコマンドの結果をパイプし、項目数をカウントする目的で使うこともよくあります。例えば以下は、`error.log` というログファイルから `ERROR` という文字列が含まれる行を検索し、その行数をカウントすることで、発生したエラーの件数を数えています。

```
$ grep 'ERROR' error.log | wc -l
```

6.8 ファイルを編集/変換/ソートするコマンド

6.8.1 BASE64 エンコード/デコードを行う「base64」

`base64` は、BASE64 のエンコードとデコードを行うコマンドです。最近ではデータを BASE64 エンコードして環境変数にセットしたり、逆に JSON で取得したデータの中身が BASE64 エンコー

ド済みだったりといったケースもよくあるでしょう。そうしたデータを作る/読む際に便利です。以下はhelloという文字列をBASE64 エンコードし、aGVsbG8Kというエンコード結果を得る例です。

```
$ echo 'hello' | base64
aGVsbG8K
```

-dオプションをつけると、逆にBASE64 エンコードされたデータをデコードします。先ほど得られたaGVsbG8Kというエンコード済みの文字列をデコードするには、以下のようにコマンドを実行します。

```
$ echo 'aGVsbG8K' | base64 -d
hello
```

6.8.2 ファイルの中身をシャッフルする「shuf」

shufはファイルの中身を行単位でシャッフルした結果を表示します。-nオプションは出力する最大行数の指定です。以下はexample.txtの中からランダムな一行を取り出す例です。

```
$ shuf -n 1 example.txt
```

6.8.3 ファイルの中身をソートする「sort」

sortはファイルの中身を行単位でソートするコマンドです。後述するuniqのように、Linux コマンドの中には、元ファイルがソート済みであることを前提としているものも多く存在します。そうしたコマンドを実行する前処理としてもよく使われるコマンドです。以下はexample.txtをソートする例です。

```
$ sort example.txt
```

よく併用されるオプションとして、文字列ではなく数値としてソートする-nオプション、逆順に表示する-rオプション、ソートした上で重複行を削除する-uオプションなどがあります。またソフトウェアのバージョン番号や月の名前などを正しくソートすることもできます。

6.8.4 文字の置換と削除を行う「tr」

trは文字の置換と削除を行うコマンドです。以下はhelloという文字列を、大文字に変換する例です。なおtrコマンドはファイルを直接開けないため、別のコマンドの出力をパイプして読み込む必要があります。

```
$ echo 'hello' | tr '[:lower:]' '[:upper:]'
HELLO
```

-dオプションをつけると、指定した文字を削除できます。以下はexample.txtから改行を削除します。

```
$ cat example.txt | tr -d '\n'
```

6.8.5 ファイルから重複を削除する「uniq」

`uniq`はファイルから重複した行を削除します。なおこの際、元のファイルはあらかじめソート済みである必要があります。そのため`sort`コマンドと組み合わせて使うのが一般的です。ですが前述の通り`sort`コマンドには`uniq`コマンドと同様の動作をする`-u`オプションが存在するため、重複を削除するだけであれば、`uniq`コマンドの出番はあまりないかもしれません。以下のふたつのコマンドは、どちらも同じ結果となります。

```
$ sort example.txt | uniq
$ sort -u example.txt
```

`uniq`コマンドには、重複した件数をカウントする`-c`オプションが存在します。このオプションを使うと、例えば Web サーバーのアクセスログから、アクセス数が多いページのランキングを作るといったこともできます。

6.9 ファイルやディレクトリに関するコマンド

6.9.1 ファイルやディレクトリをコピーする「cp」

ファイルやディレクトリをコピーする、Linux において基本中の基本とも言えるコマンドが`cp`です。もっとも基本的な使い方は、1 番目の引数に指定したファイルを、2 番目の引数に指定したファイルとしてコピーすることです。例えば以下は、`example.txt`というファイルを、`copy.txt`というファイルにコピーします。

```
$ cp example.txt copy.txt
```

ディレクトリをコピーする時は`-r`オプションを併用します。以下は`srcdir`というディレクトリを、`destdir`という名前でコピーします。なおこの際、`destdir`というディレクトリが既に存在した場合は、その中に`srcdir`がコピーされます。このように同じコマンドであっても、状況によって結果が異なることがあります。

```
$ cp -r srcdir destdir
```

複数のファイルをまとめてディレクトリにコピーすることもできます。`cp`コマンドは絶対にマスターすべきコマンドな上、非常に多くのオプションや使い方があるため、ぜひマニュアルを熟読してください。

6.9.2 ファイルのコピーと変換を行う「dd」

`dd`コマンドは、ファイルのコピーと変換を行います。主に単なるファイルのコピーよりも、USB メモリに ISO ファイルを書き込んだり、ディスク全体をゼロで上書きしたりする際によく利用されます。例えば以下のように実行すると、Ubuntu の ISO イメージファイルを USB メモリに書き込んで、インストールメディアを作ることができます。

```
$ sudo dd if=ubuntu-26.04-live-server-amd64.iso of=USBメモリのデバイスファイル名 bs=1M
```

`conv` オプションを指定すると、コピー中にファイルの中身の変換が行えます。以下は `example.txt` を `output.txt` としてコピーする例ですが、`conv=ucase` が指定されているため、テキストファイル中のアルファベット小文字が、すべて大文字に変換されてコピーされます。

```
$ dd if=example.txt of=output.txt conv=ucase
```

6.9.3 属性を指定してコピーする「install」

Linux システムの管理においては、ファイルやディレクトリをシステムの領域にコピーした上で、所有者やパーミッションを適切なものに変更する必要がある場合があります。前述した `cp` コマンドでコピーを行った後に、後述する `chown` コマンドや `chmod` コマンドで所有者やパーミッションを変更するのが一般的ですが、これを同時に行えるのが `install` コマンドです。

引数に、コピー元ファイルとコピー先ファイルを指定する点は、`cp` コマンドと同じです。その際 `-o` オプションで所有者、`-g` オプションでグループ、`-m` オプションでパーミッションを指定できます。以下は `script.sh` というスクリプトファイルを、所有者を `root`、グループを `ubuntu`、パーミッションを `755` として、`/usr/local/bin` ディレクトリにコピーする例です。

```
$ sudo install -o root -g ubuntu -m 755 script.sh /usr/local/bin/script.sh
```

また `-d` オプションを指定すると、その属性でディレクトリを作成できます。この際は、コピー元ファイルの指定は不要です。作成するディレクトリ名のみを指定してください。

6.9.4 リンクを作成する「ln」

`ln` コマンドは、あるファイルに対してハードリンクを作成します。ハードリンクとは、同じファイルシステム上に作られた、ファイルの別名です。以下は `example.txt` に対して、`link.txt` という別名を作成する例です。以後、`example.txt` と `link.txt` は同一のファイル（の実体）を指すようになります。

```
$ ln example.txt link.txt
```

`ln` コマンドに `-s` オプションをつけると、ハードリンクではなくシンボリックリンクを作成できます。シンボリックリンクもハードリンクと同様に、あるファイルに対してつけられた別名です。ハードリンクがファイルシステム上のメタ情報を元にリンクを辿るのに対し、シンボリックリンクはパスを頼りにリンクを辿ります。

ハードリンクはファイルシステム上の番号を元にリンクを辿るため、リンク先のファイルの名前を変えたり、ディレクトリを移動してもリンクが切れません。その反面、ファイルシステムを跨いでリンクを作成できないという特徴があります。シンボリックリンクはその逆で、ファイルシステムを跨いでリンクを作れる反面、リンク先の名前（パス）が変わると、リンクが切れてしまうという特徴があります。用途に応じて使い分けてください。

6.9.5 ディレクトリの内容を表示する「ls」

ディレクトリの中にあるファイルの一覧を表示するのが`ls`コマンドです。おそらく Linux の CLI において、世界で一番多く実行されているコマンドではないでしょうか。

よく使うオプションとして、隠しファイルを表示する`-a`オプション、詳細なリスト表示をする`-l`オプション、時系列に並び変える`-t`オプション、逆順で表示する`-r`オプション、数値を読みやすく表示する`-h`オプションなどがあります。以下はホームディレクトリの中身を、隠しファイルも含め、古い順に詳細なリスト表示をする例です。

```
$ ls -ltar ~
```

`ls`コマンドは基本中の基本の上、便利なオプションが色々あるため、一度はマニュアルを読んでみましょう。

6.9.6 ディレクトリを作成する「mkdir」

新しいディレクトリを作成するのが`mkdir`コマンドです。引数に作成したいディレクトリ名を指定してください。なお`-p`オプションを指定すると、親ディレクトリが存在しなかった場合は、親ディレクトリも含めて一気に作成できます。

以下はホームディレクトリ内に、`~/work/wsl/document`という階層のサブディレクトリを作成する例です。`-p`オプションが指定されているため、`work`ディレクトリや`work/wsl`ディレクトリが存在しなかった場合は、同時に作成されます。

```
$ mkdir -p ~/work/wsl/document
```

6.9.7 ファイルを移動する「mv」

ファイルやディレクトリを移動するのが`mv`コマンドです。`cp`コマンドと同様に、移動元のファイル名と、移動先のファイル名を引数に指定して使います。以下は`example.txt`というファイルを、`work`ディレクトリ内に移動する例です。

```
$ mv example.txt work/example.txt
```

副次的な使い方として、ファイルの名前の変更があります。ファイルの名前の変更とは、同じディレクトリ内で（別名のファイルとして）移動を行っていることに他なりません。例えば以下は、`example.txt`の名前を`renamed.txt`に変更しています。

```
$ mv example.txt renamed.txt
```

複数のファイルを同時に移動することもできます。ですがこの場合、移動先の名前を個別に指定することができません。そのため移動先は既存のディレクトリに限られ、かつファイル名を同時に変更することはできません。`cp`コマンド同様、引数の指定の仕方いくつかのパターンがあり、それによって挙動が微妙に変化するため、一度はマニュアルを読んでおきましょう。

6.9.8 ファイルを削除する「rm」

ファイルを削除するのが `rm` コマンドです。削除したいファイルを引数に指定して使います。ファイルは複数指定することもできます。以下は `example1.txt` と、`example2.txt` という 2 つのファイルを削除する例です。

```
$ rm example1.txt example2.txt
```

`rm` コマンドでは、ディレクトリを削除することはできません。ですが `-r` オプションをつけることで、ディレクトリとその中身を再帰的に削除することができます。以下は `work` というディレクトリと、その中身をすべて削除する例です。

```
$ rm -r work
```

ですが CLI にはゴミ箱がないため、うっかり間違ったディレクトリを削除してしまうと大事故に繋がります。ディレクトリを削除する時は、くれぐれも注意してください。

6.9.9 ディレクトリを削除する「rmdir」

ディレクトリを削除するコマンドが `rmdir` です。中身ごと消す `rm -r` コマンドと違い、`rmdir` コマンドはディレクトリの中身が空でないで失敗します。以下は `work` ディレクトリを削除する例ですが、`work` ディレクトリ内にファイルが 1 つでも残っていると、このコマンドはエラーになります。

```
$ rmdir work
```

前述の通り、`rm -r` は大事故を起こす可能性があります。重要なサーバー上で作業を行うような場合は、ディレクトリは親ごと一気に削除することは避け、確認しながら個別にファイルを削除し、中身が空になったことを確認してから `rmdir` コマンドを使うことを推奨します。

6.9.10 カレントディレクトリを表示する「pwd」

`pwd` はカレントディレクトリを表示するコマンドです。単体ではそれほど意味のあるコマンドではありませんが、別のコマンドにディレクトリのパスを引き渡したいような場合によく使われます。

具体的な例を挙げると、Docker でコンテナを起動する際に、カレントディレクトリをワーキングディレクトリとして、コンテナ内にマウントしたいというケースがあります。ですがカレントディレクトリは環境によって異なるため、スクリプトなどにハードコーディングできません。そこで `pwd` コマンドを使うことで、環境による違いを吸収し、コマンドを一般化できます。以下の `docker` コマンドは、カレントディレクトリをコンテナ内の `/var/lib/work` にマウントする例です。

```
$ sudo docker run -v $(pwd):/var/lib/work container-name
```

6.9.11 ファイルやディスクを安全に消去する「shred」

`rm` コマンドでファイルを削除しても、ディスク上には 0 と 1 のデータの羅列が残っており、やり方によってはデータを復元することができます。ですが機密データを破棄するような場合、これでは都合が悪いです。 `shred` コマンドは、ファイルやディスクを安全に消去するコマンドです。

以下はexample.txtというファイルをshredコマンドで削除する例です。-vオプションをつけることで、詳細な進捗が表示されます。ここからわかる通り、ファイル上に3回ランダムなデータを上書きしてから削除を行っています。

```
$ shred example.txt
shred: example.txt: pass 1/3 (random)...
shred: example.txt: pass 2/3 (random)...
shred: example.txt: pass 3/3 (random)...
```

削除対象にディスクのデバイスファイルを指定することで、HDDやSSD全体を消去することもできます。また-zオプションをつけると、乱数を書き込んだ後に、全体にゼロを上書きします。NSA（米国家安全保障局）ではデータの消去に際し、ディスク全体に2回乱数を書き込んだ後に、ゼロを書き込むことを推奨しています。これをshredコマンドで実行するには、回数を指定する-nオプションと-zオプションを組み合わせ、以下のように実行します。

```
$ sudo shred -v -n 2 -z /dev/ディスクのデバイスファイル名
```

6.10 ファイルの属性を変更するコマンド

6.10.1 ファイルの所有グループを変更する「chgrp」

Linuxにおいて、ファイルやディレクトリには所有者と所有グループが設定されています。そして所有者、所有グループ、その他の第三者それぞれに、アクセス権限が設定されています。chgrpはそのうち、所有グループを変更するためのコマンドです。引数に変更後のグループ名と、対象のファイルを指定して実行します。

```
$ chgrp ubuntu example.txt
```

ただし上記のように所有グループが変更できるのは、対象のファイルを自分自身が所有しており、変更後の所有グループにも自分が所属している場合に限りです。それ以外のファイルの所有グループを変更するには、root権限が必要となりますので注意してください。

なお-Rオプションをつけてディレクトリを指定すると、ディレクトリ内のファイルとサブディレクトリに対し、再帰的に変更を適用できます。

6.10.2 ファイルのパーミッションを変更する「chmod」

Linuxのファイルには、所有者、所有グループ、第三者それぞれに読み込み、書き込み、実行の3つの権限が個別に設定されています。これをパーミッションと呼びます。そしてchmodはファイルのパーミッションを変更するためのコマンドです。

設定するパーミッションは、数値とシンボルによる指定が可能です。よく目にするのが、読み込み、書き込み、実行をそれぞれ二進数のフラグに置き換え、それを8進数に変換して表す方式です。例えば以下のコマンドは、example.txtのパーミッションを644、すなわち所有者は読み書き可能、所有グループと第三者は読み込みのみ可能に設定します。

```
$ chmod 644 example.txt
```

数値の具体的な意味については、以下のコマンドでmanセクション2の、chmodシステムコールを参照してください。


```
$ man 2 chmod
```

シンボルモードは、読み込み (r)、書き込み (w)、実行 (x) のパーミッションを、所有者 (u)、所有グループ (g)、第三者 (o)、すべてのユーザー (a) の誰に対して設定するかを、文字で指定するモードです。またこれ以外にも、`SETUID`と`SETGID`という特殊なビットや、削除制限フラグや`スティッキービット`といった属性を設定できます。これらについては本記事では省略しますので、詳しくは`chmod`のマニュアルを参照してください。

6.10.3 ファイルの所有者を変更する「chown」

`chgrp` コマンドがグループのみであったのに対し、`chown` コマンドはファイルの所有者と所有グループの両方を変更するコマンドです。基本的な使い方は`chgrp` コマンドと同様ですが、所有者と所有グループを:もしくは.で区切って指定する点が異なります。また所有者を変更するため、実行には root 権限が必要となります。以下は`example.txt`の所有者を root、所有グループを ubuntu に設定する例です。

```
$ sudo chown root:ubuntu example.txt
```

なお:の前部分を省略することで、`chgrp`と同様に所有グループだけを変更することもできます。

6.10.4 ファイルのタイムスタンプを変更する「touch」

`touch` コマンドは、引数に指定したファイルのタイムスタンプを変更するコマンドです。以下は`example.txt`のタイムスタンプを現在時刻に変更する例です。前後で`ls` コマンドを実行してみると、タイムスタンプが変更されていることが確認できます。

```
$ ls -l example.txt
-rw-rw-r-- 1 mizuno mizuno 4096  1月 15 14:55 example.txt

$ touch example.txt

$ ls -l example.txt
-rw-rw-r-- 1 mizuno mizuno 4096  1月 15 16:57 example.txt
```

なお存在しないファイルを指定すると、その名前で空のファイルを作成します。どちらかと言うと、こちらの目的で利用されることが多いため、空ファイルを作成するコマンドだと誤解されているかもしれません。

6.11 ユーザーとシステムに関するコマンド

6.11.1 システムのアーキテクチャを表示する「arch」

`arch` はシステムのアーキテクチャ (x86_64 や arm64 など) を表示するコマンドです。アーキテクチャによって動作するバイナリは異なります。最近では x86_64 だけでなく、arm アーキテクチャの CPU が使われるサーバーも増えてきましたから、現在のシステムのアーキテクチャを確認する必要性に遭遇することも多いのではないのでしょうか。またインストールスクリプトの中などで利用し、アーキテクチャによってインストールするパッケージを切り替えるといった用途にも利用できます。

```
$ arch
x86_64
```

6.11.2 時刻を表示する「date」

`date` コマンドは、文字通り時刻を表示するコマンドです。引数なしで実行すると現在の時刻を表示します。

```
$ date
2026 年  7 月 22 日 水曜日 13:53:27 JST
```

`-d` オプションを使うことで、任意の時刻を、非常に柔軟に指定することもできます。以下のコマンドは、現在の 3 日後の時刻を表示する例です。

```
$ date -d '3 days'
2026 年  7 月 25 日 土曜日 13:53:14 JST
```

`+` に続いてフォーマット指示子を指定することで、出力フォーマットを指定することもできます。以下は次の日曜日 (next sunday) の日付を、`YYYY/MM/DD` のフォーマットで表示する例です。

```
$ date -d 'next sunday' '+%Y/%m/%d'
2026/07/26
```

`date` はこうした性質を利用し、時刻を元にした文字列を組み立てることに使われることもよくあります。

6.11.3 所属しているグループ名を表示する「groups」

`groups` コマンドは、引数に指定したユーザーが、所属しているグループ名を表示します。引数を省略した場合は自分自身の情報を表示します。

```
$ groups
mizuno adm dialout cdrom floppy sudo audio dip video plugdev users netdev
```

6.11.4 ユーザー ID やグループ ID を表示する「id」

`id` コマンドは、引数に指定したユーザーのユーザー ID や、所属するグループ ID を表示します。引数を省略すると、自分自身の情報を表示します。Linux ではグループごとに権限が分けられており、特定のグループに所属しないと行えない処理というものも存在します。例えば Ubuntu では、`sudo` コマンドは `sudo` グループに所属していなければ使えません。そうした場合の所属グループの確認などに利用できます。

```
$ id
uid=1000(mizuno) gid=1000(mizuno) groups=1000(mizuno),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo)
```

なお名前を表示する `-n` オプションと、すべてのグループ ID を表示する `-G` オプションを併用すると、前述の `groups` コマンドと同じ挙動になります。

```
$ id -n -G
```

6.11.5 システム情報を表示する「uname」

`uname` はシステム情報を表示するコマンドです。オプションなしで実行すると、システム名（Linux 環境であれば Linux、macOS であれば Darwin など）のみを表示します。`-m` オプションでアーキテクチャ（x86_64 や arm64 など。arch コマンドと同じ挙動になる）、`-r` オプションでカーネルのリリース、`-v` オプションでカーネルのバージョンを表示します。`-a` オプションをつけるとすべての情報を表示するため、対話的に利用するのであれば、こちらを利用しておけばよいでしょう。以下は WSL の Ubuntu 26.04 で `uname` コマンドを実行した例ですが、WSL のカーネル上で動いているため、カーネルのリリースが `microsoft-standard-WSL2` となっていることがわかります。

```
$ uname -a
Linux resolute 6.18.33.2-microsoft-standard-WSL2 #1 SMP PREEMPT_DYNAMIC Thu Jun 18 21:54:43 UTC 2026
```

またシステム名を表示できるため、このコマンドをスクリプトの中で利用し、Linux と macOS で処理を分岐するインストールスクリプトを実装するといったこともできます。

6.11.6 システムの稼動時間を表示する「uptime」

`uptime` コマンドは、現在時刻、システムが起動してからの時間、ログインユーザー数、ロードアベレージを表示するコマンドです。この中でも特に重要なのが、システムが起動してからの時間です。これはシステムが動き続けている限り、常に増え続け、減ることはありません。そのためこの値が減少していたら、システムが意図せず再起動したということを意味します。サーバーが再起動した疑いがある時は、このコマンドを実行してみましょう。

以下は `uptime` コマンドの実行例です。このサーバーは起動してから 9 日ほど経過していることがわかります。

```
$ uptime
15:25:40 up 9 days, 4:51, 1 user, load average: 0.01, 0.04, 0.00
```

6.11.7 ログイン中のユーザー情報を表示する「who」

`who` コマンドは、システムにログイン中のユーザー情報を表示します。ログイン中のユーザー名とあわせて、ログイン時刻と使用している仮想端末、リモートから接続している場合は IP アドレスも表示されます。

```
$ who
mizuno pts/0 2026-07-22 13:53 (192.168.1.141)
```

`-b` オプションをつけると、最後にシステムが起動した時刻を表示します。

```
$ who -b
system boot 2026-07-19 06:00
```

6.11.8 自分のユーザー名を表示する「whoami」

`whoami` コマンドは、自分のユーザー名を表示します。対話的に Linux 環境にログインしているのであれば、自分のユーザー名がわからなくなるということは、あまりないでしょう。これも前述の `pwd` コマンドと同様に、スクリプト内に埋め込んで、ユーザーや環境によって異なるパラメータを抽象化するような際に利用されることが多いコマンドです。

```
$ whoami
mizuno
```

6.12 プロセスに関するコマンド

6.12.1 プロセスにシグナルを送る「kill」

実行中のプロセスにはシグナルという仕組みを用いて、様々なイベントの発生を通知できます。プロセスはシグナルを受信すると、そのシグナルの種類に応じて、一時停止、再開、終了といった処理を行います。そしてプロセスに任意のシグナルを送れるコマンドが `kill` です。

`kill` コマンドは、送信するシグナルの種類と、送信先のプロセス ID を指定して使います。例えば以下は、プロセス ID が 1000 のプロセスに対し、KILL シグナル（強制終了）を送信しています。

```
$ kill -KILL 1000
```

暴走したプロセスを強制終了する最後の手段として、KILL シグナルの送信はよく使われます。覚えておくとよいでしょう。なおシグナルについて詳しくは、以下のコマンドで `man` セクション 7 の `signal` を参照してください。

```
$ man 7 signal
```

6.12.2 処理を一時停止する「sleep」

複数のコマンドを連続して実行していると、コマンドとコマンドの間にインターバルを設けたい場合がよくあります。`sleep` は指定した時間の間、処理を一時停止するコマンドで、主にシェルスクリプトの中で利用されます。引数に停止したい時間を指定して実行してください。以下のコマンドは、60 秒間処理を一時停止する例です。

```
$ sleep 60
```

数字の後には `s`、`m`、`h`、`d` というサフィックスをつけることができ、それぞれ秒、分、時間、日を表します。サフィックスを省略した場合は、`s` が指定されたものとして扱われます。上記の 60 秒のスリープは、以下のように書くこともできます。

```
$ sleep 1m
```

6.12.3 タイムアウトを指定してコマンドを実行する「timeout」

`timeout` は単独で使うコマンドではなく、別のコマンドにタイムアウトを設定するためのコマンドです。例えば `yes` コマンドは、`y` という文字を出力するコマンドですが、`Ctrl+C` キーが押されるまで無限に実行を続けます。ですが `timeout` を設定して実行することで、指定した時間が経過した時点で、自動的にコマンドの実行を打ち切ることができます。以下のコマンドを実行すると、3 秒間だけ `yes` コマンドが実行され、タイムアウトを迎えると自動的に終了します。

```
$ timeout 3 yes
```

第 7 章 シェルをカスタマイズして、コマンドラインを便利に使う

どのようなアプリにも、個人用の設定というものが存在します。そして自分用に細かくカスタマイズすることで、作業効率が大幅に上がることも決して珍しくありません。この章では CLI 操作の要となる標準シェル（Bash）の便利なカスタマイズを紹介します。

7.1 Bash の設定ファイルと読み込み順

Linux をはじめとする Unix ライクな OS では、OS やアプリの設定の多くはテキストファイルとなっています。Bash の設定もまた、テキストで記述されています。まずは Bash の設定ファイルから見ていきましょう。

Bash の設定ファイルは大きく、システム全体で利用される共通の設定ファイルと、個人ごとの設定ファイルに分けられます。システム全体で利用される設定ファイルが `/etc/profile` です。対して各ユーザーのホームディレクトリに作られる、個人用な設定ファイルは `.bash_profile`、`.bash_login`、`.profile`、`.bashrc` などに分けられています。ではカスタマイズの設定は、具体的にどのファイルに書いたらよいのでしょうか。

Bash はその起動のされ方によって、**ログインシェル**と、それ以外の**対話的シェル**に分けられます。仮想コンソールや SSH でログインしたときに起動されるシェルがログインシェルです。そして `bash` コマンドを実行するなどして起動された、それ以外のシェルが対話的シェルとなります。どのような違いがあるかと言えば、起動時に読み込まれる設定ファイルが異なります。

Bash はログインシェルとして起動されると、まず `/etc/profile` を読み込みます。その後にホームディレクトリから `.bash_profile`、`.bash_login`、`.profile` の順にファイルの存在を調べ、最初に見つかったものだけを読み込みます。そしてログインシェルでない対話的シェルとして起動した場合は、かわりにホームディレクトリの `.bashrc` を読み込みます。ここまでが標準的な Bash の挙動です。

Ubuntu では、少し事情が異なります。まずシステムワイドな設定ですが、`/etc/profile.d` というディレクトリが用意されています。そして `/etc/profile` の中で、このディレクトリ内にある、拡張子が `.sh` のファイルをすべて追加で読み込むように設定されています。そのため Ubuntu では、システムワイドな Bash の設定は、適切な粒度で別ファイルに分割できるのです。これによってメンテナンス性も向上します*1。

続いて個人用の設定ファイルですが、Ubuntu では各ユーザーのホームディレクトリに、デフォルトで `.profile` と `.bashrc` のみが作成されます。そして `.profile` の中で `.bashrc` を読み込むように設定されています。つまりログインシェルであるか対話的シェルであるかにかかわらず、`.bashrc` が必ず読み込まれるため、すべての設定はここに書くのがお勧めです。なお `.bash_profile` や `.bash_login` といったファイルを作ってもいけないという点に注意してください。これらのファイルは `.profile` よりも優先順位が上のため、もしファイルを作ってしまうと、デフォルトの設定が読み込まれなくなってしまいます。

結論をまとめると、Ubuntu における Bash の設定は以下のようになります。

- 複数のユーザーが利用するようなサーバーで、全ユーザーに共通のカスタマイズを施したい場合は、`/etc/profile.d` 以下に設定ファイルを作成すればよい
- 個人用の設定はすべて `.bashrc` に書けばよい

*1 日本語入力の設定で行った、環境変数とインプットメソッドの自動起動設定がまさにこれです。

なお Bash の設定ファイルの中身は、Bash がそのまま実行可能なシェルスクリプトそのものです。今回は具体的な設定内容の紹介に留め、シェルスクリプトの文法や理屈については解説しませんが、興味があれば専門の文献などに当たってみてください。

7.2 Bash をカスタマイズしよう

それではここからは、筆者が実際に行っているオススのカスタマイズを紹介していきます。

7.2.1 シェルオプションを使う

Bash には様々なオプションが用意されています。このオプションを設定するコマンドが `shopt` です。まずは実行してみましょう。

```
$ shopt
array_expand_once      off
assoc_expand_once      off
autocd                 off
bash_source_fullpath    off
cdable_vars            off
(... 略...)
```

`shopt` コマンドを実行するとシェルオプションの一覧と、現在の有効/無効 (on/off) が表示されました。各シェルオプションを有効にするには、`-s` オプション^{*2}に続き、シェルオプション名を指定します。またシェルオプションを無効にするには `-u` オプションとシェルオプション名を指定します。

見ての通り、非常に多くのシェルオプションが用意されていますが、その中でもおすすめなのが `autocd` です。これは `cd` コマンドを入力せず、ディレクトリ名のみでカレントディレクトリを移動できるシェルの機能です。試しに有効化してみましょう。

```
$ shopt -s autocd
```

以下のように、ディレクトリ名のパス（ここでは `/usr/bin`）を入力するだけで、内部的に `cd` コマンドが実行されます。

```
$ /usr/bin
cd -- /usr/bin

$ pwd
/usr/bin
```

Linux の CLI において、ワーキングディレクトリを変更する `cd` コマンドは、何度も繰り返し実行するコマンドです。これを省略できるのは、一見地味ですが、慣れると手放せなくなる便利機能です。ぜひ試してみてください。

以下のコマンドを実行すると、`.bashrc` の末尾に `shopt -s autocd` という一行を追記します。以後 Bash を起動すれば、自動的に `autocd` が有効になります。

```
$ echo 'shopt -s autocd' >> ~/.bashrc
```

`autocd` 以外のシェルオプションについても、マニュアル^{*3}を参考に色々試してみるとよいで

^{*2} まぎらわしいですが、ここで言う「オプション」は、コマンドラインに指定するオプションのことです。

^{*3} https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html

しょう。

7.2.2 記録できる履歴の件数を増やす

Bash の機能でもっとも便利なものとは何かと言えば、**履歴**だと筆者は考えています。Bash は過去に実行したコマンドの履歴を保存しており、簡単に呼び出すことができるのです。試してみましょう。まず以下のように `ls` コマンドを `-a` オプションつきで実行してください。

```
$ ls -a
```

通常通りコマンドが実行されて、カレントディレクトリの内容が表示されます。再びプロンプトが表示されたら、カーソルキーの `↑` か、`Ctrl+P` キーを押してみましょう。直前に実行した `ls -a` が、プロンプトの後に自動的に入力されます。つまり `↑` キーと `Enter` キーだけで、以前に実行したコマンドを再実行できるというわけです。なお履歴から呼び出したコマンドは、キーボードから入力を終えて、`Enter` キーの押下を待っているのと同じ状態になっています。つまりこの状態でカーソルキーの `←` と `→` や `BackSpace` キーを使い、一部だけを編集可能です。履歴からコマンドを呼び出し、オプションや引数だけ書き換えて再利用もできるのです。

```
$          ← プロンプトが表示されている状態で↑キーか Ctrl+P を押す
$ ls -a    ← 直前に実行したコマンドが入力された状態となる。
```

さらに `↑` キーを続けて押せば、より古いコマンドを順番に呼び出すことができます。また `↓` キーか `Ctrl+N` キーを押すと、新しい方向に履歴を辿れます。うっかり `↑` キーを押しすぎてしまったら、`↓` キーを押して戻りましょう。`ls` 程度のシンプルなコマンドであれば、それほど恩恵はないでしょう。ですが複雑なコマンドの場合、毎回キーボードからコマンドを入力していると、時間もかかりますし、タイプミスによって実行に失敗してしまうこともあるでしょう。過去に一度でも実行したコマンドは、なるべく履歴を活用することを推奨します。

履歴というのは、コマンドの実行効率を上げる資産であり、老舗のウナギ屋のタレのようなものです。Linux を使い込めば使い込むほど、その大切さが身にしみてくるでしょう。実際筆者も、新しい PC をセットアップしたら、まず最初にシェルの履歴ファイルをコピーして移植するほどです。Ubuntu のデフォルトの設定では、履歴は直近から 2000 件を `.bash_history` というファイルに保存しており、そのうち新しい方から 1000 件を Bash 起動時に読み込みます。ですが 1000 件では、まったく足りないかと筆者は考えています。そこで履歴の保存件数を増やすカスタマイズを推奨します。

Bash の履歴の件数は、`.bashrc` の 20 行目付近で設定されています。以下の部分です。

```
HISTSIZE=1000
HISTFILESIZE=2000
```

`HISTFILESIZE` が `.bash_history` に保存される件数で、`HISTSIZE` が Bash の起動時に読み込まれる件数です。ざっくりですが、どちらも 100000 くらいにしておけば十分でしょう。テキストエディタで該当部分を書き換えてから、Bash を再起動してください。

7.2.3 履歴の前方検索を有効にする

前述のカスタマイズで、履歴を 10 万件保存できるようになりました。ですが 10 万回前に実行したコマンドを呼び出したい場合に、`↑` キーを 10 万回押すのは非現実的です。そこで活用したいのが履歴の検索です。プロンプトが表示されている状態で `Ctrl+R` キーを押してください。するとプロンプト

が`reverse-i-search`という表示に変わります。ここに検索したい文字列を入力すると、履歴の中からその文字列を含むコマンドを検索できるのです。

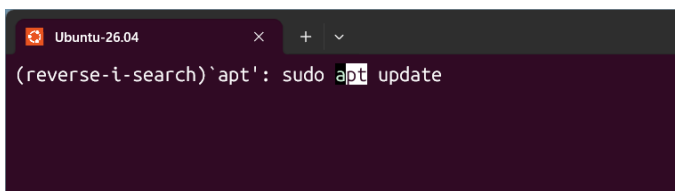


図 7.1: Ctrl+R を押した状態で「apt」と入力した例

履歴の検索はインクリメンタルサーチとなっているため、検索文字を長く入力するほど候補を絞り込めます。履歴の中に複数の候補がある場合は、文字の入力中に追加でCtrl+Rを押すことで、次の候補を呼び出せます。目的のコマンドが見つかったらEnterキーを押せばコマンドを実行できる点は、↑キーによる履歴の呼び出しと同様です。

↑キーに対する↓キーのように、履歴の検索でもCtrl+Rを押しすぎてしまった場合は、Ctrl+Sキーを押すことで、新しい履歴に戻ってくることができます。ところがUbuntuのデフォルトの状態では、このキーは動作しません。正確に言うと、このキーを押すとターミナルの出力が停止して、操作できなくなってしまうのです。この状態になってしまったらCtrl+Qキーを押しましょう。出力を再開できます。

正直なところ、ターミナルの出力を停止したいシーンというのはほとんどないでしょう。そこでこのキーバインドを解除して、履歴の順方向検索を有効にしましょう。以下のコマンドを実行して、`.bashrc`の末尾に`stty`コマンドの実行を追加します。ファイルを書き換えたら、Bashを再起動しておきましょう。

```
$ echo 'stty stop undef' >> ~/.bashrc
```

以後は履歴検索中にCtrl+Sを押すと、プロンプトの表示が`i-search`に変化し、順方向の検索が動作するようになります。

履歴の検索を利用すれば、大昔に実行したコマンドや、名前の一部だけしか覚えていないコマンドであっても、簡単に探し出すことができます。直前に実行したコマンドを繰り返す時は↑キーが便利ですが、過去のコマンドを再実行したい時は検索を活用するのがお勧めです。

7.2.4 履歴のあいまい検索を可能にする

履歴は部分一致で検索されるため、`ls -la`という履歴は`ls`、`-la`、`ls -l`のいずれで検索してもヒットしますが、`ls -a`ではヒットしません。このように、オプションの並び順が変わっただけでヒットしなくなるのは、少し使いづらいでしょう。そこでもう少し柔軟な検索が可能になるカスタマイズを紹介します。まず以下のコマンドで、あいまい検索フィルタである`fzf`^{*4}のパッケージをインストールしてください。

```
$ sudo apt install -U -y fzf
```

以下のコマンドを実行し、`.bashrc`の末尾に設定を追記してBashを再起動してください。

^{*4} <https://github.com/junegunn/fzf>

```
$ echo 'source /usr/share/doc/fzf/examples/key-bindings.bash' >> ~/.bashrc
```

再起動した Bash でCtrl+Rキーを押すと、通常の履歴検索のかわりに__fzf_history__というシェール関数が呼び出されるようになっています。この関数は Bash の履歴を表示するhistoryコマンドの出力をフィルタであるfzfコマンドに渡し、あいまいな検索を可能にしています。通常の履歴の検索と同様に、検索文字列を入力してみましょう。候補が絞り込まれていきますが、その挙動が「あいまい検索」になっていることがわかるでしょう。

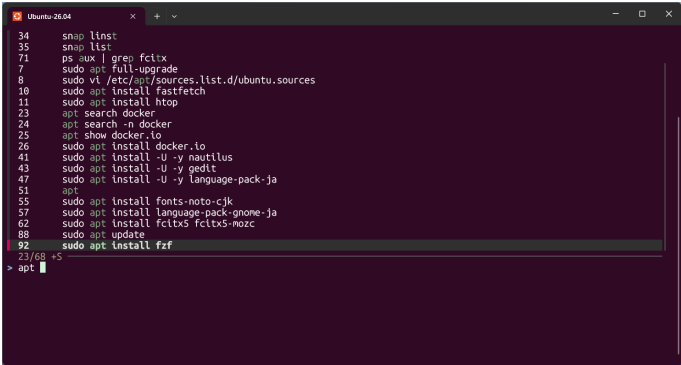


図 7.2: fzf を使って履歴を検索した状態

履歴がある程度絞り込めたら、リストの中から↑キーと↓キーで履歴を選択し、Enterキーで実行できます。ヒットした複数の候補が一覧表示されるため、一行ずつしか表示できなかった通常の検索と比較して、飛躍的に視認性も向上しています。fzfを使った履歴のあいまい検索は、筆者にとっては手放せない定番のカスタマイズとなっています。

7.2.5 ターミナル間で履歴を共有する

Linux の CLI を使いこなすようになると、複数のターミナルを起動して、異なる作業を同時並行することもよくあります。ですが標準状態の Bash では、ターミナル間で履歴を共有できません。例えばターミナル A であるコマンドを実行し、その直後にターミナル B で↑キーを押しても、ターミナル A の履歴は読み込めないのです。これは非常に不便です。そこで以下のコマンドを実行し、.bashrcの末尾に設定を追加してください。

```
$ echo 'export PROMPT_COMMAND="history -n; history -w; history -c; history -r; $PROMPT_COMMAND"' \>> ~/.bashrc
```

環境変数「PROMPT_COMMAND」には、シェルがプロンプトを表示する直前に実行するコマンドを定義できます。ここでは以下の 4 つのコマンドを順次実行するよう設定しています。

表 7.1: 実行しているコマンド

コマンド	挙動
history -n	履歴ファイルからまだ読み込んでいない履歴を読み込む。
history -w	現在の履歴をファイルに書き込む。
history -c	履歴をクリアする。
history -r	ファイルから履歴を読み込む。

つまりコマンドを実行する（次のプロンプトが表示される）たびに

- 別のシェルによって更新されている可能性のある履歴をファイルから追加で読み込む
- 現在の履歴の内容を履歴ファイルに書き込む
- 現在の履歴をクリアしてファイルから読み込み直す

という処理を行っています。これをすべてのシェルで行うことで、ターミナル間で履歴を半リアルタイムに同期できるというわけです。

注意点として、履歴ファイルを介した履歴の同期は、プロンプトが表示される直前に行われるという点です。つまりターミナル A で実行した直前のコマンドを、ターミナル B からすぐ呼び出したい場合は、ターミナル B で一度 **Enter** キーを押して、プロンプトを再表示する必要があります。

7.2.6 二段プロンプトのすすめ

Ubuntu の Bash のプロンプトには、デフォルトでユーザー名@ホスト名: カレントディレクトリ\$と表示されています。コマンドを実行する度に表示されるプロンプトは、現在の CLI の状態を確認するという意味で、非常に重要な情報源です。ここに様々な情報を追加するカスタマイズ、例えばプロジェクトのディレクトリ内では、現在の Git のブランチを表示したり、使用中の AWS のプロファイル名を表示したりなどは、シェルの定番のカスタマイズとなっています。

Bash のプロンプトは、環境変数 `PS1` でコントロールできます。実際にやってみましょう。`.bashrc` の末尾に、以下を追記してください。`git` コマンドを実行して、現在のブランチを表示するシェル関数を定義し、その関数をプロンプト文字列の中に埋め込んでいます。これは Ubuntu のデフォルトのプロンプト設定の末尾に、`$_current_branch` の部分を足しています*5。

```
function _current_branch() {
    _git_branch=$(git branch --show-current 2>/dev/null) && echo "[$_git_branch] "
}

export PS1='\[\e]0;\u@\h:\w\a\]${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\]
:\[\033[01;34m\]\w\[\033[00m\] $_current_branch)\$ '
```

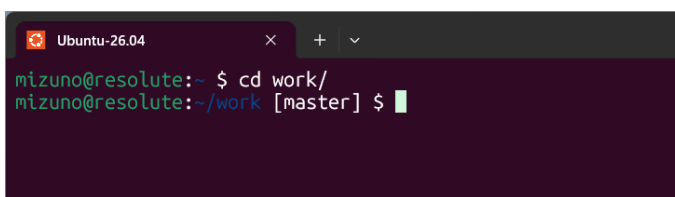


図 7.3: Git 管理下のディレクトリでは、プロンプトの最後にブランチが表示されるようになる

プロンプトのカスタマイズは便利なので、とにかく色々な情報を表示したくなります。しかしプロンプトが長くなると、相対的にコマンドを入力するスペースが狭くなり、使い勝手が低下してしまいます。そこで筆者が強く推奨するのが「二段プロンプト」です。文字通りプロンプトの中に改行を埋め込むことで、プロンプトを二段にしてみようという発想です。先ほど `.bashrc` に記述したプロンプトの設定を、以下のように書き換えてください。

*5 折り返し記号 (↵) の部分には改行を入れず、長い一行で記述してください。

```
function _current_branch() {
    _git_branch=$(git branch --show-current 2>/dev/null) && echo "[$_git_branch] "
}

function _1st_line() {
    DATE="[$(date '+%Y-%m-%d %H:%M:%S')]"
    WIDTH=$(tput cols)

    printf "%$(( $WIDTH - ${#DATE} - 1 ))s" | tr ' ' '-'
    echo " $DATE"
}

export PS1='$_1st_line)\[\e]0;\u@\h:\w\a\}${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@
\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\] $_current_branch)\$ '
```

新しく、プロンプトの一段目を表示するシェル関数を定義しています。具体的な処理としては、`date` コマンドで現在時刻を、`tput` コマンドでターミナルの横幅を取得しています。そして現在時刻の文字数をターミナルの幅から引き（実際にはスペースを入れるため、もう1文字ぶんを引いています）、その数だけ `printf` コマンドでスペースを出力し、`tr` コマンドでスペースをハイフンに置換したものを、現在時刻と合わせて出力しています。この関数を、プロンプト文字列の先頭に埋め込みました。関数の最後で時刻を表示している `echo` コマンドが改行を出力するため、プロンプト内で改行されるというわけです。セパレーターによって視認性が向上するだけでなく、コマンドを実行した時刻がターミナル上に残るため、記録としても便利です。

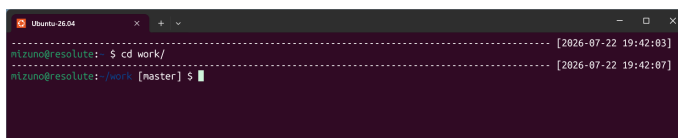


図 7.4: 二段プロンプトの例

プロンプトには様々な特殊文字^{*6}を指定できます。マニュアルを参考に、自分だけの使いやすいプロンプトを作ってみてください。

^{*6} <https://www.gnu.org/software/bash/manual/bash.html#Controlling-the-Prompt>

第 8 章 ターミナルマルチプレクサで、コマンドラインをさらに快適にしよう

Windows Terminal は非常に高機能なターミナルエミュレータですが、あくまで GUI のアプリのため、シェルの内部セッションの管理には手が出せないという問題もあります。そこで便利なのが「ターミナルマルチプレクサ」です。

8.1 ターミナルマルチプレクサとは

ターミナルマルチプレクサとは、仮想ターミナルを管理するためのソフトウェアです。複数のターミナルをセッションという単位で管理し、それらを自由に切り替えたり、分割したりを可能にします。代表的な実装としては GNU Screen^{*1}や tmux^{*2}が有名です。しかし、これらのソフトウェアは非常に高機能である反面、設定や使いこなしが難しいという欠点があります。特に初心者にとっては、複雑なキーバインドや設定ファイルの記述が大きな障壁となるでしょう。

そこで登場したのが Byobu^{*3}です。Byobu は、Ubuntu 開発者でもある Dustin Kirkland によって開発された、Ubuntu 発のソフトウェアです。もともと Byobu は **screen-profiles** という名前で開発がスタートしました。その名の通り、最初は GNU Screen の使いやすい設定セットを提供するためのものだったのです。これが GNU Screen だけでなく tmux にも対応し、さらに進化を遂げたものが現在の Byobu です。この名前はもちろん、ジャパニーズトラディショナルなマルチパネルスクリーンである屏風に由来しています。

8.2 Byobu のインストール

WSL 環境の 26.04 では、デフォルトでインストール済みとなっています。もし何らかの理由でインストールされていない場合は、以下のコマンドを実行してください。

```
$ sudo apt install -U -y byobu
```

8.3 Byobu の設定

8.3.1 基本的な起動方法

Byobu は同名の `byobu` コマンドで起動します。

```
$ byobu
```

起動すると、ターミナルの最下部にカラフルなステータスラインが表示されます。このステータスラインには、システムの様々な情報（Ubuntu のバージョン、ロードアベレージ、CPU 使用率、メモリ使用率、日時など）が表示されています。

^{*1} <https://www.gnu.org/software/screen/>

^{*2} <https://github.com/tmux/tmux/wiki>

^{*3} <https://www.byobu.org/>

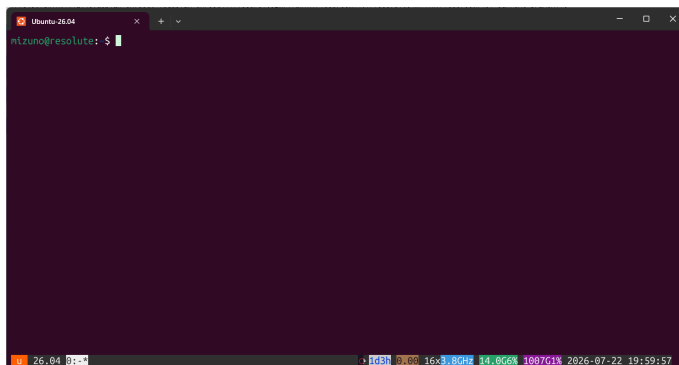
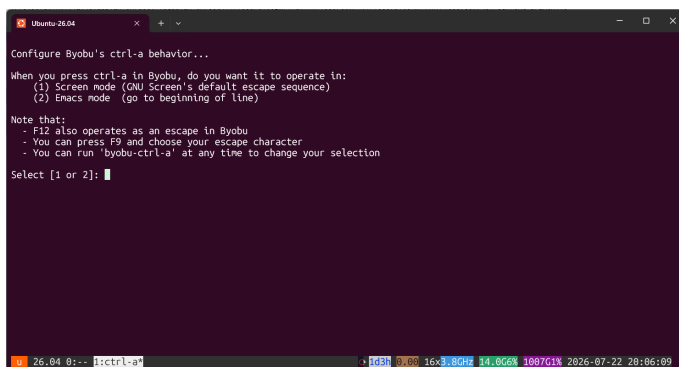


図 8.1: 起動したデフォルト状態の Byobu

8.3.2 エスケープキーの設定

Byobu では、様々な機能をキーボードショートカットで呼び出します。特に重要なのが Prefix キーと呼ばれるエスケープキーです。Prefix キーは Byobu の使い勝手を決めるキーな上、どのキーを割り当てるかは個人の好みが強くと反映されます。Byobu を使いはじめするには、まず Prefix キーの設定から行いましょう。

Byobu には `byobu-ctrl-a` というコマンドが用意されています。はじめて Byobu を起動した状態では、このコマンドが `Ctrl+A` キーに割り当てられています。そのため `Ctrl+A` をはじめて押すと、以下の画面が表示されます。

図 8.2: Byobu を起動して `Ctrl+A` を押すと、キーをどのように扱うか訊かれる

`byobu-ctrl-a` は、`Ctrl+A` を一般的なシェルと同様にカーソルの行頭移動として使うか、あるいは Byobu の Prefix キーとして使うかの選択を行なうためのコマンドです。Screen モードを選択すると、`Ctrl+A` が Prefix キーに設定されます。対して Emacs モードを選択すると、`Ctrl+A` がカーソルの行頭移動になります。`Ctrl+A` での行頭移動は頻繁に利用するため、2を入力して Emacs モードを選択しておく方が筆者は好みです。

Emacs モードを選択した場合、Prefix キーとして `F12` キーが設定されます。ですが `F12` というキーは非常に押しにくく、またキーボードによってはファンクションキーが存在しないこともあるでしょう。そこでより使いやすいキーに変更することを推奨します。`F9` キーを押すか、`byobu-config` コマンドを実行して Byobu の設定画面を表示し、「エスケープシーケンスの変更」を選択します。ここで自分にとって使いやすいキーを設定しましょう。筆者のお勧めは `Ctrl+Z` です。

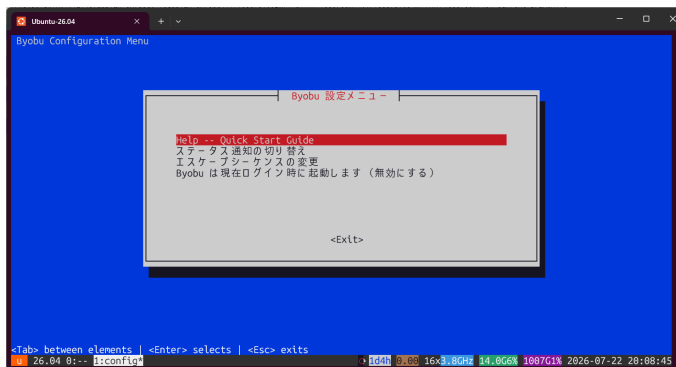


図 8.3: Byobu の設定画面

なお筆者が考える、Ctrl+Zをエスケープキーにするメリットは以下の通りです。

- 左手だけで押しやすい
- シェルのCtrl+Zの機能（ジョブのサスペンド）はそれほど頻繁には使わないため、奪ってしまっても差し支えない
- 本来の機能（サスペンド）を使いたい時は、Ctrl+Zを2回続けて押せばよいいため、サスペンドができなくなるわけではない
- むしろ誤ってCtrl+Zに触れてしまい、ジョブをサスペンドさせてしまうリスクが減るとも言える

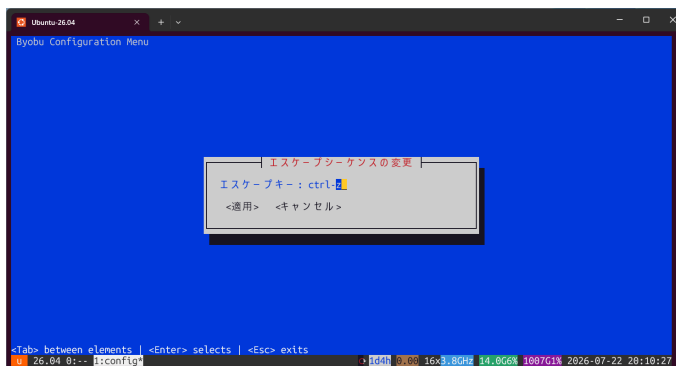


図 8.4: Prefix キーを Ctrl+Z に変更する

8.3.3 ステータスラインの設定

Byobu のステータスラインには、システムの様々な情報が表示されています。デフォルトでは左から順に Ubuntu ロゴ、Ubuntu のバージョン、ウィンドウ一覧、アップデートできるパッケージ数と再起動の要求、起動時間、ロードアベレージ、CPU コア数×周波数、メモリ容量と使用率、ストレージ容量と使用率、現在日時が表示されています。これらの表示項目は Byobu の設定画面から変更できます。必要な情報を追加、あるいは不要な情報を削除して、ステータスラインを自分好みにカスタマイズしてみましょう。F9キーもしくはbyobu-configコマンドで設定画面を表示させたら、「ステータス通知の切り替え」を選択します。

ステータスラインに表示できる項目がリストアップされます。現在「*」がついている項目が、ステータスラインに表示されます。表示したい項目をカーソルキーで選択したら、スペースキーを押して

ください。また再度スペースキーを押すことで「*」が消え、その項目を非表示にできます。

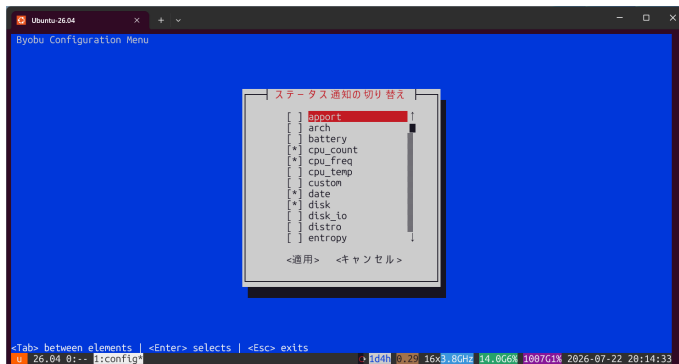


図 8.5: ステータスラインに表示する項目を選択する

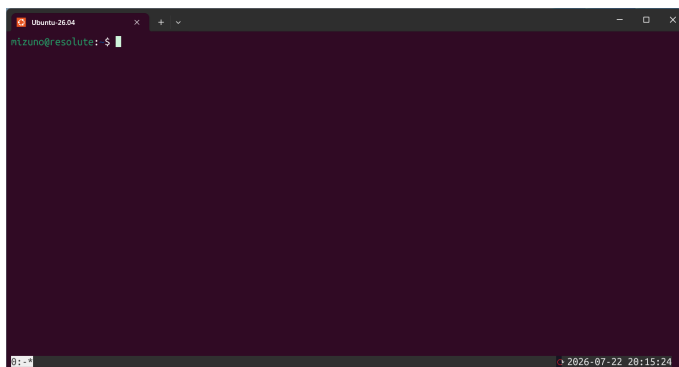


図 8.6: ウィンドウ一覧、アップデートの有無、現在時刻だけに減らしてみた例

8.4 Byobu の機能

それでは Byobu の具体的な機能を解説しましょう。

8.4.1 ウィンドウ管理

Byobu にはウィンドウという概念があります。これは Byobu 内で開かれている独立したターミナルで、Windows Terminal のタブと同じようなものだと考えるとわかりやすいでしょう。

新しいウィンドウを作成するにはF2キー、もしくはPrefix+Cを押します。これは Prefix キーをCtrl+Zに設定しているのであれば、Ctrl+Zを押してからCキーを押す操作になります。するとステータスラインのウィンドウ一覧に新しいウィンドウが追加されます。F3/F4キーかPrefix+N/Pキーを押すと、前後のウィンドウに切り替えることができます。

ウィンドウには名前をつけることもできます。F8キーかPrefix+,を押すと、ステータスラインがrename-windowというプロンプトに変化します。そこに任意のウィンドウ名を入力してEnterキーを押してください。ウィンドウに名前をつければ、複数のウィンドウを区別しやすくなります。Byobu は、複数のサーバーを並列操作するような際にもよく利用します。ウィンドウの名前は、サーバーの取り違いという重大インシデントを予防することにも役立ちます。

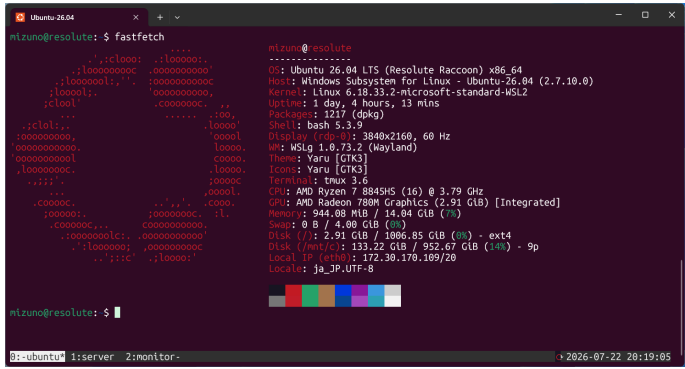


図 8.7: 名前を変えた複数のウィンドウを並べた例

8.4.2 スプリット

Byobu では、1 つのウィンドウを複数の領域に分割することができます。これをスプリットと呼びます。同一の画面に複数のシェルを並べたい場合は、ウィンドウよりもスプリットが便利でしょう。

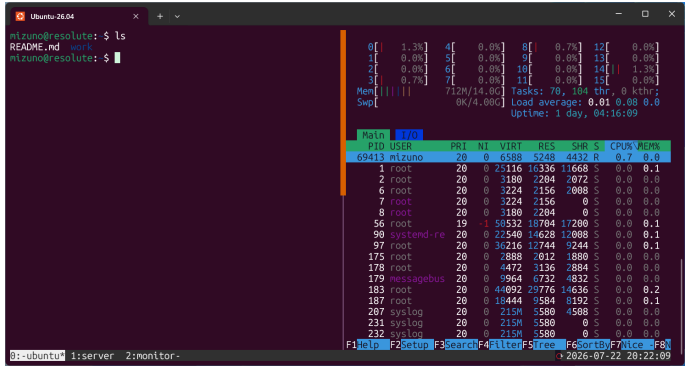


図 8.8: スプリットの例

スプリットの主な操作は以下の通りです。

表 8.1: スプリットのキー操作

キー操作	用途
Ctrl+F2	ウィンドウを左右にスプリット
Shift+F2	ウィンドウを上下にスプリット
Shift+ カーソルキー	スプリット間を移動
Ctrl+F6	現在のスプリットを閉じる

スプリットを使うと、同じ画面内に複数のターミナルを並べて表示できるため、並行して作業したい時に非常に便利です。例えば、一方でのスプリットでコードを編集しながらもう一方でコンパイルやデバッグを行ったり、一方のスプリットでコマンドを実行しながらもう一方でログを確認するといった使い方ができます。

8.4.3 コピー&ペースト

F7かPrefix+[キーを押すとコピーモードに入ります。コピーモードではカーソルを移動し、画面外へ流れていった出力をスクロールバックして読むことができます。スクロールバックは、GUIのターミナルエミュレーターでは当たり前の機能であるため、それほどありがたみはないでしょう。ですがサーバーの仮想コンソールを直接利用しているような場合は、画面外に溢れたログを読むことができません。このような場合にはByobuの出番です。

コピーモードでは、その名の通りコピーとペーストができます。コピーしたい領域の始点にカーソルを合わせてスペースキーを押してから、終点までカーソルを移動してEnterキーを押すと、その範囲がコピーされます。コピーした内容をペーストするには、Prefix+[キーを押します。

8.4.4 セッションのアタッチとデタッチ

Byobuの最も重要な機能の一つが、セッションのデタッチとアタッチです。デタッチとは、Byobuをバックグラウンドで動作させたまま、そのセッションをターミナルから切り離すことです。そしてアタッチとは、バックグラウンドで動作しているByobuセッションに再接続することです。

と聞いてもなかなかイメージしづらいと思いますので、実際に試してみましょう。まずByobuを起動してからtopコマンドを実行してください。システムの状態やプロセス一覧が表示され続けます。

```

top - 20:24:34 up 1 day, 4:18, 1 user, load average: 0.00, 0.05, 0.03
Tasks: 67 total, 1 running, 66 sleeping, 0 stopped, 0 zombie
       0.0 us, 0.0 sy, 0.0 ni, 99.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 14381.7 total, 12451.7 free, 949.4 used, 1226.0 buff/cache
MiB Swap: 4096.0 total, 4096.0 free, 0.0 used, 13432.0 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR   S   %CPU  %MEM    TIME+  COMMAND
    1 root        20   0   25116   16336  11668   S   0.0   0.1   0:07.73  systemd
    2 root        20   0   3198   2284   2972   S   0.0   0.0   0:00.06  init-systemd(Lib
    6 root        20   0   3224   2156   2988   S   0.0   0.0   0:00.02  tinit
   56 root       19  -1  58532  18788  17284   S   0.0   0.1   0:01.62  systemd-journal
   90 systemd+   20   0  22548  14626  12088   S   0.0   0.1   0:00.26  systemd-resolve
   97 root       20   0  36216  12744   9244   S   0.0   0.1   0:01.99  systemd-udev
  175 root       20   0   2888   2932   1880   S   0.0   0.0   0:00.04  chronyd-starter
  178 root       20   0   4472   3136   2884   S   0.0   0.0   0:00.26  cron
  179 message+   20   0   9964   6732   4832   S   0.0   0.0   0:01.78  dbus-daemon
  183 root       20   0  44092  29776  14636   S   0.0   0.2   0:00.19  networkd-dispat
  187 root       20   0  18444   9584   8192   S   0.0   0.1   0:00.64  systemd-logind
  207 syslogd    20   0  228548  5580   4588   S   0.0   0.0   0:00.36  rsyslogd
  235 root       20   0  123456  32664  18848   S   0.0   0.2   0:00.15  unattended-upgr
  241 root       20   0   5492   2332   2644   S   0.0   0.0   0:00.02  atetty
  266 chrony     20   0  24524  11996   7240   S   0.0   0.1   0:00.41  chronyd
  267 chrony     20   0  12092   2528   1880   S   0.0   0.0   0:00.00  chronyd
  515 mizuno     20   0  22676  13688  11052   S   0.0   0.1   0:00.25  systemd
  517 mizuno     20   0  22908   4116   2116   S   0.0   0.0   0:00.00  (sd-pam)
  651 root       20   0   8648   5576   4732   S   0.0   0.0   0:00.00  login
  696 mizuno     20   0   6136   5468   3928   S   0.0   0.0   0:00.01  bash
 1123 polkittd   20   0  306444   8496   7472   S   0.0   0.1   0:03.67  polkitd
 3318 mizuno     20   0   8948   5896   4716   S   0.0   0.0   0:00.50  dbus-daemon
  
```

図 8.9: top コマンドを実行した状態

この状態でF6キーまたはPrefix+Dを押します。デタッチすると、一見Byobuを終了したように見えますが、実はByobuはバックグラウンドで動作し続けています。

```

[detached (from session 1)]
mizuno@resolute: $
  
```

図 8.10: デタッチを行った状態。Byobu から切り離され、ターミナルの表示がByobu 実行前の状態に戻った

この状態で再びbyobuコマンドを実行すると、自動的に前回のセッションにアタッチされます。ウィンドウやスプリットの状態も復元されますし、実行中のtopコマンドも動き続けています。実行に長い

時間がかかるコマンドを開始した後にデタッチし、後で再アタッチして結果を確認するといった使い方もできます。

図 8.11: Byobu に再アタッチすると、デタッチ前の状態が復元される

簡単に言えば、Byobu という層を介することで、ターミナルとその中身を自由に切り離すことが可能になるわけです。例えばコマンドの実行中に、Windows Terminal のウィンドウを閉じてしまったことを考えてみましょう。通常であれば、そのウィンドウ内で実行中のプログラムは強制的に終了されてしまいます。ですが Byobu 上で実行されているコマンドであれば、ターミナルとの接続は切れるものの、Byobu がセッションを保護してくれるため、終了されることはありません。そのため Windows Terminal を再起動した後に、セッションにアタッチすれば、作業に復帰することができるわけです。

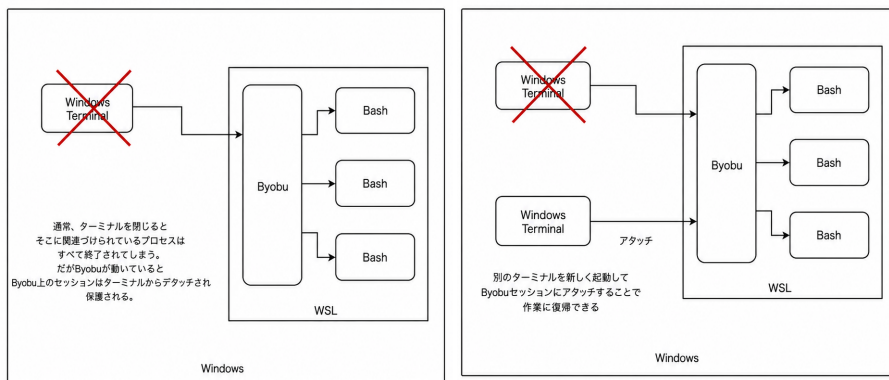


図 8.12: Byobu とセッション保護の例

セッション保護が最も効果を発揮するのは、リモートサーバー上で作業をしている時です。SSH 接続などでリモートサーバーに接続している場合、ネットワークトラブルで接続が切れてしまうと、実行中のコマンドもすべて終了されてしまいます。もしもシステムのアップデートや、データベースのマイグレーションといったメンテナンスを行っている最中にこれが起こると、サーバーが起動しないといった致命的な障害に繋がる可能性もあります。ですが Byobu を使えば、このような不意の接続断からもセッションを保護できます。Windows Terminal を閉じてコマンドが継続して動き続けるのと同様に、SSH 接続が切れてしまっても、Byobu は動き続けるためです。ただしリモートサーバー上のセッションを保護する際は、ローカルの WSL 上ではなく、保護したいリモートサーバー上で Byobu を起動する必要がある点に注意してください。

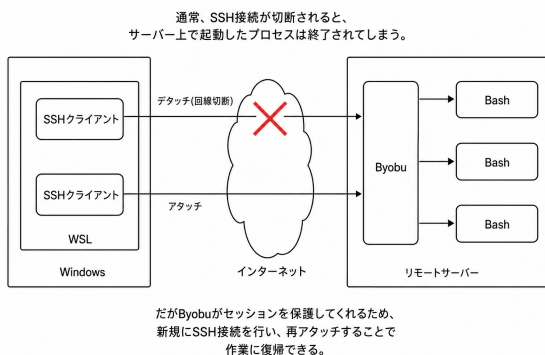


図 8.13: Byobu を使えば不意の回線切断に備えられる

なおセッションは複数同時に起動することもできます。Byobu の実行中に`Ctrl+Shift+F2`キーを押すと、新しいセッションを作れます。またセッションの一覧は、`byobu list-sessions`コマンドで確認できます。

```

mizuno@resolute: $ byobu list-sessions
1: 1 windows (created Wed Jul 22 20:15:22 2026) (group 1)
3: 1 windows (created Wed Jul 22 20:36:56 2026)
mizuno@resolute: $

```

図 8.14: `byobu list-sessions` で、複数のセッションを一覧できる

```

mizuno@resolute: $ byobu
Byobu sessions...

1. tmux: 1: 1 windows (created Wed Jul 22 20:15:22 2026) (group 1)
2. tmux: 3: 1 windows (created Wed Jul 22 20:36:56 2026)
3. Create a new Byobu session (tmux)
4. Run a shell without Byobu (/bin/bash)

Choose 1-4 [1]:

```

図 8.15: 複数のセッションが存在する状態で Byobu を起動すると、アタッチ先を選択できる

このように Byobu は、ターミナルでの作業を格段に効率化してくれるツールです。またシェル上で動作するプログラムのため、ターミナルの種類を問いません。一度その便利さを体験すれば、もう手放せなくなるでしょう。特にクラウドなどのサーバーを操作する必要がある場合は、サーバー側に Byobu をインストールしておくことを強く推奨します。単に作業効率を向上させるだけでなく、不意の回線切断といったトラブルにも備えることができるためです。

第 9 章 さらに現代的な Linux コマンドを活用しよう

近年、Linux の基本コマンドを Rust で再実装するのが流行っています。この章では日々のコマンドライン作業を快適にする、より今風で便利な実装について紹介します。

9.1 より現代的な Linux コマンドの登場

Linux の基礎コマンド群である Coreutils については、既に解説しました。Coreutils に含まれる `ls` や `cat` はもちろん、それ以外でも `grep`、`diff`、`top` のようなコマンドは、どの環境にもインストールされている基本コマンドのため、一通り使えるようになっておくべきです。

しかしこれらのコマンドは歴史が古く、今の時代からすると機能的に物足りなかったり、不親切な部分もあります。そこでこうした基本コマンドを置き換える、より現代的な実装が数多く登場しています。こうした現代的な実装は、おおむね共通して以下のような特徴を持っています。

- 従来のコマンドと比べて高いパフォーマンス
- カラフルな出力やシンタックスハイライトといった視覚的な改善
- より分かりやすいオプションや引数設計
- 現代の開発環境に合わせた Git 統合機能

共用のサーバーに個人向けのツールをインストールするのは好ましくない場合もありますが、WSL のような個人用の環境であれば、自由にカスタマイズして使いやすくしていきましょう。どのような環境でも作業ができるよう標準コマンドに習熟することと、効率を追求したカスタマイズを行うことは、どちらも同じくらい大切だと筆者は考えています。この章では筆者が有用だと考えた、いくつかの代替コマンドを紹介し、それぞれの Ubuntu でのインストール方法と基本的な使用方法を解説します。

もちろんいきなりすべてのコマンドを置き換える必要はありません。従来のコマンドと併用しながら、視覚的な改善が分かりやすいものから始めてみてはいかがでしょうか。以下で紹介するコマンドを活用して、より生産的な開発環境を構築していきましょう。なおどのコマンドも、非常に多くの機能やオプションを持っています。紹介できるのは、あくまで代表的な使い方だけに限定していますので、より詳しい使い方は各コマンドのマニュアルを参照してください。

9.2 eza

`eza` は `ls` の代替コマンドで、カラフルな出力、Git 統合、ツリー表示などの機能を提供します。もともと `exa` という実装が存在したのですが、こちらが Obsolete となってしまったため、その後継が `eza` となります。

```

mizuno@resolute:~$ eza --tree -l
drwxr-xr-x  - mizuno 23  7月 12:19  chap_03
drwxr-xr-x  - mizuno 23  7月 12:07  03.md
-rwxr-xr-x 222k mizuno 23  7月 12:07  adduser_001.png
-rwxr-xr-x  39k mizuno 23  7月 12:07  archive_001.png
-rwxr-xr-x  18k mizuno 23  7月 12:07  archive_002.png
-rwxr-xr-x 132k mizuno 23  7月 12:07  archive_003.png
-rwxr-xr-x 225k mizuno 23  7月 12:07  archive_004.png
-rwxr-xr-x 124k mizuno 23  7月 12:07  archive_005.png
-rwxr-xr-x  92k mizuno 23  7月 12:07  archive_006.png
-rwxr-xr-x  43k mizuno 23  7月 12:07  archive_007.png
-rwxr-xr-x  85k mizuno 23  7月 12:07  archive_008.png
-rwxr-xr-x 141k mizuno 23  7月 12:07  edit_001.png
-rwxr-xr-x  68k mizuno 23  7月 12:07  extract_001.png
-rwxr-xr-x 144k mizuno 23  7月 12:07  extract_002.png
-rwxr-xr-x 275k mizuno 23  7月 12:07  extract_003.png
-rwxr-xr-x  97k mizuno 23  7月 12:07  extract_004.png
-rwxr-xr-x 308k mizuno 23  7月 12:07  fstab_001.png
-rwxr-xr-x  81k mizuno 23  7月 12:07  group_001.png
-rwxr-xr-x 143k mizuno 23  7月 12:07  gui_001.png
-rwxr-xr-x 2.0M mizuno 23  7月 12:07  gui_002.png
-rwxr-xr-x  18k mizuno 23  7月 12:07  gui_003.png
-rwxr-xr-x 1.3M mizuno 23  7月 12:07  job_001.png
-rwxr-xr-x  24k mizuno 23  7月 12:07  kernel.png
-rwxr-xr-x 117k mizuno 23  7月 12:07  kill_001.png
-rwxr-xr-x 668k mizuno 23  7月 12:07  nan_001.png
-rwxr-xr-x 281k mizuno 23  7月 12:07

```

図 9.1: eza の利用例

9.2.1 インストール方法

aptでezaパッケージをインストールします。

```
$ sudo apt install -U -y eza
```

9.2.2 基本的な使用方法

ezaに限らず今回紹介するコマンドは、基本的な使い方や多くのオプションを、元のコマンドから踏襲しています。そのため特に違和感なく使えるのではないのでしょうか。ezaの便利な点として、`--git`オプションの存在があります。このオプションをつけることで、Git 上でのファイルの変更状態を表示させられます。

以下はカレントディレクトリのファイル一覧をカラフルに表示します。

```
$ eza
```

`-l`オプションで詳細情報を表示します。

```
$ eza -l
```

`-a`オプションで隠しファイルも表示します。

```
$ eza -a
```

`--git`オプションをつけると、Git 情報も表示します。

```
$ eza -l --git
```

`--tree`オプションをつけると、ツリー形式で表示します。

```
$ eza --tree
```

9.3 zoxide

zoxideは**cd**の代替コマンドで、アクセス頻度と最新性に基づいて賢いディレクトリ移動を実現します。

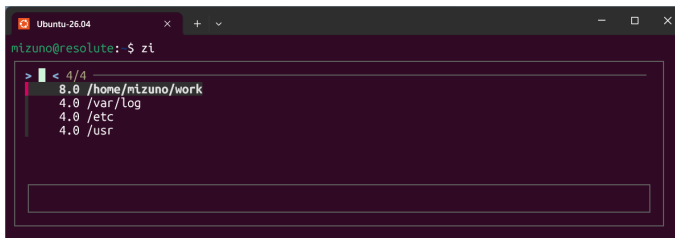


図 9.2: zoxide と fzf を組み合わせたディレクトリ移動

9.3.1 インストール方法

aptで**zoxide**パッケージをインストールします。同時にあいまい検索フィルタである**fzf**パッケージもインストールしておくといよいでしょう。

```
$ sudo apt install -U -y zoxide fzf
```

zoxide を使うには、シェルの設定を追加する必要があります。以下のコマンドを実行してください。

```
$ echo 'eval "$(zoxide init bash)"' >> ~/.bashrc
$ source ~/.bashrc
```

9.3.2 基本的な使用方法

zoxide では、**z**というコマンドでディレクトリ移動を行います。最初は通常の**cd**コマンドと同じですが、**zoxide** は一度移動したディレクトリを記憶しています。次回以降はディレクトリ名の一部分を指定するだけで、履歴からマッチする、もっとも利用頻度の高いディレクトリに移動できます。

初回は通常の**cd**と同様に、ディレクトリを移動します。

```
$ z ~/work
```

一度訪問したディレクトリは部分的な名前でもジャンプ可能になります。以下の例では、**w**だけで**~/work**に移動できます。

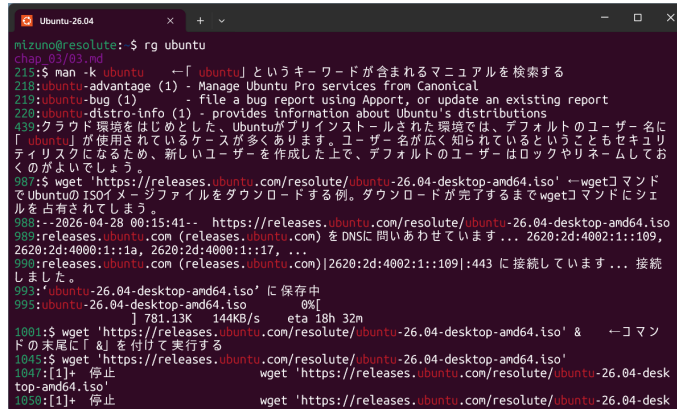
```
$ z w
```

また**zi**コマンドで、**fzf**を利用したインタラクティブなディレクトリ選択が可能になります。

```
$ zi
```

9.4 ripgrep

`ripgrep`は`grep`の代替コマンドです。高速性と、デフォルトでの再帰検索が特徴です。



```

mizuno@resolute:~$ rg ubuntu
chap_03/03.md
215:$ man -k ubuntu --「ubuntu」というキーワードが含まれるマニュアルを検索する
218:ubuntu-advantage (1) - Manage Ubuntu Pro services from Canonical
219:ubuntu-bug (1) - file a bug report using Apport, or update an existing report
220:ubuntu-dist-info (1) - provides information about Ubuntu's distributions
430:クラウド環境をはじめとした、Ubuntuがプリインストールされた環境では、デフォルトのユーザー名に「ubuntu」が使用されているケースが多くあります。ユーザー名が広く知られていることもセキュリティリスクになるため、新しいユーザーを作成した上で、デフォルトのユーザーはロックやリネームしておくのがよいでしょう。
987:$ wget 'https://releases.ubuntu.com/resolute/ubuntu-26.04-desktop-and64.iso' --wgetコマンドでUbuntuのISOイメージファイルをダウンロードする例。ダウンロードが完了するまでwgetコマンドにシェルを占有させましょう。
988:--2026-04-28 00:15:41-- https://releases.ubuntu.com/resolute/ubuntu-26.04-desktop-and64.iso
989:releases.ubuntu.com (releases.ubuntu.com) をDNSに問いあわせています... 2620:2d:4002:1::109, 2620:2d:4000:1::1a, 2620:2d:4000:1::17, ...
990:releases.ubuntu.com (releases.ubuntu.com)[2620:2d:4002:1::109]:443 に接続しています... 接続しました。
991:ubuntu-26.04-desktop-and64.iso' に保存中
992:ubuntu-26.04-desktop-and64.iso
993: 781.13K 144KB/s eta 18h 32m
1001:$ wget 'https://releases.ubuntu.com/resolute/ubuntu-26.04-desktop-and64.iso' --コマンドの末尾に「&」を付けて実行する
1045:$ wget 'https://releases.ubuntu.com/resolute/ubuntu-26.04-desktop-and64.iso'
1047:[1]+ 停止 wget 'https://releases.ubuntu.com/resolute/ubuntu-26.04-desktop-and64.iso'
1050:[1]+ 停止 wget 'https://releases.ubuntu.com/resolute/ubuntu-26.04-desk

```

図 9.3: ripgrep での検索例

9.4.1 インストール方法

`apt`で`ripgrep`パッケージをインストールします。

```
$ sudo apt install -U -y ripgrep
```

9.4.2 基本的な使用方法

`ripgrep`の使い方は、`grep`に比べてシンプルです。`grep`では検索パターンと対象ファイルを指定する必要がありましたが、`ripgrep`では対象ファイルは省略可能です。デフォルトでカレントディレクトリ以下に対して、再帰検索が行われます。

```
$ rg "検索パターン"
```

`-i`オプションをつけると、大文字小文字の違いを無視します。

```
$ rg -i "pattern"
```

`--type`オプションで、特定のファイルタイプ（ここでは Markdown）に絞って検索できます。

```
$ rg --type md "function"
```

`-l`オプションをつけると、ファイル名のみを表示します。


```
$ rg -l "pattern"
```

`-A`オプションと`-B`オプションでは、マッチした部分の前後の行を、指定した行数ぶん表示できます。

```
$ rg -A 3 -B 3 "pattern"
```

検索パターンには正規表現も使えます。

```
$ rg --type md "^#+ "
```

9.5 bat

`bat`は`cat`の代替コマンドです。キーワードに色をつけるシンタックスハイライトや、Git 統合といった機能を持っています。

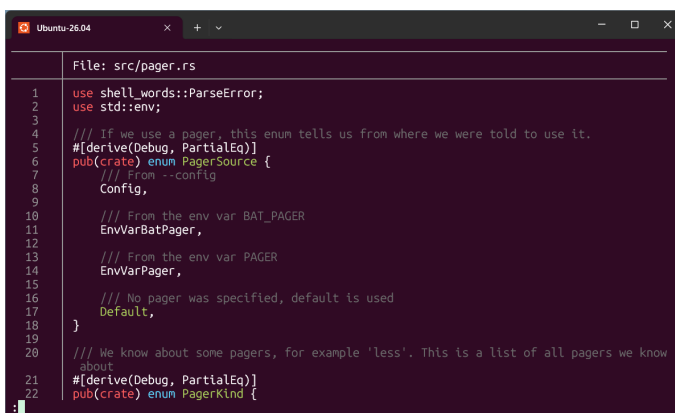


図 9.4: bat でテキストファイルを表示した例

9.5.1 インストール方法

`apt`で`bat`パッケージをインストールします。

```
$ sudo apt install -U -y bat
```

ちなみに Ubuntu では別のパッケージと競合するため、`bat`ではなく`batcat`というコマンドとしてインストールされます。そのため以下のエイリアスを設定し、`bat`として呼び出せるようにしておくとう便利です。

```
$ echo 'alias bat=batcat' >> ~/.bashrc
$ source ~/.bashrc
```

9.5.2 基本的な使用方法

`cat`と同様に、表示したいファイル名を指定して使用するのが一般的な使い方です。`cat`と異なり、デフォルトでページャを使って表示されます。また自動的にシンタックスハイライトが行われます。

```
$ bat file.py
```

`--diff` オプションをつけると、Git の変更情報も表示します。

```
$ bat --diff file.py
```

同時に複数のファイルを表示することも可能です。

```
$ bat file1.py file2.py
```

9.6 fd

`fd`は`find`の代替コマンドです。シンプルな構文や Git 統合が特徴です。

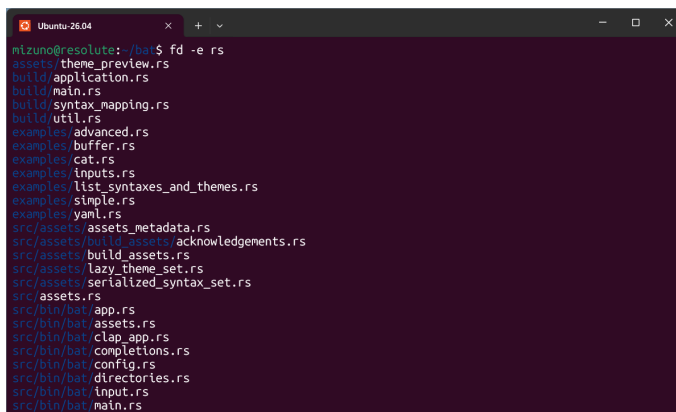


図 9.5: `fd` でファイルを検索した例

9.6.1 インストール方法

`apt`で`fd-find`パッケージをインストールします。

```
$ sudo apt install -U -y fd-find
```

なお`fd`も`bat`と同様に、別のパッケージとの競合問題があり、`fdfind`コマンドとしてインストールされます。こちらも以下のようにエイリアスを設定しておくとも便利です。

```
$ echo 'alias fd=fdfind' >> ~/.bashrc
$ source ~/.bashrc
```

9.6.2 基本的な使用方法

引数にファイル名を指定して検索するのが、基本的な使い方です。

```
$ fd ファイル名
```

`-e` オプションをつけると、ファイルの種類を限定して検索できます。以下は Markdown ファイルに限定する例です。

```
$ fd -e md
```

ファイルの種類は複数指定できます。

```
$ fd -e md -e yaml
```

`-t` オプションをつけると、ファイルタイプを限定できます。`t` でディレクトリのみを検索します。

```
$ fd -t d
```

`f` でファイルのみを検索します。

```
$ fd -t f
```

`fd` はデフォルトで隠しファイルや隠しディレクトリを検索から除外します。`-H` オプションをつけると、これらも含んで検索します。

```
$ fd -H filename
```

見つかったファイルに対して、指定したコマンドを実行することもできます。以下の例では Markdown ファイルに対して `stat` コマンドを実行しています。

```
$ fd -e md -x stat {} \;
```

9.7 delta

`delta` は `diff` の代替コマンドで、特に Git diff に特化した美しい差分表示を提供します。

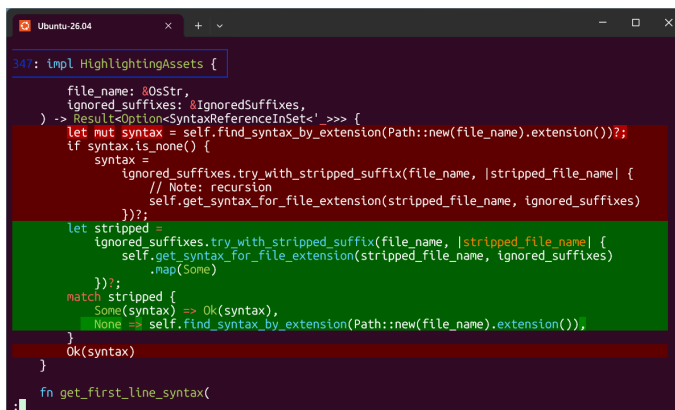


図 9.6: delta による git diff の表示例

9.7.1 インストール方法

aptでgit-deltaパッケージをインストールします。

```
$ sudo apt install -U -y git-delta
```

deltaはdiffコマンドの代わりに使うこともできますが、Git から呼び出すように設定して、その真価を発揮します。gitコマンドが暗黙的にdeltaを使用するよう、gitconfigを設定しておきましょう。

```
$ git config --global pager.diff delta
$ git config --global pager.show delta
$ git config --global pager.log delta
$ git config --global pager.reflog delta
```

9.7.2 基本的な使用方法

2つのファイルを指定することで、通常のdiffを美しく表示します。

```
$ delta file1.txt file2.txt
```

--side-by-sideオプションをつけると、2つのファイルを左右に並べるサイドバイサイド形式で表示します。

```
$ delta --side-by-side file1.txt file2.txt
```

前述のgit configを行っていると、以下のgitコマンド使用時にdeltaを使用して差分が表示されます。

```
$ git diff
$ git show
$ git log -p
```

9.8 dust

`dust`は`du`の代替コマンドです。視覚的で分かりやすいディスク使用量表示を提供します。

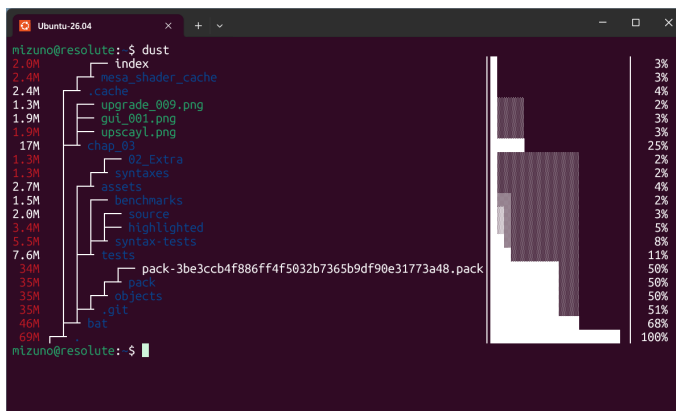


図 9.7: dust によるディスク使用量表示

9.8.1 インストール方法

`apt`で`du-dust`パッケージをインストールします。

```
$ sudo apt install -U -y du-dust
```

9.8.2 基本的な使用方法

システムを長期間運用していると、ストレージの空き容量が逼迫してくるものの、具体的にどのディレクトリが容量を食っているのかがわからないといった事態が起こりがちです。従来の`du`コマンドでは、`du -hs *`といったコマンドを繰り返し実行し、肥大化したディレクトリを掘って、対象を突き止める必要がありました。対して`dust`コマンドであれば、ディレクトリ階層を辿りながら分析し、肥大化したディレクトリを簡単に突き止めることが可能です。

以下はカレントディレクトリ以下のディスク使用量を表示する例です。

```
$ dust
```

引数にパスを指定すると、特定のディレクトリを分析できます。

```
$ dust /path/to/directory
```

`-n`オプションを指定すると、表示する項目数を制限できます。

```
$ dust -n 10
```

`-d`オプションを指定すると、表示するディレクトリ深度を制限できます。

```
dust -d 3
```

`-D`オプションを指定すると、ディレクトリのみを表示します。

```
$ dust -D
```

9.9 procs

`procs`は`ps`の代替コマンドです。プロセス情報をカラフルに、かつ追加情報とともに出力します。

PID	User	TTY	CPU	MEM	CPU Time	Command
			[%]	[%]		
1	root		0.0	0.1	00:00:09	/sbin/init
2	root		0.0	0.0	00:00:00	/init
6	root		0.0	0.0	00:00:00	plan9 --control-socket 7 --log-level 4 --serv
56	root		0.0	0.1	00:00:02	/usr/lib/systemd/systemd-journald
90	systemd-resolve		0.0	0.1	00:00:00	/usr/lib/systemd/systemd-resolved
97	root		0.0	0.1	00:00:02	/usr/lib/systemd/systemd-udev
175	root		0.0	0.0	00:00:00	/bin/sh /usr/lib/systemd/scripts/chronyd-star
178	root		0.0	0.0	00:00:00	/usr/sbin/cron -f +P
179	messagebus		0.0	0.0	00:00:02	dbus-daemon --system --address=systemd: --no
183	root		0.0	0.2	00:00:00	/usr/bin/python3 /usr/bin/networkd-dispatcher
187	root		0.0	0.1	00:00:00	/usr/lib/systemd/systemd-logind
207	syslog		0.0	0.0	00:00:00	/usr/sbin/rsyslogd -n -iNONE
235	root		0.0	0.2	00:00:00	/usr/bin/python3 /usr/share/unattended-upgrade
241	root		0.0	0.0	00:00:00	/usr/sbin/agetty --noret --noclear --issue-
266	_chrony		0.0	0.1	00:00:00	/usr/sbin/chronyd -n -F 1 -x
267	_chrony		0.0	0.0	00:00:00	/usr/sbin/chronyd -n -F 1 -x
515	mizuno		0.0	0.1	00:00:00	/usr/lib/systemd/systemd --user
517	mizuno		0.0	0.0	00:00:00	(sd-pan)
651	root		0.0	0.0	00:00:00	login -- mizuno
696	mizuno	pts/1	0.0	0.0	00:00:00	-bash
1123	polkitd		0.0	0.1	00:00:06	/usr/lib/polkit-1/polkitd --no-debug --log-le
3318	mizuno		0.0	0.0	00:00:00	/usr/bin/dbus-daemon --session --address=system
8900	root		0.0	0.0	00:00:00	/init
8901	root		0.0	0.0	00:00:00	/init

図 9.8: `procs` でプロセス一覧を表示した例

9.9.1 インストール方法

`apt`で`procs`パッケージをインストールします。

```
$ sudo apt install -U -y procs
```

9.9.2 基本的な使用方法

`procs`コマンドを実行すると、現在実行中のプロセスリストが表示がされます。また前述の`bat`コマンドと同様に、`procs`もデフォルトでページャを使って結果を表示します。

```
$ procs
```

引数に文字列を指定すると、その名前を持つプロセスを表示します。

```
$ procs python
```

引数に数字を指定すると、そのプロセス ID を持つプロセスを表示します。

```
$ procs 1234
```

`--tree` オプションを指定すると、プロセスの親子関係をツリー表示します。

```
$ procs --tree
```

`--sorta` もしくは `--sortd` オプションを指定すると、指定した項目でソート（昇順、降順）します。この例では CPU の使用率順にソートしています。

```
$ procs --sortd cpu
```

`-w` オプションを指定するとウォッチモードになり、プロセスリストをリアルタイム更新して表示し続ける、`top` のような使い方ができます。

```
$ procs -w
```

9.10 bottom

`bottom` は `top` の代替コマンドで、より見やすく使いやすいシステムモニターを提供します。

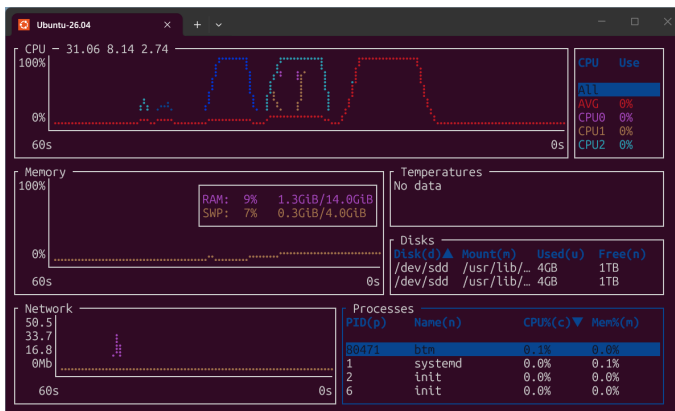


図 9.9: `bottom` の表示例

9.10.1 インストール方法

`snap` で `bottom` パッケージをインストールします。

```
$ sudo snap install bottom
```

9.10.2 基本的な使用方法

`bottom` コマンドでシステムモニターを起動します。

```
$ bottom
```

`--basic` オプションをつけると、グラフを表示しない簡易モードになります。

```
$ bottom --basic
```

`-r` オプションでは、表示の更新間隔を設定できます。

```
$ bottom -r 1s
```

`bottom` は `top` と同様に、画面全体を占有し、対話的に操作するコマンドです。主な操作は以下の通りです。より詳しくは `?` キーで表示されるヘルプを参照してください。

表 9.1: `bottom` の主なキー操作

キー	機能
j or k	プロセスの選択
dd	シグナルの送信
f	更新の停止/再開
e	ウィジェットの最大化/元に戻す
c	プロセスを CPU 使用率でソート
m	プロセスをメモリ使用率でソート
p	プロセスをプロセス ID でソート
/	検索
?	ヘルプの表示
q	終了

9.11 hyperfine

`hyperfine` は `time` の代替コマンドです。コマンドのパフォーマンスを計測できるベンチマークツールです。指定したコマンドを複数回実行し、平均実行時間をはじめとする統計情報を詳細に表示します。

```

mizuno@resolute: $ hyperfine 'dd if=/dev/urandom of=test.img bs=1M count=1024'
Benchmark 1: dd if=/dev/urandom of=test.img bs=1M count=1024
Time (mean ± σ): 2.524 s ± 0.111 s [User: 0.003 s, System: 2.393 s]
Range (min ... max): 2.455 s ... 2.789 s 10 runs
mizuno@resolute: $

```

図 9.10: `hyperfine` によるコマンドのベンチマーク

9.11.1 インストール方法

`apt` で `hyperfine` パッケージをインストールします。


```
$ sudo apt install -U -y hyperfine
```

9.11.2 基本的な使用方法

hyperfineは、引数に実行したいコマンドを指定して使います。デフォルトでは指定したコマンドを10回実行します。以下はddコマンドで1GBのランダムなダミーファイルを作成するのにかかる時間を計測します。

```
$ hyperfine 'dd if=/dev/urandom of=test.img bs=1M count=1024'
```

--runs オプションでは、実行する回数を指定できます。

```
$ hyperfine --runs 3 任意のコマンド
```

--export-json オプションをつけると、結果をJSONで出力します。

```
hyperfine --export-json results.json 任意のコマンド
```

--export-csv オプションをつけると、結果をCSVで出力します。

```
hyperfine --export-csv results.csv 任意のコマンド
```

--export-markdown オプションをつけると、結果をMarkdownで出力します。

```
hyperfine --export-markdown results.md 任意のコマンド
```

9.12 hexyl

hexylはhexdumpの代替コマンドです。バイナリダンプをカラフルで見やすく表示できます。

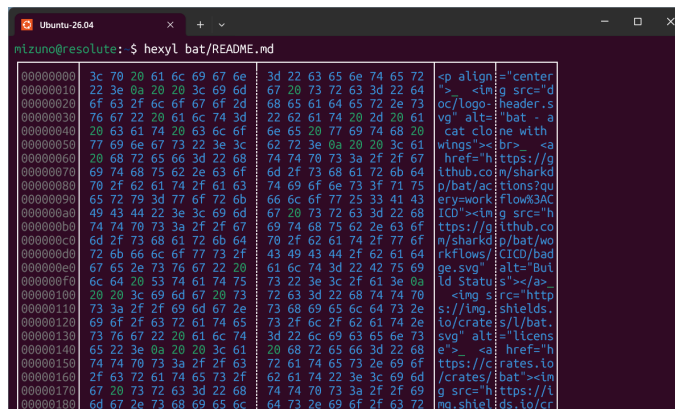


図 9.11: hexyl によるファイルの16進ダンプ

9.12.1 インストール方法

aptでhexylパッケージをインストールします。

```
$ sudo apt install -U -y hexyl
```

9.12.2 基本的な使用方法

引数に指定したファイルの内容を 16 進数にダンプして表示します。例えばテキストファイルのはずなのに、一部が化けて正しく表示されないファイルなどに遭遇することはよくあります。こうしたテキストとして表示できない文字であっても、バイナリにダンプすることで、どのようなデータが書き込まれているかを調べられます。他にもバイナリファイルのヘッダをチェックする、キャプチャしたパケットを解析するなど、様々な応用が可能なコマンドです。

引数に指定したファイルを 16 進ダンプして表示します。

```
$ hexyl ファイル
```

-n オプションで、先頭から指定したバイトのみ表示できます。

```
$ hexyl -n 10 ファイル
```

-s オプションで、先頭から指定したバイトの表示をスキップします。

```
$ hexyl -s 10 ファイル
```

9.13 lnav

lnavは、ログファイルの表示と分析に特化したテキストビューアです。従来であれば、ログファイルはcatやlessといったシンプルなコマンドで表示したり、grepを組み合わせるくらいしかできず、中身の解析はエンジニアの目に頼らざるを得ない部分がありました。ですがlnavを使えば、より効率的にログの監視や分析が可能になります。lnavではSQLクエリを発行し、ログデータを直接フィルタするといったことも可能です。

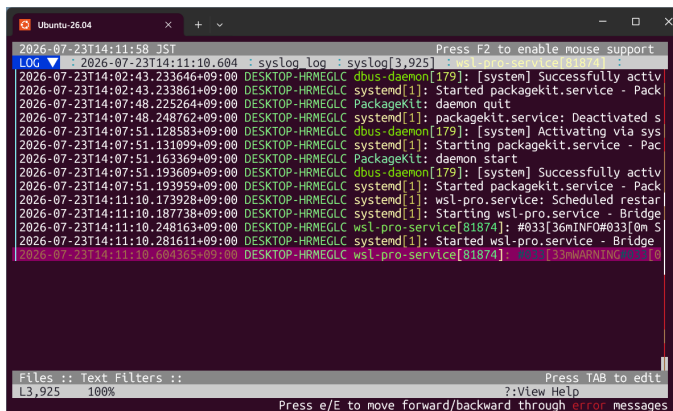


図 9.12: lnav によるログの表示例

9.13.1 インストール方法

`apt` で `lnav` パッケージをインストールします。

```
$ sudo apt install -U -y lnav
```

9.13.2 基本的な使用方法

`lnav` コマンドを引数なしで実行すると、`syslog` ファイルを開きます。`tail` コマンドとは異なり、特別なオプションをつけなくても、ログが更新されると自動的に更新部分を読み込みます。

```
$ lnav
```

引数にログファイルを指定すると、そのファイルを表示します。

```
$ lnav /var/log/auth.log
```

ワイルドカードを使い、複数のログファイルを同時に表示できます。

```
$ lnav /var/log/apache2/*.log
```

`-r` オプションをつけると、ディレクトリを再帰的にたどってすべてのログファイルを表示します。

```
$ lnav -r /var/log/
```

`docker` コマンドからパイプすることで、Pod のログを表示できます。

```
$ docker logs pod | lnav
```

lnavもまた、対話的に操作するタイプのコマンドです。主な操作は以下のようになっています。より詳しくはヘルプを参照してください。

表 9.2: lnav の主なキー操作

キー	機能
/	検索
;	SQL 入力モード
j/k	下/上に移動
n/N	次/前の検索結果にジャンプ
e/E	次/前のエラーへジャンプ
w/W	次/前の警告へジャンプ
?	ヘルプの表示
q	終了

第 10 章 標準入力と標準出力について知ろう

コマンドを効率よく使う上で必須の知識が「標準入出力」です。この章では Linux における標準入出力の仕組みを解説します。

10.1 標準入力と標準出力とは

Linux のコマンドは、キーボードから入力を受け取り、ディスプレイに結果を表示します。しかしこれは厳密には正しくありません。Linux の入出力は、**標準入力 (standard input)** と **標準出力 (standard output)** という、抽象的な仕組みとして扱われています。コマンドはキーボードやディスプレイといったデバイスを直接扱っているのではなく、標準入力からデータを受け取り、標準出力へデータを出力するという動作をしています。単にデフォルトの標準入力がキーボードに、標準出力がディスプレイに繋がっているというだけなのです。

なぜこんな面倒な仕組みになっているかというと、標準入出力を自由に切り替え可能にするためです。標準入出力という抽象化を一段挟むことで、実際のデータ入力元や出力先が変更されても、プログラムはその違いを意識する必要がなくなります。そして実際に標準入出力の切り替えを行うのが、この章で解説するリダイレクトやパイプです。

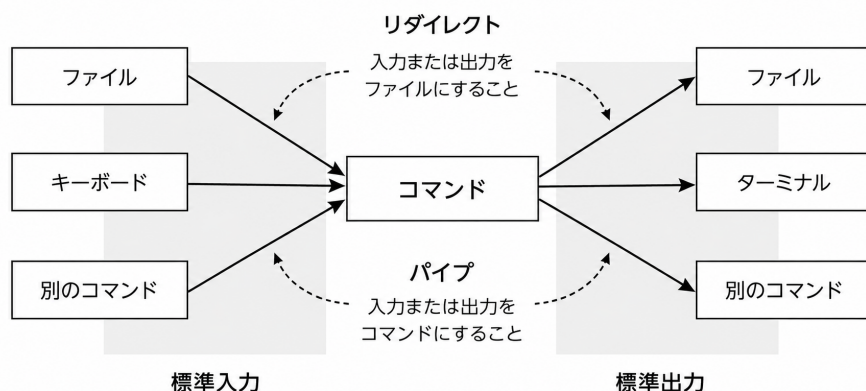


図 10.1: リダイレクトとパイプの概念図

標準入力と標準出力は Linux のもっとも基本的な概念のひとつであり、Linux のコマンドはすべてこの仕組みの上に成り立っています。しかし同時に、なんとなくコピーで、理解せずに使ってしまう部分もあります。本章を読めば、パイプは何をしているのか、リダイレクトがなぜ便利なのか、という部分を理解しましょう。仕組みさえ把握できれば、複雑なコマンドの組み合わせも、自然と理解できるようになってくるはずです。

10.2 リダイレクト

リダイレクトとは、標準入出力の接続先を変更する機能です。出力リダイレクトと入力リダイレクトに分けて説明します。

10.2.1 出力リダイレクトとは

もっともよく使われるのが、出力リダイレクトです。単にリダイレクトと言った場合、だいたい出力リダイレクトのことを指すと思ってよいでしょう。>に続いて出力したいファイルを指定して使います。

```
$ ls /usr/bin > ~/list.txt
```

ここでは`ls /usr/bin`コマンドを実行しています。これは`/usr/bin`ディレクトリにあるファイルの一覧を標準出力、つまりディスプレイに出力するコマンドです。もうお馴染みですね。その後ろにある>が出力リダイレクトの指定です。そして`~/list.txt`がリダイレクト先のファイル名となります。つまりこのコマンドを実行すると、本来ディスプレイに出力されるはずのファイル一覧が、ホームディレクトリ内の`list.txt`というファイルに書き込まれることになります。

10.2.2 「>」と「>>」の違い

また出力リダイレクトには>>というバージョンも存在します。>がリダイレクト先のファイルを新規作成するのに対し、>>は既存のファイルに追記します。なおファイルが存在しなかった場合は>と同じく新規作成します。出力先を常に初期化したいのか、あるいは追記したいのかによって使い分けましょう。以下のコマンドを実行すると、`/var/log`以下にあるファイル一覧を、`list.txt`の末尾に追記します。

```
$ ls /var/log >> ~/list.txt
```

10.2.3 よくある失敗例

よくある失敗例も紹介しましょう。以下は`example.log`というログファイルから`error`という文字を含む行を検索し、同じファイルに上書きしようとしています。つまりエラーが発生した行だけを抽出し、無関係な行を掃除した上で、ファイルを更新しようとしているわけです。ですがこのコマンドは期待通りに動きません。このコマンドを実行すると、`example.log`の中身は空になってしまいます。

```
$ grep -i error example.log > example.log
```

これはリダイレクト先が初期化される順序によるものです。上記コマンドが実行されると、まずシェルがリダイレクト先の`example.log`を空の状態で新規作成します。しかる後に`grep`コマンドが実行されます。そのため `grep` コマンドの検索対象は、空の新規ファイルになってしまうのです。リダイレクトはコマンドではなく、コマンドを起動するシェルが行うという点を意識するとよいでしょう。

10.2.4 入力リダイレクトとは

入力リダイレクトはその名の通り、標準入力をファイルに切り替えるリダイレクトです。<に続いてファイル名を指定すると、そのファイルの中身を標準出力に流し込んだこと同等になります。これが威力を発揮するのは、標準入力からしか入力を受け付けないコマンドを使う場合です。具体的な例を挙げて解説しましょう。

`grep`コマンドは指定された文字列を検索するコマンドですが、標準入力に対しても、ファイルを指定しても、同じように実行できます。以下の2つのコマンドは、どちらも`example.txt`に対して`error`と

いう文字列で検索をかけられます。`grep`のようなコマンドの場合、コマンド自体がファイルを直接開けるため、入力リダイレクトを使う意味はありません^{*1}。

```
$ grep error example.txt
$ cat example.txt | grep error
```

対して`tr`というコマンドがあります。これは標準入力から受け取った文字列に対し、文字の置換や削除を行うコマンドです。そしてポイントは、`tr`はファイルを直接開くことはできず、標準入力経由でしか入力を受け取れないという点です。以下は`example.txt`内のアルファベット小文字をすべて大文字に変換する例ですが、`tr`はファイルを直接開けないため、`cat`からパイプして渡す必要があります。

```
$ cat example.txt | tr '[:lower:]' '[:upper:]'
```

ですが入力リダイレクトを使えば、以下のようにファイルを標準入力に流し込めるのです。

```
$ tr '[:lower:]' '[:upper:]' < example.txt
```

`cat`とパイプでも代用できるため、入力リダイレクトが絶対に必要かと言われればそうではありません。出力リダイレクトと比較すると、利用頻度が低いのも事実でしょう。ですが入力リダイレクトを使いこなせば、コマンドをよりスマートに記述できます。

10.2.5 シェル全体の出力をリダイレクトする

`exec`コマンドを使うと、そのシェル全体の出力先を変更できます。例えば以下のコマンドを実行すると、これ以降、そのシェルで実行されるコマンドの標準出力はすべて`~/output.log`に書き込まれるようになります。

```
$ exec > ~/output.log
```

一般的な対話的シェルで使う機能ではありませんが、シェルスクリプト内でログを取りたい場合などには便利です。たとえば Cron で定期実行するスクリプトなどでは、

- 出力をディスプレイには一切出したくない。
- でもログは残したい。

というケースがよくあります。こういう時は都度コマンドの末尾でリダイレクトを指定するより、スクリプトの先頭で`exec`してしまう方が便利です。リダイレクトの指定が一箇所済むため、出力先を変更するのも簡単になります。

なおこのリダイレクトは、以下のコマンドで元に戻すこともできます^{*2}。

```
$ exec > /dev/tty
```

^{*1} `cat` コマンドと `grep` コマンドを繋いでいる縦棒がパイプです。パイプについては後述します。

^{*2} この方法で戻せるのは、制御端末を持つシェルに限ります。Cron から実行されたスクリプトは端末に関連づいていないため、`/dev/tty` は使えません。また通常は使う必要もありません。

10.2.6 実は 2 種類ある出力を区別する

実は出力には、標準出力とは別に**標準エラー出力**があります。デフォルトではどちらも同じように、ディスプレイに（混ざって）表示されるため、普段はあまり意識しないでしょう。ですがこれらは、個別にリダイレクトできるのです。そのため「標準出力は表示したくないが、エラー出力だけは記録したい」「それぞれを分けてファイルにリダイレクトしたい」といった制御ができます。

標準エラー出力は2で表します。以下の例では、あるコマンドの標準出力はディスプレイに出力し、エラーは**error.log**に出力しています。

```
$ コマンド 2> error.log
```

以下の例では、標準出力とエラー出力の双方を**all.log**に出力します。1は標準出力を表す数字で、2は標準エラー出力を表す数字です。2>&1で、「2（標準エラー出力）を1（標準出力）と同じ（&）場所にリダイレクト（>）する」という意味になります。2つの出力をマージする際の決まり文句として、この記法はそのまま覚えてしまってもよいでしょう。

```
$ コマンド > all.log 2>&1
```

ここで突然登場した1や2という数字は、**ファイルディスクリプタ番号**と呼ばれるものです。これはプロセスの読み書き対象となるファイルを OS が識別するための番号で、以下の3つは予約されています。

表 10.1: ファイルディスクリプタ番号とその意味

ファイルディスクリプタ番号	意味
0	標準入力
1	標準出力
2	標準エラー出力

ファイルディスクリプタについては本筋と外れてしまうため、詳しい解説は省略します。とりあえず出力とエラー出力が分かれていることと、標準入出力には固定の番号が振られており、番号でそれらを制御できることを覚えておきましょう。

10.3 パイプ

あるコマンドの標準出力を、後続のコマンドの標準入力に繋ぐのが**パイプ**です。本書でも既に何度か利用してきた機能になります。パイプを使うと標準入出力を経由して、複数のコマンドを連続実行し、データを順次加工できます。パイプは|でふたつのコマンドを繋いで記述します。以下のコマンドを実行してみましょう。

```
$ ls /usr/bin | grep ls
false
grub-menulst2cfg
inetutils-telnet
ls
lsattr
lsb_release
lsblk
lscpu
lshw
(... 略...)
```


この例では、まず `ls` コマンドが `/usr/bin` 以下にあるファイル一覧を標準出力に出力します。この出力が `grep` コマンドの標準入力に繋がります。そして `grep` コマンドは、`ls` コマンドの出力に対して `ls` という文字列でフィルタをかけています。結果として `/usr/bin` 以下にある、ファイル名に `ls` を含むコマンドのみが `grep` の標準出力（ここではディスプレイ）に出力されるというわけです。つまりパイプを活用すると、複数のコマンドをまとめて実行して、一気にデータを加工できるというわけです。`grep` や `sort` のようなコマンドを繋ぎ、データを望みの形に加工するのは、コマンドラインの定番テクニックです。

Linux のコマンドは、シンプルなコマンドを組み合わせることで複雑な処理を作ることから、レゴブロックのようだとよく言われます。そしてその中心にあるのがパイプです。コマンドを使いこなすなら、絶対にマスターしたい機能です。

10.4 リダイレクトにありがちな罠

サーバーの初期設定をする際、システム設定ファイルをコマンドで書き換えたいことはよくあります。以下は `/etc/example.conf` というファイルに、`key=value` という文字列を書き込む例です。ですがこのコマンドは、以下のように失敗します。

```
$ echo 'key=value' >> /etc/example.conf
bash: /etc/example.conf: Permission denied
```

その理由は、単純に権限の問題です。`/etc` ディレクトリ以下には、`root` 権限がないと書き込みが行えないためです。そして `root` 権限を得るためには、`sudo` が必要です。では `sudo` をつけて `echo` コマンドを実行してみましょう。

```
$ sudo echo 'key=value' >> /etc/example.conf
bash: /etc/example.conf: Permission denied
```

同様に、こちらもエラーとなってしまいました。`root` 権限でコマンドを実行しているのに、権限がないと怒られてしまっています。なぜでしょうか？

実はこれは、多くの人が経験する「あるある」です。上の例では、たしかに `echo` コマンドは `root` 権限で実行されています。ですが `root` 権限で実行されているのは、あくまで標準出力に文字列を出力しようとしている、`echo` コマンドだけなのです。先ほども `grep` の失敗例の部分で触れましたが、リダイレクトはコマンドではなくシェルが行っています。そのためいくら `echo` コマンドを `root` 権限で実行しようとも、その前段階のシェルが一般ユーザー権限で起動している以上、権限不足によって `/etc` 以下のファイルに書き込むことはできないのです。

この問題の解決策が `tee` コマンドを使うことです。`tee` は標準入力を受け取り、ファイルに書き込むと同時に、標準出力にも出力するコマンドです。以下のように使います。

```
$ echo 'key=value' | sudo tee -a /etc/example.conf
```

ファイルへの書き込みは `tee` コマンドが担当するわけですから、`tee` コマンドを `root` 権限で実行してあげれば、`/etc` 以下のファイルであっても書き換えられるのです。書き込みに `root` 権限が必要な場合は、単なるリダイレクトではなく、こうした小技を挟む必要があります。覚えておきましょう。

第 11 章 CLI でもテキストエディタを使おう

様々な設定をテキストで管理する Linux において、テキストエディタは必要不可欠なツールです。この章ではターミナル内で動く軽量の CLI のテキストエディタについて解説します。

11.1 テキストエディタの必要性

Linux をはじめとした Unix ライクな OS は、システムの様々な設定をテキストファイルの形で記述するのが基本です。今でこそ本番環境であれば、プロビジョニングツールによる OS 設定の自動化や、イミュータブル化、コンテナ化も一般的になってきましたが、個人向けの作業環境やテスト環境などでは、まだまだ人間が設定ファイルを手作業で編集することも多いでしょう。

そこで必須となるのが、対話的にテキストファイルを編集できる**テキストエディタ**です。テキストエディタと言えば、開発者の方は VS Code のような開発ツールを思い浮かべるかもしれませんが。ですが OS の設定においては、テキストの作成と、簡単な編集のみができれば十分です。VS Code のような高機能なツールは、まさに鶏を割くに牛刀を用いるが如しと言えるでしょう。またサーバー OS では不要なアプリケーションや GUI 環境はインストールしないのが鉄則です。そのためこうした重量級の GUI アプリケーションを、設定目的で使うことは通常ありません。

そこで必要になるのが、テキストのみのターミナル内で動く軽量の CLI エディタなのです。

11.2 Ubuntu の標準エディタ「nano」

nano は、Ubuntu における標準のテキストエディタです。最近では Ubuntu だけでなく、Fedora などのディストリビューションでも、後述する Vim に代わって nano が標準エディタとして採用されています。

従来、Unix ライクな OS の標準エディタと言えば **vi** でした。この vi に代わって nano が採用された最大の理由はその直感性にあります。vi は強力なエディタですが、非常に独特な操作体系を持っており、初心者には敷居が高いという問題があります。例えば vi はカーソルキーを押してもカーソルは動きませんし、キーボードをタイプしても、そのままでは文字が入力できません。エディタを終了するにしても、特殊なコマンドが必要です。対して nano は一般的なテキストエディタに近い操作感を持ち、画面下部に常に基本的なキーバインドが表示されるため、初めて使うユーザーでも迷わず操作できます。

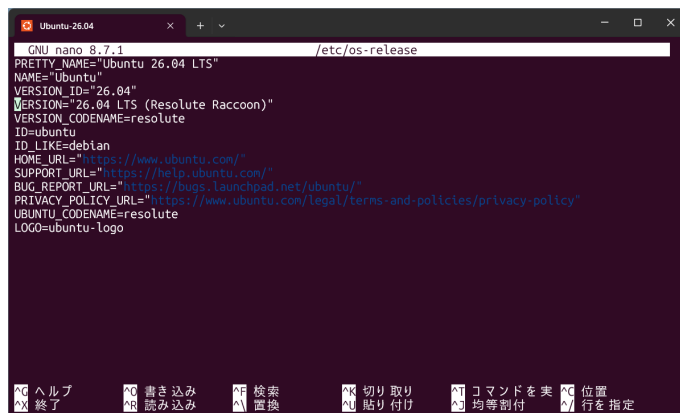


図 11.1: nano の画面

11.2.1 nano の基本操作

nano を起動するには、ターミナルで以下のコマンドを実行します。

```
$ nano ファイル名
```

これで nano が起動し、指定したファイルが開かれます。また指定したファイルが存在しない場合は、ファイルが作成されます。ファイル名を指定しないで起動した場合は、空の新規バッファが開かれます。Windows でメモ帳をはじめて起動した際と同じ挙動だと考えるとわかりやすいでしょう。

nano の画面は非常にシンプルで、上部にファイルの編集領域、下部にヘルプとなるキーバインド一覧が常に表示されています。カーソルキーでカーソルを自由に動かし、キーボードをタイプすることで文字を入力できます。まさに説明するまでもない、普通のテキストエディタだと言えるでしょう。主要な操作は以下の通りです。

表 11.1: nano の基本操作

キー操作	機能
Ctrl+G	ヘルプを表示
Ctrl+W	文字列を検索
Ctrl+K	行をカット
Ctrl+\	置換
Ctrl+U	ペースト
Ctrl+O	ファイルを保存
Ctrl+R	ファイルの内容を挿入
Ctrl+X	nano を終了

nano にはヘルプも完備されています。Ctrl+Gを押すとヘルプ画面が表示され、すべてのキーバインドと機能を確認できます。また、画面下部には常に主要なコマンドが表示されているため、操作に迷った時でもすぐに確認できます。

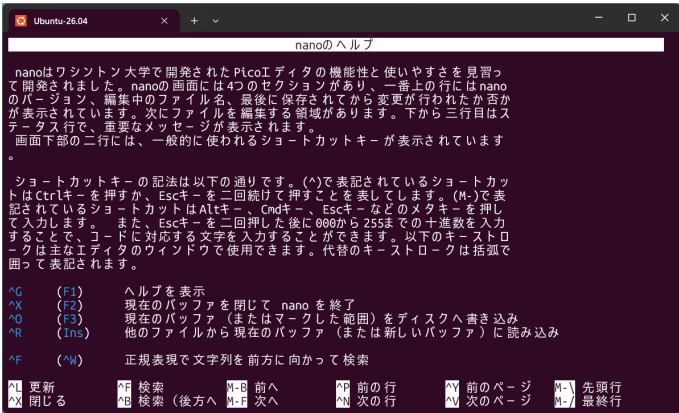


図 11.2: nano のヘルプ

11.3 Linux の定番エディタ「Vim」

もともと Unix の世界で長年親しまれ、標準エディタの地位を確立していた存在が vi です。vi は 1976 年に開発された歴史あるエディタで、現在でも多くの Unix 系システムで利用されています。ただ

し正確に言うと、現在ではオリジナルの vi ではなく、より進化した実装である **Vim (Vi IMproved)** が vi の代わりにインストールされていることがほとんどです。Ubuntu でもデフォルトで Vim（正確には機能限定版の vim.tiny）がインストールされており、vi コマンドを実行すると実際には vim.tiny が起動します。nano と同様に、編集するファイル名を指定して起動もできます。

```
$ vi ファイル名
```

なお vim.tiny は最小限の機能のみを持つ軽量版のため、プログラミング目的などで本格的に Vim を使いたい場合は、以下のコマンドで高機能版（vim.basic）をインストールすることをお勧めします。

```
$ sudo apt install -U -y vim
```

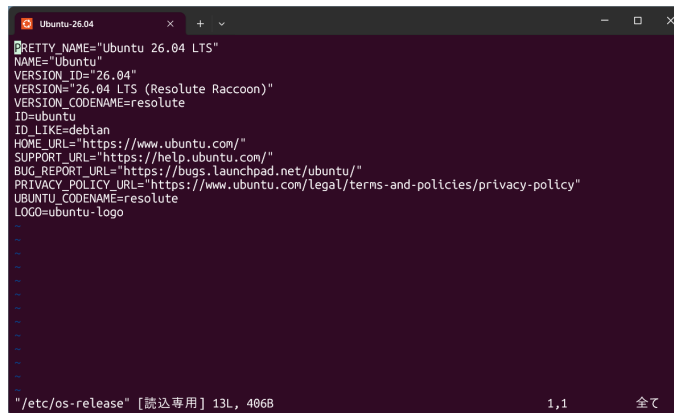


図 11.3: Vim の画面

11.3.1 Vim の基本操作

nano や Windows のメモ帳といったエディタは、カーソルキーでカーソルを動かし、キーボードを押せば、その場に直接文字を入力できます。ですが Vim はそうではありません。Vim は複数の**モード**を切り替える、非常に特殊な操作体系を持っています。

ノーマルモード

Vim 起動時のデフォルトモードが、**ノーマルモード**です。ここでは文字の入力はできず、カーソル移動やテキストの削除、コピーといった編集作業を行います。

表 11.2: ノーマルモードでの主な操作

キー操作	機能
h, j, k, l	カーソルを左、下、上、右へ移動
x	カーソル位置の文字を削除
dd	行全体を削除
yy	行全体をコピー
p	貼り付け

インサートモード

ノーマルモードで `i` や `a` キーを押すと、実際に文字を入力するインサートモードに遷移します。`ESC` キーを押すと、ノーマルモードに戻れます。

表 11.3: インサートモードでの主な操作

キー操作	機能
<code>i</code>	カーソル位置からインサートモードに遷移
<code>I</code>	行頭にカーソルを移動してインサートモードに遷移
<code>a</code>	カーソルの次の位置からインサートモードに遷移
<code>A</code>	行末にカーソルを移動してインサートモードに遷移
<code>o</code>	カーソルの次の行に新しい行を挿入してインサートモードに遷移
<code>O</code>	カーソルの前に新しい行を挿入してインサートモードに遷移
<code>Esc</code>	ノーマルモードに戻る

コマンドラインモード

コマンドラインモードは、ファイルの保存、検索、置換などを行うモードです。ノーマルモードで `:` や `/` に続いて、コマンドを入力します。

表 11.4: コマンドラインモードでの主な操作

キー操作	機能
<code>:w</code>	ファイルを保存
<code>:q</code>	Vim を終了
<code>:wq</code>	ファイルを保存して Vim を終了
<code>:q!</code>	保存せずに強制終了
<code>/文字列</code>	文字列を検索

11.4 Vim と人気を二分する「Emacs」

Emacs は、Vim と並んで Unix 系システムで長年愛用されている高機能テキストエディタです。1976 年に初版が登場して以来、現在でも活発に開発が続けられている歴史あるエディタです。

Emacs の最大の特徴は、その拡張性にあります。Emacs はテキストエディタの顔をしていますが、その実体は Emacs Lisp と呼ばれるプログラミング言語の実行環境です。この言語で書かれた任意のプログラムをエディタ上で実行可能なため、まさに無限の拡張性を持っているのです。世界中の開発者によって開発された、膨大な数の拡張機能（パッケージ）が利用可能で、VS Code に匹敵する統合開発環境として使用することも可能です。そのため「Emacs はテキストエディタではなく環境である」とも言われます。

Emacs は Vim とは異なり、非モーダルエディタです。そのため一般的なテキストエディタと同様に、起動後すぐに文字入力が可能です。なお筆者は本書を含め、ここ 20 年ほどの間のすべての原稿を Emacs で執筆しています。

11.4.1 Emacs のインストールと起動

Ubuntu で Emacs をインストールする方法は複数ありますが、最新版を利用したい場合は Snap パッケージの利用をおすすめします。ユーザーが使うアプリケーションは進化が速く、より新しいバージョ

ンを使いたいと考える人が多いでしょう。Snap であれば、こうしたバージョンアップに追従しやすいからです。

```
$ sudo snap install emacs --classic
```

Snap からインストールした Emacs は、GUI でも動作します。そのまま `emacs` コマンドを実行すると、Windows のデスクトップ上で Emacs が起動してしまいます。そのためターミナル内で起動する場合は、以下のように「`-nw`」オプションをつけて実行してください。

```
$ emacs -nw
```

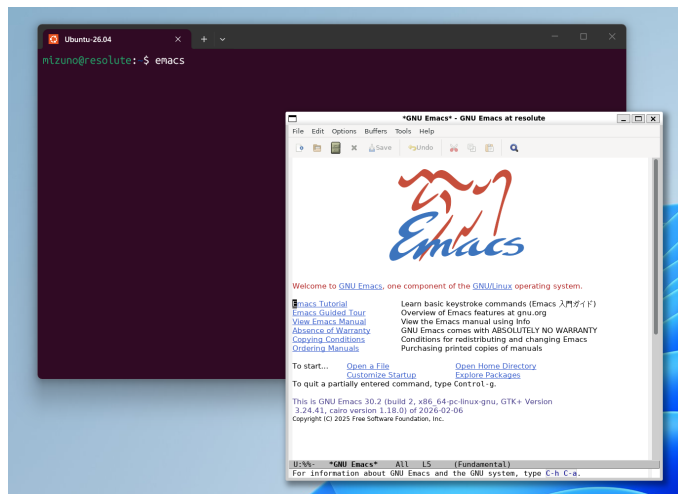


図 11.4: `-nw` オプションなしで起動した Emacs

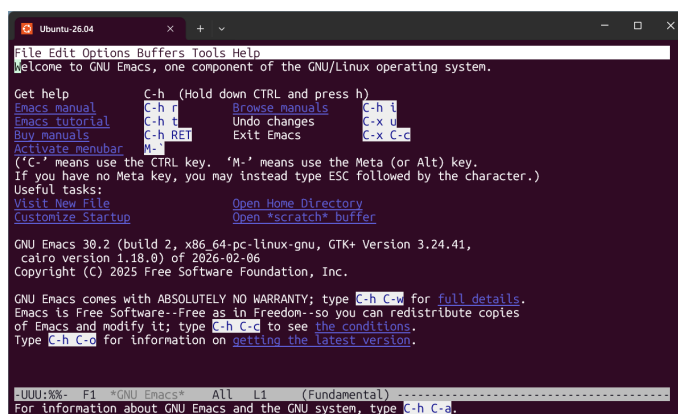


図 11.5: ターミナル内で起動した Emacs

11.4.2 Emacs の基本操作

Emacs の操作は、主に **Ctrl** キー（C と表記）や **Alt** キー（M と表記）と、別のキーの組み合わせで行います。例えばカーソルを右に動かすには、**Ctrl** キーと **f** キーを同時に押すのですが、これは **C-f** と表記します。同様に **Alt** キーと **x** キーを同時に押すことを、**M-x** と表記します。カーソル移動をはじめとした編集系のキーバインドは、シェルをはじめ、Emacs 以外の様々な環境でも一般的に採用されています。VS Code などともそうですが、キー操作を Emacs 風にする設定や拡張機能が用意されているアプリも珍しくありません。Emacs そのものは使わなくとも、この **Emacs キーバインド** に慣れておくと非常に便利です。

カーソル操作

表 11.5: カーソル操作

キー操作	機能
C-f	カーソルを右に移動
C-b	カーソルを左に移動
C-n	カーソルを下に移動
C-p	カーソルを上を移動
C-a	カーソルを行頭に移動
C-e	カーソルを行末に移動

ファイル操作

表 11.6: ファイル操作

キー操作	機能
C-x C-f	ファイルを開く
C-x C-s	ファイルを保存
C-x C-w	ファイルを別名保存
C-x C-c	Emacs を終了する

編集操作

表 11.7: 編集操作

キー操作	機能
C-k	カーソル位置から行末まで削除
C-Space	範囲選択の開始
C-w	選択範囲のカット
M-w	選択範囲のコピー
C-y	貼り付け
C-s	前方検索
C-r	後方検索

ウィンドウ操作

表 11.8: ウィンドウ操作

キー操作	機能
C-x 2	ウィンドウを水平分割
C-x 3	ウィンドウを垂直分割
C-x o	次のウィンドウに移動
C-x 0	現在のウィンドウを閉じる

11.5 Microsoft 製の新型エディタ「Microsoft Edit」

Microsoft Edit（以下 `msedit`）は、Microsoft が 2025 年に新たにリリースしたオープンソースの CLI 向けテキストエディタです。CLI に不慣れなユーザーでも使いやすいエディタを提供することを目標に開発された、期待のニューフェイスです。

`vi` が使いにくいという理由で、Ubuntu は `nano` を標準エディタとして採用したのは前述の通りです。ですが正直な話、`nano` も一般的なエディタと呼ぶには特殊すぎる操作体系を持っており、お世辞にも使いやすいとは言えません。また、`Vim` や `Emacs` は非常に高機能ですが、学習コストが高く、初心者には敷居が高いという問題がありました。そこで Microsoft が開発した、現代的で直感的な CLI エディタが `msedit` です。最近の Microsoft としては珍しくない話ですが、Windows 版の他に Linux 版も同時にリリースされています。

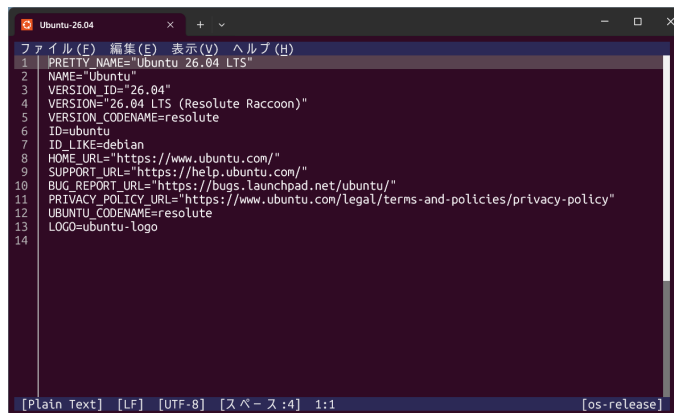


図 11.6: msedit の画面

11.5.1 msedit のインストール

`msedit` は Snap パッケージでインストールできるのですが、これはお勧めしません。なぜなら Snap パッケージ版に特有の仕様として、自分のホームディレクトリ外のファイルが編集できないためです。これは `sudo` をつかって起動しても変わりません。CLI のエディタを使う理由のひとつに、`/etc` 以下にあるシステム設定ファイルを編集したいというものがあります。Snap 版 `msedit` では、この目的を達成できないのです。

そこで以下のコマンドを実行し、GitHub^{*1}から最新のリリースを手動でインストールしてください。

^{*1} <https://github.com/microsoft/edit>


```
$ wget https://github.com/microsoft/edit/releases/download/v2.0.0/edit-2.0.0-x86_64-linux-gnu.tar.gz
$ tar xf edit-2.0.0-x86_64-linux-gnu.tar.gz
$ sudo install -m 755 -o root -g root edit /usr/local/bin/edit
```

msedit は、現代的な GUI エディタに慣れたユーザーが違和感なく使えるよう設計されています。引数にファイルを指定した場合や、ファイル名を指定しないで起動した場合の挙動は、nano と同じです。

```
$ edit ファイル名
```

11.5.2 msedit の基本操作

表 11.9: msedit の基本操作

キー操作	機能
Ctrl+n	新規バッファを開く
Ctrl+o	ファイルを開く
Ctrl+s	ファイル保存
Ctrl+z	元に戻す
Ctrl+y	やり直し
Ctrl+c	コピー
Ctrl+x	切り取り
Ctrl+v	貼り付け
Ctrl+a	すべてを選択
Ctrl+q	エディタを終了

見ての通り、一般的な GUI のアプリケーションと共通したキーバインドが採用されています。既存のエディタを使い慣れたユーザーならすぐに使いこなせるでしょう。また msedit は、ターミナル内でもマウスによる操作が可能です。GUI アプリと同じように、マウスでメニューをクリックして操作できます。まさに msedit は、従来の CLI エディタとモダンな GUI エディタの良いところを組み合わせた、新世代のテキストエディタと言えるでしょう。

11.6 Ubuntu の標準エディタを変更する

Linux において、自動的にエディタが起動する瞬間というものがあります。具体的にはシステムのスケジューラーである `crontab` の編集時、バージョン管理システムに変更をコミットする `git commit` 時、`systemd` のユニットファイルを編集する `systemctl edit` の実行時などです。前述のように Ubuntu のデフォルトエディタは nano ですから、このタイミングでシステムが自動的に nano を起動します。しかし慣れ親しんだエディタがある場合は、この標準エディタを変更したいと思うでしょう。

11.6.1 環境変数による設定

システムが起動するエディタは、環境変数 `VISUAL` と `EDITOR` で制御できます。標準のエディタを Vim にしたい場合は、以下のように環境変数を設定してください。

```
$ export VISUAL=/usr/bin/vim
$ export EDITOR=/usr/bin/vim
```

export コマンドでの設定は、シェルを終了すると失われてしまいます。恒久的に設定する場合は、シェルの設定ファイルに記述してしまうとよいでしょう。

```
$ echo 'export EDITOR=vim' >> ~/.bashrc
$ echo 'export VISUAL=vim' >> ~/.bashrc
```

なぜ環境変数がふたつあるのかと、疑問に思った方もいるでしょう。実は大昔の Unix では、全画面を占有して動くスクリーンエディタと、そうした機能がないラインエディタを区別していたためです。現在であればこうした問題はほとんどないのですが、歴史的な事情により、どちらか片方しか参照しないコマンドも存在する。そのため意図しない事故を防止する意味でも、両方の環境変数に同じ値を設定することを推奨します。

11.6.2 update-alternatives によるシステムワイドな設定

Ubuntu では、特定の機能に対して、実装が複数存在するケースがあります。例えば先ほどの Vim の例であれば、`vi` という機能に対して、機能限定版の `vim.tiny` と、より高機能な `vim.basic` というふたつの実装がありました。そしてインストールされているパッケージなどの状態によって、`vi` コマンドを実行した時に、どちらの実装が呼び出されるかが決定されます。この実装の切り替えを実現しているのが、**Alternatives** という機能です。

システムの標準エディタもまた、Alternatives で変更できます。`update-alternatives` コマンドを、以下のように実行してください。

```
$ sudo update-alternatives --config editor
```

すると以下のように、インストールされているエディタのリストと、選択肢の番号を入力するプロンプトが表示されます。デフォルトでは `nano` が選択されていますが、ここで `3` を入力すれば、デフォルトのエディタを `vim.basic` に変更できます。

```

mizuno@resolute:~$ sudo update-alternatives --config editor
alternative editor (/usr/bin/editor を提供) には 4 個の選択肢があります。

  選択肢   パス               優先度   状態
-----
* 0        /bin/nano                40      自動モード
  1        /bin/ed                 100     手動モード
  2        /bin/nano                40      手動モード
  3        /usr/bin/vim.basic       30      手動モード
  4        /usr/bin/vim.tiny        15      手動モード

現在の選択 [*] を保持するには <Enter>、さもなければ選択肢の番号のキーを押してください:

```

図 11.7: Alternatives によるデフォルトエディタの変更

なお Snap からインストールした `msedit` は、デフォルトでは Alternatives の選択肢として表示されません。そこで以下のコマンドを実行すると、`msedit` を優先度 50 で選択肢として登録できます。

```
$ sudo update-alternatives --install /usr/bin/editor editor /usr/local/bin/edit 50
```

再度 `update-alternatives --config` を実行してみると、`/usr/local/bin/edit` が追加されていることがわかります。msedit を標準エディタにしたい場合は参考にしてください。

```

mizuno@resolute: $ sudo update-alternatives --config editor
alternative editor (/usr/bin/editor を提供) には 5 個の選択肢があります。

  選択肢   パス               優先度   状態
-----
* 0        /usr/local/bin/edit   50       自動モード
1          /bin/ed              -100      手動モード
2          /bin/nano             40       手動モード
3          /usr/bin/vim.basic    30       手動モード
4          /usr/bin/vim.tiny     15       手動モード
5          /usr/local/bin/edit   50       手動モード

現在の選択 [*] を保持するには <Enter>、さもなくば選択肢の番号のキーを押してください:

```

図 11.8: Alternatives に msedit を追加した状態

11.7 エディタが存在しない時の対処法

最近では、IoT 向けやコンテナなど、人間が対話的に操作しないことが前提の Linux 環境も増えてきました。こうした環境では、無駄なソフトウェアを一切削ぎ落として軽量化するのが、セキュリティ面においても鉄則です。そのため、どの環境にも存在すると言われていた `vi`すらインストールされていないことも当たり前になってきています。

こうした環境に、デバッグ目的などで接続した際に、設定ファイルが編集できず困った経験がある方もいるのではないのでしょうか。ここでは、対話的なスクリーンエディタがない場合の回避策を紹介します。

11.7.1 ed を使った編集

これまで紹介した nano、Vim、Emacs、msedit は、いわゆる **スクリーンエディタ**と呼ばれる種類のエディタです。スクリーンエディタは、ターミナル画面全体を使ってファイルの内容を表示し、カーソルを自由に移動させて編集を行います。対して `ed` は **ラインエディタ**と呼ばれる、より古い形式のエディタです。ラインエディタは画面全体を使わず、一行ずつコマンドを入力してファイルを編集します。お世辞にも使いやすいエディタではなく、常用するようなものではありません。ですが nano や vi がない環境では、ed が最後の砦となることもあります。

ed の基本操作

ラインエディタでファイルを編集する例を示します。まず `ed` コマンドで起動します。

```
$ ed
```

`ed` ではアルファベット一文字のコマンドで操作します。`a` を押すと、テキストの追加モードになりますので、文字列を入力してください。入力を終了するには、`.` (ドット) を入力します。

```

a          ← a を入力して追加モードに入る
This is sample text.
.          ← 文字列を入力する
.          ← ドットを入力して、追加モードを終了する

```

入力した内容をファイルに保存するには `w` とファイル名を入力します。終了するには `q` を入力します。

```
w example.txt    ← 入力した内容を example.txt に保存する
q                ← ed を終了する
```

より詳細な編集コマンドについては、マニュアル^{*2}を参照してください。

11.7.2 sed によるファイル編集

sed はストリームエディタの略で、人間が対話的に操作するのではなく、主にスクリプトの中などで文字列を加工するために使われるコマンドです。対話的に操作しないため、命令はすべてコマンドラインから引数の形で与えます。こうした特徴を持つため、人間が対話的に操作しないコンテナなどであっても、インストールされていることの多いツールです。

sed の基本操作

sed は非常に高機能なエディタですが、その中でよく使うであろう表示、削除、置換、追加の例を紹介します。

`example.txt` の 2 行目を表示します。

```
$ sed -n '2p' example.txt
```

`example.txt` の 3 行目を削除して上書き保存します。

```
$ sed -i '3d' example.txt
```

`example.txt` 中の `old` を `new` に置換して上書き保存します。

```
$ sed -i 's/old/new/g' example.txt
```

`example.txt` 中の 2 行目の後ろ (3 行目) に `new line` という新しい行を追加します。

```
$ sed -i '2a\n new line' example.txt
```

sed は普段からよく使う Linux の超定番ツールなので、これらの使い方は覚えておいて損はありません。

11.7.3 cat によるファイルの作成と追記

cat はファイルの中身を表示する目的でよく使用されますが、その実態は標準入力を標準出力にそのまま出力するコマンドです。つまり、標準入力をキーボードとし、標準出力をファイルにリダイレクトすれば、キーボードからの入力をそのままファイルに保存することができます。既存ファイルの編集こそできませんが、新規ファイルの作成や、ファイル末尾への追記であれば、エディタや **sed** を使わずとも、**cat** だけで実現可能です。

^{*2} https://www.gnu.org/software/ed/manual/ed_manual.html

cat の基本操作

`cat` コマンドを引数なしで実行すると、標準入力から入力を受け付けます。これをファイルにリダイレクトすることで、簡易的なテキストエディタのように利用できます。複数行の入力も可能で、終了したい場合は `Ctrl+D` を押してください。

以下は `example.txt` を新規作成して文字を入力する例です。

```
$ cat > example.txt
ここに文字列を入力する
Enter を押すと改行する
複数行の入力も可能
(Ctrl+D を押すと終了する)
```

リダイレクトを `>>` と指定することで、既存ファイルの末尾に文字列を追記できます。使い方は前述の新規ファイル作成と同じで、任意の文字列を入力した後、`Ctrl+D` を入力して終了してください。以下既存ファイルの末尾にテキストを追記する例です。

```
$ cat >> example.txt
```

スクリプトの中では、人間が直接キーボードから入力できません。こうした場合は **ヒアドキュメント** を使うと便利です。ヒアドキュメントとは、シェルにおいて複数行のテキストをコマンドに渡すための仕組みです。例えば以下のように `cat` コマンドを実行すれば、複数行からなる `config.conf` という設定ファイルを自動的に生成できます。

```
$ cat > config.conf << EOF
server_name=example.com
port=8080
debug=true
EOF
```

この方法は、サーバーを自動的に初期化するような際に便利です。

Linux システムを操作する上で、CLI のテキストエディタへの習熟は避けて通れません。どのエディタを選ぶかは個人の好みや用途によりますが、少なくとも一つに習熟しておけば、Linux 上で効率よく作業を進められるようになるでしょう。

普段使いのエディタにどれを選ぶかは個人の好みですが、いざ本番環境に触れる際に、「この環境には Emacs がインストールされていないので、私には設定できません」というのでは困ります。nano や vi といった、システムに標準でインストールされているツールも、一通りは動かせるようになっておきましょう。またエディタが利用できない環境も、昨今では珍しくありません。sed や cat などのコマンドも、基本的な使い方は把握しておきましょう。

第 12 章 ワンライナーで作業を劇的に効率化しよう

パイプを使って複数のコマンドを連携させれば、より複雑な処理を自由に組み立てられます。この章では主にパイプを活用した、今すぐ使える便利なコマンドの具体例を紹介します。

12.1 ワンライナーとは

1つのコマンドを入力して実行し、完了したら次のコマンドをまた入力し、というのが通常のシェルの動作です。ですがパイプやループ、論理演算などを組み合わせれば、複数のコマンドを1行に詰め込むことも可能です。

```
$ grep ERROR access.log | sort | uniq -c | sort -nr
```

上記の例は、「access.log というログファイルから ERROR という文字が含まれる行を抽出して、エラーの種類ごとに数を集計し、多い順に並べる」という処理を、たった1行で行っています。こうした1行のコマンドだけで処理を完結させる書き方を**ワンライナー**と呼びます。ワンライナーはプロンプト上に直接コマンドを入力するため、テキストエディタを開いてスクリプトを組む必要もなく、非常に手軽で便利です。つまり複雑な処理をその場でインスタントに組み立て、同時に使い捨てられるため、作業効率は飛躍的に上がります。

12.2 今日から使えるワンライナーの例

日常のよくあるタスクを解決するための実践的なワンライナーの例を、簡単な例から複雑なものまで紹介します。なお各コマンドの詳細な使い方やオプションの意味は、適宜マニュアルを参照してください。

あれとあれを組み合わせれば、こんな便利なことができるかも？ と考えられるようになれば、中級者への第一歩です。ぜひ自分だけの便利なコマンドの活用例を考えてみてください。

12.2.1 条件に合ったファイルを一括削除する

まずはシンプルな例から見ていきましょう。**find**コマンドを使って、条件に合ったファイルをまとめて削除する例です。**find**コマンドは、指定したパス以下にあるファイルやディレクトリを検索するコマンドですが、条件や、見つかったファイルに対するアクションを指定できます。以下の例は、検索対象はカレントディレクトリ以下（.）、検索対象はファイル（**-type f**）、最終更新日が30日以上前（**-mtime +30**）という条件でファイルを検索し、見つかった対象を削除（**-delete**）するコマンドです。古いバックアップファイルをまとめて削除する、といった時に使えます。

```
$ find . -type f -mtime +30 -delete
```

他にも期間を分単位で指定したり、直近 xx 日以内を指定したり、更新日ではなくアクセス日で検索するといったことも可能です。

12.2.2 重複したファイルを検出する

もうひとつ、**find**を使って重複ファイルを検出する例を紹介しましょう。

```
$ find . -type f -exec md5sum {} + | sort | uniq -w32 -d
```

先ほどの例では `-delete` オプションで、見つかった対象を削除しました。今回は見つかった対象に任意のコマンドを実行する `-exec` オプションを使用します。このオプションは、その後に実行したいコマンドを記述します。またこの際、`{}` が対象のファイル名に置き換えられます。ここでは見つかったファイルのハッシュ値を、`md5sum` コマンドで計算しているわけです。その後の `+` はコマンドの終端を意味するターミネーターです。

`md5sum` コマンドの出力結果は、`sort` コマンドにパイプしています。つまりハッシュ値でソートしているわけです。そして続いて `uniq` コマンドにパイプしています。`uniq` はソート済みのファイルから、重複した行を検出するコマンドです。デフォルトでは行全体を比較しますが、ここでは `-w32` オプションを使って先頭 32 文字、つまり MD5 チェックサム分のみを比較しています。`-d` オプションは、重複があった行のみを表示するオプションです。つまりこれで、同じハッシュ値を持つファイルを検出できるというわけです。

このワンライナーを実行すると、重複したファイルのハッシュ値とファイル名のリストが得られます。なのでさらに `cut` コマンドにパイプしてファイル名だけを取り出し、`xargs` コマンドにパイプして `rm` コマンドで削除する、といった応用も可能です。

12.2.3 ファイルの拡張子を変更する

Bash では、ループを一行で書くこともできます。以下はシンプルな `for` ループを使い、ファイルの拡張子を変更する例です。

```
$ for file in *.txt; do mv "$file" "${file%.txt}.md"; done
```

ここではカレントディレクトリにある、拡張子が `.txt` のファイルを対象にループを行っています。ループが回ると、対象となるファイル名が順次、シェル変数 `$file` に格納されます。ループの内部ではこの変数を使い、`mv` コマンドでリネームを行っているわけです。ここでの特徴は `${file%.txt}` という記述です。これはパラメータ展開と呼ばれる機能で、変数 `$file` の末尾から、`.txt` という文字列を削除して展開するという意味になります。展開結果の後に `.md` を付加しているため、これで `example.txt` というファイル名を `example.md` に変更できるというわけです。

ループを即興で書けるようになると、コマンドでの作業効率が一気に上がります。ぜひマスターしましょう。

12.2.4 ファイルを一括でリネームする

これは前述の拡張子変更の応用で、基本的な構造は同じです。

```
$ for file in *\ *; do mv "$file" "${file// /_}"; done
```

違うのは、対象のファイルを `*\ *` と指定している点です。これによって、ファイル名にスペースを含むファイルが対象になります。またパラメータ展開が `${file// /_}` になっています。これは変数内のスペースを `_` に置換するという意味になります。つまりこれはファイル名に含まれるスペースを、アンダースコアに置き換えるワンライナーです。

ファイル名にスペースが含まれていると、コマンドやスクリプトによっては誤動作の原因になるため、アンダースコアに置き換えるというのは一般的なワークアラウンドです。こうした作業もワンライナーを使えば、一発で行えるわけです。

12.2.5 ファイルを並列ダウンロードする

`xargs` コマンドは、標準入力からスペースや改行で区切られた文字列を読み込み、それらの文字列を引数として別のコマンドを起動します。簡単に言えば、引数のリストを渡すことで、引数が異なる同じコマンドを複数並列に起動できるわけです。以下は `xargs` を使って `curl` コマンドを複数起動し、ファイルを並列ダウンロードする例です。

```
$ xargs -P 8 -n 1 curl -O <<'EOF'
https://releases.ubuntu.com/24.04/ubuntu-24.04.4-desktop-amd64.iso
https://releases.ubuntu.com/24.04/ubuntu-24.04.4-live-server-amd64.iso
https://releases.ubuntu.com/22.04/ubuntu-22.04.5-desktop-amd64.iso
https://releases.ubuntu.com/22.04/ubuntu-22.04.5-live-server-amd64.iso
EOF
```

ダウンロードしたいファイルの URL を、`<<'EOF'` から `EOF` の間に 1 行 1 URL で列挙します。この書き方は **ヒアドキュメント** と呼ばれ、標準入力に渡す内容をインラインで記述できる便利機能です。ここではバージョンの異なる Ubuntu のインストールメディア、4 種類の URL を渡しています。`xargs` はこの 4 行を受け取り、`curl` コマンドを 4 並列で実行します。

標準入力は、ファイルや別のコマンドからだけでなく、このように手動で流し込むこともできるので、一時ファイルを作らなくてよい、使い捨ての処理を書く際に便利です。

12.2.6 肥大化しているディレクトリを探す

ディスクの空き容量が少なくなった時は、余計なファイルを掃除して、空き容量を確保するのが鉄則です。どのディレクトリがどのくらいの容量を使っているか、それを把握するのに便利なのが `du` コマンドです。`du` は引数に指定したファイルやディレクトリが、どのくらいの容量を消費しているかを表示します。

```
$ du -hs * | sort -rh | head
```

ここでは `du` に、2 つのオプションをつけています。1 つは `-s` オプションで、これは引数に指定された項目ごとの合計サイズを表示します。つまり引数にディレクトリを指定した場合、そのディレクトリが含むすべてのサブディレクトリとファイルの合計サイズを表示できるわけです。もう 1 つが `-h` オプションで、これは人間にとって見やすく表示するためのものです。引数には `*` を指定し、カレントディレクトリの全ディレクトリとファイルを対象としました。

その結果を `sort` コマンドにパイプしてソートします。`-r` オプションを指定しているため、ソートした項目が降順、つまりサイズが大きい順に並びます。最後にこれを `head` コマンドにパイプしています。`head` は標準入力から渡された内容の、先頭 10 行だけを表示するコマンドです。つまりディスク容量を消費しているワースト 10 のディレクトリを表示できるというわけです。

12.2.7 Yes/No プロンプトを作る

`apt` コマンドでパッケージのインストールを行おうとすると、本当に実行してよいか、**Yes/No** を入力する確認プロンプトが表示されます。こうした確認プロンプトを自前で実装できるのが以下のワンラ

イナーです。シェルスクリプトの内部などで、本当に実行してよいかの確認を促す場合に利用してみてください。

```
$ read -p "Continue? [y/N]: " yn && [[ $yn == y ]] && echo 'Run!'
```

`read` コマンドは、標準入力から入力を読み取り、その結果を変数に格納するコマンドです。ここではキーボードからの入力を、変数 `yn` に格納しています。続いて `[[ ]]` (二重ブラケット) が登場しました。これは Bash の条件判定です。シェル変数 `$yn`、つまり押されたキーが `y` かどうかを判定しています。もし `y` キーが押されていた場合、最後の `echo` コマンドが実行されます。実際はこの `echo` コマンド部分を、任意のコマンドに置き換えてみてください。これであるコマンドを実行してよいかの確認プロンプトが作れます。

この例では、コマンド同士を繋ぐのにパイプではなく `&&` が使われています。これは **AND** を表す論理演算で、左辺が真だった場合に右辺の評価を行います。つまり 2 つのコマンドを `&&` で繋ぐと、左辺 (最初に実行するコマンド) が成功した場合に、右辺 (後続のコマンド) を実行するという意味になります。例えば以下は、`example` というディレクトリを作成し、作成に成功したら `cd` コマンドで移動する例です。こうしたコマンドの書きかたはシェルの定石ですので、この機会に覚えておきましょう。

```
$ mkdir example && cd example
```

ちなみに 2 つのコマンドを `||` で繋ぐと、左辺のコマンドが失敗した場合に、後続のコマンドを実行するという意味になります。こちらはエラー時の後処理などによく利用されています。以下は、あるコマンドが失敗した場合に `exit` でシェルを終了する例です。スクリプト内のエラーハンドリングで、こうした書き方をよくします。これもセットで覚えておくといでしょう。

```
$ 何らかのコマンド || exit 1
```

12.3 WSL で使えるワンライナーの例

WSL 環境では、Windows 側のコマンドを直接叩くことができるなど、通常の Linux 環境よりも多くのことができます。こうした WSL ならではのワンライナーも紹介しましょう。

12.3.1 エクスプローラで今のディレクトリを開く

WSL では、Windows にインストールされている `exe` ファイルをシェルから起動できます。例えば `explorer.exe` の引数にカレントディレクトリを指定すれば、そのディレクトリをエクスプローラで開けます。

```
$ explorer.exe .
```

WSL で作業中に、GUI のファイルマネージャーを操作したくなることもあるでしょう。何度もフォルダをダブルクリックして、WSL の深いディレクトリ階層を辿るのは面倒ですが、このコマンドであれば目当てのディレクトリを一発で開けて便利です。

12.3.2 Windows と WSL のパスを相互変換

Windows のエクスプローラから、WSL 内のファイル进行操作できます。また WSL 内からも、C ドライブ上にある Windows のファイル进行操作できます。このように Windows/Linux という異なる環境

でありながら、シームレスに相互のファイル进行操作できるのが WSL のメリットのひとつです。ですが Windows と Linux では、ファイルのパス表記が異なります。また WSL2 は Hyper-V で動いている仮想マシンですから、ファイルはディスクイメージ内に保存されており、C ドライブ以下のツリー上にあるわけではありません。9p のネットワーク共有を経由してアクセスする必要があり、UNC パスを指定しなければなりません。

そこで Windows と WSL のパスを相互変換するコマンドが `wslpath` です。引数に Windows のパスを指定すると、WSL 上からアクセスできるパスに変換します。以下は Windows ユーザーのホームフォルダである `C:\Users\mizuno` を指定した例です。 `/mnt/c/Users/mizuno` という変換結果が表示されました。WSL 上からは、このパスで Windows のホームフォルダにアクセスできるというわけです。

```
$ wslpath "C:\Users\mizuno"  
/mnt/c/Users/mizuno
```

また `-w` オプションと WSL のパスを指定すると、Windows 上からアクセスできる UNC パスに変換してくれます。以下は WSL 内のホームディレクトリである `/home/mizuno` を指定した例です。UNC パスである `\wsl.localhost\Ubuntu-26.04\home\mizuno` という変換結果が表示されました。Windows のエクスプローラのアドレスバーにこのパスを入力すると、エクスプローラから WSL 内のホームディレクトリを閲覧できます。

```
$ wslpath -w /home/mizuno  
\\wsl.localhost\Ubuntu-26.04\home\mizuno
```

12.3.3 WSL から Windows の既定アプリで開く

Windows には `start` というコマンドがあります。これは macOS でいう `open` コマンドに相当するもので、Windows の関連づけに従ってアプリを起動できます。以下は WSL 上にあるテキストファイルを開く例です。

```
$ cmd.exe /c start $(wslpath -w ./example.txt)
```

WSL から Windows の `cmd.exe` (コマンドプロンプト) を呼び出しています。 `cmd.exe` の `/c` オプションは、その後に続くコマンドを実行して、コマンドプロンプトを終了するためのものです。そして実行するコマンドとして `start` コマンドを、 `start` コマンドの引数として、カレントディレクトリにある `example.txt` のパスを渡しています。この際、内部的に前述の `wslpath` を呼び出し、WSL 上のファイルパスを Windows 上からアクセスできるパスに変換しています。

つまりこのコマンドを実行すると、Windows 上から見える `example.txt` を引数として、 `start` コマンドが実行されます。拡張子が `.txt` のファイルですから、(デフォルトでは) メモ帳が起動し、このファイルが編集可能な状態になるというわけです。

WSL 上のファイルを、Windows のアプリに渡したい時に便利なワンライナーです。

12.3.4 WSL からクリップボードにコピーする

WSL のターミナルに表示された結果を、Windows から利用したいことはよくあります。例えばコマンドの出力を、Windows のテキストエディタにコピペしたいような場合です。Windows ターミナル上の出力をドラッグして選択した上で、右クリックすればコピーできるのはご存知の通りです。ですが実は、クリップボードへのコピーを行うコマンドが存在します。それが `clip.exe` です。

```
$ echo "hello" | clip.exe
```

`clip.exe`は標準入力から渡された内容を、クリップボードにコピーします。上記の例では、`Hello`という文字列がクリップボードにコピーされます。プラットフォームをまたいだクリップボードの共有が可能になるというわけです。いちいち手でコピーせずとも、コマンドの出力を直接 Windows に渡せるため、ドキュメントの作成などに威力を発揮するでしょう。

ただし WSL 上で出力した日本語をコピーし、Windows 上でペーストすると、文字化けしてしまいます。これは WSL が UTF-8 で日本語を扱うのに対し、Windows では Shift-JIS を期待しているために起こる現象です。そのような場合は `clip.exe` に渡す前に、`nkf`などをパイプで挟んで、文字コードの変換を行うとよいでしょう。なお `nkf` コマンドは `nkf` パッケージでインストールできます。

```
$ echo "ハロー" | nkf --sjis | clip.exe
```

12.3.5 クリップボードの内容を WSL で整形する

`clip.exe`はクリップボードへコピーするためのコマンドであり、ペーストはできません。クリップボードの中身を読み出すには、PowerShell の `Get-Clipboard` コマンドレットを使います。これでクリップボードの中身を、WSL の標準出力へ読み出せます。

```
$ echo "hello" | clip.exe
$ powershell.exe -command "Get-Clipboard" | tr '[:lower:]' '[:upper:]'
HELLO
```

標準出力に読み出せるのですから、あとはパイプを繋いで自由に編集が可能です。上記はまず `clip.exe` でクリップボードに `hello` という文字列をコピーした上で、PowerShell を使ってそれを取り出し、`tr` コマンドにパイプしてアルファベット小文字を大文字に変換しています。

Windows のアプリ（Web ブラウザなど）でコピーした文字列を、WSL 上で加工して利用したいような場合に便利です。ここからさらに `clip.exe` にパイプすれば、クリップボードの中身を変換するワンライナーも作れます。

第 13 章 コマンドを並列実行して効率をアップしよう

CPU やネットワークに余裕があるのであれば、複数のコマンドを並列に実行することで、処理速度を向上できます。この章では GNU Parallel を使ったコマンドの並列実行方法を紹介합니다。

13.1 コマンドの並列実行とは

WSL を使う大きな動機のひとつが、強力な Linux のコマンドを Windows 上でシームレスに動かせることです。CLI ツールを使いこなせば、複雑な処理の自動化や大量のデータの一括処理の効率を、飛躍的に向上させられます。ですが処理をシーケンシャルに行っていると、どうしてもある所で速度は頭打ちになってしまいます。そこで登場するのが並列化です。今回はジョブ（コマンドやスクリプト）を並列実行するためのツール、**GNU Parallel**^{*1}の使いこなしを紹介します。

13.2 GNU Parallel とは

Linux のシェルではループ構文を用いて、大量のデータ処理を自動化することができます。ですが愚直にループを回すだけだと、データはひとつずつ、シーケンシャルに処理されることになります。一度に 1 つの処理しか行わないのであれば、現代的なマルチコア CPU を活かせず、効率化も早々に頭打ちとなってしまいうしょう。この問題を解決するには、ジョブを適度な粒度で分割し、並列実行するしかありません。

冒頭で述べた通り GNU Parallel は、同じコマンドを複数の入力に対して並列実行するツールです。その真価は大量に繰り返すタイプの処理を、安全に高速化する場面で発揮されます。`for`や`xargs`コマンドを`parallel`に置き換えることで、CPU コアや I/O 帯域を効率よく使って処理を高速化できます。また`parallel`は、並列実行した各ジョブの出力が混ざらないよう制御されるため、逐次実行時と同様に他のコマンドへパイプで渡せます。

13.3 GNU Parallel のインストール

以下のコマンドで`parallel`パッケージをインストールしてください。

```
$ sudo apt install -U -y parallel
```

13.4 GNU Parallel の基本的な使い方

GNU Parallel の基本的な使い方を簡単に解説します。まずファイル名や数値、文字列のリストなど、複数の入力を列挙します。Parallel はこれらに対してコマンドを適用し、それを複数並列に実行します。一般的なプログラミング言語で言うイテレータが、いい感じに処理を並列化してくれるようなものだと考えるとよいでしょう。基本的な構文は以下の通りです。

```
parallel コマンド ::: 引数 1 引数 2 引数 3 (... 略...)
```

例として、以下のコマンドを実行してみましょう。

^{*1} <https://www.gnu.org/software/parallel/>

```
$ parallel echo ::: A B C
```

実行するコマンドは `echo` で、引数として `A`、`B`、`C` を取る 3 パターンを並列実行するという意味となります。当然、出力は以下の通りとなります。

```
A
B
C
```

これは以下のコマンドを実行したのと同じことですが、3 つの `echo` コマンドが順次実行されているわけではなく、内部的には並列に実行されています。

```
$ echo A
$ echo B
$ echo C
```

13.5 GNU Parallel の具体的な活用シーン

それでは現実の運用で、これは効くという利用シーンを、具体例とともに紹介しましょう。

13.5.1 大量ファイルの一括処理

まずは大量ファイルの一括変換です。 `magick`^{*2} コマンドを使い、JPEG 画像のリサイズを行ってみましょう。ここでは例として、8K の写真を 100 枚用意しました。すべての写真を、横幅 1920 ピクセルに縮小してみます。

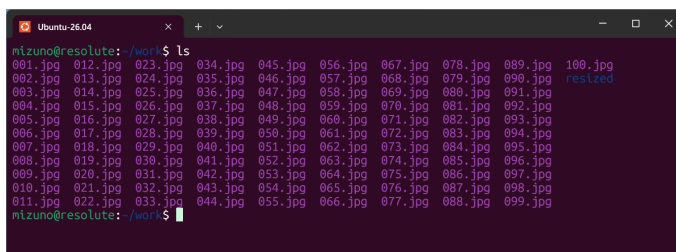


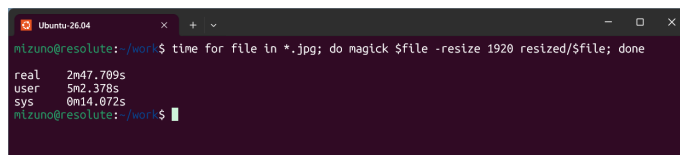
図 13.1: 001.jpg～100.jpg という 100 枚の 8K 写真、おおよそ 800MB 強のデータを用意した

普通に変換しようとする、以下のように `for` ループを使うことになるでしょう。

```
$ mkdir resized
$ for file in *.jpg
do
  magick $file -resize 1920 resized/$file
done
```

実際に実行し、かかった時間を計測してみました。筆者の環境では約 2 分 47 秒でした。

^{*2} 最近の ImageMagick では、`convert` ではなく `magick` コマンドを使います。



```

mizuno@resolute: /work$ time for file in *.jpg; do magick $file -resize 1920 resized/$file; done
real    2m47.709s
user    5m2.378s
sys     0m14.072s
mizuno@resolute: /work$

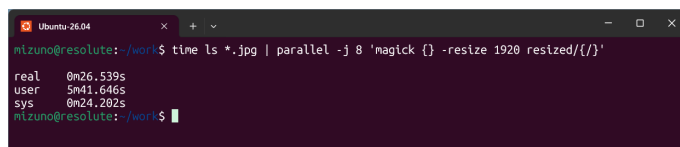
```

図 13.2: time コマンドでコマンド全体の実行時間を計測できる。real が実際にかかった時間

これを GNU Parallel を使って並列化すると、以下のように書き換えられます。ls コマンドでファイルのリストを作成し、パイプを使って parallel に渡しています。-j オプションは並列数です。8 コアの CPU を搭載した環境のため、8 を指定しています。{} は特殊な置換記号で、入力値そのもの（ここではパイプで渡されたファイル名）に置き換わります。また {/} はファイル名からディレクトリ名を取り除いたものに置き換わります。

```
$ ls *.jpg | parallel -j 8 'magick {} -resize 1920 resized/{/}'
```

実際に実行してみた結果が以下の通りです。かかった時間は約 26 秒で、単純なループに比べて 6 倍以上も高速化できました。



```

mizuno@resolute: /work$ time ls *.jpg | parallel -j 8 'magick {} -resize 1920 resized/{/}'
real    0m26.539s
user    5m41.646s
sys     0m24.202s
mizuno@resolute: /work$

```

図 13.3: CPU コア数に合わせて並列化したところ、大幅な高速化が実現できた

13.5.2 大量データ生成

テスト目的などで、大量のダミーデータを生成したいことがあります。例として、01~08 まで 8 個のディレクトリを作成し、それぞれの中に 1MB のランダムなダミーファイルを 1024 個作成してみましょう。つまり合計で 8192 個、8GiB のダミーデータを生成します。

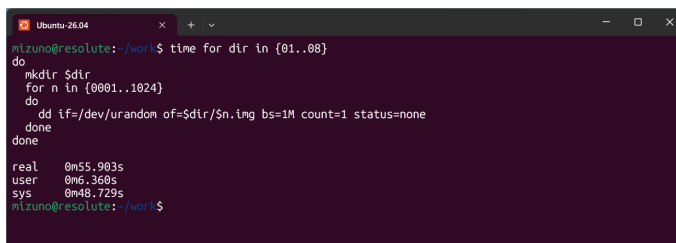
for ループで作ろうとすると、ディレクトリ 8 個とファイル 1024 個の、二重ループになるでしょう。

```

$ for dir in {01..08}
do
  mkdir $dir
  for n in {0001..1024}
  do
    dd if=/dev/urandom of=$dir/$n.img bs=1M count=1 status=none
  done
done

```

実際に実行してみました。結果としては約 55 秒かかりました。



```

mizuno@resolute: /work$ time for dir in {01..08}
do
  mkdir $dir
  for n in {0001..1024}
  do
    dd if=/dev/urandom of=$dir/$n.img bs=1M count=1 status=none
  done
done
real    0m55.983s
user    0m6.368s
sys     0m48.729s
mizuno@resolute: /work$

```

図 13.4: 二重ループで愚直にファイルをひとつずつ作った例

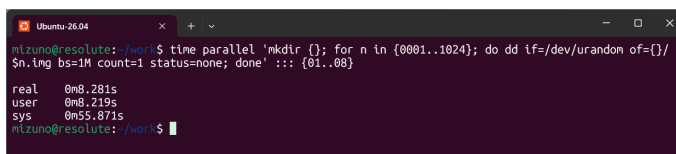
これも GNU Parallel で並列化してみましょう。ファイルを生成する 1024 回のループはそのままに、外側のディレクトリごとのループ単位で 8 並列化します。

```

$ parallel \
'mkdir {}'; \
for n in {0001..1024}; \
do dd if=/dev/urandom of={}/$n.img bs=1M count=1 status=none; \
done' \
 ::: {01..08}

```

結果は 8 秒で、これも 7 倍に高速化できました。このように適した粒度で並列化でき、かつリソースに余力があれば、ほぼリニアに性能は向上します。



```

mizuno@resolute: /work$ time parallel 'mkdir {}'; for n in {0001..1024}; do dd if=/dev/urandom of={}/$n.img bs=1M count=1 status=none; done' ::: {01..08}
real    0m8.281s
user    0m8.219s
sys     0m55.871s
mizuno@resolute: /work$

```

図 13.5: 8 並列で実行した結果。並列化したぶんだけ高速化できた

13.5.3 複数サーバーへのコマンド同時投入

サーバーの管理を行っているとき、複数のサーバーに対して同じ処理を行いたことがあります。こういう作業は並列化の出番です。まず対象サーバーの IP アドレスを、1 行ごとに列挙した「hosts」というファイルを用意しました。ここでは LAN 内にある 3 台のサーバーを対象とします。

```

192.168.1.213
192.168.1.238
192.168.1.182

```

for ループを使って、各サーバーにコマンドを投入する例が以下になります。ここでは各サーバーのマシン ID を出力し、1 秒スリープし、現在時刻を出力し、さらに 1 秒スリープするというコマンドを投入しています。

```

$ for host in $(cat hosts)
do
  ssh $host 'cat /etc/machine-id;sleep 1;date;sleep 1'
done

```

実行した結果は以下の通りです。

```

mizuno@resolute: $ time for host in $(cat hosts); do ssh $host 'cat /etc/machine-id;sleep 1;date;sleep 1'; done
6923a9d7f306490b8f5cf5edc11c623
木 7月 23 08:04:34 UTC 2026
5d92e47e774c455c94c8ae69c23f30a4
木 7月 23 08:04:37 UTC 2026
d58849d897f546f99774630928c22201
木 7月 23 08:04:39 UTC 2026

real    0m6.568s
user    0m0.011s
sys     0m0.033s
mizuno@resolute: $

```

図 13.6: 各処理の中で 2 秒スリープしているため、合計の実行時間は 3×2 で 6 秒ちょっととなる

これも以下のように並列化しました。3 台のサーバーに対して同時に SSH セッションを開き、コマンドを投入しています。

```
$ cat hosts | parallel ssh {} 'cat /etc/machine-id;sleep 1;date;sleep 1'
```

当然ですが並列実行されるため、実行時間は 2 秒に短縮されました。

```

mizuno@resolute: $ time cat hosts | parallel ssh {} 'cat /etc/machine-id;sleep 1;date;sleep 1'
6923a9d7f306490b8f5cf5edc11c623
2026年 7月 23日 木曜日 17:05:07 JST
5d92e47e774c455c94c8ae69c23f30a4
2026年 7月 23日 木曜日 17:05:07 JST
d58849d897f546f99774630928c22201
2026年 7月 23日 木曜日 17:05:07 JST

real    0m2.242s
user    0m0.099s
sys     0m0.112s
mizuno@resolute: $

```

図 13.7: 並列化すれば、理論上は 1 台の処理にかかる時間ですべてのサーバーを処理できる

物理的に異なるサーバーが複数並んでいるなら、こうした処理を 1 台ずつ順番に行うのは時間の無駄です。複数サーバーの管理は、まさに並列化が一番輝くシーンのひとつだと言えるでしょう。

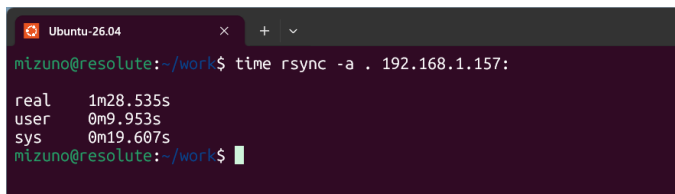
13.6 GNU Parallel が効かない例

リモートサーバーへ大量のファイルをコピーする際に、よく使われるコマンドが **rsync** です。ですが回線が十分に高速な場合、単発の **rsync** ではウィンドウサイズや輻輳制御の都合上、どこかで頭打ちとなってしまう、回線の帯域幅を十分に使い切れないケースがあります。また暗号化処理が CPU の 1 コアに寄ってしまうため、マルチコア CPU の恩恵も受けられません。**rsync** の特徴は差分転送ですが、その差分計算処理がシングルスレッド寄りなことも、パフォーマンスを十分に出せない理由となります。そこで送信するデータを適宜分割し、並列処理することで、パフォーマンスが向上するか確認してみましょう。

先ほど作成した 8GiB のダミーデータを、LAN 内の別のサーバーへコピーしてみましょう。ダミーデータの 01~08 ディレクトリがカレントディレクトリにある場合の **rsync** の実行例は、以下のようになります。

```
$ rsync -a . (送信先サーバーの IP アドレス):
```

約 1 分 28 秒かかりました。

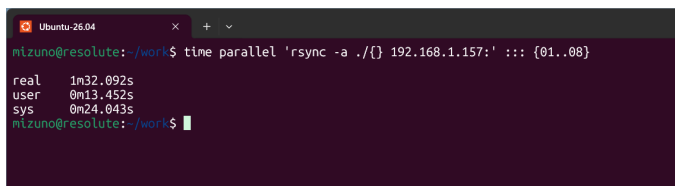


```
Ubuntu-26.04
mizuno@resolute: /work$ time rsync -a . 192.168.1.157:
real    1m28.535s
user    0m9.953s
sys     0m19.607s
mizuno@resolute: /work$
```

図 13.8: 一般的な rsync の例

続いてこちらも並列化します。先ほどと同様に、ディレクトリ単位で 8 並列化してみましょう。

```
$ parallel 'rsync -a ./{} (送信先の IP アドレス):' ::: {01..08}
```



```
Ubuntu-26.04
mizuno@resolute: /work$ time parallel 'rsync -a ./{} 192.168.1.157:' ::: {01..08}
real    1m32.092s
user    0m13.452s
sys     0m24.043s
mizuno@resolute: /work$
```

図 13.9: rsync を並列実行した例。今回の例ではネットワークがボトルネックになっているため、並列実行しても速度は上がらない

8 倍に高速化できるかと思いきや、なんと 1 分 32 秒で結果はほぼ同一、むしろ遅くなっています。これは一体どういうことでしょうか？

そもそも `parallel` で性能が向上するのは、シングルスレッドでは性能を使い切れない状況においても、並列化することでリソースを効率よく使えるからです。実はこの環境では、単発の `rsync` の時点で、ネットワークの帯域をほぼ 100% 使い切っている状態でした。そのためどれだけ並列化しても、ハードウェアの限界に突き当たっている以上、性能は向上しないというわけです。

このように、並列化すれば常に性能が向上するわけではありません。 `rsync` の場合は回線の帯域はもちろん、読み書き双方の SSD の速度などもボトルネックになり得ます。使い切れていないリソースがあるのか、並列化する意味があるのかどうかを確認することも重要です。

第 2 部

開発環境構築編

第 14 章 設定ファイルを管理しよう

快適な開発環境を構築する前に、個人用の設定ファイルを管理できるツールについて解説します。

14.1 dotfiles とは

Unix ライクなシステムでは、ソフトウェアの設定をプレーンテキスト形式のファイルに保存します。そのようなファイルを**設定ファイル**と呼びます。そしてプログラムのソースコードと同じく、こうした設定ファイルもバージョン管理するのがエンジニアの間では一般的です。そして設定ファイルや設定ファイルが保存されるディレクトリ名は、慣習的にドット (.) から始まるため、これらを管理するリポジトリのことを **dotfiles** と呼びます。

dotfiles とは、Bash の設定ファイルである `.bashrc` や、Emacs の設定ディレクトリである `.emacs.d` など、自分がよく使うソフトウェアの設定ファイルをバージョン管理するリポジトリです。設定ファイルは最初こそ細かくチューニングするものの、一度設定が固まってしまうと、その後設定を変更する機会も減っていきます。そのため後から見たときに、なぜこのような設定をしたのか、理解できなくなっていることもしばしばあります。そこでバージョン管理を行いエビデンスを残すことで、変更履歴を追跡できるようにしようというわけです。またこのリポジトリをリモートリポジトリと定期的に同期しておくことで、バックアップとしても役立ちます。

では、どのように設定ファイルを管理するのが理想的でしょうか。実は明確な答えはありません。GitHub などで開催されている様々な dotfiles を見てみると、各々が好きなように管理していることがわかります。ただしどのような dotfiles であっても、設定ファイルがリポジトリで管理され、それを何らかの方法でホームディレクトリにコピー、あるいは Symlink を作成して設定を反映しているところはおおむね共通しています。

この辺りを上手く実装するのがエンジニアの腕の見せ所でもあるのですが、単純に設定ファイルの管理さえできていればいいという方にとっては、単に面倒くさいだけでしょう。そこでそのような手間を避けるため、dotfiles を管理するためのツールも存在しています。

14.2 chezmoi とは

chezmoi^{*1}は、こうした dotfiles を安全に管理するためのソフトウェアです。読み方は**シェモア**、フランス語で「私の家」という意味があるそうです。

Go 言語で開発されており、シングルバイナリで配布されています。マルチプラットフォームのアプリであり、Linux、macOS、Windows などに対応しています。

chezmoi のメリットのひとつが、暗号化ツールと連携し、指定のファイルを暗号化できることです。たとえばクラウドサービスのクレデンシャルなど、機密情報を含むファイルであっても、暗号化することで安全に管理できます。対応している暗号化ツールは **GnuPG**^{*2}と **age**^{*3}です。

chezmoi の仕組みは簡単です。まず `chezmoi init` コマンドを実行することで、ローカルに Git リポジトリを作成します。 `chezmoi add` コマンドで特定のファイルを dotfiles の管理下に置き、 `chezmoi edit` コマンドで設定ファイルを編集します。このままでは変更が反映されないので、 `chezmoi apply` コマンドでホームディレクトリに設定を反映します。 `chezmoi git` コマンドでリポジトリに対して Git の操作を行えますので、変更をコミットしたりリモートにプッシュしたりできます。

^{*1} <https://www.chezmoi.io>

^{*2} <https://www.gnupg.org/>

^{*3} <https://github.com/filosottile/age>

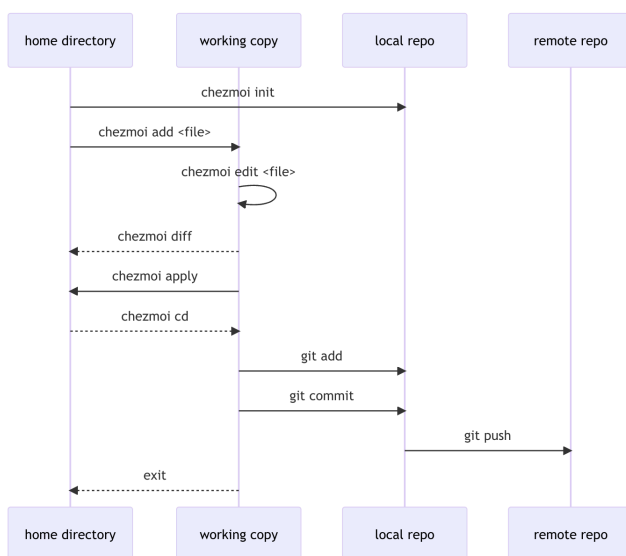


図 14.1: 公式ドキュメントにも記載されているフロー図

14.3 chezmoi の使い方

14.3.1 chezmoi のインストール

インストール方法については公式ドキュメント^{*4}に詳しく書かれています。こちらをあわせて参照してください。

Ubuntu では `snap` でインストールできます。

```
$ sudo snap install chezmoi --classic
```

バックエンドに Git を利用しますので、未インストールの場合は `apt` でインストールします。

```
$ sudo apt install -U -y git
```

14.3.2 リポジトリの初期化

dotfiles を管理するため、ローカルにリポジトリを作成します。下記のコマンドを実行してください。`~/.local/share/chezmoi` にリポジトリが作成されます。

```
$ chezmoi init
```

14.3.3 リポジトリへの移動

作成されたローカルリポジトリに移動するには、`chezmoi cd` コマンドを使用します。

^{*4} <https://www.chezmoi.io/install/>

```
$ chezmoi cd
```

`chezmoi cd` コマンドはリポジトリルートで新しいシェルを起動します。`exit` コマンドでシェルを終了すると、`chezmoi cd` を実行した時点のディレクトリに簡単に戻れます。

シェルを新たに起動するのではなく、単に移動したい場合は `chezmoi source-path` コマンドでリポジトリのパスを確認し、素の `cd` コマンドを使うのがよいでしょう。

```
$ chezmoi source-path
~/.local/share/chezmoi

$ cd ~/.local/share/chezmoi
```

14.3.4 設定ファイルの追加

リポジトリに設定ファイルを追加するには、`chezmoi add` コマンドを使用します。このコマンドは引数に与えられたファイルをローカルリポジトリにコピーします。ここでは Bash の設定ファイルである `.bashrc` を追加してみましょう。

```
$ chezmoi add ~/.bashrc
```

`chezmoi add` コマンドを実行すると `~/.local/share/chezmoi/dot_bashrc` というファイルが作成されます。このファイルは追加した `.bashrc` と全く同じ内容ですが、`dotfiles` リポジトリにコピーされた、別のファイルとして管理されています。ですので、片方のファイルを変更しただけでは、変更内容が同期されず、内容の分断が起きてしまいます。またデフォルトの状態では、単に `add` しただけでは Git の管理下に置けないことに注意してください。

14.3.5 設定ファイルの中身を表示する

設定ファイルの中身を表示するには、`chezmoi cat` コマンドを使用します。このコマンドを使用すると、リポジトリ内のファイルを直接 `cat` で表示できます。

```
$ chezmoi cat ~/.bashrc
```

なお `chezmoi cat` コマンドは、暗号化したファイルの中身も表示できます。

14.3.6 設定ファイルを変更する

設定ファイルに変更を加えるには、`chezmoi edit` コマンドを使用します。このコマンドは `dotfiles` のリポジトリ内にあるファイル、つまり `~/.local/share/chezmoi/dot_bashrc` を変更します。ホームディレクトリの `.bashrc` が変更されることはありません。コマンドを実行すると、自動的にデフォルトのテキストエディタが起動します。

```
$ chezmoi edit ~/.bashrc
```

14.3.7 設定ファイルの変更点を確認する

`chezmoi edit`で行った変更の確認には、`chezmoi diff`コマンドを使用します。ここでは例として、`.bashrc`内の環境変数`HISTSIZE`の値を変更しています。1000から2000に値が変更されていることが確認できました。

```
$ chezmoi diff
diff --git a/.bashrc b/.bashrc
index e8117c62d4a39ba4f0bdb36c546dfd1a69ad7826..b19223b0f4dfa4341c350c5c1aa25028faa94d0d 100644
--- a/.bashrc
+++ b/.bashrc
@@ -1,1 @@
-HISTSIZE=1000
+HISTSIZE=2000
```

なおコマンドの引数にファイルを指定すると、指定されたファイルの差分のみを表示できます。

```
$ chezmoi diff ~/.bashrc
```

14.3.8 変更を反映する

`chezmoi edit`で行った変更を実際に反映するには、`chezmoi apply`コマンドを使用します。

```
$ chezmoi apply
```

コマンドの引数にファイルを指定すると、指定されたファイルのみ設定を反映できます。

```
$ chezmoi apply ~/.bashrc
```

14.3.9 不要なファイルを chezmoi の管理から外す

様々な設定ファイルを管理していると、時には管理を止めたい設定ファイルも出てきます。そんな時は`chezmoi forget`コマンドを使用します。すると以下の様にプロンプトが表示されます。`y`もしくは`a`をタイプすることで、リポジトリから設定ファイルを削除できます。

```
$ chezmoi forget ~/.bashrc
Remove /home/mizuno/.local/share/chezmoi/dot_bashrc? yes/no/all/quit
```

ただしこれだけでは、ホームディレクトリ内のファイルは削除されません。`chezmoi`の管理から外すと同時に、ホームディレクトリ内のファイルも削除したい場合は、`chezmoi destroy`コマンドを使用しましょう。このコマンドを実行すると、以下の様にプロンプトが表示されます。ここでも`y`もしくは`a`をタイプすることで、リポジトリとホームディレクトリの両方からファイルを削除できます。

```
$ chezmoi destroy ~/.bashrc
Destroy /home/mizuno/.bashrc and /home/mizuno/.local/share/chezmoi/dot_bashrc? yes/no/all/quit
```

14.3.10 設定ファイルを Git の管理下に置く

デフォルト状態の chezmoi では、設定ファイルを追加、編集しても、Git による管理下には置かれません。これでは都合が悪いので、あわせて Git を使って管理する方法を解説します。

chezmoi には Git を操作するための、`chezmoi git` コマンドが用意されています。しかし chezmoi を通して Git を操作するのは少々クセ*5*6があるため、今回は `git` コマンドを直接操作します。

まずリポジトリのルートに移動します。

```
$ chezmoi cd
```

ワーキングディレクトリ内の設定ファイルを Git に追加します。この作業は `chezmoi add` の後に行ってください。

```
$ git add .
```

変更をコミットします。`-m` オプションはコミットメッセージを指定するためのオプションです。

```
$ git commit -m 'Add bashrc' dot_bashrc
```

必要であれば、リモートリポジトリと同期します。例えば GitHub にリモートリポジトリを用意したのであれば*7、以下のようにコマンドを実行してください。

リモートリポジトリをローカルリポジトリに追加します。

```
$ git remote add origin https://github.com/OWNER/REPOSITORY.git
```

リモートリポジトリの変更を取り込みます。

```
$ git pull origin main
```

リモートリポジトリに push します。

```
$ git push origin main
```

以上でリモートリポジトリとの同期は完了です。

14.4 システムワイドな設定ファイルを管理するには

ここまでは、各ユーザーのホームディレクトリにある設定ファイルの管理について解説してきました。ではシステムワイドな設定ファイルを管理するソフトウェアも chezmoi でいいのでしょうか？

システムワイドな設定ファイルは `/etc` 以下に格納されていますが、このディレクトリは一般ユーザーでは変更を書き込むことができません。もっと言えば、読み込みすら許可されていないファイルも存在

*5 `git` コマンドのオプションを chezmoi に解析させないため、「chezmoi git --」のようにセパレータを追加する必要があります。

*6 「chezmoi git commit ~/.bashrc」は正しく動作しません。リポジトリ内のファイルを指定して「chezmoi git commit dot_bashrc」とする必要があります。

*7 <https://thinkit.co.jp/article/37866>

します。chezmoi はユーザー固有の設定ファイルを管理するためのツールですので、こうした管理者権限が必要となるシステムワイドな設定を管理するには不向きと言えます。

そこでシステムワイドな設定ファイルを管理するのであれば **etckeeper** を推奨します。etckeeper は、`/etc` 以下の設定ファイルをすべて、バージョン管理下に置くための仕組みです。

14.4.1 etckeeper のインストール

`apt` で `etckeeper` パッケージをインストールしてください。これだけで自動的にローカルリポジトリが作成され、`/etc` のファイルがコミットされます。

```
$ sudo apt install -U -y etckeeper
```

14.4.2 etckeeper の自動コミット

`etckeeper` パッケージをインストールすると、`apt install` や `apt upgrade` など、システムを変更するコマンドを実行したタイミングで、変更点が自動的にコミットされるようになります。また未コミットの変更があると、毎日自動的にコミットを行うようになっています。したがってインストールしておくだけで、特別な操作を行わなくとも、自動的に `/etc` 以下の変更点が記録されていくことになります。

`etckeeper` をインストールするデメリットはなく、運用コストもゼロに近いので、とりあえずインストールしておくことをお勧めします。

14.4.3 手動で変更をコミットする

変更点を手動でコミットすることも可能です。自動コミットではコミットログが記録されないため、何らかの意図をもって設定を変更したのであれば、やはり手動でコミットを行うべきでしょう。

`etckeeper` はデフォルトで、バージョン管理に `Git` を使用します。ですが通常は `git` コマンドを直接ではなく、`etckeeper` コマンドを介して操作します。例えば変更をコミットするには、以下のコマンドを実行します。デフォルトのテキストエディタが起動しますので、コミットログを入力してください。

```
$ sudo etckeeper commit
```

この際、コマンドラインから直接コミットログを入力することもできます。`-m` オプションは不要です。

```
$ sudo etckeeper commit '●●の機能を有効にするため、××の設定を追加'
```

`etckeeper vcs` を使うと、`git` のコマンドを直接実行できます。`etckeeper` コマンドには、`git diff` や `git log` に相当するコマンドが存在しないため、差分やログを見たい場合は以下のように、`git` のコマンドを実行してください。

```
$ sudo etckeeper vcs log
$ sudo etckeeper vcs diff
```


第 15 章 バージョンマネージャーを使いこなそう

プロジェクトごとに、要求されるツールのバージョンが異なることはよくあります。この章では同一ツールの複数バージョンを使い分けられる、バージョンマネージャーについて解説します。

15.1 なぜバージョンマネージャーが必要なのか

近年、様々なツールやプログラミング言語を駆使してサービス開発を行うことは一般的となりました。実行環境の構築には **IaC** ツールを使いますし、フロントエンドとバックエンドの開発には、それぞれ別のプログラミング言語を使うことでしょう。これは、それぞれの言語やツールの特性を理解し、使い分けることで効率化を図るのが目的です。ただし使用するツールや言語が増えてくると、今まで動いていたはずのコードがある時点から動かなくなったり、誰かの環境でしか再現しない問題などが発生することも多々あります。これには色々な原因が考えられますが、その 1 つにバージョンの差異が挙げられます。

いくつかのプロジェクトを掛け持ちして、開発に参加することもあるでしょう。あるプロジェクトでは古いバージョンに依存している一方で、別のプロジェクトでは最新のバージョンを使いたいと言ったケースも珍しくありません。このような状況で、うかつにツールやプログラミング言語をシステムワイドにインストールしてしまうと、別のプロジェクトではバージョンが合わず、開発に支障が出てしまいます。

そこで便利なのが、プロジェクトごとに適切なツールのバージョンを管理できる**バージョンマネージャー**です。バージョンマネージャーにも複数の実装が存在しますが、有名なのが **asdf**^{*1}でしょう。

ちなみに **asdf** は、Linux と macOS 向けにしかリリースされていないため、Windows 上で直接動かすことはできません。ですがこうした Linux 向けのツールであっても、手軽に Windows 上に導入できるのが WSL のメリットだと言えます。

15.2 そもそもバージョンマネージャーとは

バージョンマネージャーとは、プログラミング言語やツールの複数のバージョンをシステム内で共存させ、必要に応じて切り替えを可能にするツールです。バージョンマネージャーを使用することで、開発者はシステム全体の設定を変更することなく、プロジェクト毎にツールの最適なバージョンを選択できます。複数のプロジェクトで開発に参加している場合には、必須のツールと言えるでしょう。

バージョンマネージャーには様々な実装が存在します。たとえば、Python のバージョンを管理するツールとしては **pyenv**^{*2}が有名です。他にも Go は **goenv**^{*3}、Node.js は **nodenv**^{*4}など、`~env` という名前で開催されていることが多くあります。もちろん、このルールから外れているツールもいくつか存在していますが、どれもソフトウェアとバージョンを管理できるという点では同じです。

これらのバージョンマネージャーは、プログラミング言語やツール毎に実装され、それぞれ使い方も異なります。つまり使用するツールやプログラミング言語が増えていくと、それに応じてバージョンマネージャーの種類も増え、それぞれの使い方を覚えていく必要がでてくるわけです。これでは効率化を図るために使い始めたツールが、効率化の足枷となってしまうと言っても過言ではありません。そこで登場するのが、様々なツールや言語をまとめて管理できる **asdf** というわけです。

*1 <https://asdf-vm.com/ja-jp/>

*2 <https://github.com/pyenv/pyenv>

*3 <https://github.com/go-nv/goenv>

*4 <https://github.com/nodenv/nodenv>

15.3 asdf とは

asdf とは、複数のプログラミング言語やツールのバージョンを統合的に管理できる、バージョンマネージャーです。繰り返しになりますが、従来のバージョンマネージャーはツールや言語毎に実装されていたため、複数のバージョンマネージャーを使う必要がある上、その使い方もツール毎にバラバラでした。しかし asdf は 1 つのツールで複数の言語、ツールのバージョンを統合管理できるため、asdf さえインストールしておけばよいケースも多く、ツールに起因する煩雑さを解消できます。

asdf で管理するツールのバージョンは、テキストベースの設定ファイル `.tool-versions` でコントロールできます。そのためこのファイルをプロジェクトのリポジトリにコミットしておけば、チーム全員がツールや言語のバージョンを、簡単に揃えられるのです。

15.4 asdf を使ってみよう

asdf には、非常に充実した日本語の公式ドキュメントが存在します。もし手順で迷った場合は、ドキュメント^{*5}も合わせて参照してください。

15.4.1 依存パッケージのインストール

asdf は Git に依存しています。下記のコマンドを実行し、`apt` で `git` パッケージをインストールしてください。

```
$ sudo apt install -y -U git
```

また必須ではないものの、`build-essential` パッケージもインストールしておくといいかもしれません。というのも、asdf はツールのインストール時に、そのツールの依存ライブラリやツール自体のビルドを行うことがあるからです。必要に応じてインストールしてください^{*6}。

```
$ sudo apt install -y -U build-essential
```

15.4.2 asdf のインストールと設定

asdf は `apt` や `snap` でインストールができません。そこでリリースされているバイナリを手動でインストールします。以下のコマンドを実行してください。なおこれは、2026 年 7 月現在の最新リリースである、バージョン 0.20.0 の手順です。もしより新しいバージョンがリリースされている場合は、適宜ファイル名を変更してください。

```
$ wget https://github.com/asdf-vm/asdf/releases/download/v0.20.0/asdf-v0.20.0-linux-amd64.tar.gz
$ tar xf asdf-v0.20.0-linux-amd64.tar.gz
$ sudo install -m 0755 asdf /usr/local/bin/asdf
```

asdf が実行できるか確認してみましょう。

^{*5} <https://asdf-vm.com/ja-jp/guide/getting-started.html>

^{*6} ただし `build-essential` パッケージをインストールすると、`Coreutils` を GNU 版に切り替えられなくなるという Ubuntu の仕様については留意してください。

```
$ asdf --version
asdf version v0.20.0 (revision 150aaf0)
```

ダウンロードしたバージョンが表示されていればインストールは完了です。続いて `asdf` のコマンドの補完を有効にしましょう。

```
$ echo '. <(asdf completion bash)' >> ~/.bashrc
$ source ~/.bashrc
```

これで `asdf` と入力した後に `Tab` キーを押すと、`asdf` のサブコマンドなどが補完できるようになります。

`asdf` でインストールしたツールは、`~/.asdf/shims` 以下に展開されます。しかしこのディレクトリにはパスが通っていないはずなので、環境変数 `PATH` に追加する必要があります。下記のコマンドを実行しましょう。

```
$ echo 'export PATH="$HOME/.asdf/shims:$PATH"' >> ~/.bashrc
$ source ~/.bashrc
```

これでインストールは完了です。

15.4.3 ツールバージョン管理の手順を理解する

`asdf` では、特定のプログラミング言語やツールをどうインストールし、どう管理するかを **プラグイン** というモジュールで管理しています。そのためまずはじめに、使いたいツールに対応したプラグインの追加が必要です。プラグインを追加してはじめて、そのツールのバージョン一覧の取得やインストール、バージョンの切り替えなどが行えるようになります。全体の流れとしては下記の手順になります。

1. プラグインの検索
2. プラグインの追加
3. 利用可能なバージョンを調べる
4. 特定のバージョンを指定してインストール
5. インストールしたバージョンを有効化

このうち、プラグインの検索や利用可能なバージョンを調べる作業は、必ずしも必要ではありません。とはいえこれらは非常にありがちな作業のため、この機会に覚えておきましょう。

15.4.4 ツールの特定バージョンをインストールする

`asdf` が動くようになったはずですので、続いてツールのインストールを行います。今回は例として **Node.js** の最新版をインストールしてみます。まずは `asdf` に `Node.js` のプラグインがあるか確認してみましょう。

```
$ asdf plugin list all | grep nodejs
nodejs      https://github.com/asdf-vm/asdf-nodejs.git
```

見つかりました。以下のコマンドでこのプラグインを追加します。

```
$ asdf plugin add nodejs
```

プラグインが追加できたか見てみましょう。

```
$ asdf list
nodejs
  No versions installed
```

プラグインは追加できましたが、Node.js 自体のインストールはまだ行なっていないため、「No versions installed」と表示されます。では利用可能な Node.js のバージョンを確認してみましょう。

```
$ asdf list all nodejs
(... 略 ...)
26.3.0
26.3.1
26.4.0
26.5.0
```

出力された中から、使用したいバージョンを選択してインストールします。最新のバージョンが知りたい場合は下記のコマンドを実行してください。

```
$ asdf latest nodejs
26.5.0
```

執筆時点では26.5.0が最新のようなので、このバージョンをインストールしましょう。

```
$ asdf install nodejs 26.5.0
```

ここではバージョン26.5.0を直接指定しましたが、とにかく最新でいいという場合は、かわりにlatestと指定することで、最新のバージョンをインストールできます。

この latest タグは優秀で、latest:versionと指定すると、versionで指定したメジャーバージョン系列の最新リリースをインストールできます。具体的にはlatest:26とすると、Node.js のメジャーバージョンが26で、マイナーバージョン、パッチバージョンは現時点での最新がインストールされます。latest:25とすれば 25 系の最新がインストールされるというわけです。

余談ですが、プログラミング言語やツールのバージョンは、XX.YY.ZZという、ドットで区切った3つの数字で表されることがよくあります。これはセマンティックバージョンングと呼ばれ、XXがメジャーバージョン、YYがマイナーバージョン、ZZがパッチバージョンです。メジャーバージョンは互換性のない変更を加えた時にカウントアップされます。マイナーバージョンは後方互換性が保たれた機能追加、パッチバージョンは後方互換性を保ったバグ修正などでカウントアップされます。つまりYY.ZZの部分は、バージョンが上がったとしても互換性には影響しないことになります。そのためメジャーバージョンのみを固定し、マイナーバージョン、パッチバージョンは常に最新にするという運用もよく行なわれます。latest:versionは、こうした運用に向けた指定というわけです。

少し話が逸れましたが、指定のバージョンが正しくインストールされたか見てみましょう。

```
$ asdf list nodejs
26.5.0
```

インストールしたバージョンが表示されれば成功です。

15.4.5 プロジェクト毎にバージョンを使い分ける

ここまでの説明でツールのインストール方法は理解できたかと思います。続いてはプロジェクト毎にツールのバージョンを使い分けます。これが `asdf` の本命の機能と言えます。今回は架空のプロジェクトである、`project_a` と `project_b` ディレクトリを用意し動作を説明します。まずはディレクトリを作成しましょう。

```
$ mkdir ~/project_{a,b}
```

`project_a` ディレクトリに移動し、バージョンをセットします。

```
$ cd ~/project_a
$ asdf set nodejs 26.5.0
```

このコマンドを実行すると、`project_a` ディレクトリに `.tool-versions` という名前のファイルが作成されます。このファイルには使用するツールとそのバージョンが記述されています。このファイルを Git を経由して共有することで、他の環境でも同じ環境を再現できるというわけです。

```
$ node --version
v26.5.0
```

続いて `project_b` に移動して、別のバージョンをインストールしてみましょう。

```
$ cd ~/project_b
$ asdf install nodejs latest:25
$ asdf set nodejs latest:25
$ node --version
v25.9.0
```

バージョン 25 系の最新が表示されれば成功です。現在どのバージョンが使用されているかを見てみましょう。

```
$ asdf list nodejs
*25.9.0
26.5.0
```

先頭に `*` がついているバージョンが、現在のプロジェクトで使用中のバージョンです。このシステムには `25.9.0` と `26.5.0` がインストールされており、このプロジェクトでは `25.9.0` が使われていることがわかります。`project_a` ディレクトリにも移動して確認してみてください。

15.4.6 チームメンバーが設定したバージョンを使う

ここまでは自分で特定のバージョンをインストールし、使用できるようインストールしました。自分がこのプロジェクト内で、バージョン選定のその権限を持っているのならこれで構いません。ですが自分以外にテックリードが存在し、バージョンの決定権がその人にある場合もあるでしょう。ここでは他の人がセットアップした `.tool-versions` を使って、自分の環境をセットアップしてみます。

`project_c` ディレクトリを作成し、他の人が用意したという体の `.tool-versions` を用意します。

```
$ mkdir ~/project_c
$ cd ~/project_c
$ echo 'golang 1.24.2' > .tool-versions
```

このプロジェクトは Go 言語を使用するため、`golang` プラグインを追加しましょう。

```
$ asdf plugin add golang
```

以下のコマンドを実行すると、`.tool-versions`の内容に従って、指定のバージョンがインストールされます。

```
$ asdf install
```

`asdf install` コマンドに、インストールするツールもバージョンも指定していません。ですがカレントディレクトリに `.tool-versions` があった場合、`asdf` はこのファイルを自動的に読み、そこに記載されたバージョンを自動でインストールするのです。ただしプラグインの追加だけは、事前に自分で行う必要があることに注意してください。

```
$ go version
go version go1.24.2 linux/amd64
```

指定された Go のバージョンがインストールされていれば成功です。このようにバージョンマネージャーを使えば、プロジェクトで指定されている環境を簡単に再現できるというわけです。

15.5 その他のバージョンマネージャー

15.5.1 mise

`asdf` は様々なプラグインでバージョンを管理でき、非常に便利です。ただし 1 つ、懸念点もあります。それはプラグインの管理方法です。`asdf` にはプラグインを管理する 2 つの仕組みが存在します。1 つは `asdf-community`^{*7} という GitHub オーガナイズーションにリポジトリを作成する方法、もう 1 つは `asdf-plugins`^{*8} リポジトリに PR を作成し、マージしてもらう方法です。

`asdf-community` は `asdf` のコミュニティで、プラグインをメンテナンスするため比較的安全と言えます。`asdf` もこちらの方法を推奨しています。

対して `asdf-plugins` は一定のルールに則って PR を出せば、明らかにバグっていない限りは誰でもプラグインを追加できます。そしてそのプラグインのセキュリティについては、プラグインの作成者に委ねられます。また、使用者側も自己責任で使用するようになっています^{*9}。

この問題を解決しようとしているツールとして、`mise`^{*10} というものがあります。`mise` は独自のプラグインリポジトリを設けており、オーナーだけがプラグインのコミットを許可されています。そのため悪意のあるコードや、1 度しか使用しないであろうプラグインが追加されることを阻止できます。

`mise` は `asdf` の `.tool-versions` ファイルを読み込めるため、`asdf` とツール互換性があります。基本的な使い方も `asdf` と同様です。つまりツール移行にかかるコストをほぼ無視できるというわけです。

^{*7} <https://github.com/orgs/asdf-community/repositories>

^{*8} <https://github.com/asdf-vm/asdf-plugins>

^{*9} <https://github.com/asdf-vm/asdf-plugins?tab=readme-ov-file#security>

^{*10} <https://mise.jdx.dev/>

15.5.2 aqua

aqua^{*11}は、日本人によって開発されているバージョンマネージャーです。aqua の開発者は **tfaction** と呼ばれる GitHub Actions の開発者でもあります。開発者による日本語のドキュメントが充実しているため、導入も簡単でしょう。

ただし aqua は asdf と互換がないため、aqua を使用するなら aqua 用のセットアップが必要になります。

^{*11} <https://aquaproj.github.io/>

第 16 章 プロジェクト単位で環境変数を管理しよう

環境変数は API キーやデータベースの接続先、実行モードなど、アプリケーションの動作を環境ごとに切り替えるための重要な設定です。この章ではプロジェクトに固有の環境変数を上手に管理する方法について解説します。

16.1 環境変数の重要性

近年アプリケーションの実行環境として最も選択されているのはクラウドです。運用の簡素化やコスト効率など様々なメリットがありますが、クラウドを選択する理由の 1 つとしてスケーラビリティがあります。オンデマンドに環境やリソースを追加できるのは、クラウド利用の最大のメリットと言えるでしょう。

クラウド利用を念頭においた設計思想を「クラウドネイティブ」と呼びます。クラウドネイティブなアプリケーションには以下のような特徴があります。

- Twelve-Factor App^{*1}に準拠
- コンテナ化とオーケストレーション
- マイクロサービスアーキテクチャ
- クラウドの特徴を活かした設計

この中の 1 つ、**Twelve-Factor App** には「設定を環境変数に格納する^{*2}」と明示されています。アプリケーションの設定は、それぞれの環境（開発、ステージング、本番）で異なります。たとえば DB への接続情報やホスト名などです。とはいえこれらの機密情報を、アプリケーションと同じリポジトリで管理するわけにはいきません。そのためこうした設定値は、環境変数を經由してアプリに渡せる方がよりセキュアなわけです。

このようなアプリケーションを開発するには、開発環境でも環境変数を管理する必要があります。しかし複数のプロジェクトを同時に扱う場合や、複数の環境で作業する場合、煩雑になる環境変数の切り替えが問題となります。このような課題を解決するツールとして、**direnv**^{*3}があります。

16.2 direnv とは

現代のアプリケーション開発では、環境変数を使用して設定を管理することが一般的です。しかし、以下のような課題があります。

- プロジェクトごとに異なる環境変数が必要
- 環境変数の設定を忘れてしまう
- 環境変数の管理が煩雑
- チームメンバー間での環境変数の共有が難しい

この問題を解決するのが **direnv** です。**direnv** は、ディレクトリごとに環境変数を自動的に設定、切り替えるためのユーティリティです。これにより、プロジェクトや環境ごとの環境設定を簡単に管理できるようになります。例えば、以下のようなシーンで **direnv** は特に有用です。

- 複数の Kubernetes クラスターを管理する場合
- 異なる環境の認証情報を扱う場合

^{*1} <https://12factor.net/ja/>

^{*2} <https://12factor.net/ja/config>

^{*3} <https://direnv.net/>

- プロジェクト固有の環境変数を設定する場合

この章では `direnv` を活用しながら、よりスムーズな開発体験を実現する方法を紹介します。

16.3 `direnv` の特徴

`direnv` の特徴は、まず自動的な環境変数の設定です。そのディレクトリに入ったときに自動的に環境変数を設定し、ディレクトリから出たときに自動的に環境変数を解除します。

次にシンプルな設定ファイルです。`.envrc` というファイルで環境変数を管理します。このファイルは非常にシンプルな構文で設定可能です。

3 番目はセキュリティです。設定ファイルの変更を検知したり、信頼できる設定ファイルのみを実行できます。

最後はクロスプラットフォーム対応です。Linux, macOS, Windows という主要な OS とシェルに対応しています。

16.4 `direnv` のインストール

`direnv` は `apt` でインストールできます。`direnv` パッケージをインストールしてください。

```
$ sudo apt install -U -y direnv
```

続いてフックを設定します。以下のコマンドを実行してください。

```
$ echo 'eval "$(direnv hook bash)"' >> ~/.bashrc  
$ source ~/.bashrc
```

16.5 基本的な使い方

今回は仮に `project_a` と `project_b` ディレクトリで動作確認を行います。なお前章でこのディレクトリを既に作成済みの場合は、あらかじめ削除しておいてください。

```
$ mkdir ~/project_{a,b}
```

`project_a` ディレクトリ内に、`.envrc` ファイルを作成します。ここでは `PROJECT` という環境変数に `A` という値を設定しています。

```
$ echo 'export PROJECT=A' > ~/project_a/.envrc
```

作成した設定ファイルを信頼します。以下のコマンドを実行してください。`direnv allow` に続き、信頼する設定ファイルが置かれているディレクトリを引数に指定します。

```
$ direnv allow ~/project_a
```

動作を確認してみましょう。カレントディレクトリを `project_a` ディレクトリに変更すると、自動的に `.envrc` が読み込まれ、`PROJECT` 環境変数に `A` がセットされます。

```
$ cd ~/project_a
direnv: loading ~/project_a/.envrc
direnv: export +PROJECT

$ echo $PROJECT
A
```

`project_a`ディレクトリから出ると、セットした環境変数が解放されます。

```
$ cd ..
direnv: unloading

$ echo $PROJECT
← 変数の中身が空になっている
```

続いて`project_b`ディレクトリにも、`.envrc`ファイルを作成しましょう。こちらは`PROJECT`環境変数に`B`という値を設定しています。

```
$ echo 'export PROJECT=B' > ~/project_b/.envrc
```

同じく設定ファイルを信頼します。

```
$ direnv allow ~/project_b
```

今度は`project_b`ディレクトリに移動してみましょう。同様にディレクトリ内の`.envrc`が読み込まれ、`PROJECT`環境変数に`B`がセットされます。

```
$ cd ~/project_b
direnv: loading ~/project_b/.envrc
direnv: export +PROJECT

$ echo $PROJECT
B
```

`project_b`ディレクトリから出ると、同様に環境変数が解放されます。

```
$ cd ..
direnv: unloading

$ echo $PROJECT
← 変数の中身が空になっている
```

16.6 よくある使用例

開発と本番という、複数の環境を対象に作業していることを想定してみましょう。これ以降の作業は`project_c`以下で行いますので、ディレクトリを作成し移動しておきます。

```
$ mkdir ~/project_c
$ cd ~/project_c
```

アプリ開発のため、

- データベースの接続情報
- デバッグモードの有無
- API エンドポイントの設定

という 3 つの環境変数が必要だとしましょう。project_c ディレクトリ内に、以下の内容で .envrc を作成してください。

```
export DATABASE_URL="postgres://localhost:5432/dev_db"
export DEBUG=true
export API_ENDPOINT="http://localhost:3000"
```

.envrc を編集すると direnv が変更を検知し、以下のメッセージが表示されます。

```
direnv: error /home/mizuno/project_c/.envrc is blocked. Run 'direnv allow' to approve its content
```

これは project_c ディレクトリの .envrc はまだ信頼されていないため、読み込みをブロックしたという意味です。以下のコマンドで信頼しましょう。

```
$ direnv allow ~/project_c
```

これで、開発環境における環境変数が自動的にセットされるようになりました。ここまでは既に解説した通りです。

この設定だけだと、開発環境向けの環境変数しか設定できていません。本番環境と簡単に切り替えられると便利です。そこで追加で .env というファイルを、以下の内容で作成してください。このファイルは dotenv というライブラリがデフォルトで読み込む設定ファイルで、ここに書かれた値は環境変数として利用されます。ここでは開発環境 (DEV) と本番環境 (PROD) それぞれの、API キーを環境変数にセットしています。また ENV 環境変数に prod を指定しています。

```
DEV_API_KEY=0123456789
PROD_API_KEY=9876543210
ENV=prod
```

.env はあくまで環境変数のセットを管理するだけです。環境そのものの切り替えはできません。そこで direnv を挟むことで、環境の切り替えを実現してみましょう。先ほど作成した .envrc を、以下の内容で書き換えてください。

```
# .envrc と同じ階層に .env ファイルがあれば読み込む
if [ -f "$PWD/.env" ]; then
  source "$PWD/.env"
fi

# ENV に prod が設定されていれば本番用の値を環境変数にセット
if [ "$ENV" = "prod" ]; then
  export DATABASE_URL="postgres://prod-db:5432/prod_db"
  export DEBUG=false
  export API_ENDPOINT="http://prod-api:3000"
  # .env から読み込んだ本番用の API_KEY をセット
  export API_KEY=$PROD_API_KEY
fi

# ENV が prod 以外の場合は開発用の値を環境変数にセット
```

```
else
  export DATABASE_URL="postgres://localhost:5432/dev_db"
  export DEBUG=true
  export API_ENDPOINT="http://localhost:3000"
  # .env から読み込んだ開発用の API_KEY をセット
  export API_KEY=$DEV_API_KEY
fi
```

ここまでの説明でお気づきの方もいるかもしれませんが、`.envrc`の内容は、実はただのシェルスクリプトです。ディレクトリに入るとスクリプトが実行され、`export`した環境変数がこのディレクトリ下で有効になります。言いかえると`direnv`を使用すれば、ディレクトリへの移動をトリガーとして、任意の処理を実行できるというわけです。

この状態で`project_c`ディレクトリへ移動すると、`direnv`がシェルスクリプトを実行し、`.env`に記述された`ENV`環境変数の値に応じて、`API_KEY`や`API_ENDPOINT`を切り替えてくれます。

16.7 想定される運用

プロジェクト内で共通の環境変数を使用する場合は、`.envrc`をプロジェクトのリポジトリに含めるとよいでしょう。ただしリポジトリで管理するのなら、機密情報を書くべきではありません。その場合は`.env`に機密情報を記述し、それを別途読み込むようにするといいいでしょう。当然ですが、`.env`はリポジトリに含めてはいけません。

第 17 章 Windows でも Docker を動かそう

今時のアプリケーション開発において、避けては通れないのが Docker です。ですが Docker は Linux でなければ動きません。この章では WSL を使って Windows 上で Docker を動かす方法を解説します。

17.1 Docker とは

今時のアプリケーション開発において、避けては通れないトレンドとなっているのが **コンテナ** です。そしてその中でも、世界で最も利用されているコンテナ実行環境が **Docker** ではないでしょうか。Docker は、自身を「アプリケーションを開発、転送、実行するためのプラットフォーム」と位置づけています。Docker はアプリケーションの実行環境一式をカプセル化し、ポータビリティを持たせる技術です。これによって環境に左右されず、同じ条件でアプリケーションを動かすことができ、効率のよい開発やデリバリーを実現できるというわけです。

Docker は、ユニバーサルパッケージである Snap と、非常によく似た設計思想を持っています。コンテナといえば軽量な仮想環境というイメージを持っている方も多いかもしれませんが、Docker はむしろ、アプリケーションのデリバリーを目的としたパッケージシステムに近いシステムだと言えるでしょう。

Docker は Linux カーネルの機能を利用しているため、Linux でなければ動かすことはできません。そのため Windows 上で Docker を使うには、仮想マシンを併用するのが定石でした。ですが現在では、より高速に動作する WSL が存在しますので、これを利用しない手はないでしょう。この章では WSL を使って、Windows 上で Docker を動かす方法を紹介します。

17.2 Docker Desktop とは

Windows や macOS で Docker を動かそうと思った際、最初に候補に上がるのが **Docker Desktop**^{*1}でしょう。Docker Desktop は、**Docker Engine**、クライアントとなる **docker** コマンド、GUI フロントエンド等々の Docker 実行環境を、オールインワンで提供するプロダクトです。その特徴としては、内部的に仮想マシンを起動して、Docker を動かしているという点が挙げられます。Docker Engine を仮想マシンで提供することより、Windows/macOS/Linux が混在する環境であっても、同じバージョンの Docker Desktop をインストールすれば、開発環境を同一に揃えることができるのです。これは大規模なチームによる開発プロジェクトにおいて、非常に大きなアドバンテージとなります。

従来の Windows 版の Docker Desktop は、バックエンドとして Hyper-V の仮想マシンを利用していました。ですが WSL がインストールされている環境であれば、より軽量で高速な WSL をかわりに利用できます。

17.3 Docker Desktop のインストールと起動

それでは実際に Docker Desktop をインストールしてみましょう。Docker Desktop のインストールドキュメント^{*2}を参考に、Windows 用のインストーラーをダウンロードして実行してください。

インストールが完了したら、Docker Desktop を起動しましょう。初回起動時は以下のダイアログが表示され、Docker Subscription Service Agreement への同意の確認が行われます。内容を確認した後、問題がなければ「Accept」をクリックします。Docker Desktop は、一定以上の規模の企業では、ライセンス契約なしに使用することができません。業務で利用する場合はくれぐれも気をつけてください。

^{*1} <https://docs.docker.com/desktop/>

^{*2} <https://docs.docker.com/desktop/setup/install/windows-install/>

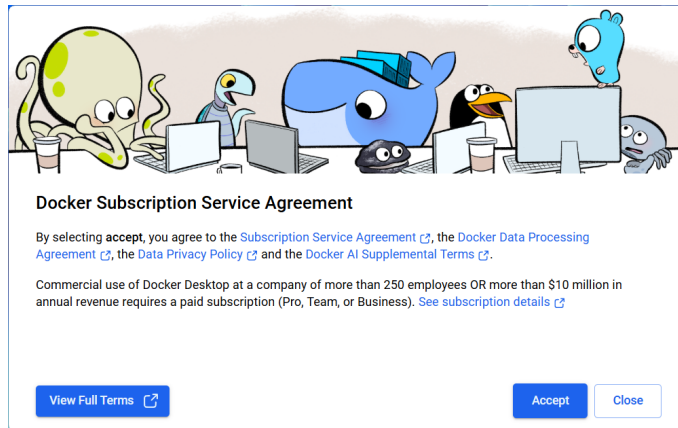


図 17.1: Docker Subscription Service Agreement への同意

Docker Desktop を起動すると、WSL 上で `docker-desktop` という名前のディストリビューションが起動します。Docker Engine 自体はここで動いています。PowerShell を起動し、以下のコマンドを実行すると、その存在を確認できます。

```
$ wsl --list -v
```



図 17.2: docker-desktop 用のディストリビューションがインストールされていることがわかる

ここまで解説してきた Ubuntu 26.04 とは異なり、`docker-desktop` ディストリビューション内に、ユーザーが直接ログインして操作することはありません。あくまで Docker Desktop のバックエンドとして、Docker Engine を動かすためのものとなります。通常の利用の範疇においては、その存在を意識する必要もないでしょう。詳しくは後述しますが、CLI で実行する `docker` コマンドを、Windows 上の PowerShell から直接実行することも可能です。

17.4 Docker コンテナを動かす

これだけで、WSL をバックエンドとして Docker コンテナを動かせます。試しに nginx Web サーバーをコンテナで動かしてみましょう。

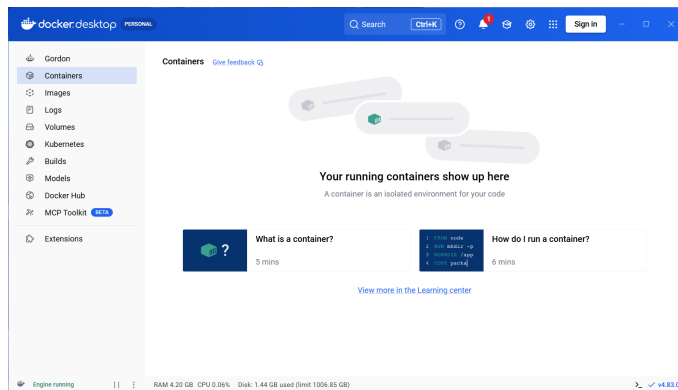


図 17.3: 起動した Docker Desktop の画面

Docker Desktop を起動したら、ウィンドウ上部にある検索ボックスに「nginx」と入力してください。複数のイメージがヒットしますが、（おそらく）一番上に表示されているであろう、公式のイメージを選択します。Run をクリックしてください。

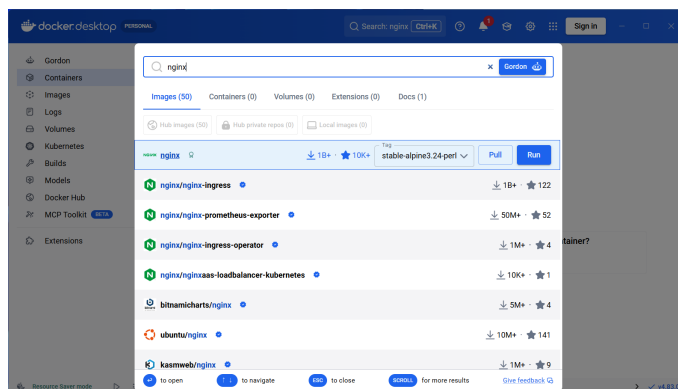


図 17.4: nginx のイメージを取得して実行する

コンテナを起動しようとする、以下のウィンドウが表示されます。nginx のコンテナは、ホストが待ち受けるポート番号や、コンテナ内にマウントするボリュームを起動時に指定できます。これはその設定を行うためのウィンドウです。Optional settings をクリックして展開したら、必要な値を入力してください。ここではコンテナに nginx という名前をつけ、ホストのポートに 8080 を指定しました。設定値を入力したら Run をクリックしてください。

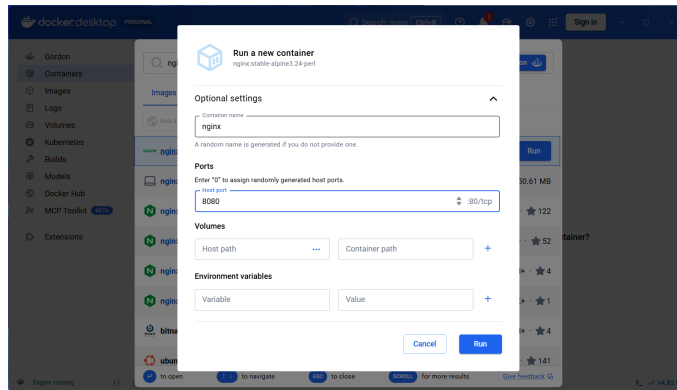


図 17.5: 起動前にコンテナの設定を行う

これでコンテナが起動します。Windows で Web ブラウザーを起動して、`http://localhost:8080`^{*3} にアクセスしてみましょう。nginx のサンプルページが表示されたら成功です。

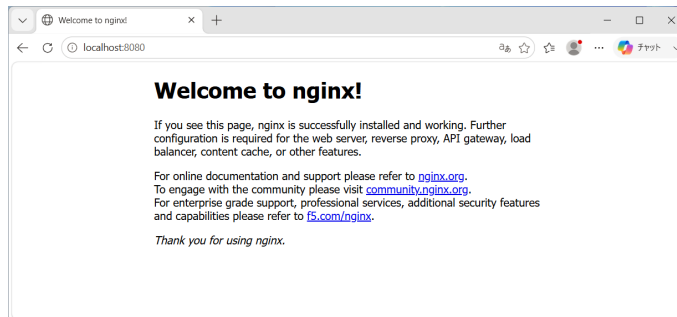


図 17.6: Windows から、コンテナ内で起動している nginx にアクセスできた

基本はたったこれだけです。WSL があるおかげで、Windows 上でも簡単に Docker を動かせるのが理解できたのではないのでしょうか。Docker Desktop 自体のより詳しい使い方は、ドキュメント^{*4}も参照してください。

17.5 コマンドラインから Docker を使う

Docker Desktop のメリットのひとつは、GUI から操作できる点です。ですが「普段からコマンドラインを使っているため、CLI でも操作したい」「自動化したいのでマウス操作は困る」といった方もいるでしょう。Docker Desktop には CLI ツールである `docker` コマンドも同梱されているため、コマンドラインからの操作も可能です。

PowerShell を起動して、以下のコマンドを実行してみましょう。先ほど pull した nginx のコンテナイメージや、(コンテナを終了していなければ) 起動中の nginx のコンテナが表示されます。

```
$ docker images
$ docker ps
```

^{*3} 8080 以外のポートを指定してコンテナを起動した場合は、ポート番号を適宜変更してください。

^{*4} <https://docs.docker.com/desktop/use-desktop/>

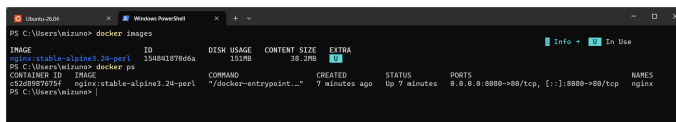


図 17.7: Windows の PowerShell から docker コマンドを実行できる

少し内部の仕組みを見てみましょう。以下のコマンドを実行してください。

```
$ docker context inspect
```

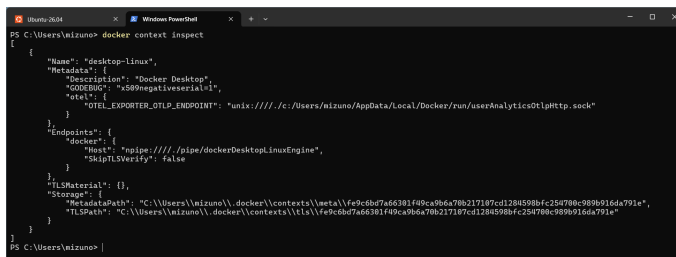


図 17.8: Docker のコンテキストの状態

`docker` コマンドは、Docker Engine との通信に必要な、エンドポイント情報やセキュリティ情報などをコンテキストという単位で管理しています。`docker` コマンドは使用するコンテキストを切り替えることで、複数の Docker Engine と通信できるのです。Docker Desktop をインストールすると、`desktop-linux` という名前のコンテキストが作成され、デフォルトでこれを利用するように設定されていることがわかります。またそのエンドポイントとして、Windows の名前つきパイプである `.\pipe\dockerDesktopLinuxEngine` が指定されています。この名前つきパイプを経由して、WSL 内の Docker Engine と通信を行っているというわけです。こうした仕組みによって、WSL の外側にある PowerShell から Docker が使えるわけです。

17.6 WSL 内の Linux から Docker Desktop を使う

PowerShell から Docker を使えることは、先に述べた通りです。ですが普段から Linux サーバーに慣れている方からすると、「PowerShell はちょっと不慣れ」「やっぱりコマンド操作は Linux の Bash から行いたい」という方も多いのではないのでしょうか。筆者もそうです。せっかく WSL ディストリビューションとして Ubuntu がインストールされているのですから、Ubuntu のシェルから Docker Desktop を利用できたら便利です。

Docker Desktop のウィンドウを開いたら、右上にある歯車のアイコンをクリックして、設定を開いてください。左ペインから **Resources** をクリックし、**WSL integration** のタブを開きます。すると以下の画面が表示されますので、「Enable integration with my default WSL distro」にチェックを入れてください。これでデフォルトの WSL ディストリビューション（本書の通りにインストールしているのであれば Ubuntu 26.04）において、Docker Desktop との連携設定が行われます。もし複数のディストリビューションをインストールしており、個別に連携設定を行いたい場合は、「Enable integration with additional distros」を個別にオンにしてください。

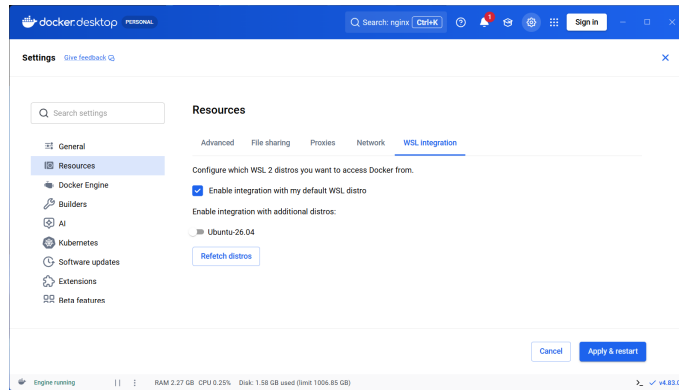


図 17.9: WSL ディストリビューションと Docker Desktop を連携する

Apply & restart をクリックして設定を適用したら、連携を設定した WSL の Linux ディストリビューションを起動してください。するとディストリビューション内でインストール作業を行っていないにもかかわらず、`docker` コマンドが使えるようになっていることに気づくでしょう。

`/usr/bin/docker` コマンドの実体を調べてみると、それは `/mnt/wsl` 以下にマウントされたファイルへのシンボリックリンクとなっていることがわかります。このファイルの実体は、Windows 上にある `docker-wsl-cli.iso` という ISO イメージ内に含まれています。Windows 上にインストールされた ISO イメージ^{*5}をループバックマウントすることで、WSL の Linux 内から利用可能にしているというわけです。

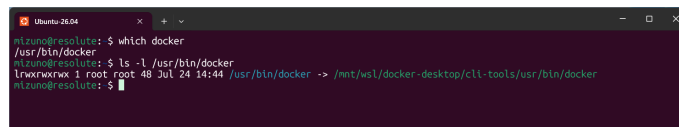


図 17.10: docker コマンドの実体は Windows 側にある

また Linux 内から `docker context` を確認すると、エンドポイントが `/var/run/docker.sock` という Unix ドメインソケットを向いていることがわかります。ですが Docker Desktop がこのソケットをプロキシしているため、`docker-desktop` ディストリビューションの Docker Engine と通信できるようになっています。

実際に Ubuntu 上から `docker` コマンドを実行してみましょう。先ほど Windows 上で pull したイメージや、実行中のコンテナが表示されます。つまり同一の Docker Engine と通信できていることがわかります。

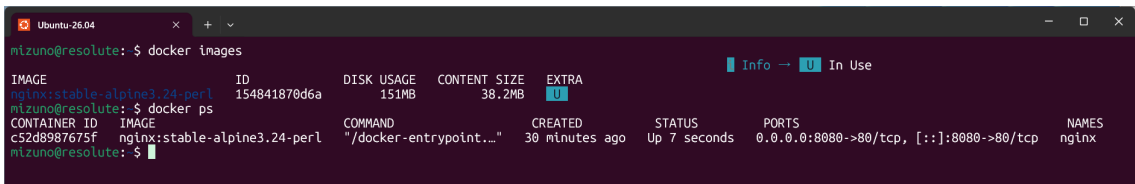


図 17.11: WSL の Ubuntu 内から docker コマンドを実行した例

^{*5} ユーザーごとに Docker Desktop をインストールした場合、このファイルは `%LOCALAPPDATA%\Programs\DockerDesktop\resources\wsl\docker-wsl-cli.iso` にあります。

Docker Desktop の WSL 連携を使うことで、クライアント環境としては好みのディストリビューションを使いつつ、Windows/WSL 間で共通の Docker Engine を利用できるのです。

17.7 Docker Desktop を使わない方法

このように至れり尽くせりな Docker Desktop ですが、企業での利用となると、そのライセンスが問題となることもあります。大規模な企業の業務では無償利用ができないため、現場レベルで気軽に試すといったことができません。

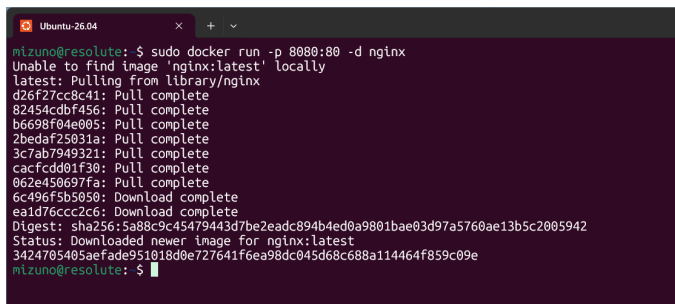
勘違いされる方もいるのですが、こうしたライセンスが課されているのは **Docker Desktop** です。コンテナの実行に必要な Docker Engine 自体は、Apache License 2.0 で公開されているオープンソースソフトウェアで、当然自由に利用できます。つまり Docker Desktop を使わず、Docker Engine をインストールすればよいということになります。そして Ubuntu には、**Docker** のパッケージが用意されており、これは WSL の中でも動かすことができます。

Ubuntu のシェルを起動したら、以下のコマンドを実行してください。なお Ubuntu 内に直接 Docker をインストールする際は、意図しないトラブルを避けるため、Docker Desktop の WSL 連携は無効にしておきましょう。あるいは Docker Desktop 自体をアンインストールしてしまってもよいでしょう。

```
$ sudo apt install -U -y docker.io
```

これだけで Docker が動き出します。前述の例と同じく、nginx のコンテナを動かすには以下のようにコマンドを実行してください。なお Ubuntu のパッケージから Docker をインストールした場合、**docker** コマンドの実行には root 権限が必要になります。

```
$ sudo docker run -p 8080:80 -d nginx
```



```
Ubuntu-26.04
mizuno@resolute:~$ sudo docker run -p 8080:80 -d nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
d26f27cc8c41: Pull complete
82454c0bf456: Pull complete
b6698f04e005: Pull complete
2bedaf25031a: Pull complete
3c7ab7949321: Pull complete
cacfcdd01f30: Pull complete
062e450697fa: Pull complete
6c496f5b5050: Download complete
eaid76ccc2c6: Download complete
Digest: sha256:5a88c9c45479443d7be2eadc894b4ed0a9801bae03d97a5760ae13b5c2005942
Status: Downloaded newer image for nginx:latest
3424705405aefade951018d0e727641f0ea98dc045d68c688a114464f859c09e
mizuno@resolute:~$
```

図 17.12: Ubuntu 内にインストールした Docker を使ってコンテナを起動する

先ほどと同様に、Windows の Web ブラウザーから <http://localhost:8080> にアクセスしてみましょう。WSL には自分自身（Windows）宛ての通信を、WSL が動作している仮想マシンへ転送する仕組みがあるため、これで問題なくコンテナ内と通信できます。

第 18 章 Kubernetes クラスターを構築してみよう

複数のコンテナを協調動作させるには、オーケストレーションツールの存在が欠かせません。この章ではコンテナオーケストレーションのデファクトスタンダードでもある Kubernetes を動かします。

18.1 コンテナオーケストレーションツールとは

ここまでの解説で、Docker を使ってコンテナを動かせるようになりました。アプリケーションをホスト OS から隔離して動かすコンテナは、現在のアプリケーションのデリバリーにおいて欠かせない技術となっています。

手元の PC でコンテナをひとつ起動するだけであれば、Docker を直接実行すればよいでしょうし、実際にそうしている開発者の方も多いはずです。ところが本番環境ではそうはいきません。システムは複数のコンテナで構成されることが一般的ですが、複数のコンテナを協調して動かそうとすると、様々な問題が浮上してきます。複数のノード上で動くコンテナ間の通信はどうやって実現したらよいのでしょうか？ 複数のアプリケーションコンテナが吐き出すログを一箇所に集約するには？ コンテナの死活監視や負荷分散、オートスケール、物理ノードがダウンした時の対応も考えなければいけません。

こうした問題を解決するのが、**コンテナオーケストレーション**と呼ばれるツールです。コンテナオーケストレーションは、複数のコンテナ化されたアプリケーションのデプロイ、スケーリング、管理を自動化します。数十から数百、場合によっては数千のコンテナを、本番環境で運用する場合には、自動化されたコンテナオーケストレーションが必要不可欠と言えるでしょう。そして現在、コンテナオーケストレーションの中でもデファクトスタンダードの地位を築いているのが **Kubernetes**（略して k8s と呼ばれます）です。

k8s はもともと、Google が社内で使用していた **Borg** というシステムをベースに開発されました。2014 年にオープンソース化され、現在ではクラウドネイティブアプリケーションの標準的なプラットフォームとして、多くの企業で採用されています。また AWS EKS、Google GKE、Azure AKS など、主要なクラウドプロバイダーもマネージド Kubernetes サービスを提供しています。

本番環境を k8s で構築するのであれば、当然開発者の手元にも開発用の k8s が必要になるでしょう。そこで WSL 上で簡単に k8s を動かす方法を紹介します。

18.2 Windows で Kubernetes を動かす方法

Kubernetes は Linux 上で動作するソフトウェアですから、Windows 上で動かすためには、バックエンドとして何らかの Linux 環境が必要となります。そこで WSL の Ubuntu 上に、**MicroK8s**^{*1}を直接インストールして構築します。

MicroK8s とは、Canonical 社が提供する軽量で導入しやすい Kubernetes ディストリビューションです。Kubernetes に必要な主要コンポーネントを小さくまとめており、非常に軽量、かつ簡単にインストール可能なのが特徴です。数ある k8s ディストリビューションの中でも、特に 1 台の PC による検証環境の構築に向いています。

なお Windows で Kubernetes を動かすには、MicroK8s 以外にも選択肢があります。例えば「Docker Desktop の Kubernetes 機能を使う」「Minikube や Kind でインストールする」などです。ですが今回は上記のようなメリットも考慮し、本格的な学習環境を構築可能な microk8s を選択しました。また WSL においてはいつものことですが、Ubuntu 上でのノウハウをそのまま流用できるのもメリットです。

^{*1} <https://microk8s.io/>

18.3 MicroK8s のインストール

それでは MicroK8s をインストールしていきましょう。MicroK8s は Snap パッケージとして提供されています。

```
$ sudo snap install microk8s --classic
```

`sudo` なしで `microk8s` コマンドを実行できるように、現在のユーザーを `microk8s` グループに追加します。

```
$ sudo gpasswd -a $USER microk8s
$ sudo chown -f -R $USER ~/.kube
```

グループの設定を反映するためには、一度 Ubuntu のシェルをすべて閉じ、開き直します。続いて以下のコマンドを実行してください。「microk8s is running」と表示されれば、MicroK8s が正常にインストールされ起動しています。

```
$ microk8s status --wait-ready
microk8s is running
(... 略 ...)
```

18.4 コマンドにエイリアスを設定する

Kubernetes のあらゆる操作は、REST API を通じて行われています。そのため API を叩くことさえできれば、クライアントの種類は問いません。実際 `curl` コマンドで Kubernetes を操作することも可能です。ですがそれでは使い勝手が悪いので、専用の CLI コマンドを使うのが一般的です。それが `kubectl` コマンドです。

Snap から MicroK8s をインストールすると、`kubectl` コマンドも同時にインストールされます。ですが Snap からインストールした環境はやや特殊で、各種コマンドには `microk8s.` というプリフィックスが付きまします。つまり `kubectl` コマンドは `microk8s.kubectl` という名前でインストールされているのです。毎回長いコマンドを入力するのは面倒なので、エイリアスを設定して一般的な `kubectl` として使えるようにしましょう。以下のコマンドを実行してください。

```
$ sudo snap alias microk8s.kubectl kubectl
```

これで `kubectl` コマンドが使えるようになります。エイリアスが正しく設定されたことを確認しましょう。

```
$ kubectl version --client
Client Version: v1.35.6
Kustomize Version: v5.7.1
```

Kubernetes クライアントのバージョン情報が表示されれば成功です。また何らかの理由でエイリアスを解除したい場合は、以下のコマンドを実行してください。

```
$ sudo snap unalias microk8s.kubectl
```

18.5 クラスターの情報を確認する

`kubectl cluster-info` コマンドで、k8s クラスターが正常に動作していることを確認できます。

```
$ kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:6443
CoreDNS is running at https://127.0.0.1:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Kubernetes コントロールプレーンのエンドポイントや、動いているコアサービスの情報が表示されます。

18.6 ノードの状態を確認する

`kubectl get nodes` コマンドを実行すると、クラスターに参加しているノードの情報を表示できます。今回はシングルノードのクラスターですので、自分自身が「Ready」として表示されることを確認しましょう。

```
$ kubectl get nodes
NAME          STATUS    ROLES    AGE   VERSION
resolute      Ready    <none>   8m16s v1.35.6
```

18.7 サービスを動かす

実際に Kubernetes 上でアプリケーションを動かしてみましょう。今回はシンプルな例として、Web サーバーの nginx を使用します。

Kubernetes は、いわゆる**宣言型**のアプローチを採用しています。Kubernetes では、クラスター内で実行したいリソースを**マニフェスト**という形で宣言します。ユーザーがシステムのあるべき姿を定義すると、具体的なリソースの作成や削除といった調整作業は、Kubernetes が自動的に行うのです。

以下のコマンドを実行し、`nginx-deployment.yaml` というファイルを作成してください。

```
$ cat <<EOF > nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 80
```

```
resources:
  requests:
    memory: "64Mi"
    cpu: "250m"
  limits:
    memory: "128Mi"
    cpu: "500m"
  readinessProbe:
    httpGet:
      path: /
      port: http
    initialDelaySeconds: 3
    periodSeconds: 5
  livenessProbe:
    httpGet:
      path: /
      port: http
    initialDelaySeconds: 10
    periodSeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: http
      nodePort: 30080
EOF
```

以下のコマンドで、マニフェストを適用します。

```
$ kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
service/nginx-service created
```

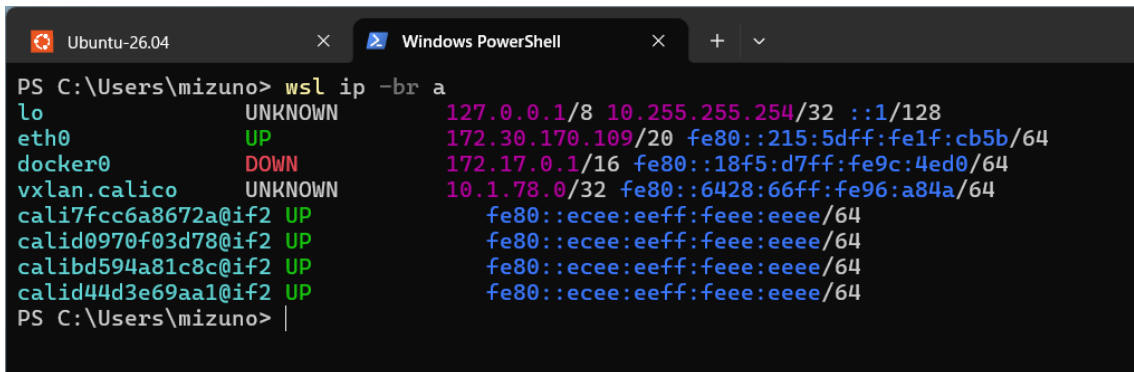
これだけで nginx の Pod^{*2}が起動します。また Pod にアクセスするためのサービスも作成されます。今回は NodePort タイプのサービスを作成していますので、ホストの IP アドレスでサービスに直接アクセスが可能です。以下のコマンドを実行してみましょう。nginx のサンプルページの HTML が表示されるはずです。

```
$ curl http://localhost:30080
```

Windows 上の Web ブラウザからアクセスもできます。ただしこの場合、localhost ではなく WSL が動作している仮想マシンの IP アドレスを指定する必要があります。PowerShell を起動して、以下のコマンドを実行してください。

^{*2} Pod は 1 つ以上のコンテナの集まりで、これが Kubernetes が管理するコンテナの最小単位となります。

```
$ wsl ip -br a
```



```

PS C:\Users\mizuno> wsl ip -br a
lo                UNKNOWN      127.0.0.1/8  10.255.255.254/32  ::1/128
eth0              UP          172.30.170.109/20  fe80::215:5dff:fe1f:cb5b/64
docker0          DOWN       172.17.0.1/16  fe80::18f5:d7ff:fe9c:4ed0/64
vxlans.calico     UNKNOWN    10.1.78.0/32  fe80::6428:66ff:fe96:a84a/64
cali7fcc6a8672a@if2 UP        fe80::ecce:eeff:feee:eeee/64
calid0970f03d78@if2 UP        fe80::ecce:eeff:feee:eeee/64
calibd594a81c8c@if2 UP        fe80::ecce:eeff:feee:eeee/64
calid44d3e69aa1@if2 UP        fe80::ecce:eeff:feee:eeee/64
PS C:\Users\mizuno> |

```

図 18.1: WSL の VM の IP アドレスを調べる

eth0に設定されているのが、WSL のネットワークインターフェイスに割り当てられた IP アドレスです。ここでは172.30.170.109となります。この IP アドレスとポート番号 (今回の例では 30080) を、Web ブラウザのアドレスバーに入力してください。

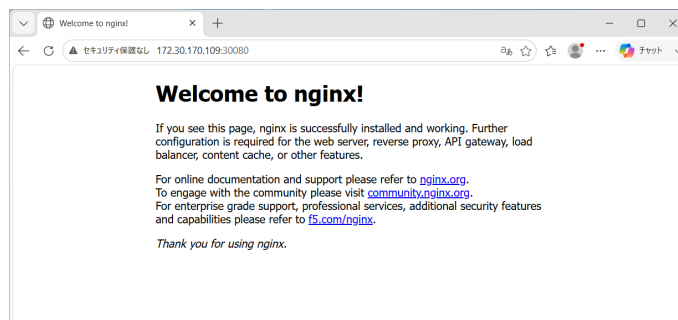


図 18.2: Windows から MicroK8s 上のサービスにアクセスした例

以下のコマンドを実行すると、Pod 内で起動している nginx のログを確認することができます。

```
$ kubectl logs deployment/nginx-deployment
```

動作テストが完了したら、以下のコマンドを実行してください。作成したリソース一式を削除できます。

```
$ kubectl delete -f nginx-deployment.yaml
```

18.8 アドオンを有効化する

MicroK8s には多数の便利なアドオンが用意されています。今回は、Kubernetes クラスターを視覚的に管理できる **Kubernetes Dashboard** を有効化してみましょう。以下のコマンドを実行してくだ

さい。Dashboard 用の Pod やサービスが自動的にデプロイされます。なお実行の途中で `sudo` のパスワードを求められます。

```
$ microk8s enable dashboard
```

続いて以下のコマンドを実行して、生成されたダッシュボードのトークンを取得します。これは Web UI からダッシュボードにログインする際に必要となります。表示された値を控えておいてください。

```
$ kubectl describe secret -n kube-system microk8s-dashboard-token
```

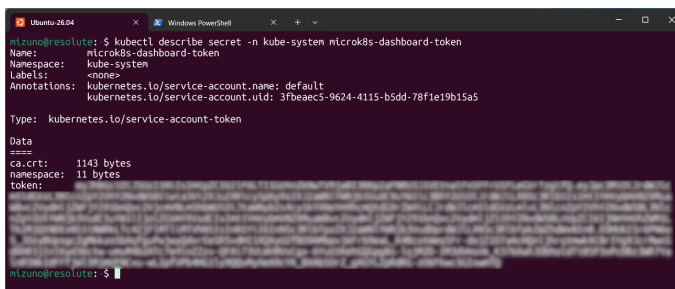


図 18.3: token 以下の値を控えておく

ダッシュボードのサービスは、先ほどの NodePort とは異なるため、ホストの IP アドレスで直接アクセスすることができません。そこでポートフォワードを行います。以下のコマンドを実行してください。コマンドの実行中に限り、外部からの接続をサービスヘフォワードできます。

```
$ kubectl port-forward -n kube-system service/kubernetes-dashboard 8443:443
```

Windows の Web ブラウザを開き、`https://localhost:8443` にアクセスしてください。スキーマが HTTPS であることと、今回は `localhost` でよい点に注意してください。なお SSL/TLS 証明書のセキュリティ警告が表示されますが、気にせずアクセスしてください。ログイン画面が表示されますので、先ほど控えたトークンを入力して `サインイン` をクリックします。

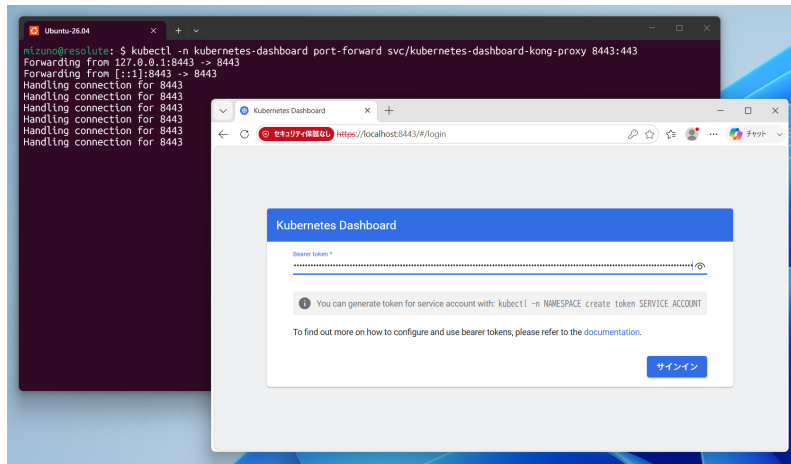


図 18.4: 控えたトークンをペーストしてサインインする

ダッシュボードが表示されれば成功です。

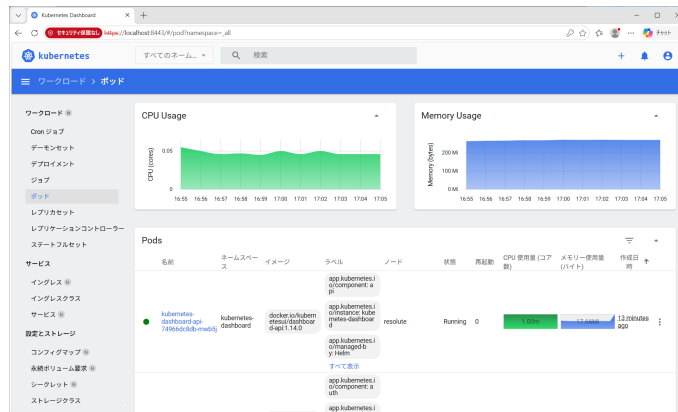


図 18.5: ダッシュボードの画面

Kubernetes は非常に複雑なシステムのため、慣れないうちは何がどうなっているのか、イメージしづらいことも多いでしょう。ダッシュボードは、Kubernetes クラスターの様々な側面をグラフィカルに表現してくれます。特にデプロイされているリソースを把握しやすいため、文字情報だけの CLI に比べ、学習の助けとなるでしょう。

コンテナを使うのであれば、オーケストレーションツールの学習もまた避けては通れません。Windows 上でも本格的な Kubernetes 環境を構築し、コンテナオーケストレーションの学習や開発作業を効率的に進めましょう。

第 3 部

WSL 応用編

第 19 章 WSL のカスタムディストリビューションを作ってみよう

WSL のディストリビューションは、ストアで配布されているものだけではありません。それどころか独自のカスタムを施したディストリビューションを作ることにも可能です。この章では Ubuntu をベースに、カスタムディストリビューションを作る方法を解説します。

19.1 カスタムディストリビューションとは

どのような OS であれ、インストール直後の状態で仕事をすることはできないでしょう。必要に応じて、様々なミドルウェアやアプリケーションをインストールし、環境に固有の設定も行う必要があります。個人用のデスクトップはもちろん、開発環境であればなおさらです。そこで本書では、Ubuntu へのパッケージのインストール方法や、カスタマイズ方法、設定ファイルの管理方法などを紹介してきました。要件に合わせて Ubuntu をカスタマイズし、開発環境として仕立て上げる方法が理解できてきたのではないのでしょうか。

ですが Linux に熟練したユーザーであればともかく、Linux の操作に慣れていない人の場合、開発環境の立ち上げに手こずって数日かかるといったこともありがちです。また仮に滞りなく操作ができるとしても、チーム全員が同じ作業を個別に行うのは無駄ですし、同じパッケージを各自がダウンロードするのもエコではありません。

そんな時に便利なのが、要件に合わせてあらかじめカスタマイズされた WSL ディストリビューションです。プロジェクトに必要なアプリやミドルウェア、社内環境に必要な設定などを、ディストリビューションのレベルであらかじめ施しておけば、WSL 上に展開するだけで開発環境を立ち上げることも可能なわけです。

従来の WSL ディストリビューションは、**APPX** という形式で配布されていました。ところがこの形式は、作成が非常に面倒くさい^{*1}という問題があります。また WSL ディストリビューションは、原則的に Microsoft Store 経由で配布する必要があり、これもハードルを上げる一因となっていました。

ところが 2024 年 11 月にリリースされた WSL のバージョン 2.4.4²以降では、Linux のルートファイルシステムにいくつかのメタデータファイルを追加し、tar 形式でアーカイブするだけで、WSL のディストリビューションを作成できるようになったのです。また作成したアーカイブの拡張子を **.wsl** に変更しておけば、エクスプローラー上でダブルクリックするだけで、インストールすることも可能になりました。例えばカスタマイズした Ubuntu の環境をアーカイブし、社内のファイルサーバーなどに配置しておけば、セットアップ済みの WSL 環境を、簡単に配布展開できるというわけです。

19.2 Ubuntu のルートファイルシステムを構築する

本書では Ubuntu をベースにカスタマイズを行っていきます。そのため、まずは素の Ubuntu のルートファイルシステムを作らなくてはなりません。

ルートファイルシステムとは、Linux のディレクトリツリーの頂点である **/** にマウントされるファイルシステムのことです。ルートファイルシステムには、システムが起動するために必要なバイナリ、ライブラリ、設定等のファイル一式が含まれています。具体的には **/bin** ディレクトリとその中の実行コマンドであったり、**/etc** ディレクトリとその中の設定ファイルであったりなどです。

^{*1} <https://learn.microsoft.com/ja-jp/windows/win32/appxpkg/how-to-create-a-package>

Ubuntu には、ルートファイルシステムを構築するためのツールがいくつか用意されています。その中でも一番有名なのが、**debootstrap**^{*2}です。debootstrap は何もない状態から、クリーンな Debian/Ubuntu のルートファイルシステムを構築するためのツールとして、広く普及しています。

ですが今回は debootstrap ではなく、その代替ツールである **mmdebstrap**^{*3}を採用しました。mmdebstrap は debootstrap と異なり、内部的に APT を利用してルートファイルシステムを構築します。そのため debootstrap よりも高速で、また複数のミラーリポジトリにも対応しており、サードパーティのリポジトリを追加することもできます。カスタマイズした開発環境を構築しようとする、非公式な外部リポジトリが必要になることもあるでしょう。そうした用途を考慮すると、mmdebstrap の方が向いていると判断したためです。

それでは実際に、カスタマイズ作業を行っていきましょう。まず Ubuntu に、以下のコマンドで **mmdebstrap** パッケージをインストールしてください。

```
$ sudo apt install -U -y mmdebstrap
```

以下のコマンドで、作業用のディレクトリを作成しておきましょう。mmdebstrap はルートファイルシステムをサブディレクトリ内に作成するため、この作業自体は必須ではありません。ですがひとつディレクトリを掘っておいた方が、後片付けがしやすい等のメリットがあるためです。

```
$ mkdir ~/work && cd ~/work
```

続いて **mmdebstrap** コマンドを実行してください。

```
$ sudo mmdebstrap \  
--components=main,restricted,universe,multiverse \  
--include=git,nginx \  
resolute \  
resolute-rootfs \  
http://ftp.uxd.icscoe.jp/Linux/ubuntu
```

--components オプションには、利用する Ubuntu リポジトリのコンポーネントを、スペースもしくはカンマ区切りで列挙します。ここでは 4 つのコンポーネントをすべて使用する設定としています。

--include オプションには、ルートファイルシステムに追加するパッケージを、スペースもしくはカンマ区切りで列挙します。ここではパッケージが追加されていることがわかりやすいよう、例として **git** パッケージと、**nginx** のパッケージを指定しています。実際はカスタマイズしたい内容に合わせて、必要なパッケージを列挙してください。

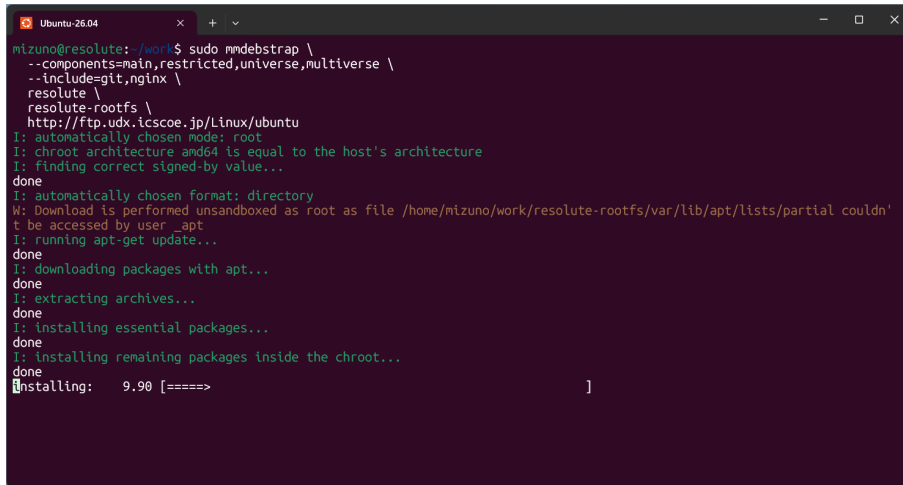
続いて Ubuntu のバージョンを指定します。ここでは **resolute** と指定しているため、Ubuntu 26.04 のルートファイルシステムが構築されます。慣れないと少しわかりづらいですが、Ubuntu ではリリースごとに、頭韻を踏んだ英単語で「形容詞 + 動物名」というコードネームを、アルファベット順につけるという慣習があります。このうち形容詞部分が、リポジトリのディレクトリ名などとして利用されるのです。Ubuntu 26.04 LTS のコードネームは **Resolute Raccoon** のため、ここが **resolute** になるというわけです。

その後の **resolute-rootfs** は、ターゲットとなるサブディレクトリ名です。ここでは先ほど作成した **work** ディレクトリ内で作業をしているため、**~/work/resolute-rootfs** というサブディレクトリが作成され、その中にルートファイルシステムが構築されることになります。

^{*2} <https://wiki.debian.org/Debootstrap>

^{*3} <https://gitlab.mister-muffin.de/josch/mmdebstrap>

最後の URL は、使用するリポジトリの URL です。ここでは日本国内にある、IPA のミラーサーバーを指定しています。リポジトリのサーバーは複数指定することも可能です。より詳しくはマニュアル^{*4}を参照してください。



```

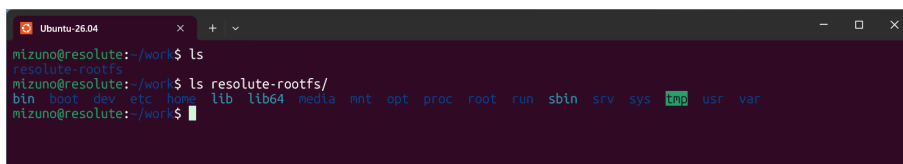
mizuno@resolute: /work$ sudo mmdebstrap \
--components=main,restricted,universe,multiverse \
--include=git,nginx \
resolute \
resolute-rootfs \
http://ftp.uxl.cscoe.jp/Linux/ubuntu
I: automatically chosen mode: root
I: chroot architecture amd64 is equal to the host's architecture
I: finding correct signed-by value...
done
I: automatically chosen format: directory
W: Download is performed unsandboxed as root as file /home/mizuno/work/resolute-rootfs/var/lib/apt/lists/partial couldn't be accessed by user _apt
I: running apt-get update...
done
I: downloading packages with apt...
done
I: extracting archives...
done
I: installing essential packages...
done
I: installing remaining packages inside the chroot...
done
Installing: 9.90 [=====]

```

図 19.1: mmdebstrap でルートファイルシステムを構築する

19.3 WSL 構成ファイルを作成する

mmdebstrap コマンドの実行が完了すると、resolute-rootfs サブディレクトリ内に、Ubuntu のルートファイルシステムが構築されています。軽く中を覗いてみましょう。



```

mizuno@resolute: /work$ ls
resolute-rootfs
mizuno@resolute: /work$ ls resolute-rootfs/
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys usr var
mizuno@resolute: /work$

```

図 19.2: bin や etc など、ルートファイルシステムとなるディレクトリツリー一式が構築されている

この Ubuntu 環境を、WSL ディストリビューションとして構成するためには、いくつかのメタデータファイルが必要です。これらのファイルも、ルートファイルシステム内に含める必要があります。まず WSL の起動方法を制御するファイルを用意します。以下のコマンドを実行してください。

```

$ cat <<'EOF' | sudo tee ~/work/resolute-rootfs/etc/wsl.conf
[boot]
systemd=true
EOF

```

続いて WSL を使用して初めて起動した際に、そのディストリビューションをどのように構成するかを制御する、~/work/resolute-rootfs/etc/wsl-distribution.conf というファイルを作成します。

^{*4} <https://manpages.ubuntu.com/manpages/resolute/man1/mmdebstrap.1.html>

```
$ cat <<'EOF' | sudo tee ~/work/resolute-rootfs/etc/wsl-distribution.conf
[oobe]
command = /etc/oobe.sh
defaultUid = 1000
defaultName = ubuntu-vtj

[shortcut]
enabled = true
icon = /usr/lib/wsl/vtj.ico

>windowsterminal]
enabled = true
profileTemplate = /usr/lib/wsl/terminal-profile.json
EOF
```

`command`は、はじめてシェルを起動した際に実行されるコマンドの指定です。ここでは`/etc/oobe.sh`というシェルスクリプトを起動して、初期化処理を行うよう設定しています。このスクリプトについては後述します。

`defaultUid`は、ディストリビューション内に作成されるユーザーの、既定のユーザー ID です。Ubuntu の場合は1000のままにしておいてください。

`defaultName`は、ディストリビューションの名前です。ここでは`ubutnu-vtj`としていますが、任意の名前に変更してください。

`icon`は、ショートカット等に表示されるアイコンファイルです。ルートファイルシステム内に任意のico形式のファイルをコピーした上で、そのパスを指定してください。

`ProfileTemplate`は、Windows Terminal 用に作成するプロファイルのテンプレートです。これも後述します。

19.4 初期化スクリプトの作成

先ほど`command`に指定した、初期化用のスクリプトを用意します。以下のコマンドで`~/work/resolute-rootfs`というファイルを作成してください。

```
$ cat <<'EOF' | sudo tee ~/work/resolute-rootfs/etc/oobe.sh
#!/bin/bash

set -ue

DEFAULT_GROUPS='adm,cdrom,sudo,dip,plugdev'
DEFAULT_UID='1000'
LANG=C.UTF-8

echo 'Please create a default UNIX user account. The username does not need to match your Windows user'
echo 'For more information visit: https://aka.ms/wslusers'

if getent passwd "$DEFAULT_UID" > /dev/null ; then
    echo 'User account already exists, skipping creation'
    exit 0
fi

while true; do
    read -p 'Enter new UNIX username: ' username

    if /usr/sbin/adduser --uid "$DEFAULT_UID" --quiet --comment ' ' "$username"; then
        if /usr/sbin/usermod "$username" -aG "$DEFAULT_GROUPS"; then
            break
        else
            /usr/sbin/deluser "$username"
        fi
    fi
done
```

```
done
EOF
```

スクリプトには、実行権限をつけておきます。

```
$ sudo chmod +x ~/work/resolute-rootfs/etc/oobe.sh
```

19.5 プロファイルの作成とアイコンファイルの配置

まず以下のコマンドで、`/usr/lib/wsl`ディレクトリを作成します。

```
$ sudo mkdir ~/work/resolute-rootfs/usr/lib/wsl
```

そしてこのディレクトリに、先ほど`wsl-distribution.conf`ファイルで指定したアイコンファイルをコピーしておいてください。

Windows Terminal 用のプロファイルを作成します。以下のコマンドを実行してください。

```
$ cat <<'EOF' | sudo tee ~/work/resolute-rootfs/usr/lib/wsl/terminal-profile.json
{
  "profiles": [
    {
      "colorScheme": "Ubuntu VTJ",
      "cursorShape": "bar",
      "cursorHeight": 25,
      "padding": "12, 8, 12, 8",
      "antialiasingMode": "grayscale",
      "font": {
        "face": "Cascadia Mono",
        "size": 11,
        "weight": "normal"
      }
    }
  ],
  "schemes": [
    {
      "name": "Ubuntu VTJ",

      "foreground": "#B8E6B8",
      "background": "#300A24",
      "cursorColor": "#C8F7C8",
      "selectionBackground": "#5E2750",

      "black": "#2E3436",
      "red": "#E05A5A",
      "green": "#72D572",
      "yellow": "#D6C56C",
      "blue": "#6FA8DC",
      "purple": "#B58CCB",
      "cyan": "#63C5C5",
      "white": "#D8E8D8",

      "brightBlack": "#6B776B",
      "brightRed": "#FF7777",
      "brightGreen": "#98F598",
      "brightYellow": "#F0DD88",
      "brightBlue": "#8FC7F0",
      "brightPurple": "#D3A9E8",
      "brightCyan": "#88E5E5",
      "brightWhite": "#FFFFFF"
    }
  ]
}
```



```

    ]
}
EOF

```

ここでは Ubuntu VTJ というカラスキーマを新規作成した上で、それを参照するよう設定しています。フォントや色は、好みの設定に変更しても構いません。なお JSON ファイルのフォーマットについて、詳しくはドキュメント^{*5}を参照してください。

19.6 その他のファイルの配置

この段階でディレクトリ内にファイルを追加すれば、WSL ディストリビューションにそのファイルを含めることができます。例えばインストールしたアプリが社内サーバーを使うよう、デフォルトの接続設定を記述したファイルを入れておく、といった具合です。用途に応じてカスタマイズを行ってください。ここではわかりやすいよう、デフォルトで用意されている `/var/www/html/index.nginx-debian.html` を削除し、カスタムした `index.html` を配置してみます。

```

$ sudo rm -f ~/work/resolute-rootfs/var/www/html/index.nginx-debian.html
$ cat <<'EOF' | sudo tee ~/work/resolute-rootfs/var/www/html/index.html
Hello WSL
EOF

```

19.7 ディストリビューションのアーカイブ

カスタマイズと、必要なメタデータファイルの配置が完了したら、以下のコマンドでルートファイルシステムをアーカイブします。ルートファイルシステムを構築したディレクトリごとではなく、その中身を直接アーカイブする点に気をつけてください。本記事の例で言えば、`resolute-rootfs` ディレクトリをアーカイブするのではなく、このディレクトリ内に入って、中身をアーカイブしなくてはなりません。

```

$ cd ~/work/resolute-rootfs
$ sudo tar --numeric-owner --absolute-names -c * | gzip --best > ~/work/ubuntu-vtj.tar.gz

```

`~/work` ディレクトリ内に `ubuntu-vtj.tar.gz` というアーカイブが作成されますので、拡張子を `.wsl` に変更しておきましょう。

^{*5} <https://learn.microsoft.com/en-us/windows/terminal/json-fragment-extensions#structure-of-the-json-files>

```
$ mv ~/work/ubuntu-vtj.{tar.gz,wsl}
```

出来上がった wsl ファイルを、インストール先の Windows マシンにコピーしてください。

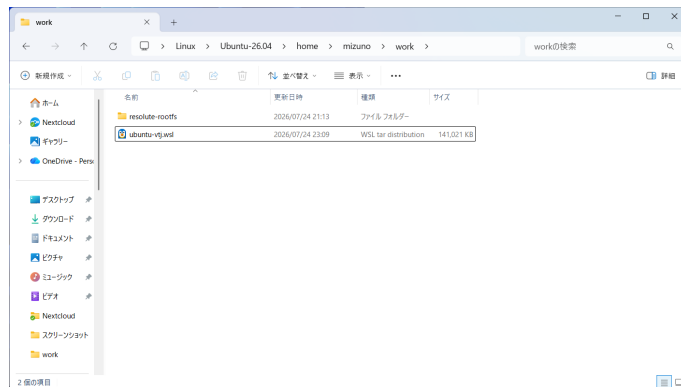


図 19.3: エクスプローラで WSL 内のフォルダを開くと、成果物を取り出しやすい

19.8 ディストリビューションのインストール

拡張子が `.wsl` のファイルは、エクスプローラー上でペンギンのアイコンで表示されます。このファイルをダブルクリックすると、自動的に Windows Terminal が起動し、インストール作業が行われます。

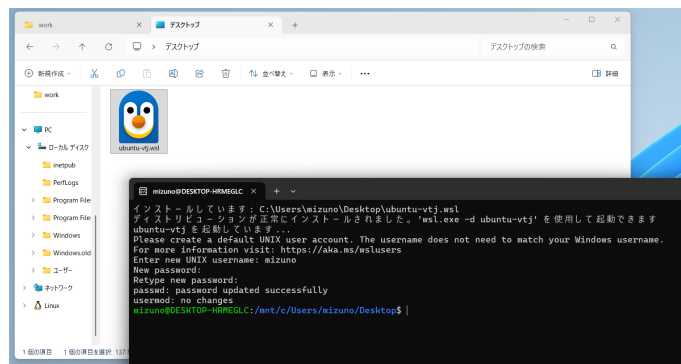


図 19.4: 作成した oobe.sh が自動的に実行され、ユーザーの作成も行われる



図 19.5: 同時にスタートメニューへの追加も行われる

19.9 カスタムディストリビューションの起動

インストールが完了したら、一度 Windows Terminal を再起動してください。再起動後にタブバーにある下向き矢印をクリックすると、`ubuntu-vtj` という (`wsl-distribution.conf` で指定した名前の) プロファイルが追加されているはずです。これを選択してディストリビューションを起動してみましょう。文字の色がプロファイルテンプレートで指定した、カスタムカラーになっていることもわかるでしょう。

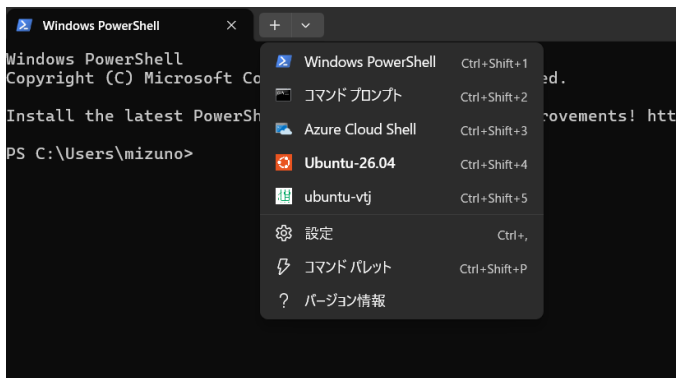


図 19.6: 新しいプロファイルが作成されている

追加した `nginx` の動作も確認してみましょう。`ss -lntp` コマンドを `root` 権限で実行すると、`nginx` のプロセスがポートを待ち受けていることがわかります。

```
$ sudo ss -lntp
```

Windows で Web ブラウザーを起動して、`http://localhost` にアクセスしてみましょう。作成した `index.html` が表示されれば成功です。

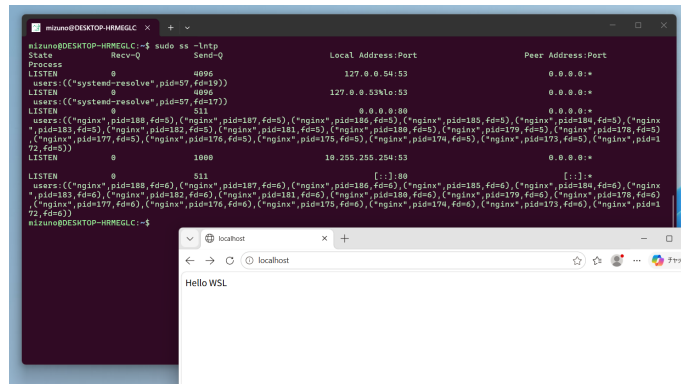


図 19.7: カスタムディストリビューション内で動作している Web サーバーにアクセスできた

このようにまっさらなルートファイルシステムを用意し、任意のパッケージを追加したり、独自のファイルを配置することで、簡単に WSL 用のカスタムディストリビューションが作成できます。またこれらの作業は、Ubuntu サーバーさえあれば自動化することも、それほど難しくありません。CI/CD を使って、定期的に自社向けのディストリビューションを更新するというのも面白そうです。

第 20 章 WSL 環境の初期セットアップを自動化してみよう

クラウド環境では、インスタンスの起動時に自律的な初期化を行うのが基本です。この章では WSL 環境のセットアップを自動化する方法について解説します。

20.1 cloud-init とは

開発に使うミドルウェアやアプリがプリインストールされたカスタムディストリビューションを作れば、開発環境の構築をより効率化できるでしょう。ですが、カスタムディストリビューションに含めにくい、個別のカスタマイズというものも存在します。例えば以下のようなものです。

- 個人のメールアドレスや Git 設定
- ログインに使う SSH 公開鍵
- .bashrc などの個人向け設定ファイル
- プロジェクト固有の環境変数

これらの個人設定は、共通のイメージに含めることはできません。開発者ごとに異なる設定が必要な上、項目によってはセキュリティリスクになることもあるためです。こうした環境ごとに異なる設定は、初回起動時に自律的に行うべきです。そしてこれを実現するのが **cloud-init** です。

cloud-init は、クラウドインスタンスの初期化を自動化するためのツールの名称です。元々は Amazon EC2 などのクラウド環境で、仮想マシンの初回起動時に自律的な初期化を行うために開発されました。現在では AWS や Azure といったクラウドのみならず、VMware や Proxmox などの仮想化ソフトウェア、LXD/LXC などのコンテナプラットフォーム、MAAS などのベアメタル環境、そして本章で紹介する WSL など、幅広い環境で利用されています。

20.2 user-data とは

cloud-init では、起動時にインスタンスに流し込まれた **user-data** と呼ばれるデータをもとに初期化を行います。user-data は複数のフォーマットをサポートしていますが、その中でもよく使われるのが、**Cloud config** と呼ばれる YAML 形式のデータです。具体的には以下のようなものです。

```
#cloud-config
users:
  - name: developer
    sudo: ALL=(ALL) NOPASSWD:ALL
    ssh_authorized_keys:
      - ssh-rsa AAAAB3Nza...(略)

packages:
  - nginx

write_files:
  - path: /etc/nginx/sites-available/myapp
    content: |
      server {
        listen 80;
        server_name myapp.example.com;
        (... 略...)
```

```

    }

    runcmd:
      - apt -U -y full-upgrade

```

内容について、少し詳しく解説します。まず Cloud config は、`#cloud-config` という行で開始する必要があります。

```
#cloud-config
```

`users` セクションでは、作成するユーザーを設定します。ここでは `developer` というユーザーを作成し、パスワードなしの `sudo` を許可し、SSH ログイン用の公開鍵を設定しています。

```

users:
  - name: developer
    sudo: ALL=(ALL) NOPASSWD:ALL
    ssh_authorized_keys:
      - ssh-rsa AAAAB3Nza...(略)

```

`packages` セクションでは、システムに追加でインストールするパッケージを指定します。ここでは `nginx` パッケージをインストールするよう指定しています。

```

packages:
  - nginx

```

`write_files` セクションでは、任意のファイルを指定した内容で作成できます。ここで `nginx` のホストの設定を追加しています^{*1}。

```

write_files:
  - path: /etc/nginx/sites-available/myapp
    content: |
      server {
        listen 80;
        server_name myapp.example.com;
        (... 略...)
      }

```

`runcmd` セクションでは、任意のコマンドを実行できます。ここでは OS 全体をアップデートする `apt` コマンドを実行します。

```

runcmd:
  - apt -U -y full-upgrade

```

^{*1} ただしこのファイルを作成しただけでは、ホスト設定は有効になりません。実際にこのホストを運用するには、このファイルに対し `/etc/nginx/sites-enabled` 以下からシンボリックリンクを張る必要があります。

なおその他の user-data のフォーマットについては、ドキュメント^{*2}を参照してください。

20.3 cloud-init で WSL を初期化する

それでは実際に user-data を作成し、WSL のディストリビューションを初期化してみましょう。Windows キー +Rを押して「ファイル名を指定して実行」のダイアログを表示します。ここに%USERPROFILE%と入力してEnterを押してください。エクスプローラーで、自分のユーザーフォルダが開きます。ここに .cloud-init という名前で、新しいフォルダを作成してください。

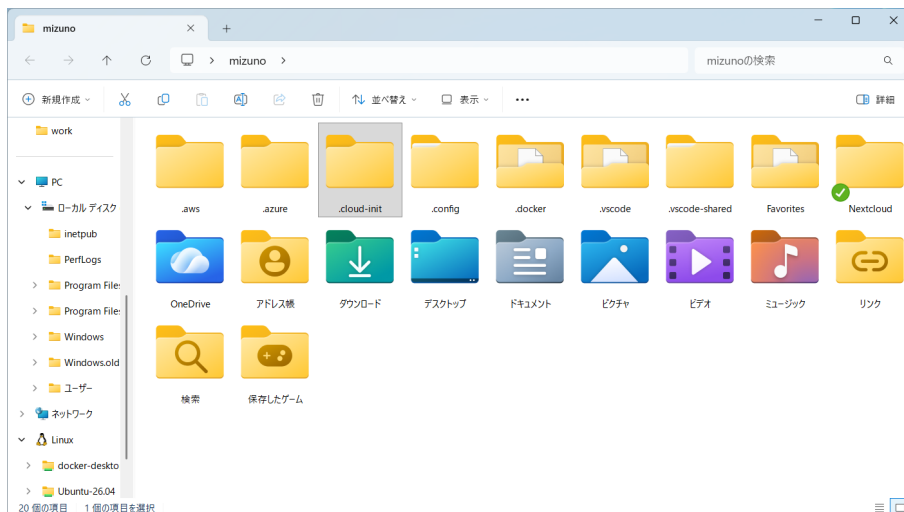


図 20.1: .cloud-init フォルダを作成する

.cloud-init フォルダの中に、以下の内容のファイルを ubuntu-all.user-data という名前で作成してください。

```
#cloud-config
users:
  - name: developer
    groups: adm, dialout, cdrom, floppy, sudo, audio, dip, video, plugdev, netdev
    sudo: ALL=(ALL) NOPASSWD:ALL
    shell: /bin/bash

timezone: Asia/Tokyo

locale: ja_JP.UTF-8

packages:
  - vim
  - fzf
  - curl

write_files:
  - path: /etc/wsl.conf
```

^{*2} <https://cloudinit.readthedocs.io/en/latest/explanation/format.html>

```

append: true
content: |
    [user]
    default=developer

runcmd:
- apt -U -y full-upgrade
- sudo -u developer git config --global user.name (自分の名前)
- sudo -u developer git config --global user.email (自分のメールアドレス)

```

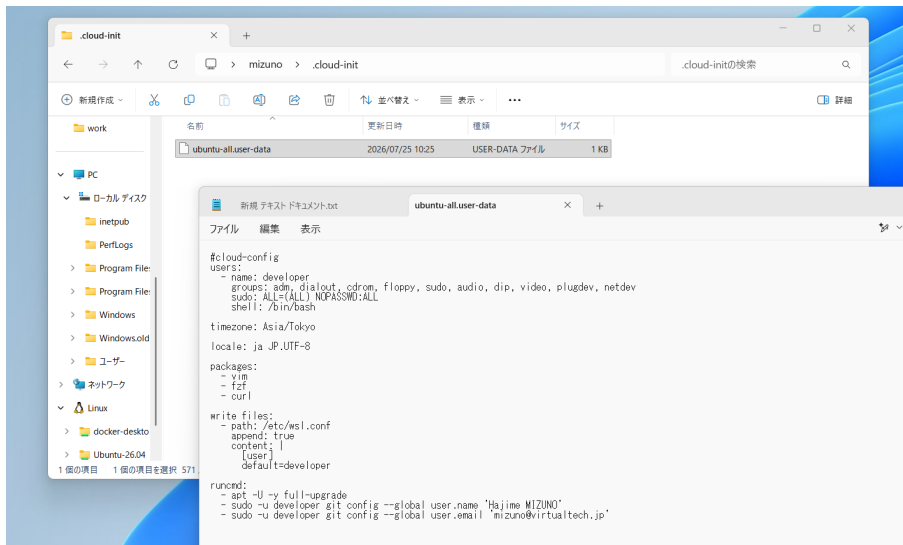


図 20.2: 作成した user-data

user-data が作成できたら、Ubuntu 26.04 を新しくインストールして、初期化が正しく行われるか見てみましょう。PowerShell を起動して、以下のコマンドを実行してください。ここでは `--name` オプションを使用して、ディストリビューションに `cloud-init-test` 名前をつけています。WSL では固有の名前をつけることで、同一のディストリビューション（ここでは Ubuntu 26.04）の独立した実体を、PC 上に複数インストールできます。これを **インスタンス** と呼びます。

```
$ wsl --install -d Ubuntu-26.04 --name cloud-init-test
```

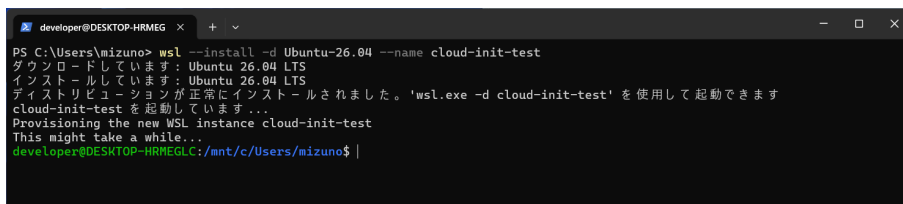
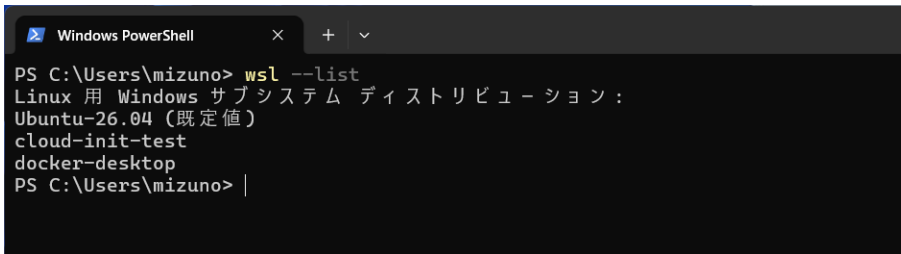


図 20.3: 名前をつけてディストリビューションをインストール

PowerShell で以下のコマンドを実行してみてください。既にインストールされている `Ubuntu-26.04`

(デフォルトのインスタンス名) にくわえて、`cloud-init-test`がインストールされていることがわかるでしょう*³。

```
$ wsl --list
```



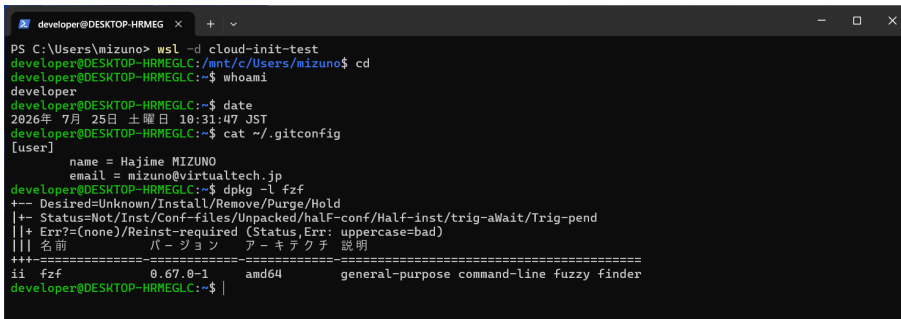
```
Windows PowerShell
PS C:\Users\mizuno> wsl --list
Linux 用 Windows サブシステム ディストリビューション :
Ubuntu-26.04 (既定値)
cloud-init-test
docker-desktop
PS C:\Users\mizuno> |
```

図 20.4: インストールされているインスタンスのリスト

ディストリビューションは、以下のように `wsl` コマンドの `-d` オプションに名前を指定して起動できます。

```
$ wsl -d cloud-init-test
```

インストールしたディストリビューションを起動したら、`cloud-init` の内容が反映されていることを確認しましょう。インストール直後に、ユーザーの作成プロンプトが表示されなかったことから察せられると思いますが、既に `developer` ユーザーが作成されています。タイムゾーンは日本になっていますし、Git の個人的な設定が既に行われており、`fzf` パッケージも追加されています。



```
developer@DESKTOP-HRMEGLC: ~
PS C:\Users\mizuno> wsl -d cloud-init-test
developer@DESKTOP-HRMEGLC: /mnt/c/Users/mizuno$ cd
developer@DESKTOP-HRMEGLC: ~$ whoami
developer
developer@DESKTOP-HRMEGLC: ~$ date
2026年 7月 25日 土曜日 10:31:47 JST
developer@DESKTOP-HRMEGLC: ~$ cat ~/.gitconfig
[user]
  name = Hajime MIZUNO
  email = mizuno@virtualtech.jp
developer@DESKTOP-HRMEGLC: ~$ dpkg -l fzf
+--- Desired=Unknown/Install/Remove/Purge/Hold
+--- Status=Not/Inst/Conf-files/Unpacked/half-inst/trig-aWait/Trig-pend
+--- Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||| 名前 バージョン アーキテクチ 説明
+---+
ii fzf 0.67.0-1 amd64 general-purpose command-line fuzzy finder
developer@DESKTOP-HRMEGLC: ~$ |
```

図 20.5: `cloud-init` による初期化が行われているか確認する

`/etc/wsl.conf` にも、デフォルトユーザーの設定が追記されています。

*³ Docker Desktop をインストール済みの場合は、Docker 用のインスタンスも存在するはずです。

```

developer@DESKTOP-HRMEG  x  +  v
developer@DESKTOP-HRMEGLC:~$ cat /etc/wsl.conf
[boot]
systemd=true
[user]
default=developer
developer@DESKTOP-HRMEGLC:~$ |

```

図 20.6: wsl.conf ファイルへの追記も行われている

20.4 バージョンごとに初期化プロセスを使い分ける

先ほどの YAML ファイルは `ubuntu-all.user-data` という名前で作成しました。ここから想像がつくかもしれませんが、WSL の `cloud-init` では、ディストリビューションやバージョンによって異なる `user-data` を自動的に適用できます。この `user-data` のファイル名は `<ID>-<VERSION_ID>.user-data` というフォーマットとなっています。

ID と VERSION_ID には、対象となるディストリビューションの ID とバージョンに置き換えます。例えば Ubuntu 26.04 上で、以下のコマンドを実行してみてください。

```
$ grep -E '\b(VERSION_ID|ID)\b' /etc/os-release
```

```

developer@DESKTOP-HRMEG  x  +  v
developer@DESKTOP-HRMEGLC:~$ grep -E '\b(VERSION_ID|ID)\b' /etc/os-release
VERSION_ID="26.04"
ID=ubuntu
developer@DESKTOP-HRMEGLC:~$ |

```

図 20.7: Ubuntu 26.04 の ID と VERSION_ID を /etc/os-release から調べる

Ubuntu 26.04 は、ID が `ubuntu`、VERSION_ID が `26.04` であることがわかります。そのため `ubuntu-26.04.user-data` という名前で YAML を用意すると、その初期化処理は Ubuntu 26.04 に対してのみ適用されます。ただし先ほどは VERSION_ID 部分を `all` としていました。この場合はすべてのバージョンの Ubuntu が対象となります。これを利用することでパッケージ名が変更されているといった、Ubuntu のバージョンごとの差異を吸収することができます。

ファイル名を `default.user-data` とすると、すべてのディストリビューションに適用される `user-data` を作成することができます。また特定の名前をつけたインスタンス（先ほどの例では `cloud-init-test`）にのみ適用される `user-data` も作れます。その場合はファイル名を `インスタンス名.user-data` としてください。

ここまで聞くと、Ubuntu 以外のディストリビューション向けの `user-data` も作れると思うかもしれませんが、原理的にはそうなのですが、初回起動時に `%USERPROFILE%\cloud-init` 以下から `user-data` を読み込むという挙動が正しく行われるのは、Ubuntu 側のカスタマイズによるものです。たとえば **AlmaLinux 10** などをそのままインストールしても、`cloud-init` による初期化はできません。もしも Ubuntu と同様のことを他のディストリビューションでも行いたいのであれば、`cloud-init` を適切に設定した、カスタムディストリビューションを作成する必要があるでしょう。

このように `cloud-init` を利用すれば、これまで手動で行っていた個人レベルの設定や、プロジェクト固有の環境構築を完全に自動化できます。開発環境は、何度も作り直すことがあるでしょう。何度も

行う作業であればあるほど、自動化による恩恵は大きくなります。バージョンごとに異なる user-data を用意できる柔軟性により、環境の差異を適切に吸収できるのも大きなメリットです。cloud-init 自体は、クラウド環境の初期化においても必須のツールです。user-data の記述方法に慣れておくと、本番の構築においても役立つでしょう。まずは手元の環境から、cloud-init をぜひ活用してみてください。

第 21 章 WSL 環境をまるごとバックアップしよう

WSL 環境は、別インスタンスを作ることも、再インストールすることも簡単に行えます。とはいえ万が一に備え、現状の環境をバックアップしたり、複製したいと思うこともあるでしょう。この章では WSL 環境のバックアップ方法を解説します。

21.1 WSL 環境をエクスポートする

カスタムディストリビューションや cloud-init を活用すれば、同じ環境をすばやく、何度でも作り直せます。しかしゼロから環境をクリーンに構築し直すのではなく、今動いている環境をまるごとバックアップしたいというニーズもあるでしょう。たとえば、開発環境に大きな変更を加える前にバックアップを取っておきたい、同じ環境を別の PC でも使いたい、問題が発生した環境を複製してデバッグしたい、といった場合です。

まず PowerShell を起動し、`wsl --list` コマンドを実行してください。インストールされているディストリビューションの（インスタンスの）一覧が表示されます。

```
$ wsl --list
Linux 用 Windows サブシステム ディストリビューション:
Ubuntu-26.04 (既定値)
docker-desktop
```

上記の例では Ubuntu-26.04 と、docker-desktop の 2 つがインストールされています。このうち Ubuntu-26.04 をバックアップしてみます。WSL 環境をバックアップするには、`wsl --export` コマンドを使います。このコマンドは、指定したインスタンス全体を `tar` アーカイブとしてエクスポートできます。実際に Ubuntu-26.04 をエクスポートするコマンドが以下の例です。

```
$ wsl --export Ubuntu-26.04 ubuntu.tar
```



図 21.1: インストールされているインスタンスの確認とエクスポートの実行

エクスポートが完了すると、カレントディレクトリ（上記の例では Windows のホームディレクトリ）に、`ubuntu.tar` という名前でアーカイブファイルが作成されます。この `tar` ファイルには、WSL インスタンスのルートファイルシステム (`/`) 以下の全ファイルを含んでいます。通常の Linux ツールで中身を展開したり、確認することもできます。実際に WSL の Ubuntu 上から、`tar` の中身を確認してみましょう。

```
$ tar tvf /mnt/c/Users/(Windows のユーザー名)/ubuntu.tar
./
./lib
./dev/
./lib64
./opt/
./sbin
./home/
./usr/
(... 略...)
```

21.1.1 エクスポートにおける注意点

エクスポートの実行に際しては、いくつか注意点があります。まずエクスポートを実行する際、対象の WSL ディストリビューションが自動的に停止される点です。実行中のプロセスがある場合は終了されるため、作業中のデータは事前に保存し、実行中のアプリケーションは終了しておきましょう。ディストリビューションの起動状態は、`wsl --list -v` コマンドで確認できます。

```
$ wsl --list -v
NAME                STATE      VERSION
* Ubuntu-26.04      Stopped    2
docker-desktop      Stopped    2
```

エクスポートした後の `tar` ファイルは、ディストリビューションの使用量に応じて大きくなります。数 GB～数十 GB になることもあるため、エクスポート先に十分な空き容量を確保してください。そして当然ですが、ファイルサイズによっては、エクスポートに数分以上の時間がかかることがあります。

Linux 環境において、`tar` アーカイブは `gzip` などを併用して圧縮するのが一般的です。ですが `wsl --export` が生成するのは、無圧縮の `tar` ファイルとなります。ネットワーク越しに転送するような場合は、必要に応じて、後から圧縮ツールで圧縮することも検討してください。

21.2 エクスポートしたアーカイブをインポートする

エクスポートした `tar` ファイルは、`wsl --import` コマンドでインポートできます。インポートすることで、バックアップから環境を復元したり、同じ環境を複製したりできます。

例えば先ほどエクスポートした `ubuntu.tar` を `Ubuntu-Imported` という名前で、`C:\wsl` ディレクトリ以下にインポートする場合は、以下のように実行します。まずインポート先となる `C:\wsl` ディレクトリを作成します。

```
$ mkdir c:\wsl
```

続いて以下のコマンドを実行します。

```
$ wsl --import Ubuntu-Imported C:\wsl\Ubuntu-Imported ubuntu.tar
```

このコマンドを実行すると、内部的に以下の処理が行われます。

1. `C:\wsl\Ubuntu-Imported` ディレクトリが作成される
2. `tar` ファイルが展開され、仮想ディスクファイル `ext4.vhdx` として保存される
3. `Ubuntu-Imported` という名前のディストリビューションとして WSL に登録される

インポート完了後に `wsl --list -v` コマンドを実行してみましょう。Ubuntu-Importedが追加されていることがわかります。

```
$ wsl --list -v
NAME                STATE      VERSION
* Ubuntu-26.04      Stopped    2
  Ubuntu-Imported    Stopped    2
  docker-desktop     Stopped    2
```

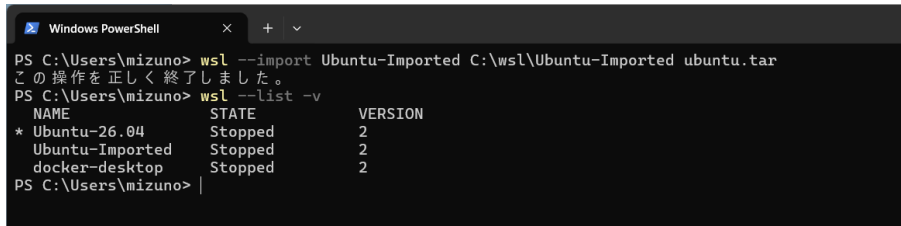


図 21.2: インポートの実行例

これで稼働中のディストリビューションの複製ができました。ちなみに、さらに異なる名前を指定することで、同じ環境を複数作成することも可能です。以下は同一のアーカイブから、開発用、テスト用、本番用の環境を作る例です。

```
$ wsl --import Ubuntu-Dev C:\wsl\Ubuntu-Dev ubuntu.tar
$ wsl --import Ubuntu-Test C:\wsl\Ubuntu-Test ubuntu.tar
$ wsl --import Ubuntu-Prod C:\wsl\Ubuntu-Prod ubuntu.tar
```

21.3 インポート先のディレクトリはどうするか

`wsl --import` コマンドでは、インポート先のディレクトリを明示的に指定する必要がありました。では WSL のディストリビューションは、どこに保存すべきでしょうか。

`wsl --install` コマンドで Microsoft Store からインストールできる、公式のディストリビューションのインストール先は以下コマンドで調べられます。

```
$ Get-ItemProperty -Path Registry::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Lxss\*
DistributionName BasePath
-----
Ubuntu-26.04     C:\Users\mizuno\AppData\Local\wsl\{5f639734-5179-4d37-a394-c32f04801887}
```

この BasePath、つまり `C:\Users\mizuno\AppData\Local\wsl\{ディストリビューション ID}` 以下に、仮想ディスクファイル `ext4.vhdx` が保存されています。ですが筆者は、自分でインポートするディストリビューションは、上記のストアディストリビューションとは別のディレクトリに集約して管理することをお勧めします。例えば以下のようになると管理しやすいでしょう。

```
C:\wsl\
├── Ubuntu-Dev\      # 開発環境
├── Ubuntu-Test\     # テスト環境
└── Ubuntu-Backup\   # バックアップ環境
```

こうすることで、自分が手動でインポートしたディストリビューションが一目瞭然となり、管理しやすくなります。またデフォルトのインストール先はディレクトリのパスが深い上、ディレクトリ名が ID のため、どのディストリか判断しづらかったりしますが、そうした問題も解消できます。なおインポート先のディレクトリ名に、日本語や特殊文字を含むことは避けましょう。余計なトラブルを避けるためにも、英数字のみのパスを利用するのが安全です。

21.4 エクスポートとインポートの実践的活用例

ここでは、エクスポートとインポート機能の実践的な活用シーンを紹介します。

21.4.1 バックアップと復元

もっともありがちなのが、環境に大きな変更を加える前にバックアップとして取得するパターンです。万が一、OS が起動しなくなるような問題が発生しても、即座に元の状態に戻せます。

```
$ wsl --export Ubuntu-26.04 ubuntu-before-upgrade.tar
```

もしも問題が発生した場合、一度壊れた `Ubuntu-26.04` を削除した上で、同名でインポートして復元します。

```
$ wsl --unregister Ubuntu-26.04
$ wsl --import Ubuntu-26.04 C:\wsl\Ubuntu-2604 ubuntu-before-upgrade.tar
```

環境全体をいつでも復元できる安心感は、新しい技術や設定に挑戦するハードルを下げてくれるでしょう。

21.4.2 環境のマシン間移行

例えば新しい PC を購入した際に、古い PC の WSL 環境をそのまま移行できます。まず古い PC で WSL 環境をエクスポートします。

```
$ wsl --export Ubuntu-26.04 ubuntu-migration.tar
```

作成された `tar` ファイルを USB メモリやクラウドストレージに保存しましょう。必要であれば `tar` を圧縮して、転送効率を上げるのもよい考えです。そして続いて新しい PC に、WSL 本体をインストールしましょう。

```
$ wsl --install --no-distribution
```

ディレクトリを作成し、`tar` ファイルをインポートします。

```
$ mkdir c:\wsl
$ wsl --import Ubuntu-26.04 C:\wsl\Ubuntu-2604 ubuntu-migration.tar
```

これで開発環境を再構築する手間なく、すぐに作業を再開できます。

21.4.3 デバッグ: 問題が起きた環境を複製して調査

他の開発者の環境で問題が発生した場合、その環境をそのまま複製することで、別メンバーがデバッグ環境として使えます。問題が起きた環境で、エクスポートを実行します。

```
$ wsl --export Ubuntu-26.04 ubuntu-issue.tar
```

他の開発者の PC に tar をコピーし、デバッグ用に複製します。

```
$ wsl --import Ubuntu-Debug C:\wsl\Ubuntu-Debug ubuntu-issue.tar
```

問題の環境を起動し、デバッグ環境で問題の切り分け、ログ確認、修正の試行といった作業を行いましょう。

```
$ wsl -d Ubuntu-Debug
```

これにより自分の環境ではバグが再現しないといった問題を防ぎ、確実に問題を調査できます。

21.5 タスクスケジューラで WSL 環境を自動バックアップ

wsl --export コマンドを使えば、WSL 環境の完全なバックアップを作成できます。バックアップを日々作成することで、データ損失のリスクを大幅に軽減できるでしょう。ですが人間は面倒くさい作業はついサボってしまうものです。そのためバックアップのような定期的に行う作業は自動化すべきです。ここでは Windows のタスクスケジューラを使って、WSL 環境を自動的にバックアップする方法を紹介します。

21.5.1 バックアップスクリプトの作成

Windows タスクスケジューラとは、指定した時刻やイベントに応じて、プログラムやスクリプトを自動実行する Windows の機能です。まずバックアップを実行する PowerShell スクリプトを作成しましょう。以下は、日付付きファイル名でバックアップを保存する PowerShell スクリプト例です。

```
# バックアップ対象の名前
$DistroName = "Ubuntu-26.04"

# バックアップ先ディレクトリ
$BackupDir = "C:\wsl\" + $DistroName + "-Backup"

# ログファイルのパス
$LogFile = Join-Path $BackupDir "backup.log"

# バックアップ先ディレクトリが存在しない場合は作成する
if (!(Test-Path $BackupDir)) {
    New-Item -ItemType Directory -Path $BackupDir | Out-Null
}

# 日付付きファイル名を生成する
$Date = Get-Date -Format "yyyyMMdd-HH:mm:ss"
$BackupFile = Join-Path $BackupDir "$DistroName-$Date.tar"
```



```
# 開始ログ記録
$LogMessage = "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss') - Backup Start: $BackupFile"
Write-Output $LogMessage
Add-Content -Path $LogFile -Value $LogMessage

# WSL をシャットダウン
wsl --terminate $DistroName

# エクスポート実行
try {
    wsl --export $DistroName $BackupFile
    $LogMessage = "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss') - Success: $BackupFile"
    Write-Output $LogMessage
    Add-Content -Path $LogFile -Value $LogMessage
} catch {
    $LogMessage = "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss') - Fail: $_"
    Write-Output $LogMessage
    Add-Content -Path $LogFile -Value $LogMessage
    exit 1
}

# 終了ログ記録
$LogMessage = "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss') - Backup finished: $BackupFile"
Write-Output $LogMessage
Add-Content -Path $LogFile -Value $LogMessage
```

メモ帳などで上記スクリプトを、`C:\wsl\backup.ps1`として保存してください。なおバックアップ対象のディストリビューション名は、適宜変更してください。

デフォルト状態の Windows11 では、自作の PowerShell スクリプトはセキュリティ上の理由で実行できなくなっています。そのため実行ポリシーを変更します。なおポリシーの変更にはセキュリティリスクが存在することを理解した上で、自己責任の上で実行してください。管理者権限で PowerShell を開き、以下のコマンドを実行します。これは、インターネットからダウンロードしたスクリプトは信頼された発行元による署名が必要で、ローカルで自作したスクリプトは署名なしで実行できるというポリシーになります。

```
$ Set-ExecutionPolicy RemoteSigned
```

コマンドを実行すると、変更の確認が表示されます。`Y`を入力して`Enter`を押してください。

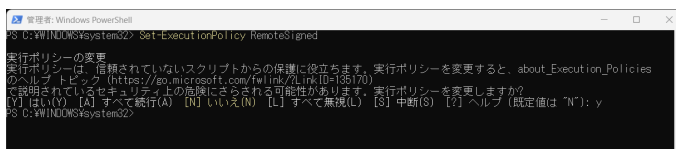


図 21.3: 管理者権限の PowerShell で実行ポリシーを変更する

21.5.2 タスクスケジューラの設定

次に、タスクスケジューラでこのスクリプトを定期実行する設定を行います。Windows キー +Rを押して「ファイル名を指定して実行」を開いてください。そこで`taskschd.msc`と入力して`Enter`を押します。これでタスクスケジューラが起動します。

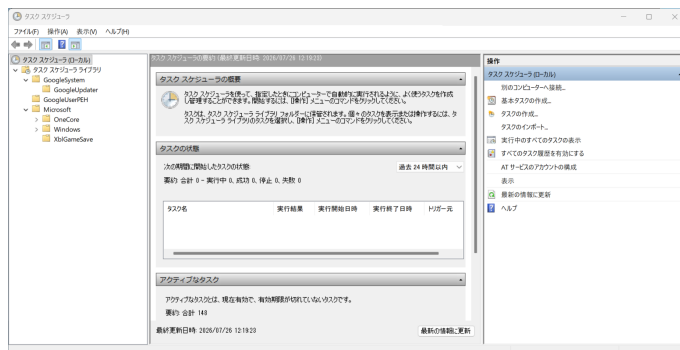


図 21.4: タスクスケジューラの画面

右側の操作パネルからタスクの作成...をクリックします。基本タスクの作成ではなく、詳細設定ができるタスクの作成を選択する点に注意してください。すると以下のダイアログが開きます。まずは全般タブを開き、以下の内容を入力してください。

名前はWSL バックアップとしました。セキュリティオプションは、「ユーザーがログオンしているかどうかにかかわらず実行する」を選択します。また「最上位の特権で実行する」にチェックを入れてください。

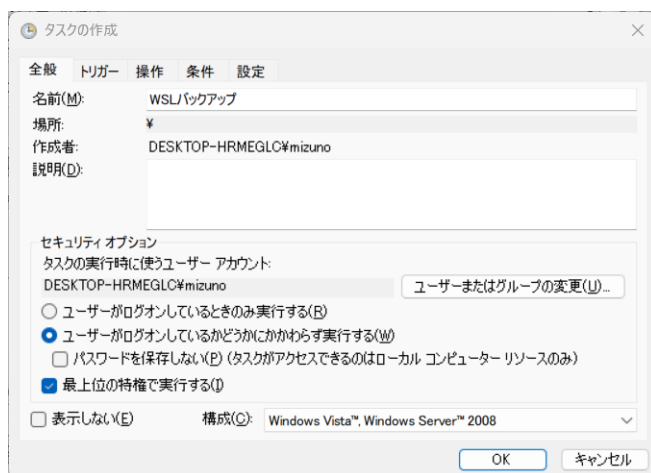


図 21.5: タスクの作成

トリガータブを開き、新規ボタンをクリックします。新しいトリガーのダイアログが開きますので、以下の内容を入力してください。

タスクの開始は、「スケジュールに従う」とします。設定は「毎日」、開始には希望する時刻を入力します。間隔は「1 日」とし、「有効」のチェックボックスにチェックを入れてください。



図 21.6: 新規トリガーの作成

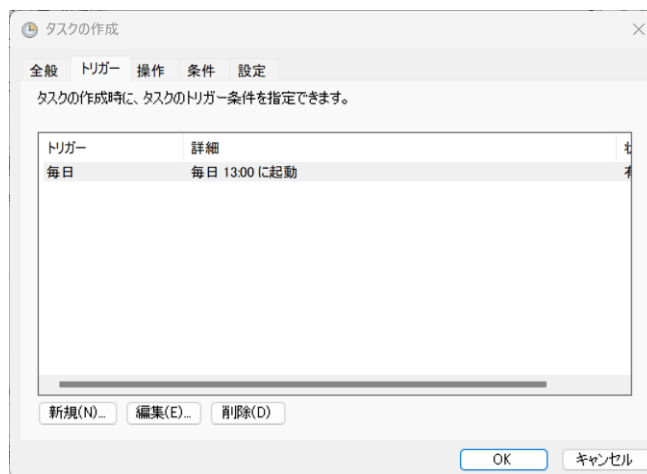


図 21.7: 作成したトリガー

続いて操作タブを開き、新規ボタンをクリックします。新しい操作のダイアログが開きますので、以下の内容を入力してください。

操作は「プログラムの開始」とします。プログラム/スクリプトはpowershell.exe、引数の追加に-File "C:\wsl\backup.ps1"と入力します。

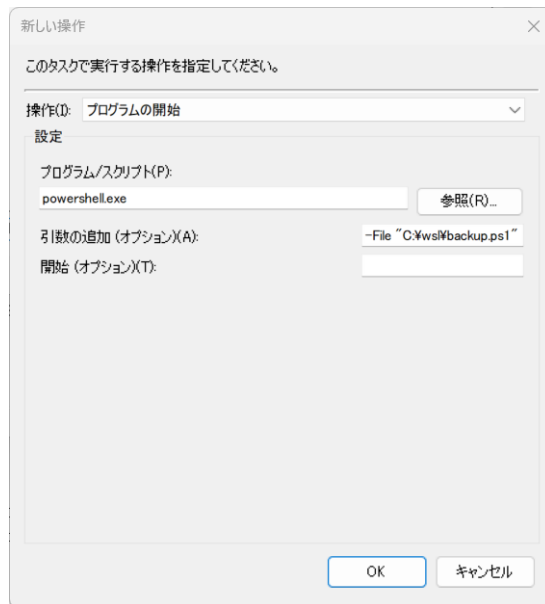


図 21.8: 新しい操作の作成

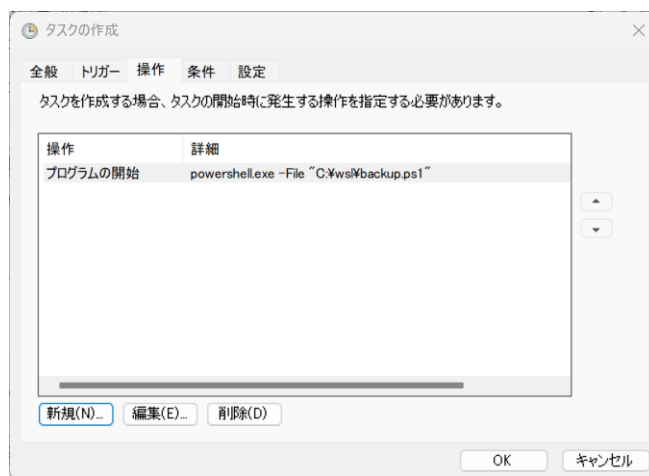


図 21.9: 作成した操作

最後に OK をクリックしてタスクを保存します。タスクを実行するユーザーの確認が表示されるため、そのユーザーのパスワードを入力してください。



図 21.10: ユーザーのパスワードを入力する

設定が完了したら、手動でタスクを実行して動作を確認しておきましょう。左のコンソールツリーからタスクスケジューラライブラリをクリックします。タスクの一覧が表示されますので、作成した WSL バックアップを右クリックし、コンテキストメニューから**実行する**をクリックします。これでタスクが実行されますので、バックアップが作成されることを確認してください。当然ですが、翌日の設定した時刻にも、タスクが起動することを確認しておきましょう。これで毎日、WSL 環境のバックアップを自動で取得できるようになりました^{*1}。



図 21.11: タスクの手動実行

WSL の `--export` と `--import` コマンドを活用すれば、動作中の WSL 環境を簡単に複製できます。一方、既に紹介したカスタムディストリビューションや `cloud-init` は、環境をレシピとして再現可能にする方法です。用途に応じて使い分けると良いでしょう。

定期的なバックアップは、予期しないトラブルからシステムを守る最も確実な方法です。タスクスケジューラを使った自動バックアップを設定しておけば、日々の運用で意識することなく、常に最新の環境が保護されます。ぜひ本章の内容を参考に、安全で効率的な WSL 環境の運用を実現してください。

^{*1} このスクリプトを実行し続けていると、無限にバックアップが増え、ストレージを圧迫してしまいます。実際に運用する場合は、古いバックアップを適度に削除するようなタスクを追加するとよいでしょう。バックアップファイル名を日時ではなく曜日に変更するというのもよい考えです。

第 22 章 Windows ファイルシステムへのアクセスを高速化しよう

WSL には従来の WSL1 と、より新しい WSL2 が存在します。この章ではあえて WSL1 を活用するシーンのひとつである、Windows ファイルシステムへのアクセスについて解説します。

22.1 WSL1 と WSL2 の違い

本書ではこれまで、実際の Linux カーネルを軽量 VM 上で動かす WSL2 を前提に解説してきました。WSL1 は、Windows 上で動作する API 変換レイヤーでした。対して WSL2 は、Hyper-V 上で本物の Linux カーネルを動かしています。これにより WSL2 は WSL1 と比較し、システムコールの高い互換性を実現しています。ところが WSL2 は WSL1 の完全な上位互換というわけではありません。

WSL1 と 2 の違いを簡単にまとめたのが、以下の表です。

表 22.1: WSL1 と 2 の違い

項目	WSL1	WSL2
コア構造	Linux システムコールを Windows の API に変換	Hyper-V 上で Linux カーネルを実行
システムコールの互換性	一部制限あり	ほぼ完全な互換性
ルートファイルシステム	Windows の NTFS 上に独自の変換レイヤーを用意	仮想ディスク上の ext4
Linux ファイルシステムへのアクセス	低速	高速
Windows ファイルシステムへのアクセス	高速	低速
ネットワーク	Windows と同一 IP アドレス、同一ホストとして動作	独立した仮想 NIC による NAT 構成
起動速度	高速	軽量 VM を起動するためやや低速
メモリ使用量	少ない	やや多い
Docker の動作	非対応	対応
systemd の動作	非対応	対応

ネイティブに近い Linux 環境を再現するという点においては、WSL2 が圧倒的に優れています。WSL1 はコアの構造上、どうしてもカーネルに強く依存した機能の動作に制限があります。また Linux プロセスのパフォーマンス面においても、WSL2 に劣ります。

ですが注目すべきポイントもあります。それが、Windows ファイルシステムへのアクセスが高速である点。そしてネットワークをシンプルに保ち、外部からの接続を受け入れやすい点です。Linux には `grep`、`sed`、`awk`、`rsync` といった便利なツールが多数存在します。そのため Windows 上にある大量のファイルを、Linux のツールを使って操作したいというケースもあるでしょう。こうしたケースでは、敢えて WSL1 を選択するアドバンテージがあるのです。

22.2 WSL1 への切り替え方法

インストール済みの WSL2 のディストリビューションを、WSL1 へ切り替える方法を紹介します。まず WSL1 を使うため、**Linux 用 Windows サブシステム**を有効にします。スタートメニューから Windows の機能の有効化または無効化を開いてください。

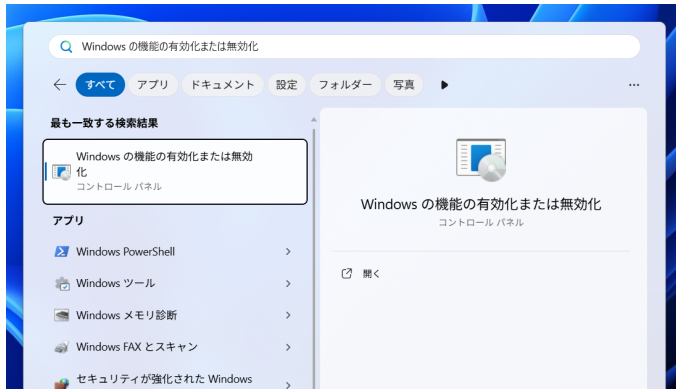


図 22.1: Windows の機能の有効化または無効化を開く

Linux 用 Windows サブシステムにチェックを入れて、OKをクリックします。インストールが完了したら、Windows を再起動してください。

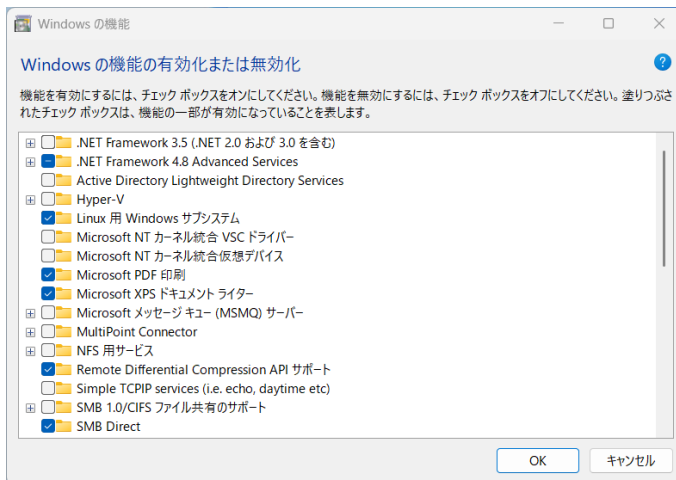
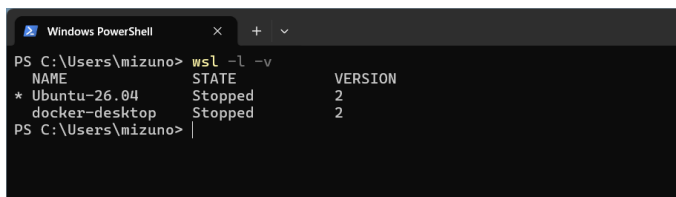


図 22.2: Linux 用 Windows サブシステムにチェックを入れる

インストールされているディストリビューションのバージョンを確認しましょう。PowerShell を起動して、以下のコマンドを実行してください。

```
$ wsl -l -v
```



```

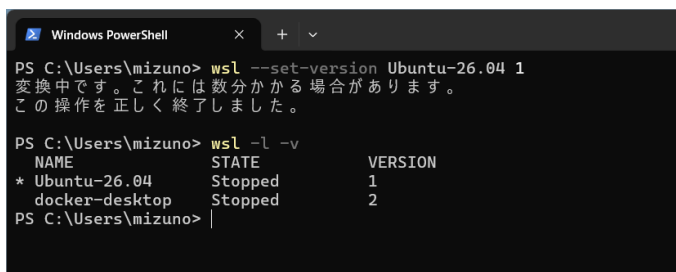
Windows PowerShell
PS C:\Users\mizuno> wsl -l -v
  NAME                STATE      VERSION
* Ubuntu-26.04        Stopped    2
  docker-desktop      Stopped    2
PS C:\Users\mizuno>

```

図 22.3: 現在インストールされている Ubuntu-26.04 は WSL2

以下のコマンドで、WSL2 から WSL1 へ変換できます。

```
$ wsl --set-version Ubuntu-26.04 1
```



```

Windows PowerShell
PS C:\Users\mizuno> wsl --set-version Ubuntu-26.04 1
変換中です。これには数分かかる場合があります。
この操作を正しく終了しました。

PS C:\Users\mizuno> wsl -l -v
  NAME                STATE      VERSION
* Ubuntu-26.04        Stopped    1
  docker-desktop      Stopped    2
PS C:\Users\mizuno>

```

図 22.4: インストール済みのディストリビューションを WSL1 に変換する

22.3 新規インストール時に WSL1 を指定する

ディストリビューションの新規インストール時に、WSL1 を明示的に指定することも可能です。例えば Debian GNU/Linux を WSL1 でインストールしたい場合は、以下のコマンドを実行します。

```
$ wsl --install -d Debian --version 1
```

また以下のコマンドを実行しておくで、バージョンを指定しなかった場合のデフォルトを WSL1 に設定できます。WSL1 の方が使う機会が多いような場合は、設定しておいてもよいでしょう。

```
$ wsl --set-default-version 1
```

22.4 ディスクアクセス速度を計測する

以下のコマンドで Ubuntu 26.04 の、WSL1 と 2 のインスタンスをそれぞれ用意しました。

```

$ wsl --install -d Ubuntu-26.04 --name ubuntu-wsl1 --version 1
$ wsl --install -d Ubuntu-26.04 --name ubuntu-wsl2 --version 2

```

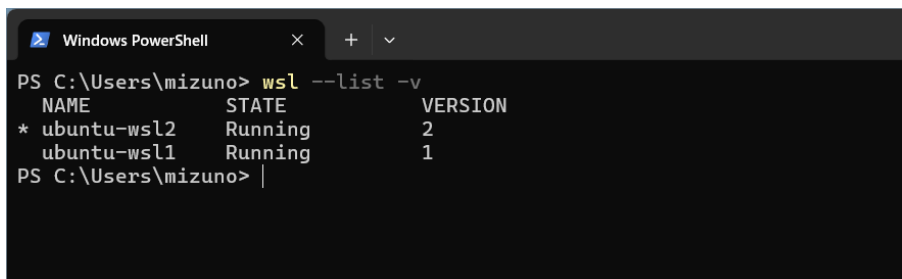



図 22.5: 名前をかえて、同一の Ubuntu 環境を WSL1/2 それぞれ用意した

それぞれの環境から、Windows ファイルシステムに対する読み書きをテストし、そのパフォーマンスを比べてみましょう。ちなみにテストに利用した PC の SSD は、Windows ネイティブ環境で、単一キュー単一スレッドでもシーケンシャル Write 速度がおおよそ 4.1GB/s 程度出ることを確認しています。

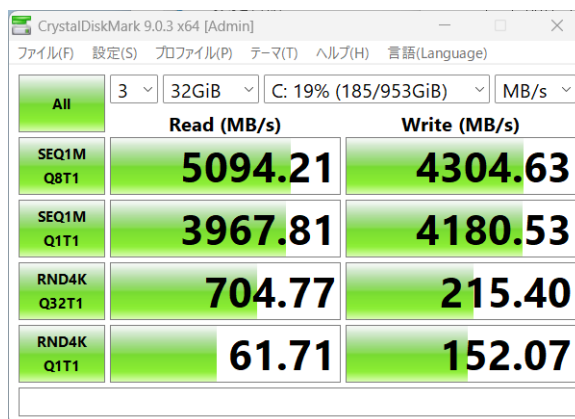


図 22.6: CrystalDiskMark の結果

まず計測コマンドである `fio` パッケージを、それぞれの環境にインストールします。

```
$ sudo apt install -U -y fio
```

続いて以下のコマンドを実行します。これは `fio` コマンドを使い、WSL から Windows 側のファイルシステムへ、32GiB のファイルを 1MiB 単位で同期的に連続書き込みするテストです。

```

$ fio \
--name=wsl-winsfs-seqwrite \
--filename="/mnt/c/Users/<ユーザー名>/testfile" \
--size=32G \
--rw=write \
--bs=1M \
--ioengine=sync \
--iodepth=1 \
--direct=1 \
--fdatasync=1 \
--numjobs=1 \
--group_reporting

```

以下は WSL2 からの書き込みテストの結果です。筆者の環境では約 233 秒かかりました。書き込み速度はおおよそ 140MB/s で、SSD を搭載した今のシステムとしてはあきらかに遅いと言えます。

```

wsl-winsf-seqwrite: {groupid=9, jobid=1} error: pid=1368: Sun Jul 26 13:49:17 2026
write: IOPS=140, BW=140MiB/s (1407MB/s)(32.0GiB/233743msec); 0 zone resets
clat (usec): min=3329, max=13621, avg=4283.34, stdev=371.76
lat (usec): min=3351, max=13634, avg=4293.46, stdev=374.58
clat percentiles (usec):
| 1.00th=[ 3621], 5.00th=[ 3752], 10.00th=[ 3884], 20.00th=[ 4047],
| 30.00th=[ 4146], 40.00th=[ 4228], 50.00th=[ 4228], 60.00th=[ 4293],
| 70.00th=[ 4424], 80.00th=[ 4498], 90.00th=[ 4621], 95.00th=[ 4817],
| 99.00th=[ 5684], 99.50th=[ 5932], 99.90th=[ 6521], 99.95th=[ 7898],
| 99.99th=[16683]
bw ( MiB/s): min=92168, max=161792, per=100.00%, avg=143618.74, stdev=8353.86, samples=467
iops: min= 90, max= 158, avg=148.25, stdev= 8.16, samples=467
lat (msec): min=1.09%, max=22.91%, avg=8.62%, stdev= 2.08%
fsync(fdatasync)/sync_file_range:
sync (usec): min=1544, max=92353, avg=2836.17, stdev=1529.38
sync percentiles (usec):
| 1.00th=[ 1631], 5.00th=[ 1696], 10.00th=[ 1745], 20.00th=[ 1844],
| 30.00th=[ 1958], 40.00th=[ 2147], 50.00th=[ 2587], 60.00th=[ 3228],
| 70.00th=[ 3425], 80.00th=[ 3621], 90.00th=[ 3782], 95.00th=[ 3851],
| 99.00th=[ 7832], 99.50th=[ 9583], 99.90th=[16188], 99.95th=[24249],
| 99.99th=[35914]
cpu: user=0.34%, sys=5.25%, ctx=622689, majf=0, minf=14
IO depths: 1=200.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
submit: 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete: 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=0,32768,0,0 short=32767,0,0 dropped=0,0,0
latency: target=0, window=0, percentile=100.00%, depth=1
Run status group 0 (all jobs):
WRITE: bw=140MiB/s (1407MB/s), 140MiB/s-140MiB/s (1407MB/s-1407MB/s), io=32.0GiB (34.4GB), run=233743-233743msec
wslun0@DESKTOP-HRMEGLC:~$

```

図 22.7: WSL2 での書き込みテスト

続いて、同じコマンドを WSL1 で実行してみました。その結果は約 17 秒で、書き込み速度はおおよそ 1.9GB/s で、WSL2 の 13 倍以上高速です。

```

wsl-winsf-seqwrite: {groupid=9, jobid=1} error: pid=1368: Sun Jul 26 13:49:17 2026
write: IOPS=1889, BW=1890MiB/s (1982MB/s)(32.0GiB/17338msec); 0 zone resets
clat (usec): min=158, max=1959, avg=223.23, stdev=79.95
lat (usec): min=161, max=1963, avg=227.02, stdev=81.08
clat percentiles (usec):
| 1.00th=[ 163], 5.00th=[ 167], 10.00th=[ 172], 20.00th=[ 182],
| 30.00th=[ 192], 40.00th=[ 196], 50.00th=[ 200], 60.00th=[ 206],
| 70.00th=[ 227], 80.00th=[ 262], 90.00th=[ 289], 95.00th=[ 318],
| 99.00th=[ 494], 99.50th=[ 652], 99.90th=[ 1237], 99.95th=[ 1319],
| 99.99th=[ 1778]
bw ( MiB/s): min=1402, max=2167, per=99.86%, avg=1887.35, stdev=208.25, samples=34
iops: min=1402, max=2167, avg=1887.12, stdev=208.22, samples=34
lat (msec): min=0.7781%, max=22.81%, avg=8.64%, stdev= 2.08%
fsync(fdatasync)/sync_file_range:
sync (usec): min=235, max=56923, avg=380.94, stdev=813.97
sync percentiles (usec):
| 1.00th=[ 245], 5.00th=[ 245], 10.00th=[ 247], 20.00th=[ 247],
| 30.00th=[ 249], 40.00th=[ 249], 50.00th=[ 251], 60.00th=[ 253],
| 70.00th=[ 255], 80.00th=[ 260], 90.00th=[ 269], 95.00th=[ 293],
| 99.00th=[ 489], 99.50th=[ 1004], 99.90th=[ 8029], 99.95th=[15533],
| 99.99th=[38540]
cpu: user=1.17%, sys=57.05%, ctx=0, majf=0, minf=39
IO depths: 1=200.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
submit: 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete: 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=0,32768,0,0 short=32767,0,0 dropped=0,0,0
latency: target=0, window=0, percentile=100.00%, depth=1
Run status group 0 (all jobs):
WRITE: bw=1890MiB/s (1982MB/s), 1890MiB/s-1890MiB/s (1982MB/s-1982MB/s), io=32.0GiB (34.4GB), run=17338-17338msec
wslun0@DESKTOP-HRMEGLC:~$

```

図 22.8: WSL1 での書き込みテスト

読み込みのテストも行ってみましょう。先ほど書き込んだファイルを、以下の `fiio` コマンドで読み込んでみます。

```

$ fio \
--name=wsl-winsf-seqwrite \
--filename="/mnt/c/Users/<ユーザー名>/testfile" \
--size=32G \
--rw=read \
--bs=1M \
--ioengine=sync \
--iodepth=1 \
--direct=1 \
--numjobs=1 \
--group_reporting

```

こちらが WSL2 の実行結果です。書き込みよりは高速であるものの、おおよそ 200MB/s 強の速度しか出せず、32GiB の読み込みに 162 秒かかりました。

```

mizuno@DESKTOP-HRMGGLC x mizuno@DESKTOP-HRMGGLC x + v
--direct=1 \
--numjobs=1 \
--group_reporting
wsl-wslfs-seqwrite: (g=0): rw=read, bs=(R) 1024KiB-1024KiB, (W) 1024KiB-1024KiB, (T) 1024KiB-1024KiB, ioengine=sync, iod
eph=1
fio-3.41
Starting 1 process
Jobs: 1 (f=1): [R(1)][100.0%][r=221MiB/s][w=221 IOPS][eta 00m:00s]
wsl-wslfs-seqwrite: (groupid=0, jobs=1): err=0: pid=1425: Sun Jul 26 14:06:20 2026
read: IOPS=261, BW=201MiB/s (211MB/s)(32.0GiB/162824msec)
clat (usec): min=462, max=16018, avg=1067.10, stdev=1331.21
lat (usec): min=4602, max=16018, avg=4907.20, stdev=1381.21
clat percentiles (usec):
| 1.00th=[ 416], 5.00th=[ 4228], 10.00th=[ 4293], 20.00th=[ 4359],
| 30.00th=[ 4424], 40.00th=[ 4490], 50.00th=[ 4555], 60.00th=[ 4621],
| 70.00th=[ 4752], 80.00th=[ 5080], 90.00th=[ 5880], 95.00th=[ 6325],
| 99.00th=[1460], 99.50th=[1383], 99.90th=[14740], 99.95th=[15000],
| 99.99th=[15270]
bw ( KiB/s): min=129024, max=237668, per=100.00%, avg=206130.83, stdev=23296.84, samples=325
iops : min= 126, max= 232, avg=201.32, stdev=22.75, samples=325
lat (msec): min=96.55%, max=3.45%
cpu : user=0.65%, sys=0.91%, ctx=557067, majf=0, minf=263
IO depths : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=32768,0,0,0 short=0,0,0 dropped=0,0,0
latency : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
READ: bw=201MiB/s (211MB/s), 201MiB/s-201MiB/s (211MB/s-211MB/s), io=32.0GiB (34.4GB), run=162824-162824msec
mizuno@DESKTOP-HRMGGLC ~$

```

図 22.9: WSL2 での読み込みテスト

対して WSL1 では、2.6GB/s の速度が出ており、約 12 秒で完了しています。

```

mizuno@DESKTOP-HRMGGLC x mizuno@DESKTOP-HRMGGLC x + v
--numjobs=1 \
--group_reporting
wsl-wslfs-seqwrite: (g=0): rw=read, bs=(R) 1024KiB-1024KiB, (W) 1024KiB-1024KiB, (T) 1024KiB-1024KiB, ioengine=sync, iod
eph=1
fio-3.41
Starting 1 process
Jobs: 1 (f=1): [R(1)][100.0%][r=2493MiB/s][w=2493 IOPS][eta 00m:00s]
wsl-wslfs-seqwrite: (groupid=0, jobs=1): err=0: pid=752: Sun Jul 26 14:09:12 2026
read: IOPS=2693, BW=2693MiB/s (2730MB/s)(32.0GiB/12587msec)
clat (usec): min=58, max=10607, avg=382.97, stdev=678.56
lat (usec): min=58, max=10607, avg=383.09, stdev=678.55
clat percentiles (usec):
| 1.00th=[ 70], 5.00th=[ 73], 10.00th=[ 75], 20.00th=[ 78],
| 30.00th=[ 85], 40.00th=[ 117], 50.00th=[ 122], 60.00th=[ 141],
| 70.00th=[ 184], 80.00th=[ 465], 90.00th=[ 1434], 95.00th=[ 1713],
| 99.00th=[ 1909], 99.50th=[ 1991], 99.90th=[ 9503], 99.95th=[ 9896],
| 99.99th=[10290]
bw ( MiB/s): min= 2028, max= 2864, per=100.00%, avg=2693.91, stdev=245.82, samples=25
iops : min= 2028, max= 2864, avg=2693.64, stdev=245.71, samples=25
lat (usec): min=34.45%, max=39.94%, 50th=6.60%, 75th=3.12%, 100th=1.24%
lat (msec): min=14.10%, max=0.23%, 10th=0.20%, 20th=0.04%
cpu : user=0.37%, sys=0.62%, ctx=0, majf=0, minf=294
IO depths : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=32768,0,0,0 short=0,0,0 dropped=0,0,0
latency : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
READ: bw=2693MiB/s (2730MB/s), 2693MiB/s-2693MiB/s (2730MB/s-2730MB/s), io=32.0GiB (34.4GB), run=12587-12587msec
mizuno@DESKTOP-HRMGGLC ~$

```

図 22.10: WSL1 での読み込みテスト

WSL2 から Windows ファイルシステムへのアクセスが遅い主な原因は、WSL2 が軽量の仮想マシン内で動作しており、Linux カーネルが Windows の NTFS を直接操作しているわけではないためです。

/mnt/c以下は、Windows ホスト側の NTFS を WSL2 の仮想マシン内へ共有ファイルシステムとして公開したものです。そのアクセスには **9P (Plan 9 Filesystem Protocol)** を基礎とする仕組みが使われます。ファイルを読み書きするたびに、Linux カーネル内だけで処理が完結するのではなく、要求を仮想マシンの境界を越えて Windows 側へ渡し、その処理結果が返ってくるのを待つ必要があります。

さらに Linux と Windows では、ファイルの属性、アクセス権、シンボリックリンク、ファイルロックなどの扱いが異なるため、両者のファイルシステム仕様を変換する処理も必要です。こうした I/O 要求ごとの固定コストがボトルネックとなるため、高速な SSD を搭載していても、/mnt/cへのアクセスでは SSD 本来の性能を十分に引き出せないことがあります。特に、小さなファイルを大量に扱う処理や、ファイルの属性を頻繁に参照する処理では、性能低下が顕著になります。

これに対して WSL1 は、独立した Linux カーネルを持たず、Linux プログラムが発行したシステムコールを Windows 側の処理へ変換する方式です。/mnt/cへのアクセスも、WSL2 のように仮想マシン内の Linux から共有ファイルシステムを経由する必要がありません。そのため Windows ファイルシステム上のファイルを扱う処理では、WSL 2 よりも高速になる場合があります。

このように Windows 上のファイルを直接扱うスクリプトや、大量のファイルを検索、変換、生成する処理を WSL2 で実行すると、大きな性能上のペナルティが生じることがあります。WSL2 を利用する場合は、Linux のツールでおもに扱うファイルを /homeなどの Linux ファイルシステム側へ置くのが

基本です。一方、Windows アプリケーションと Linux ツールの双方から同じファイルを頻繁に扱う必要があり、ファイルを Windows ファイルシステム上に置かなければならない用途では、あえて WSL1 を選択することも検討してみてください。

WSL2 のメリットは、実際の Linux カーネルによる高い互換性を備え、Linux ファイルシステム上でのファイル操作にも優れている点です。WSL1 と WSL2 は単純な新旧関係ではなく、それぞれ得意な処理が異なります。ファイルの置き場所や実行する処理の性質を踏まえ、用途に応じて使い分けることが重要です。

第 23 章 WSL 上のサーバーを公開してみよう

WSL 上で開発しているアプリに対し、外部からアクセスさせたいこともあるでしょう。この章では WSL1/2 それぞれにおいて、外部からネットワークアクセスする方法を紹介します。

23.1 WSL1 と 2 におけるネットワークアーキテクチャの違い

既に解説した通り、WSL2 は Hyper-V の軽量仮想マシン上で本物の Linux カーネルを動かしています。これによって非常に高いシステムコール互換性を実現しているわけです。ところが実体が仮想マシンであるため、ホストの Windows とは別マシンになってしまうという問題があります。これはネットワークについても例外ではなく、WSL2 は Hyper-V の仮想スイッチによる独立した仮想ネットワークを持っており、外部ネットワークには NAT を経由して出て行きます。家庭内 LAN に所属する PC が、ルーターを経由してインターネットに出ていくのと、構成的には同じです。そのためインターネットから家庭内の PC に直接アクセスができないのと同様に、WSL2 内で立ち上げたテスト環境にも、LAN 内の別の PC から直接接続することはできません。

対して WSL1 はシステムコールの変換レイヤーであるため、Linux のプロセスを Windows 上で直接動作させています。ネットワークも Windows の TCP/IP スタックを直接利用しており、Windows と IP アドレスも共有しているのです。そのため WSL1 を使えば、Windows Defender ファイアウォールの設定のみで、外部から WSL 環境へのアクセスを実現できます。

23.2 WSL1 環境の用意

まずはシンプルな、WSL1 を使った方法を紹介します。以下のコマンドでまっさらな、WSL1 の Ubuntu 26.04 環境を用意してください。

```
$ wsl --install -d Ubuntu-26.04 --name ubuntu-wsl1 --version 1
```

23.3 Streamlit のインストール

今回は Web アプリ開発の例として、Python ベースのフレームワーク Streamlit^{*1}を使って解説します。WSL1 の Ubuntu 26.04 に、以下のコマンドで Streamlit をインストールしてください。なおここでは、後述のスクリプトで利用する `psutil` と `plotly` というライブラリも同時にインストールしています。

```
$ sudo apt install -U -y python3-venv
$ mkdir ~/work && cd ~/work
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip install streamlit psutil plotly
```

23.4 Streamlit で Web アプリを立ち上げる

作成した `~/work` ディレクトリ内に、以下の Python スクリプトを `dashboard.py` という名前で作成してください。

^{*1} <https://streamlit.io/>

```

import streamlit as st
import psutil
import pandas as pd
import plotly.graph_objects as go
from datetime import datetime
import time

st.set_page_config(page_title="システムモニター", layout="wide")
st.title("WSL システムモニタリング")
st.markdown("---")
auto_refresh = st.sidebar.checkbox("自動更新", value=True)
refresh_interval = st.sidebar.slider("更新間隔 (秒)", 1, 10, 2)

col1, col2, col3, col4 = st.columns(4)

cpu_percent = psutil.cpu_percent(interval=1)
col1.metric(
    label="CPU 使用率",
    value=f"{cpu_percent}%",
    delta=f"{cpu_percent - 50}%" if cpu_percent > 50 else None
)

memory = psutil.virtual_memory()
col2.metric(
    label="メモリ使用率",
    value=f"{memory.percent}%",
    delta=f"{memory.used / (1024**3):.1f}GB 使用中"
)

disk = psutil.disk_usage('/')
col3.metric(
    label="ディスク使用率",
    value=f"{disk.percent}%",
    delta=f"{disk.used / (1024**3):.1f}GB 使用中"
)

col4.metric(label="プロセス数", value=len(psutil.pids()))
st.markdown("---")
col_left, col_right = st.columns(2)

with col_left:
    st.subheader("CPU コア別使用率")
    cpu_per_core = psutil.cpu_percent(percpu=True)

    fig_cpu = go.Figure(data=[
        go.Bar(
            x=[f"Core {i}" for i in range(len(cpu_per_core))],
            y=cpu_per_core,
            marker_color='lightblue'
        )
    ])
    fig_cpu.update_layout(
        yaxis_title="使用率 (%)",
        height=300,
        showlegend=False
    )

```

```

    st.plotly_chart(fig_cpu, use_container_width=True)

with col_right:
    st.subheader("メモリ使用状況")
    fig_mem = go.Figure(data=[
        go.Pie(
            labels=['使用中', '利用可能'],
            values=[memory.used, memory.available],
            hole=0.4,
            marker_colors=['#ff6b6b', '#51cf66']
        )
    ])
    fig_mem.update_layout(height=300)
    st.plotly_chart(fig_mem, use_container_width=True)

st.subheader("ネットワーク統計")
net_io = psutil.net_io_counters()

net_col1, net_col2, net_col3, net_col4 = st.columns(4)
net_col1.metric("送信", f"{net_io.bytes_sent / (1024**2):.2f} MB")
net_col2.metric("受信", f"{net_io.bytes_recv / (1024**2):.2f} MB")
net_col3.metric("送信バケット", f"{net_io.packets_sent:,}")
net_col4.metric("受信バケット", f"{net_io.packets_recv:,}")

st.subheader("CPU 使用率上位プロセス")
processes = []
for proc in psutil.process_iter(['pid', 'name', 'cpu_percent', 'memory_percent']):
    try:
        processes.append(proc.info)
    except (psutil.NoSuchProcess, psutil.AccessDenied):
        pass

df = pd.DataFrame(processes)
df = df.sort_values('cpu_percent', ascending=False).head(10)
df['memory_percent'] = df['memory_percent'].round(2)
df['cpu_percent'] = df['cpu_percent'].round(2)

st.dataframe(
    df,
    column_config={
        "pid": "PID",
        "name": "プロセス名",
        "cpu_percent": "CPU (%)",
        "memory_percent": "メモリ (%)"
    },
    hide_index=True,
    use_container_width=True
)

if auto_refresh:
    time.sleep(refresh_interval)
    st.rerun()

```

以下のコマンドでアプリを起動します。

```
$ cd ~/work  
$ streamlit run dashboard.py
```

初回起動時、アプリへのネットワークアクセスを許可してよいかを尋ねてきます。ここで許可をクリックすると、Python が待ち受けているポートに対してアクセスを許可するルールが、**Windows Defender ファイアウォール**に追加され、外部からのアクセスが可能になります。



図 23.1: Python へのネットワークアクセスを許可する

起動が完了するとターミナル内に、アプリへのアクセスに使用する IP アドレスとポートが表示されます。2 番目に表示されているのが、Windows のプライベート IP アドレスです。Web ブラウザから、このアドレスとポートへアクセスしてみましょう。

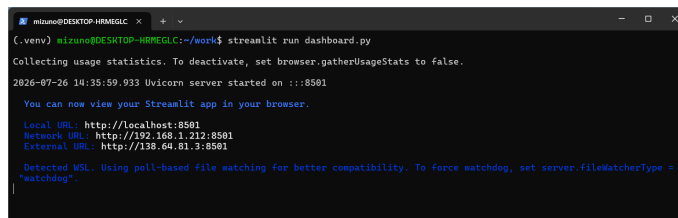


図 23.2: ここでは 192.168.1.212 が Windows の IP アドレスになる



図 23.3: ブラウザに Streamlit で作成したダッシュボードが表示された

23.5 初回起動時にアクセスを許可しなかった場合の対処

初回起動時のダイアログでキャンセルを選択すると、Windows Defender ファイアウォールに、明示的にアクセスをブロックするルールが追加されています。うっかりこの状態になってしまったら、一度ルールを削除しない限り、アクセスを許可できなくなります。コントロールパネル→システムとセキュリティ→Windows Defender ファイアウォールを開き、左ペインにある詳細設定をクリックします。

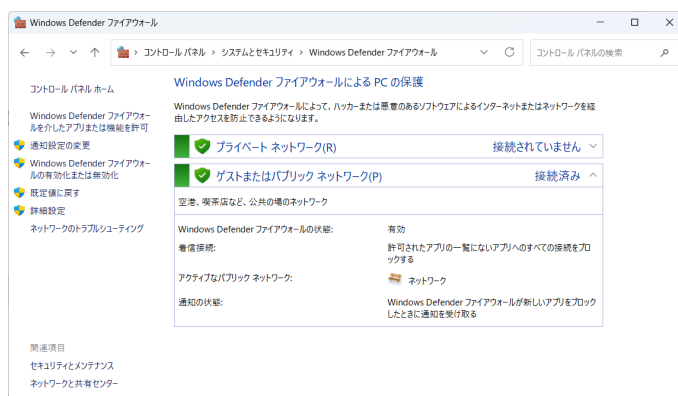


図 23.4: Windows Defender ファイアウォールのルールを確認する

受信の規則をクリックすると、ファイアウォールのルール一覧が表示されます。その中に Python をブロックするルールが 2 つ登録されていますので、これを削除してください。

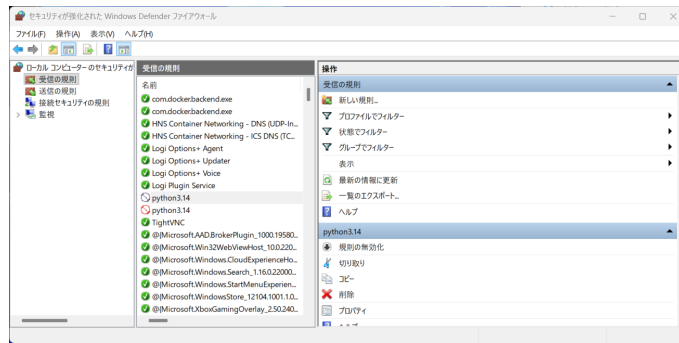


図 23.5: python がブロックされている状態

この状態でもう一度 Streamlit を起動すれば、再度許可を求めるダイアログが表示されます。

23.6 WSL2 に外部からアクセスする方法

23.6.1 ポート転送でアクセスする

ここまで説明したように、WSL1 は Windows とネットワークを共有しているため、Windows 側でファイアウォールにルールを追加するだけで、WSL 上のプロセスにアクセスできました。ですが WSL2 のネットワークは、ホストの Windows とは異なる仮想ネットワークです。そのため WSL2 で同様に外部からのアクセスを可能にするには、別途ポート転送の設定が必要となります。

PowerShell から以下のコマンドを実行し、WSL2 に割り当てられている IP アドレスを調べてください。この例では `172.30.170.109` が該当のアドレスになります。

```
$ wsl -d (WSL2 のディストリビューション名) hostname -I
172.30.170.109
```

続いて管理者権限で新しい PowerShell を起動し、以下のコマンドを実行してください (`connectaddress` に指定する IP アドレスは、上記のコマンドで調べたものに読み替えてください)。これで Windows のポート 8501 番へのアクセスが、WSL2 に転送されるようになります。

```
$ netsh interface portproxy add v4tov4 listenport=8501 listenaddress=0.0.0.0 connectport=8501 connectaddress=172.22.112.136
```

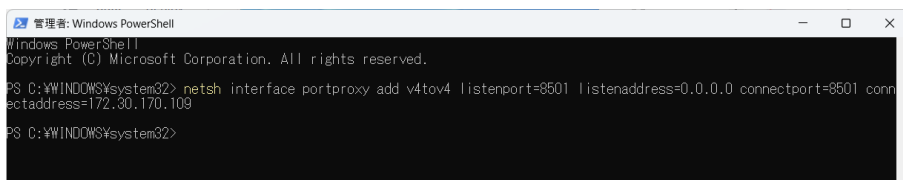


図 23.6: 管理者権限の PowerShell でポート転送を設定する

これで Windows の 8501 番ポートへのアクセスが WSL2 に転送されるようになるため、外部からアクセスが可能になります。

ただしこの方法には問題もあります。それは WSL2 の IP アドレスは再起動などによって動的に変化するという点です。netsh によるポート転送は、ターゲットの IP アドレスが変化すると機能しなくな

ります。そのため都度、ポート転送の設定をやり直す必要があるのです。恒久的に利用するには向きません。

23.6.2 ミラーモードでアクセスする

これを解決する手段が、ホストである Windows の NIC を WSL2 にミラーリングする**ミラーモード**です。この機能を有効にすると、WSL2 と Windows で同じ IP アドレスが使えます。もし前述のポート転送を設定済みの場合は、管理者権限の PowerShell で以下のコマンドを実行し、ルールをすべて削除しておいてください。

```
$ netsh interface portproxy reset
```

ミラーモードを有効にするには、スタートメニューから**WSL Settings**を起動してください。**ネットワーク**→**ネットワークモード**を、既定の**Nat**から**Mirrored**に変更します。



図 23.7: ネットワークモードを Mirrored に変更する

変更の適用をクリックすると、WSL のシャットダウンが必要だというダイアログが表示されます。**WSL を今すぐシャットダウン**をクリックして、WSL を再起動してください。



図 23.8: WSL を一度シャットダウンする

再度一度 WSL2 の IP アドレスを確認してみると、ホストの Windows と同一の IP アドレスが割り当てられていることがわかります。

```
$ wsl -d (WSL2 のディストリビューション名) hostname -I  
192.168.1.212
```

ミラーモードは、WSL2 のネットワークが WSL1 のように、Windows と統合されます。そのためサービスに外部からアクセスさせるには、Windows Defender ファイアウォールの設定が別途必要となる点に注意してください。Streamlit にアクセスさせるには、管理者権限の PowerShell で以下のコマンドを実行し、8501 番ポートへのアクセスを許可する必要があります。

```
$ New-NetFirewallRule -DisplayName "Streamlit" -Direction Inbound -LocalPort 8501 -  
-Protocol TCP -Action Allow
```

以後は WSL2 であっても、Windows の IP アドレスで、Streamlit にアクセスできるようになります。それだけならば WSL1 と同じに過ぎませんが、WSL2 は systemd が使えます。つまり複数のデーモンを動かし続けるような用途では、WSL2+ ミラーモードの方が便利でしょう。例えば WSL2 内で SSH サーバーを起動し、以下のコマンドで 22 番ポートへのアクセスを許可すれば、別の PC から SSH で WSL を操作できます。

```
$ New-NetFirewallRule -DisplayName "SSH" -Direction Inbound -LocalPort 22 -Protocol TCP -  
-Action Allow
```

このように、WSL は 1 でも 2 でも、外部からアクセス可能なサービスを動かします。ただし WSL2 は、ミラーモードを使わない場合は少々設定が面倒です。またミラーモードはすべての WSL2 ディストリビューションに設定が反映されてしまうという問題もあります。ネットワークの用途によっても、WSL のバージョンの使い分けを検討してみてください。

第 24 章 WSL におけるデータレスキュー方法を知ろう

WSL のディストリビューションが、うまく起動しなくなってしまうこともあるでしょう。この章ではそうした場合のデータ救出方法を解説します。

24.1 WSL にログイン不能になる時とは

PC を使っていると、どうしても様々なトラブルに遭遇します。OS が起動しなくなったり、設定ミスによってログインができなくなったり、そんなトラブルを経験したことのある方も多いのではないのでしょうか。WSL2 も Linux という OS を動かしている仮想マシンですから、こうしたトラブルとは無縁ではありません。トラブルが起きた際も、元通り復旧できれば理想的です。ですがどうしようもないケースも存在します。そんな最悪の場合でも、バックアップされていないデータだけは救出したいでしょう。

WSL に正しくログインできなくなる原因はいくつか考えられます。今回はそのうち、よくありがちな以下の 2 つについて考えてみました。

1. WSL2 は正しく起動しているが、ユーザーレベルの設定の問題でログインできない状態
2. 重要なシステムファイルの破損などが原因で、WSL2 の起動そのものが失敗する状態

以下ではそれぞれのパターンについて、対応方法を紹介します。

24.1.1 実施にあたっての注意

本章では解説の都合上、WSL2 のシステムを意図的に破壊します。そのため記事の内容を試す場合は、必ず試験用の環境を作ってから実施してください。実際に運用している環境では、絶対に行わないでください。

24.2 デフォルトユーザーでログインできない時は

WSL では初回起動時にユーザーを作成しますが、設定ミスで、このユーザーでログインできなくなった状況を考えてみましょう。例えばログインシェルが起動しなくなってしまった状態などです。

Ubuntu をはじめとする Linux では、ログインシェルをユーザーが自由に選択できます。Ubuntu のデフォルトのシェルは `bash` ですが、これを `zsh` や `tcsh`、`fish` などに変更できるわけです。各ユーザーのログインシェルは、`/etc/passwd` ファイルに記述されています。`/etc/passwd` は、1 行につき 1 ユーザーの設定が記述されています。行はコロンで区切られた 7 つのフィールドで構成されており、最初のフィールドがユーザー名、最後のフィールドがログインシェルです。例えば以下の例では、`mizuno` ユーザーのログインシェルは `/bin/bash` であることがわかります。

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
(... 略 ...)
mizuno:x:1000:1000:,,,:/home/mizuno:/bin/bash
```

`/etc/passwd` を眺めていると、ログインシェルに `/usr/sbin/nologin` という、見慣れないコマンドが指定されているユーザーがいることに気づくでしょう。`nologin` はそのユーザーアカウントが使用できない旨のメッセージを表示し、終了するだけのコマンドです。つまりこのコマンドがログインシェル

に指定されているユーザーは、`bash`のような対話的シェルが起動しないため、ログインできなくなるというわけです。

こんなコマンドが何の役に立つのかと思うかもしれませんが、これは主にセキュリティ確保のための措置です。Linux には、バックグラウンドでプログラムを起動するためだけのユーザーというものが多く存在します。こうしたユーザーは人間が対話的な操作を行わないため、直接ログインする必要がないためです。仮にこうしたユーザーがログイン可能に設定されていると、もしもアカウントを乗っ取られた際、攻撃者に攻撃の手段を与えることになってしまいます。

それではサービス起動用のユーザーのシェルを変更しようとしたら、うっかり別のユーザーを変更してしまい、ログイン不能になってしまったというシナリオを想定して、自身のユーザーのログインシェルを `/usr/sbin/nologin` に変更してみましょう。ログインシェルは以下のコマンドで変更できます。

```
$ sudo usermod -s 指定するシェル ユーザー名
```

`mizuno` ユーザーのログインシェルを `/usr/sbin/nologin` にする具体的なコマンドは以下の通りです。

```
$ sudo usermod -s /usr/sbin/nologin mizuno
```

コマンドを実行したら、一度 WSL からログアウトしてから、再度ログインを試みてみましょう。ログインシェルとして「`/usr/sbin/nologin`」が起動されるため、所定のメッセージが表示され、ただちに終了してしまいます。いわゆる「やらかしてしまった」状態ですね。

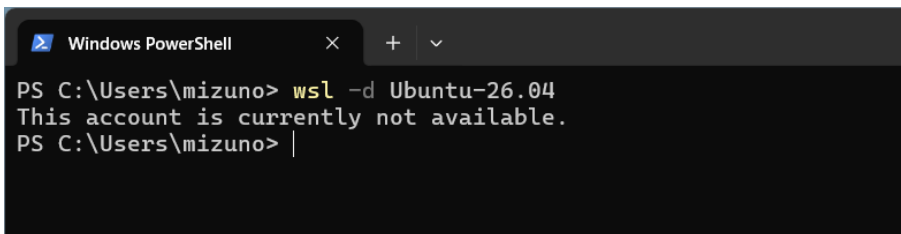
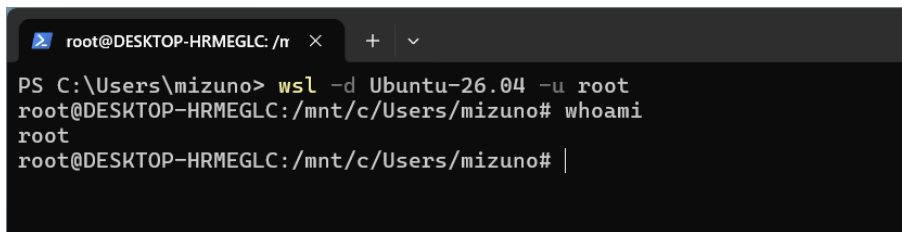


図 24.1: `/bin/bash` が起動しないため、WSL にログインできない状態

ところが、これはそれほど大した問題ではありません。というのも、特定のユーザーの設定がおかしくなっているだけであれば、別ユーザーでログインすることでバイパス可能だからです。WSL は起動時に `-u` オプションで、ログインするユーザーを指定できます。例えば以下のコマンドを実行すれば、`root` ユーザーとして `Ubuntu-26.04` にログインできるのです。

```
$ wsl -d Ubuntu-26.04 -u root
```



```
root@DESKTOP-HRMEGLC: /mnt/c/Users/mizuno#  
PS C:\Users\mizuno> wsl -d Ubuntu-26.04 -u root  
root@DESKTOP-HRMEGLC:/mnt/c/Users/mizuno# whoami  
root  
root@DESKTOP-HRMEGLC:/mnt/c/Users/mizuno# |
```

図 24.2: root ユーザーで WSL にログインした例

root のシェルが取れば、復旧は簡単です。以下のコマンドを実行して、ユーザーのログインシェルをデフォルトの `bash` に戻してあげましょう。なおここでは `root` として直接ログインしているため、`sudo` は不要になります。

```
# usermod -s /bin/bash mizuno
```

今回はシェルの変更を例に挙げましたが、それ以外のトラブルにも対応できます。例えばユーザーのホームディレクトリの中身を削除して、初期状態に戻してあげるなどです。Ubuntu において、日常的に `root` アカウントを直接使用することは推奨されていません。ですが緊急時の伝家の宝刀として、`root` で直接ログインすれば、ユーザーレベルの設定に起因する問題ば解決できることは覚えておきましょう。

24.3 ディスクイメージを別環境にマウントする

前述の通り、WSL が起動さえしているのであれば、なんとでもやりようはあります。本格的にまずいのは、WSL そのものが起動しない状態です。具体的にはディスク障害や、重要なシステムファイルが欠損したなどの理由で、OS そのものが起動に失敗してしまうケースです。それでは `sudo rm -rf /usr` コマンドをうっかり実行して、OS の起動に必要なファイルをディレクトリごと消し飛ばして、WSL を起動不能にしてみます。

繰り返しになりますが、以下の手順では WSL2 のシステムを破壊します。ユーザーが作成したデータは救出可能ですが、環境そのものは破棄して作り直す以外にありません。また WSL では、`/mnt` 以下にホストである Windows のドライブがマウントされています。誤ったファイルを削除すると、ホストに影響が出ることもあります。実際に運用している環境では、本章の内容は絶対に行わないでください。

```
$ sudo rm -rf /usr
```

```

rm: cannot remove '/usr/lib/wsl/drivers/wvmic_timesync.inf_and64_a1aaae70047b4c60/wvmic_timesync.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/wvmic_vss.inf_and64_aae11b51bd5a1c65/wvmic_vss.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/wvms_mp_windows.inf_and64_ff9a4204bc669b25/wvms_mp_windows.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/wvms_pp.inf_and64_bfcd0b01c43ee91/wvms_pp.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/wvms_vsft.inf_and64_bf03868e5ea7bcdd/wvms_vsft.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/wvms_vssp.inf_and64_ca9d783358d61f7d/wvms_vssp.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/wvpcl.inf_and64_4c62ba1a38ecd6e8/vpcl.sys': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/wvpcl.inf_and64_4c62ba1a38ecd6e8/vpcl.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/wvpclvsp.inf_and64_1836471f9e0b1b11/vpclvsp.sys': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/wvpclvsp.inf_and64_1836471f9e0b1b11/vpclvsp.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/xboxgip.inf_and64_72d29a62cc4c6f6f/devauth.sys': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/xboxgip.inf_and64_72d29a62cc4c6f6f/xboxgip.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/xboxgip.inf_and64_72d29a62cc4c6f6f/xboxgip.sys': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/xboxgipsynthetic.inf_and64_c64c355119650251/xboxgipsynthetic.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/xinputhid.inf_and64_842a5b4e321d8ac6/xinputhid.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/xinputhid.inf_and64_842a5b4e321d8ac6/xinputhid.sys': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/xusb22.inf_and64_6629bfd686c5bd4/xusb22.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/xusb22.inf_and64_6629bfd686c5bd4/xusb22.sys': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/yklnx64.inf_and64_5f83c04f4adb7630/yklnx64.inf': Read-only file system
rm: cannot remove '/usr/lib/wsl/drivers/yklnx64.inf_and64_5f83c04f4adb7630/yklnx64.sys': Read-only file system
rm: cannot remove '/usr/lib/wsl/lib': Device or resource busy
sed: command not found
mizuno@DESKTOP-HRMEGL: $

```

図 24.3: /usr 以下をまるごと削除してみた

削除が完了したら、一度 WSL からログアウトして、もう一度ログインしてみましょう。ログインシェルである `bash` も削除されているため、ログインに失敗することがわかります。

```

<3>WSL (1240 - Relay) ERROR: CreateProcessCommon:818: execvp(bin/bash) failed: No such file or directory
[プロセスはコード 1 (0x00000001) で終了しました]
このターミナルを Ctrl+D で閉じるか、Enter キーを押して再起動できます。

```

図 24.4: bash が存在しないというエラーが出る

PowerShell から一度 WSL をシャットダウンし、再起動してみましょう。致命的なエラーで起動に失敗します。

```
$ wsl --shutdown
$ wsl -d Ubuntu-26.04
```

```

PS C:\Users\mizuno> wsl --shutdown
PS C:\Users\mizuno> wsl -d Ubuntu-26.04
致命的なエラーです。
エラー コード: Wsl/Service/CreateInstance/E_UNEXPECTED
PS C:\Users\mizuno>

```

図 24.5: なにか起きてはいけないエラーが起きていることがわかる

起動しない状態の WSL から、ファイルをなんとか救出する方法を考えてみましょう。と言ったものの、実はそれほど難しい問題ではありません。実際のサーバーでも、ディストロトラブルなどで OS が起動しなくなってしまうことはよくあります。ですが OS は起動しなくなっても、ディスクそのものは残っているわけです。つまり USB メモリなどから別の Linux システムを起動したり、ディスクを取り出して USB 経由で別マシンにマウントすれば、データの救出は可能です。WSL2 も中身は仮想マシンで、そのディスクはイメージファイルとして存在します。このファイルを別の WSL ディストリビューションにマウントしてあげればよいわけです。

そんなことをしなくても、Windows のエクスプローラーから WSL 内のファイルにアクセスできるじゃないかと思われるかもしれません。実はこの機能は、WSL のディスクイメージの中身を Windows が直接見ているわけではなく、WSL に内蔵された 9p サーバーとの、ネットワークを経由したファイル共有によって実現されています。よって WSL が起動しない状態では、エクスプローラーから WSL 内のファイルにアクセスはできません。

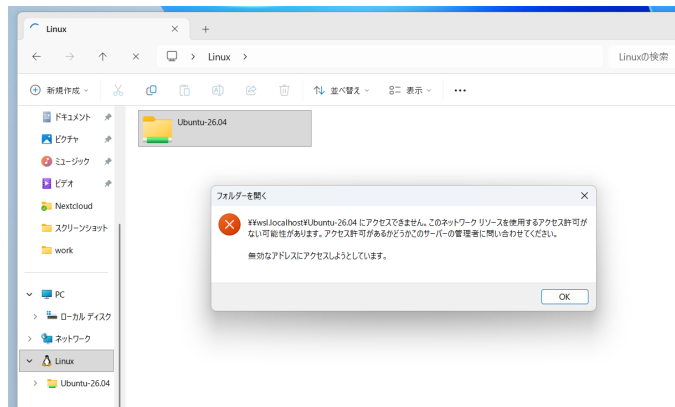


図 24.6: WSL が起動しない状態では、エクスプローラーからアクセスはできない

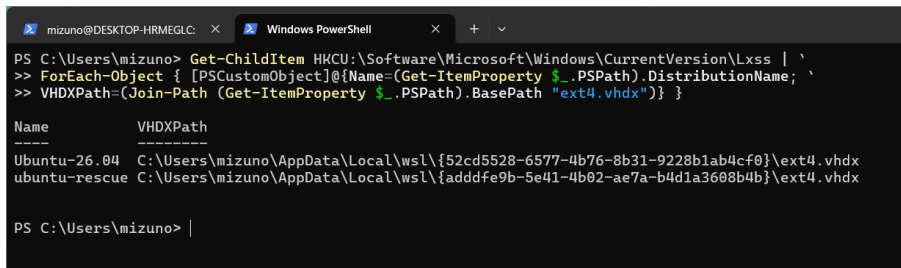
それではレスキュー用の WSL 環境を用意しましょう。以下のコマンドで、Ubuntu-26.04 を別名でインストールします。

```
$ wsl --install -d Ubuntu-26.04 --name ubuntu-rescue
```

続いて壊したディスクイメージを、この環境にマウントします。しかし WSL2 のディスクイメージである vhdx は少々特殊な形式をしており、一般的な仮想マシンイメージのように、Linux から簡単にマウントできません。そこで現在の WSL では `wsl --mount` という専用のコマンドが用意されています。まずは対象の vhdx ファイルを探しましょう。以下のコマンドを実行してください。

```
$ Get-ChildItem HKCU:\Software\Microsoft\Windows\CurrentVersion\Lxss | `
ForEach-Object { [PSCustomObject]@{Name=(Get-ItemProperty $_.PSPath).DistributionName; `
VHDXPath=(Join-Path (Get-ItemProperty $_.PSPath).BasePath "ext4.vhdx")} }
```

実行すると以下の画像のように、インストールされている WSL ディストリビューションのインスタンスごとに、vhdx ファイルのパスが表示されます。先ほど壊したのは `Ubuntu-26.04` ですから、この vhdx ファイルへのパスを控えておきましょう。



```
PS C:\Users\mizuno> Get-ChildItem HKCU:\Software\Microsoft\Windows\CurrentVersion\Lxss | `
>> ForEach-Object { [PSCustomObject]@{Name=(Get-ItemProperty $_.PSPath).DistributionName; `
>> VHDXPath=(Join-Path (Get-ItemProperty $_.PSPath).BasePath "ext4.vhdx")} }

Name                VHDXPath
-----
Ubuntu-26.04        C:\Users\mizuno\AppData\Local\wsl\{52cd5528-b577-4b76-8b31-9228b1ab4cf0}\ext4.vhdx
ubuntu-rescue       C:\Users\mizuno\AppData\Local\wsl\{adddf9b-5e41-4b02-ae7a-b4d1a3608b4b}\ext4.vhdx

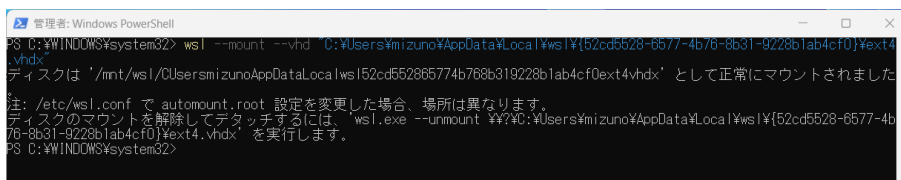
PS C:\Users\mizuno> |
```

図 24.7: インスタンスごとに vhdx ファイルのパスが表示される

続いてディスクイメージをマウントするのですが、これには管理者権限が必要となります。管理者権限で新しい PowerShell を起動して、以下のコマンドを実行してください。vhdx へのパスは、先ほど控えた実際のパスに置き換えてください。

```
$ wsl --mount --vhd "vhdx へのパス"
```

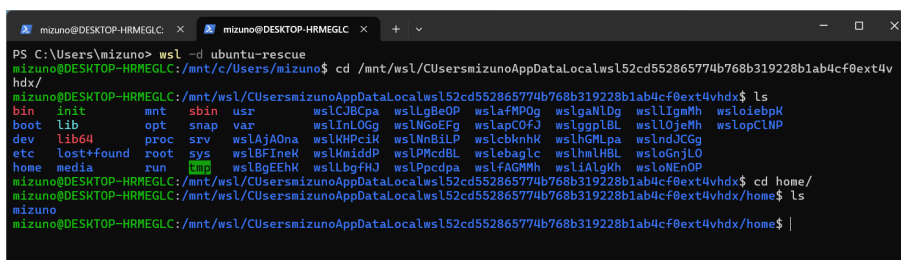
するとすべての WSL 内から、/mnt/wsl/(vhdx ファイルのパスから特殊文字を除いたものというパスで、この vhdx の中身にアクセスできるようになります。



```
PS C:\WINDOWS\system32> wsl --mount --vhd "C:\Users\mizuno\AppData\Local\wsl\{52cd5528-b577-4b76-8b31-9228b1ab4cf0}\ext4.vhdx"
ディスクは '/mnt/wsl/CUsersmizunoAppDataLocalwsl52cd5528b5774b768b319228b1ab4cf0ext4.vhdx' として正常にマウントされました
は、/etc/wsl.conf で automount.root 設定を変更した場合、場所は異なります。
ディスクのマウントを解除してデタッチするには、'wsl.exe --unmount %?'%0:~\Users\mizuno\AppData\Local\wsl\{52cd5528-b577-4b76-8b31-9228b1ab4cf0}\ext4.vhdx' を実行します。
PS C:\WINDOWS\system32>
```

図 24.8: 正常に vhdx をマウントできた

あとは先ほどインストールしたレスキュー用のディストリビューションを起動し、必要なファイルを救出してください。



```
PS C:\Users\mizuno> wsl -d ubuntu-rescue
mizuno@DESKTOP-HRMEGLC: /mnt/c/Users/mizuno$ cd /mnt/wsl/CUsersmizunoAppDataLocalwsl52cd5528b5774b768b319228b1ab4cf0ext4v
hdx/
mizuno@DESKTOP-HRMEGLC: /mnt/wsl/CUsersmizunoAppDataLocalwsl52cd5528b5774b768b319228b1ab4cf0ext4vhdh$ ls
bin    init      mnt      sbin     usr      wslCJBcpa  wslLgBeOp  wslafMP0g  wslgaNLdg  wslLlgMh  wslloiebpK
boot   lib       opt      snap     var      wslInLoGg  wslVGoEfG  wslapCOFj  wslggpLdL  wslldjehh  wsltopCLNP
dev    lib64     proc     srv      wslAJAOna  wslWHPciK  wslNnBtLp  wslcbknhk  wslh2MLpa  wslnd3CGg
etc    lost+found  root    sys      wslBFIneK  wslWaidcP  wslPMcdBL  wsllebaglc  wslhmlHdL  wslGnJlO
home   media     run      wslBgEEhK  wslLbgfHJ  wslPpcdpa  wslfAGMMh  wslAlgKh  wslNEnOP
mizuno@DESKTOP-HRMEGLC: /mnt/wsl/CUsersmizunoAppDataLocalwsl52cd5528b5774b768b319228b1ab4cf0ext4vhdh$ cd home/
mizuno@DESKTOP-HRMEGLC: /mnt/wsl/CUsersmizunoAppDataLocalwsl52cd5528b5774b768b319228b1ab4cf0ext4vhdh/home$ ls
mizuno
mizuno@DESKTOP-HRMEGLC: /mnt/wsl/CUsersmizunoAppDataLocalwsl52cd5528b5774b768b319228b1ab4cf0ext4vhdh/home$ |
```

図 24.9: レスキュー環境からマウントしたディスク内を見る

必要なファイルのレスキューが完了したら、管理者権限の PowerShell で、以下のコマンドを実行してください。vhdx をアンマウントできます。なおアンマウントの際には --vhd オプションは不要です。

```
$ wsl --unmount "vhdx へのパス"
```

最後に、破損して起動しなくなった WSL ディストリビューションは、以下のコマンドで削除してしまいましょう。

```
$ wsl --unregister Ubuntu-26.04
```

WSL 環境にログインできなくなったり、そもそも起動しなくなってしまったとしても、ディスクそのものが読める形で残っているのであれば、まだ救いはあります。トラブルは起きないに越したことはありませんが、それでも起こるべくして起こってしまうものです。いざという時に備え、こうしたレスキュー方法もぜひ覚えておきましょう。

あとがき

ここまで本書をお読みいただき、ありがとうございました。

WSL は、登場した当初こそ「Windows 上で Linux のコマンドを使うための便利な仕組み」という印象の強い機能でした。しかし現在では、本物の Linux カーネルを備え、GUI アプリや systemd、GPU、コンテナ、さらには Kubernetes まで扱える、本格的な Linux 実行環境へと成長しています。Windows を普段の作業環境として使いながら、必要なときにはすぐ Linux を呼び出せるという使い方は、もはや特殊なものではありません。

本書では、WSL の導入から Linux の基本操作、ターミナルやシェルのカスタマイズ、開発環境の構築、Docker や Kubernetes の利用、そしてバックアップやデータレスキューといった実践的な運用までを紹介してきました。すべてを一度に覚える必要はありません。まずは興味を持った章から試し、日々の作業の中で少しずつ自分の環境を育てていってください。

Linux を使いこなすうえで重要なのは、コマンドを暗記することではなく、必要な情報を調べ、試し、失敗したときに原因を切り分ける力です。エラーメッセージを読み、マニュアルや公式ドキュメントを確認し、小さな実験を繰り返すことが、結局はいちばんの近道になります。

近年は、わからないことを AI に尋ねる機会も増えました。AI は調査の入口や作業の補助として非常に便利ですが、必ずしも正しい答えを返すとは限りません。特に、自分自身が正解を判断できない分野で回答をそのまま信用するのは危険です。実際にコマンドを実行し、結果を確認し、公式情報と照らし合わせる姿勢は、これまで以上に重要になるでしょう。

WSL や Ubuntu は、今後も変化を続けていきます。本書の内容の多くも、きっとすぐに時代遅れになってしまうでしょう。しかし、ここで身につけた Linux の基本的な考え方や、環境を自分で構築して問題を解決する経験は、10 年戦える力になります。

本書が、Linux を使いはじめるきっかけとなり、読者のみなさんの Windows 環境を少し便利に、そして技術を学ぶ時間を少し楽しいものにできたなら、著者としてこれ以上うれしいことはありません。



日本仮想化技術株式会社 X: @VirtualTech_jp

日本で唯一の仮想化技術に特化した R&D 企業。その中でも DevOps チームでは、クラウドやコンテナの導入、IaC によるインフラ管理、CI/CD による自動化などをはじめとする、DevOps を実現するための支援サービスを提供中。

<https://virtualtech.jp>



水野源 X: @mizuno_as

スーファミは 1 台しか持ってなかったけど、メガドライブは 3 台持っていたので、単純計算でセガのハードは任天堂の 3 倍売れているはずという持論を今日も崩さない。Emacs だけのこ派。



たなかともあき

十数年前に Linux と Windows のデュアルブート環境を壊してしまってから以来、Windows には縁がありません。メインの OS は Arch Linux、Window Manager と CLI ツールいじりが趣味。3 匹の猫に囲まれて生活しています。

Windows ユーザーのための WSL で始める Linux 環境構築術

2026 年 7 月 27 日 発行

発行所 VirtualTech Japan Inc.

連絡先 VirtualTech Japan Inc. (<https://virtualtech.jp/>)

印刷所 株式会社アクセア

Windowsのまま、 Linuxの世界へようこそ!

WSLを使えば、Windows環境をそのままに、
Linuxのコマンドラインや豊富なツール、コンテナ環境を
手軽に利用できます。

本書では、WSLの導入から初期設定、基本操作、
Dockerとの連携まで、Windowsユーザー向けに
やさしく、わかりやすく解説します!

コマンドラインに
チャレンジ!



WSLなら
Windowsとの
連携もスムーズ!



Dockerで
コンテナ環境も
カンタン活用!



Linuxの知識が
きっと武器になる!



こんな人におすすめ!

- ✓ Linuxに触ってみたいWindowsユーザー
- ✓ Dockerを使ってみたい人
- ✓ コマンドラインに慣れたい人
- ✓ 業務や学習にLinux環境を活用したい人

本書でできること



WSLの導入から
初期設定まで
やさしく解説!



基本コマンドを
しっかり学べる!



Dockerとの連携で
コンテナ活用も
バッチリ!



初心者でも安心の
ステップ形式で
迷わず進める!

Windows環境を活かしながら、
Linuxの可能性を広げてみませんか?