今日からはじめる

DevOps

実践ガイド

今日からはじめる **DevOps** 実践ガイド

VirtualTech Japan Inc.　著

2025-10-20 版　　**VirtualTech Japan Inc.** 発行

まえがき

みなさんは **DevOps** という言葉を聞いて、何を思い浮かべるでしょうか。開発チームと運用チームの連携強化、組織文化の変革、アジャイル開発の推進……これらはすべて正しい理解です。しかし多くの現場では、「DevOps とは何か」はなんとなく理解できても、「では具体的に何から始めればよいのか」という疑問に明確な答えを見つけられずにいるのが現実ではないでしょうか。これは DevOps という言葉が指す範囲が非常に広く、抽象度の高い言葉であることに起因しているのかもしれません。

DevOps は、単なる技術の導入だけで実現できるものではありません。ビジネスプロセスの見直しはもちろん、組織の文化的な改革にまで踏み込まなければならないケースもあります。しかしそうした大きな変革を待っていては、いつまでたっても第一歩を踏み出すことができません。そこで筆者は「習うより慣れろ」をコンセプトに、まずは技術面から DevOps の世界に足を踏み入れてもらおうと考え、本書を執筆しました。具体的なツールに触れ、実際に手を動かしてみることで、DevOps 的な考え方を体得していただくことを目指しています。

本書は、DevOps を支える技術要素を体系的に学習できるように構成しました。前半（第 1 章〜第 6 章）では、**コンテナ技術**、**バージョン管理**、**CI/CD**、**Infrastructure as Code**、**監視**といった、DevOps に必要な技術の概念と全体像を解説します。後半（第 7 章〜第 9 章）では、**Terraform**、**GitHub Actions**、**Amazon CloudWatch** という代表的なツールを例に挙げ、実際の動かし方を紹介します。この構成により、DevOps に関する知識を体系的に身につけると同時に、実際に手を動かして学習を進めることができます。理論だけでなく、まずは手を動かしてみましょう。

現代のソフトウェア開発において、DevOps はもはや特別なものではありません。そして DevOps で使われる Docker、Kubernetes、Git、CI/CD、Infrastructure as Code、監視といった技術は、個々に学ぶべき専門分野ではなく、すべてのエンジニアが身につけるべき**基礎体力**だと筆者は考えています。本書を通じて、DevOps が決して高度で難しいものではなく、日々の開発業務を効率化し、品質を向上させるための実用的なアプローチであることを実感していただければと思います。

それでは、DevOps の世界への第一歩を踏み出してみましょう。

なお本書は、ThinkIT の連載「DevOps を実現するために行うこと・考えること[*1]」をリライトし、再構築しなおしたたものとなります。

[*1] https://thinkit.co.jp/series/10843

目次

第 1 章　DevOps のフローと実践に必要な技術

DevOps を実践したいが何から始めればよいかわからない。そんな疑問に答えるため、この章では技術面から DevOps にアプローチします。開発から運用まで一連の流れで使われるツールと、エンジニアに求められるスキルの変化を解説します。

1.1 はじめに

みなさんは DevOps を実践していますか？　当然実践しているという方も、実践していないという方もいるでしょう。ではどちらにしろ、何を基準にしている、していないの判断をしたのでしょうか。

実は「何を実現すれば DevOps と呼べるのか」に、明確な答えはありません。とはいえ、ゆるやかな合意というものは存在します。あえて言えば**開発チームと運用チームのコラボレーションを強化し、品質の高いソフトウェアを頻繁にリリースする**というのが最大公約数的なコンセプトです。この部分に異論のある方はいないでしょう。

このコンセプトは、単に新しいツールを導入するだけでは実践できません。ビジネスのあり方そのものを変える必要があったり、古い考えを改めるなど、組織の文化改革が必要となることもあります。つまり組織によって、必要な対応策が異なってくるのです。その上、どのような方法でこれらのコンセプトを実現していくのかを、具体的に示した資料は多くありません。こうした背景もあり、「自分たちの組織が DevOps を実現するためにすべきこと」を理解するためには、決して短くない時間がかかります。そして時間をかけたとしても、その答えを得られない可能性もあるのです。これが DevOps の参入障壁のひとつなのではないでしょうか。

文化や組織のあり方を変えていくのはもちろん重要ですが、そこで足踏みしていては前に進むことができません。そこで筆者は、「習うより慣れろ」をコンセプトに、技術面から DevOps を実践していくのがよいのではないかと考えました。本書では文化や組織の話はいったん切り離し、エンジニアが DevOps を実践するための具体的な技術やツール、環境構築と、その運用法について学んでいただきます。まずはこれらの技術に触れ、とにかく手を動かしてみてください。具体的なツールの動きを知ることで、DevOps 的な考え方を理解できるはずです。

1.2 DevOps の一連の流れと技術

DevOps では、その一連の流れを**インフィニティループ**という図で表現します。これは機能の企画、開発、ビルド、テスト、デプロイ、リリース、運用、監視、そしてフィードバックといったステップが、また次の企画に繋がっていくループです。このループを開発チームと運用チームで協力しながら繰り返し実行し、品質の高いソフトウェアをリリースしていくのが DevOps です。

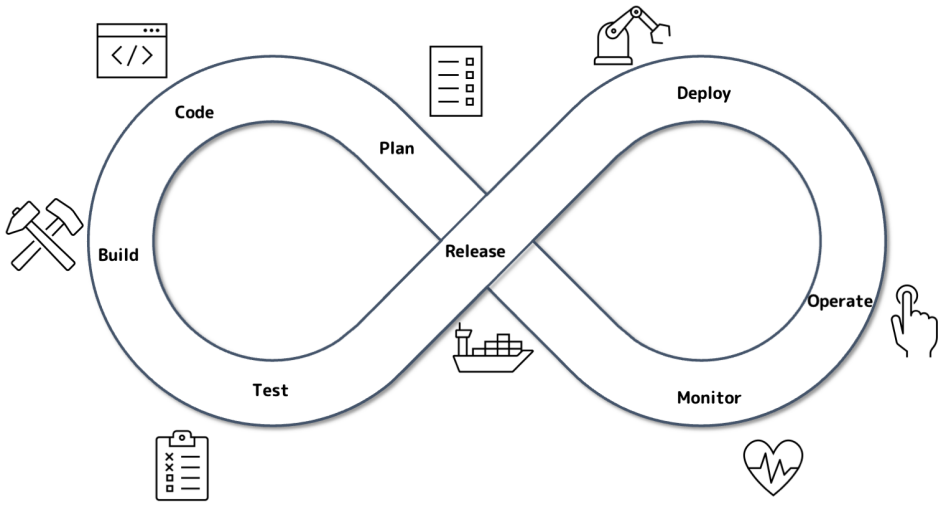

図 1.1: DevOps のインフィニティループ

このループを回し続けるためには、前述した通り組織の文化やビジネスのあり方を変えるだけでなく、それに加えて適切なツールを利用する必要があります。ですが DevOps は、何かひとつのツールを導入すれば実践できるというようなものではありません。それぞれのフェーズに合った適切なツールを、複数組み合わせて利用する必要があります。もちろん、適切なツールの組み合わせはそれぞれの状況によって異なるため、これが絶対の正解という組み合わせはありません。そのため本書では、筆者が支援する案件でよく使われるツールを例に紹介します。まずは全体像を掴むという意味で、DevOps を実現するための技術を俯瞰的に見ていきましょう。

1.2.1 実行環境

実行環境とは、開発するアプリケーションを配置し実行する環境のことです。DevOps では、ターゲットとなるアプリケーションの種類そのものは問いません。しかし最も DevOps と相性が良いのは、コンテナ技術を使った Web アプリケーションでしょう。そのため、本書では、コンテナ技術を使用した Web アプリケーションを題材とします。

コンテナとは、アプリケーションの実行環境をカプセル化し、ポータビリティを高める技術です。コンテナ内には、アプリケーションが動作するために必要な環境一式が含まれています。そのためコンテナは依存関係を持たず、どのような環境であっても同じように、アプリケーションを動かすことができます（可搬性）。また同一のイメージを利用することで、開発環境から本番環境まで一貫して、同じ環境を使うことができるため、環境依存のトラブルを大幅に減らすことができます（環境再現性）。

コンテナとは、あくまでこうした仕組みの名称です。そのため実際にコンテナを動かす技術には、様々な実装が存在します。その中でも、世界的に最も普及しているのが **Docker** でしょう。Docker を使えば、簡単にコンテナイメージをビルドしたり、そのイメージを使ってアプリケーションを動かすことができます。

一般的に、本番環境のアプリケーションが、単一のコンテナで完結しているケースはほとんどありません。機能ごとに分割された複数のコンテナが、ネットワークを通じて協調動作します。またコンテナを動かすホストも、複数用意するのが基本です。こうした複数ホスト、複数コンテナの構成を、うまく

ハンドリングするためのツールが**コンテナオーケストレーションツール**です。そしてオーケストレーションツールの分野でデファクトスタンダードとなっているのが、**Kubernetes** です。

最近では、こうしたコンテナ実行環境はクラウド上に用意するのが一般的です。そのため AWS、Azure、Google Cloud といったクラウドプロバイダーは、これらのコンテナ技術を簡単に利用できる、マネージドサービスを用意しています。

1.2.2 開発環境

開発環境とは、アプリケーションを開発するための環境のことです。Visual Studio Code のような、コードを書くツールを指して、（狭義の）開発環境とするケースもあるでしょう。ですが本書では、開発時に開発者が使用する環境全般を、（広義の）開発環境と定義します。具体的には開発者の PC 上で動作している Docker や、クラウド用に用意された開発用サーバーなども開発環境です。

アプリケーションの元となるのがソースコードです。そのため DevOps をはじめるにあたっては、**ソースコード管理**が欠かせません。最近ではチームでの開発のしやすさ等の理由から、分散バージョン管理システムを使ったソースコードの管理が一般的です。そしてほとんどのケースで、**Git** という分散バージョン管理システムが利用されます。Git には、気軽に作成できるブランチ機能や、非常に賢いマージ機能が搭載されており、複数のメンバーが並行して作業を進めやすくなっています。

また、狭義の開発環境である**開発ツール**も重要な技術です。かつてはテキストエディターでコードを書き、ビルド用に Makefile を書き、CLI から make コマンドを実行するといった、それぞれのステップで異なるツールを使用するのが一般的でした。しかし最近では、コードを書くだけではなく、Git やビルドツール、テストツール、デバッグといった、さまざまな作業を 1 つのツール上から実行できる統合開発環境を使用するケースが増えています。

アプリケーション開発は、コードを書いたらそれで終わりというわけではありません。書いたコードが正しく動作するための**検証環境**も必要となってきます。検証環境は開発したアプリケーションを動かしてみるだけではなく、自動化されたテストを実行するターゲットとして利用されることもあります。コンテナ技術を使用するシステムでは、本番環境と同じコンテナイメージを作成し、本番と同じ環境でアプリケーションの動作を確認します。

1.2.3 CI/CD

開発者がコードを書いた後、アプリケーションのビルドやテスト、デプロイを自動で行う一連の仕組みが **CI/CD** です。CI は**継続的インテグレーション（Continuous Integration）**、CD は**継続的デリバリー（Continuous Delivery）**の略語です。CI/CD は DevOps サイクルにおいて、Dev から Ops への流れを作る、中核的な技術となっています。そしてその目的は、繰り返し行う作業の自動化です。

統合（Integration）とは、コードをリリースブランチにマージ可能な状態とすることです。そのためには、コードに問題がなく、リリース可能であることを担保しなくてはなりません。そのためには「テスト」が必要となってきます。CI の主目的は、追加されたコードに対するテストの自動化であると考えて差し支えありません。

CI は、主にソースコード管理ツールに対するコードのプッシュやマージをトリガーに、CI ツールがビルドやテストを自動的に実行します。この CI ツールにどのような処理をどういった順番で自動的に実行させるのか定義したものを「CI パイプライン」と呼びます。プロジェクトとしてどのような CI パイプラインを定義し、運用していくかが、開発プロセスを自動化するための重要な作業となります。

CI が完了し、リリース可能と判断されたアプリケーションは、実行環境にデプロイしなくてはなりません。これを担うのが CD です。先ほど CD は継続的**デリバリー**と紹介しましたが、実はもう 1 つ別の定義があります。それが**継続的デプロイメント（Continuous Deployment）**です。継続的デリバリーはアプリケーションをリリース可能な状態にまでデリバリーし、最後に実行環境へリリースするのは手動で行う手法です。一方、継続的デプロイメントはアプリケーションを実行環境にリリースする

までを含めて自動化する手法です。どちらを選択するかはアプリケーションの位置付けや、開発プロセスにおける考え方によって変わります。

さらに、アプリケーションをデプロイする方法としても 2 つの考え方があります。1 つは CI パイプラインの中でデプロイまで行ってしまう方法で、もう 1 つがアプリケーションのパッケージやコンテナイメージの更新を検知して自動的にデプロイを行う方法です。これらは前者を **CIOps**、後者を **GitOps** と呼んでいます。

1.2.4 インフラ構築

実行環境の例として、Docker、Kubernetes、クラウドについて紹介しました。コンテナは、どのような場所であっても、一貫して同じアプリケーション実行環境を再現できます。ですがコンテナを実行するホスト側は、それぞれのプラットフォームに応じた設定が必要です。

従来であれば、このような環境構築の作業は手順書に則ってエンジニアが手作業で行うことが一般的でした。しかし手作業で環境を構築する場合、いくら手順書に従って作業をしてもヒューマンエラーを防ぐことはできません。そこで最近使用されるようになったのが **IaC（Infrastructure as Code）** という技術です。

IaC はインフラの設定をコード化して管理し、記述されたコードに従って自動的に構築する技術です。そのため、何度作業をしても人間に起因する障害が発生せず、同じ結果となることを担保できます。そのため繰り返し実行する作業は、コード化することが強く推奨されています。

IaC に対して批判的な意見に、「二度とやらない作業だから、コード化のコストが割に合わない」というものがあります。ですが一度しか構築しないような環境でも、IaC 化する意味はあります。まずやはり、作業ミスを防げるという点です。ヒューマンエラーが発生しないため、もし構築した環境に問題がある場合には、コードに問題があることになります。もしも手作業で構築した場合は、手順書に問題があるのか、作業時にヒューマンエラーが発生したのかの切り分けが必要になりますが、IaC を利用している場合はそのコードのみを検証するだけで良くなります。また IaC コードに対する自動テストも可能なため、間違いがないか事前にデバッグすることも可能になります。そして設定値をコードとして残すことで、パラメータシートのようなものも不要になります。

1.2.5 運用監視

アプリケーションが本番環境にデプロイされたら、それで終わりではありません。どのようなシステムも、運用を継続していると、必ず障害を起こします。そのためアプリケーションやサービスが正しく機能しているか、常に監視しなくてはなりません。これが**運用監視**です。運用監視の必要性は DevOps に限らず、どのようなシステムでも同様ですが、DevOps においては「開発チームと運用チームのコラボレーションを強化する」という観点から、従来の運用監視よりも積極的に障害の可能性を検知し、開発チームにフィードバックしていくことが求められています。

まず基本となるのが**メトリクス監視**です。ただし、やたらめったらとメトリクスを並べても意味がありません。どのコンポーネントをどのメトリクスで監視するか、その取捨選択も重要になります。またアプリケーションだけでなく、ネットワークやハードウェア、クラウドのサービスステータスなども監視の対象となります。

リソースの変化はメトリクスで監視できますが、「その時、何が起きたか」を追跡するために必要なのが**ログ**です。ログを分析して管理するには、統合型の監視ツールを使用するのが一般的です。統合型の監視ツールを使うことで、様々なコンポーネントから取得してきた情報を一箇所で分析し、運用状況を判断できます。また最近では障害が発生してからアラートを出すのではなく、障害発生の兆候を掴むことで障害が発生する前に対応するといった積極的な運用監視も行われています。

このような形で運用中のシステムの状態を把握し、問題が発生した、もしくは発生しそうな場合は開発チームにフィードバックし、対応を依頼する必要があります。ただし、なんでもかんでもフィードバックをして対応してもらうわけにもいきません。問題の原因と影響を分析し、対応するものしないもの、開発チームに依頼するもの、運用チームで対応するものといった切り分けを行う必要があります。

残念ながらこの部分に関しては未だに人間の力に頼る必要がありますが、対応するとなった場合には月次ミーティングなどで共有していては「頻繁にリリースする」メリットが消えてしまいます。そこで、イシュー管理ツールやタスク管理ツールを使って発生した問題の情報を即座に共有しつつ、対応を依頼するという方法が採られるようになりました。最近では統合型監視ツールからこのようなイシュー管理/タスク管理ツールと連携して簡単に情報を収集し、フィードバックできるような仕組みもあります。

また最近では、AI を活用した障害予測なども登場しています。

1.3 DevOps に関わるエンジニアの役割

ここまで DevOps の一連の流れと、そこで使用されるツールや環境について解説してきました。これらのツールや環境は当然ながら、まずそれらを利用できるよう、環境を構築する必要があります。前述の通り DevOps においては今までとは異なる技術を使ったり、同じ技術でも考え方を変えたりする必要がありますが、それに伴ってエンジニアが果たすべき役割の範囲は異なってきます。

ここからは、DevOps におけるエンジニアにはどのような役割があるのか、また、それぞれがどのような形で各技術に関わっていくのかを解説します。

1.3.1 開発チームと運用チーム

DevOps では**開発チームと運用チームのコラボレーションを強化する**のが重要なコンセプトのひとつです。開発チームと運用チームの境目を曖昧にするという表現もよく使われます。しかし、日本のシステム開発の実態は、DevOps が前提とする開発チームと運用チームのあり方とは異なる部分が少なくありません。特にエンタープライズ領域で見られるようなシステムでは、開発を外部のシステム開発会社に発注し、運用自体は自社の IT 部門で行うというケースが多く見られます。このような形態では、開発チームと運用チームでそもそも会社が異なるという分断が起き、境界を曖昧にすることが難しくなってしまっています。

とは言え、分断があるからといって DevOps が実践できないわけではありません。かつては「コラボレーションを強化する」と言うと、開発チームと運用チームが仲良くするといった人的なコラボレーションが想像されることもありました。しかし、DevOps で重要なのは**プロセス面と情報面でのコラボレーション**です。これらのコラボレーションはチーム同士が同じ空間にいなくても技術や仕組みで実現できます。

具体的な例を考えてみましょう。継続的デリバリーにおいて、リリース準備までを開発チームが責任を持ち、本番環境へのデプロイを運用チームが責任を持つという役割分担がされているとします。その際、CI/CD の技術を使えば、開発チームがテスト環境へアプリケーションをデプロイする仕組みと、運用チームが本番環境へとデプロイする仕組みを共通化できます。また、先ほどの運用監視からフィードバックへの流れも運用チームから開発チームへのコラボレーションを強化する仕組みとなります。

もちろん、開発チームと運用チームのコラボレーションを強化するには、技術的アプローチだけでなく組織的、文化的アプローチも加えて実施する必要があります。ですが仕組みを抜きにしては、DevOps におけるコラボレーションは実現できません。

1.3.2 開発エンジニアとインフラエンジニア

エンジニア個人のスキルに注目すると、**開発エンジニア**と**インフラエンジニア**に大別できるでしょう。これは従来のシステム開発にも存在していた役割ですが、DevOps ではそれぞれの役割が変化します。

まず開発エンジニアの視点では、自動化への対応強化が必要となります。開発エンジニアの主な役割は、アプリケーションのコードを作成し、正しく動作するアプリケーションを作ることです。そして正しく動くアプリケーションを作るため、必要不可欠なのがテストです。従来の開発では手作業でテストを行うこともありましたが、DevOps におけるテストは、CI パイプラインの中で自動化されるのが前提です。そのため、テストケースもコード化する必要があります。

ここで言うテストには、単体テストだけでなく、結合テストや E2E テストなども含まれる場合があります。ただし結合テストや E2E テストは自動化の難易度が高いため、従来通りの手作業でテストを実施する場合もあります。しかし単体テストに関して言えば、自動化は必須です。

アプリケーションが実行時に使用するライブラリは、コンテナのビルド時に、コンテナ内に埋め込むのが基本です。そのため開発者は、コンテナの仕様にも精通している必要があります。従来であれば、開発エンジニアはアプリケーションのコードだけに注力すればよく、インフラについてはインフラエンジニアに一任していました。ですがコンテナ化により、開発エンジニアにもインフラの知識が要求されるようになってきています。

一方、インフラエンジニアの視点では「手段」と「範囲」の両面で変化があります。まず手段の面では、インフラ構築が手順書に沿った手作業から IaC へ変化します。主な作業は IaC のコード作成と、その結果構築された環境が正しく動作するかの検証です。IaC ツールに準じたコードの書き方を学ぶ必要はありますが、手順書を書いて手作業で構築するよりも作業量は減り、なおかつ構築の確実性が増します。

範囲の面では、CI/CD の環境構築も、インフラエンジニアの役割になります。CI/CD のインフラ構築にあたっては単に CI/CD ツールを導入するだけでなく、CI/CD のプロセスを実行する設定、テストやビルドに必要な環境の設定なども必要になってくるでしょう。

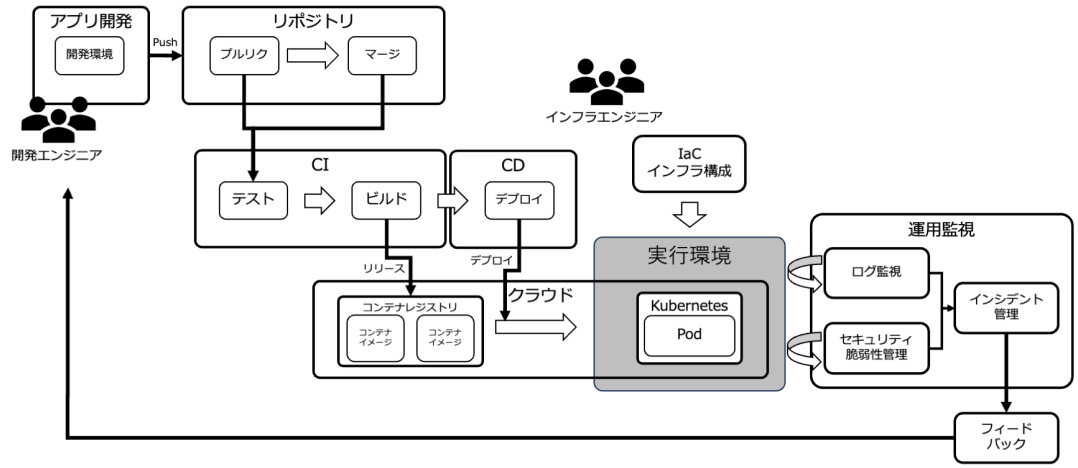

図 1.2: 開発エンジニアとインフラエンジニア

第 2 章　DevOps の中核となるコンテナとクラウド

DevOps を支える技術の中でも、特に重要なのがコンテナとクラウドです。Docker によるアプリケーションのコンテナ化、Kubernetes によるオーケストレーション、そしてクラウドプラットフォームの活用により、高速な開発サイクルと安定した運用が実現できます。この章ではこれらの技術の基本概念と、DevOps における役割を解説します。

2.1 コンテナについて知ろう

まずは基本に立ち返り、そもそもコンテナとはなんなのか、なぜこれだけ流行したのかという視点から、コンテナについて説明します。

2.1.1 コンテナとは

DevOps を実践するにあたって、切っても切り離せないツールとなっているのが**コンテナ**です。コンテナは今や、開発の現場において、なくてはならない技術に成長しています。

コンテナといえば真っ先に思い浮ぶのは Docker でしょう。ですがコンテナと Docker はイコールというわけではありません。Docker はあくまでも、世の中に数多く存在する、コンテナ実行環境のひとつに過ぎないためです。とはいえ、世界で最も普及しているコンテナ実行環境が、Docker であることもまた事実です。そのため DevOps の文脈においては、Docker のことを指して「コンテナ」と呼ぶこともよくあります。

Docker の普及によって、コンテナはぐっと身近なものになりました。特に Docker が持つ、コンテナレジストリを中心としたエコシステムの存在もあり、専門的な知識はなくとも、いくつかのコマンドを覚えるだけで、簡単にコンテナを使用できてしまいます。そのため Docker は使っているが、コンテナのことはよくわかっていないという方も少なくないでしょう。

コンテナを一言で説明すると、**プロセスを専用の空間に隔離して動かす技術**です。と言われても、なんのことか理解できないかもしれません。もう少し噛み砕いて説明しましょう。

Linux は今時の OS の例の漏れず、マルチプロセスの OS です。つまり複数のプロセスを同時に動かすことができるわけです。そのため 1 台のサーバー上で、Web サーバーの Apache HTTP Server、メールサーバーの Postfix、データベースサーバーの MySQL といった複数のプロセスを、共存させることができます。そしてこれらのプロセスはどれも、同じ OS の空間内で起動します。そのため同じルートファイルシステムにアクセスでき、/etc 以下の設定ファイルや、ネットワークインターフェイス、IP アドレスなどを共有しています。

これはこれで便利なのですが、サーバー用途では、これが問題となることがあります。例えば同じ IP アドレスを共有しているため、用途の異なる Web サーバーをふたつ同時に起動しようとすると、ポートの競合が発生してしまいます。ルートファイルシステムも共有していますから、/etc 以下にある OS レベルの設定を、プロセスごとに変えるといったこともできません。

コンテナは、Linux カーネルが持つ **namespace** や **cgroup** といった機能を使い、プロセスごとに専用の空間を作り出します。この空間はホスト OS から隔離されており、各コンテナは独立した IP アドレスや名前空間を持ちます。そしてルートファイルシステムも、専用のものがアタッチされます。コンテナ内で動作しているプロセスは、ホストから見れば単なるプロセスに過ぎません。ですがコンテナ内から見ると、あたかも OS を専有しているように振舞えるのです。従来であれば、Web サーバーやデータベースサーバーといった用途ごとに専用の仮想マシン（VM）を用意し、そのプロセスに OS を占有させていました。コンテナを使えば、それと同じことを、非常に低コストで実現できるというわけです。

図 2.1: コンテナの模式図

シングルバイナリで構成されたような一部のアプリケーションを除き、一般的なアプリケーションは機能の多くを、外部の共有ライブラリに「依存」しています。そのためアプリケーションのインストール時には、実行時に要求されるライブラリを、別途インストールしなくてはなりません。これは非常に手間な上、「開発環境では動くけど本番環境では正常に動作しない」「OS をアップグレードしたら動かなくなった」といった問題の原因となっていました。

コンテナは、**コンテナイメージ**と呼ばれるイメージファイルから起動されます。コンテナイメージの中には、アプリケーション本体と、その動作に必要なランタイム・ライブラリなど、依存関係のあるすべてのものが含まれています。つまりコンテナ化されたアプリケーションは自己完結しているため、ライブラリの不足やバージョンの不一致といった問題が起きません。そのためコンテナはどこで実行しても、同じように動作します。

コンテナはよく、海上コンテナに例えられます。鉄でできた大きな箱は国際的に標準化されており、どのような船舶・鉄道・自動車にも乗せることができます。これはコンピューターの世界でも同じです。Docker コンテナの仕様は、Open Container Initiative(OCI)[*1]という標準化団体によって標準化されています。コンテナイメージのフォーマットや、実行環境に関する標準規格が定められているため、OCI 標準に準拠していれば、どのような実行環境を使っても、同じようにコンテナを動かせるのです。例えば Docker 互換のコンテナ実行環境として、Red Hat が開発する Podman[*2]があります。

[*1] https://opencontainers.org/
[*2] https://podman.io/

2.1.2 コンテナイメージとは

コンテナイメージは、コンテナの元となるファイルシステムです。具体的には、イメージを構成するレイヤーのファイルシステムのアーカイブと、実行コマンドやパラメータなどのメタ情報で構成されています。そして通常、一度完成したコンテナイメージを書き換えることはしません。そしてコンテナはこのファイルシステムを、ルートファイルシステムとしてマウントします。これにより、どのような場所で起動しても、常に同一の環境が再現できることわけです。

独自のコンテナイメージは、ベースとなるイメージに、変更レイヤーを加えることで作成します。ベースイメージには最小限のコンポーネントで構成された OS イメージや、nginx などのミドルウェアがインストールされたイメージ、Python や Ruby などのプログラミング言語ランタイムがインストールされたイメージなど、自由に選択できます。ベースイメージは、Docker Hub などのコンテナレジストリから入手可能です。

ベースイメージに対して行われた変更差分は、**レイヤー**という単位で管理されます。レイヤーは行った変更ごとに作成され、すべてのレイヤーをミルフィーユ状に重ねたものを**イメージレイヤー**と呼びます。イメージレイヤーは基本的に読み取り専用になります。

コンテナを起動すると、イメージレイヤーの上に、書き込み可能なレイヤーが追加されます。これを**コンテナレイヤー**と呼びます。そしてイメージレイヤーとコンテナレイヤーを重ね合わせたものが、最終的なルートファイルシステムとしてコンテナにマウントされます。これは **OverlayFS** という技術で実現されています。

コンテナレイヤーは一時的なレイヤーのため、コンテナを削除すると、その内容は失われます。コンテナ内で行った変更は、イメージレイヤーに反映されることはありません。これは一見不便なようですが、**常に同じ状態で起動する、コンテナを作り直せば初期状態に戻せる**という特性は、水平スケールと相性がよいという面もあります。また機能不全を起こしたコンテナは削除して作り直してしまえばよいため、障害対応もシンプルになるというメリットがあります。

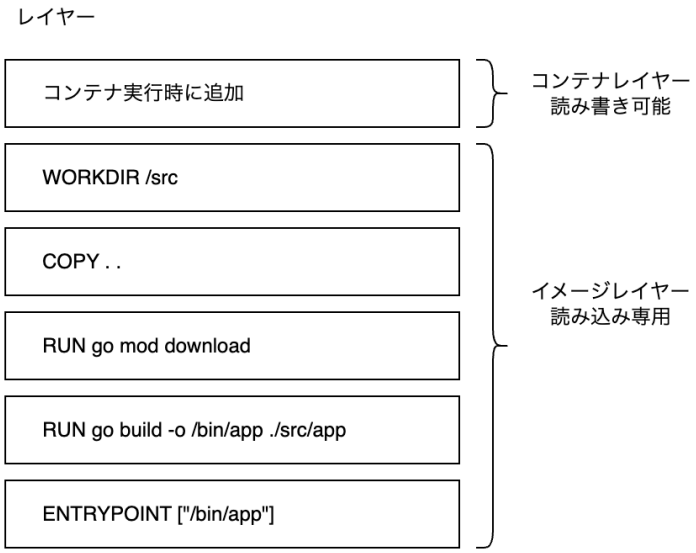

図 2.2: コンテナレイヤーの模式図

2.1.3 実は古くからあるコンテナ

コンテナと言うと、ここ 10 年ほどで登場した、新しい技術のように思えるかもしれません。ですがコンテナや、それに類似した技術は古くから多数存在します。いくつか例をあげてみます。

- 1979 年、Unix Vertion 7 に新機能として追加された **chroot** というシステムコールが、コンテナという考え方の起源だとされています。chroot はあるプロセスのルートディレクトリを別のディレクトリに変更することでプロセスを特定のディレクトリ以下に閉じ込めます。例えばセキュリティ上の理由で、FTP サーバーのリモートユーザーがアクセスできるディレクトリを制限する、といった用途でよく利用されました。
- 2000 年、**FreeBSD jail** が登場します。FreeBSD jail は chroot の機能を強化しファイルシステムやユーザー、ネットワークなどを分離できます。現在使用しているコンテナの原型とされています。
- 2005 年、**OpenVZ** が登場します。これは Parallels Virtuozzo Containers for Linux のオープンソース版で、1 台の物理サーバ上に、複数の独立した Linux 環境を構築する目的で利用されました。マルチテナントのレンタルサーバーでの採用例も見られました。
- 2008 年、**Linux Containers(LXC)**[*3]が登場します。後発の Docker も、初期の頃は内部的なコンテナライブラリとして LXC を使用していました。現在の Linux コンテナの形を作ったと言えます。
- 2013 年、dotCloud 社によって **Docker**[*4]が発表されます。Docker はコンテナイメージのビルドから配布、実行までを一環したツールで簡単に行うことができます。またコンテナレジストリという、誰もが自由にイメージを配布、利用できるエコシステムが用意されたことにより、一気に普及しました。

OpenVZ や LXC のコンテナは**システムコンテナ**とも呼ばれ、内部でフル機能の Linux OS が動作しています。いわば軽量な仮想マシンのようなコンテナでした。これに対して Docker は、コンテナ内では単一のプロセスのみを起動するというアプローチを採用しました。これを**アプリケーションコンテナ**と呼びます。アプリケーションコンテナは起動時に、PID 1 として init ではなく、対象のアプリケーションを直接実行します。こうすることでシステムの初期化や、その初期化に必要なプログラムが不要になり、コンテナはさらに軽量で高速になりました。Docker の普及により、現在では単に「コンテナ」と言った場合は、暗黙的にアプリケーションコンテナを指すことも多くなりました。

2.2 コンテナと仮想マシン

コンテナと仮想マシンは、目的に応じて専用の環境を用意するという目的だけを見れば同じものです。しかし開発の現場では、仮想マシンからコンテナへの移行が進んでいます。この 2 つの技術の違いを見ていきましょう。

2.2.1 仮想マシンとは

仮想マシンとは、**物理的なハードウェアをソフトウェア的に再現したもの**です。1 台のコンピューターの上に、複数の仮想マシンを作成できます。

仮想マシンを作成および実行するには、仮想化ソフトウェアが必要です。そして仮想化ソフトウェアは、大きく**ホスト型**と**ベアメタル型**の二種類に分類されます。

ホスト型は、Linux をはじめとするホスト OS 上に、仮想化ソフトウェアをインストールして実行します。導入が簡単なため誰でもすぐに始められるのが、ホスト型のメリットです。ホスト型の仮想化ソフトウェアとしては、VMware Player や VirtualBox があります。仮想マシンはホストから完全に独

[*3] https://linuxcontainers.org/ja/
[*4] https://www.docker.com/

立しているため、ホスト OS とは別の OS を実行できます。例えば Windows の上で Linux の仮想マシンを動かすことも、その逆も可能です。

ベアメタル型は、ハードウェア上で直接、仮想化ソフトウェアを実行する方式です。そのためホスト OS を必要とせず、ホスト型よりもハードウェアを効率よく使用できます。ベアメタル型の仮想化ソフトウェアとしては、VMware ESXi が有名です。

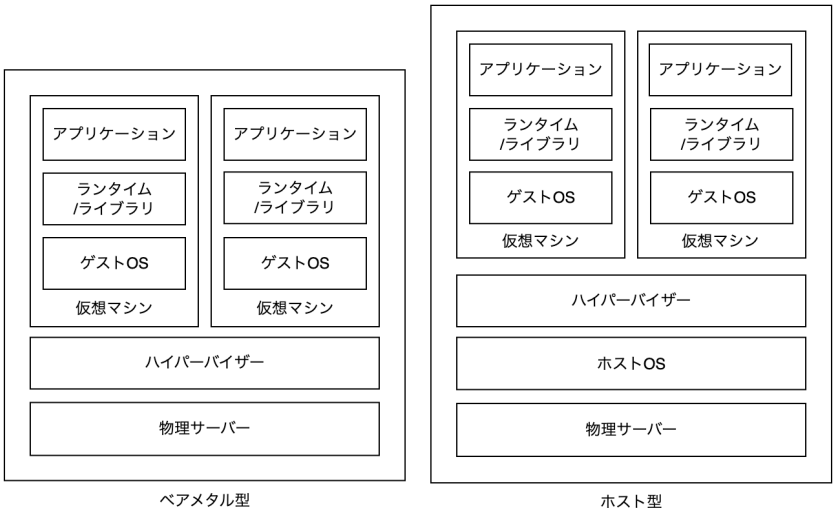

図 2.3: ベアメタル型とホスト型の仮想化ソフトウェア

2.2.2 仮想マシンの目的

仮想マシンは複数のサーバーを 1 台のコンピューターに集約する目的で使用されます。例えば、メールサーバー、ファイルサーバー、Web サーバーという、3 つのサーバーがあったとしましょう。これらのソフトウェアが、それぞれ違う OS を要求していると、当然ですが物理マシンが 3 台必要になってしまいます。ソフトウェアを追加するたびに、物理マシンを調達していては、時間やお金もかかってしまいます。ですが仮想マシンを使えば、複数のサーバーを 1 台の物理サーバー上に集約できるのです。

また大抵のサーバーは、CPU やメモリといったリソースを常に 100% 使っているわけではなく、使われていない「余剰リソース」が存在します。物理マシンを使っている場合、この余剰リソースは、遊ばせておくことしかできません。ですが仮想マシンでサーバーを集約すれば、この余剰部分に他のサーバーを詰め込むことができ、結果として効率のよい運用が可能になるのです。仮想マシンを使うことで、マシン調達にかかる時間やマシンの運用費を節約できます。

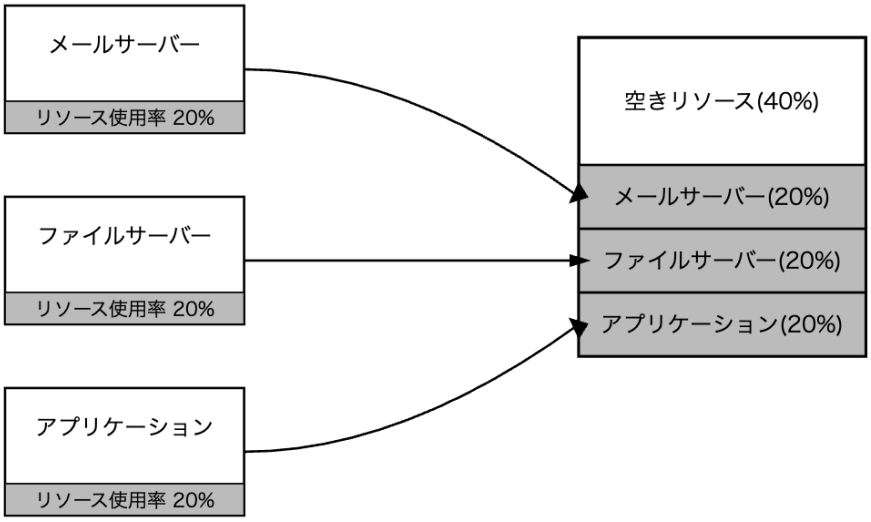

図 2.4: 仮想マシンを利用したサーバーの集約

2.2.3 コンテナの目的

一般的にコンテナは、アプリケーションのポータビリティを向上させる目的で使用されます。コンテナはどのような環境で起動しても、同じ状態を再現できます。そのため開発環境と本番環境の差異をなくしたり、逆に本番環境で発生した障害を、開発環境で再現することも容易です。

コンテナは依存関係を持たず、実行環境一式を持ち運べるため、アプリケーションのデプロイにかかるコストも低減できます。例えばアプリケーションのバージョンアップは、検証も含めて非常に面倒な作業です。ですがコンテナであれば、新しいバージョンのイメージをデプロイするだけで、バージョンアップを行えます。また通常、アプリケーションのデプロイは不可逆な作業のため、古いバージョンへのロールバックは面倒な作業です。ですがコンテナであれば、これも旧バージョンのイメージをデプロイし直すだけになります。「試しに使ってみて、困ったら元に戻す」といった戦略も取りやすくなり、インフラ管理の労力を大幅に軽減することができます。

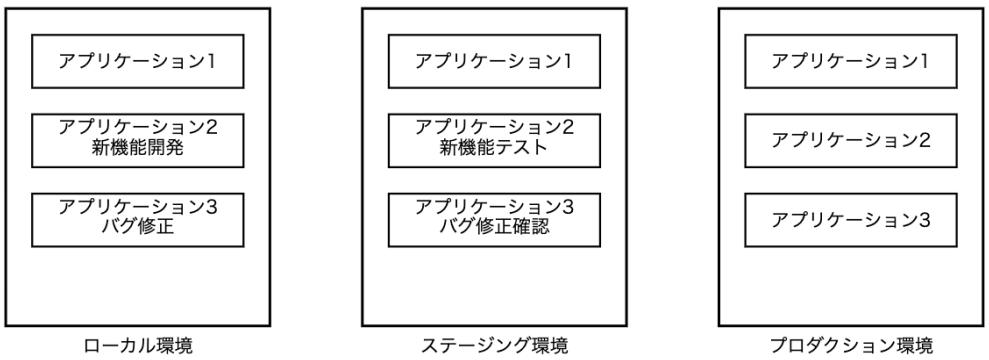

図 2.5: コンテナ運用例

2.3 コンテナの普及

昨今のクラウドネイティブなアプリケーション開発では、デプロイのしやすさや、高いスケーラビリティなどが求められます。コンテナは軽量で高速に動作する上、揮発性であるという特性から水平スケールが行いやすく、クラウドネイティブなアプリケーション開発にはなくてはならない技術となっています。

2.3.1 軽量・高速

コンテナは仮想マシンよりも軽量です。仮想マシン内には、完全な OS がインストールされています。Linux をインストールしていれば数 GB、Windows をインストールしていれば数十 GB 以上のストレージが必要になるでしょう。一方でコンテナ内には、アプリケーションと、その依存関係にあるコンポーネントのみしか含まれないため、仮想マシンと比べて軽量です。コンテナの作りにも依存しますが、数 MB 程度のサイズで提供されているコンテナも存在します。

コンテナを起動するには、コンテナイメージを**コンテナレジストリ**から入手しなくてはなりません。これにはネットワークを経由したダウンロードが必要となるため、コンテナイメージは軽量であるに越したことはありません。

一般的に、コンテナは仮想マシンよりも高速に起動します。仮想マシンはハードウェアをエミュレートするため、起動時に様々な初期化処理が行なわれます。またブートローダーがカーネルを呼び出し、通常の OS の起動プロセスを実行するため、アプリケーションにアクセスできるまでには、数分のオーダーで時間がかかります。一方でコンテナは、ホスト OS 上でプロセスを実行するだけです。そのため数秒でアクセス可能となります。こうした起動の速さは、急なアクセス増加に対応するようなケースでの、スケーラビリティに直結します。

2.3.2 効率的

仮想マシンは、起動している OS が必要とする分のリソースを要求します。対してコンテナは必要なプロセスのみしか起動しないため、そのプロセスが必要とするリソースのみで動かすことができます。リソース消費が少なければ、同時に実行できるアプリケーション数も増え、サーバーの集約効率も上がります。

2.4 Docker とは

ここから Docker について、もう少し詳しく解説していきましょう。そもそも Docker とは、コンテナを簡単に扱うためのツールです。開発元によれば、**アプケーションを開発・配布・実行をするためのプラットフォーム**だと定義されています。Docker は、アプリケーションの動作環境をコンテナ内に構築し、パッケージ化することで、ポータビリティを向上させます。これによってアプリケーションは、インフラストラクチャから切り離され、様々な環境へ持ち運び、同じ方法で実行できます。仮想マシンと比較して語られることの多いコンテナですが、どちらかと言えば Docker は、Ubuntu の APT や Ruby の Gems のような、パッケージシステムに近い立ち位置だと考えてよいでしょう。

Docker は、Linux カーネルの機能を使用してコンテナの分離を実現しています。具体的にはホスト上で実行されているプロセスを、カーネルの名前空間 (Namespaces) と呼ばれる機能を使用して、分離された環境で実行します。

以下の例では ubuntu:22.04 のコンテナで sleep コマンドを実行しています。

```
$ docker run --rm -d ubuntu:22.04 sleep 100
7cb93f83191b833d2a8bd57d0c8640ac626b972b3746fd050f4be9befa0b0ca8
```

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND       CREATED         STATUS         PORTS     NAMES
7cb93f83191b   ubuntu:22.04   "sleep 100"   3 seconds ago   Up 2 seconds             kind_wing
```

実行中のコンテナで ps コマンドを実行すると先に実行していた sleep コマンドと ps コマンドの 2 つが確認できます。

```
$ docker exec kind_wing ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 11:51 ?        00:00:00 sleep 100
root           7     0  0 11:52 ?        00:00:00 ps -ef
```

ホスト側からプロセスを確認するとコンテナで実行している sleep のプロセスが確認できます。

```
$ pgrep -a sleep
11522 sleep 100
```

また、sleep コマンドがコンテナ上では PID 1 で実行されている点にも注目してください。通常の Linux であれば、systemd に代表される init プロセスが PID 1 として実行され、システムの稼動に必要な様々なプロセスは、init の子プロセスとして実行されます。ですが Docker コンテナにおいては、ターゲットとなるプロセス（ここでは sleep）が、PID 1 として直接実行されます。繰り返しになりますが、このようなコンテナは**アプリケーションコンテナ**と呼ばれています。アプリケーションコンテナは、アプリケーションのプロセスのみを起動するため、高速な上、オーバーヘッドが少ないとされています。

2.5 Docker のアーキテクチャ

Docker はクライアント・サーバ型のアーキテクチャで構成されています。Docker クライアントと Docker デーモンの間の通信には、RESTful API が用いられます。Docker の動きを図で示すと次のようになります。それぞれのコンポーネントについて簡単にご紹介しましょう。

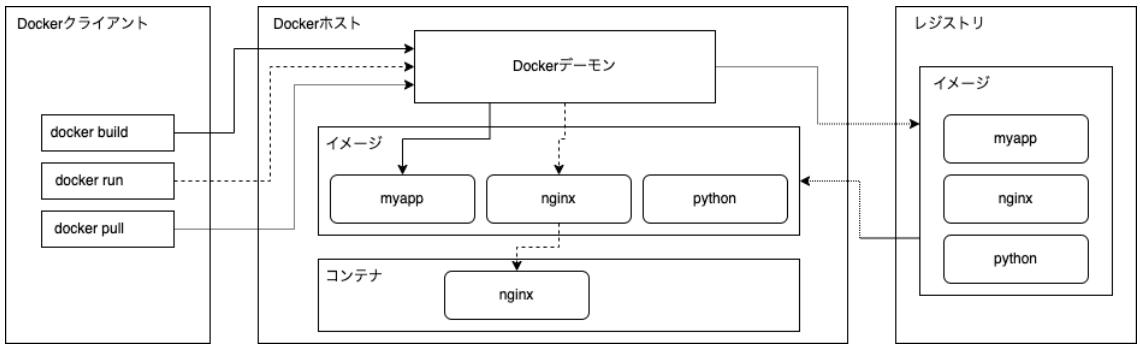

図 2.6: Docker の構成要素

2.5.1 Docker デーモン

Docker デーモンは Docker API リクエストを待ち受け、イメージ・コンテナ・ボリューム・ネットワークなどの Docker オブジェクトを管理します。Docker デーモンを実行しているホストを Docker ホストと呼びます。

2.5.2 Docker クライアント

Docker クライアントは、Docker デーモンに命令を送るためのフロントエンドです。前述の通り、RESTful API を叩ければ Docker デーモンに命令を送ることはできるため、極論、curl コマンドでも Docker を使うことはできます。ですがそれでは非常に使いづらいため、`docker` という専用のコマンドが用意されています。`docker` コマンドは、デフォルトでローカルにインストールされたデーモンと通信しますが、リモートの Docker デーモンを利用することもできます。

2.5.3 Docker レジストリ

コンテナイメージの保管庫が **Docker レジストリ**です。Docker はレジストリから、コンテナイメージを取得します。Docker のデフォルトのコンテナレジストリは **Docker Hub**[*5]です。名前からもわかるように、Docker Hub は Docker 社が運営をしており、コンテナレジストリのデファクトスタンダードとなっています。Docker Hub には **Docker Official Image**[*6]や **Verified Publisher**[*7]と呼ばれるイメージ群があります。これらのイメージは、公式の開発者や、認証されたベンダーがセキュリティを担保しているため、安心して利用できます。

Docker Hub 以外のよく使われるレジストリとしては、GitHub Container Registry や、主要なパブリッククラウドが提供しているレジストリなどがあります。また、セルフホステッド型のコンテナレジストリも存在します。

レジストリによってはイメージの取得に認証が必要な**プライベートレジストリ**と呼ばれるタイプのレジストリも存在します。独自にビルドしたイメージを、外部に公開せず利用したい場合は、プライベートレジストリを使用します。

2.5.4 Dockerfile

Dockerfile は、コンテナイメージを作成するための設計図となるファイルです。中身はプレーンなテキストファイルであり、Dockerfile 特有のフォーマットと命令を使用して、アプリケーションの動作環境を構築するためのすべての手順を記述します。Docker は Dockerfile に記述されている命令を順次実行し、コンテナイメージを作成します。

2.5.5 Docker オブジェクト

Docker は、コンテナイメージ、起動したコンテナ、コンテナが通信に使うネットワーク、データを保存するボリュームといったリソースを、**オブジェクト**として扱います。オブジェクトは docker コマンドを以下のようなフォーマットで用いて管理します。

```
docker [オブジェクト] [サブコマンド]
```

以下に簡単な使用例を紹介します。

コンテナの一覧

[*5] https://hub.docker.com/
[*6] https://hub.docker.com/search?badges=official
[*7] https://hub.docker.com/search?badges=verified_publisher

```
$ docker container ls
CONTAINER ID   IMAGE          COMMAND       CREATED          STATUS          PORTS     NAMES
f07a669741ca   ubuntu:22.04   "sleep 100"   35 seconds ago   Up 34 seconds             busy_haslett
```

コンテナイメージの一覧

```
$ docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED              SIZE
ubuntu       22.04     9b8dec3bf938   About a minute ago   117MB
```

コンテナイメージの削除

```
$ docker image rm 9b8dec3bf938
Untagged: ubuntu:22.04
Deleted: sha256:9b8dec3bf938bc80fbe758d856e96fdfab5f56c39d44b0cff351e847bb1b01ea
```

すべてのオブジェクトを削除

```
$ docker system prune --all --force --volumes
Deleted Images:
untagged: docker.io/library/ubuntu:22.04
deleted: sha256:9b8dec3bf938bc80fbe758d856e96fdfab5f56c39d44b0cff351e847bb1b01ea
deleted: sha256:b4b521bfcec90b11d2869e00fe1f2380c21cbfcd799ee35df8bd7ac09e6f63ea
deleted: sha256:3565a89d9e81a4cb4cb2b0d947c7c11227a3f358dc216d19fc54bfd77cd5b542
deleted: sha256:37aaf24cf781dcc5b9a4f8aa5a99a40b60ae45d64dcb4f6d5a4b9e5ab7ab0894

Total reclaimed space: 29.54MB
```

2.6 Docker のメリット・デメリット

オンプレミスや仮想マシンからコンテナへの移行が、最近のトレンドです。オンプレミスや仮想マシンと比べると、Docker には様々なメリットが存在します。ですがメリットばかりではなく、当然デメリットも存在します。Docker のメリットとデメリットについて、簡単に解説します。

2.6.1 Docker のメリット

少ない変更で環境構築が可能

物理マシンや仮想マシンに OS をクリーンインストールしてから、必要なミドルウェアやプログラミング言語のランタイム、ライブラリなどをインストールし、アプリケーションが動作する環境を構築していくのが、従来のやり方でした。そのため環境構築には、その OS を扱う作法や、様々なパッケージシステムに対する知識が必要になります。

一方 Docker は、アプリケーションを動かすベースとして最適なイメージをコンテナレジストリから探し、そこにわずかな変更を加えることで、アプリケーション実行環境を構築します。そして Docker Hub には、こうしたベースとなるイメージが、数多く存在します。例えば Ruby で書かれたアプリケーションを動かしたいのであれば、公式の Ruby コンテナイメージをベースとすることで、OS やランタイムのインストールは省略し、アプリケーションのインストールのみに注力することができるわけです。最適なベースイメージを選択することで、セキュアで無駄のない環境を、最小の手順で構築できます。

アプリケーションやインフラストラクチャをコードで管理

コンテナイメージを構築するための手順は、Dockerfile にすべて記述されています。Dockerfile はテキストファイルであるため、バージョン管理システムと相性がよく、構成のレビューや変更の追跡が容

易です。環境構築に際して一切の手作業が不要になるため、どこで誰が実行しても同じイメージが作成できます。

イメージの共有が容易

Docker には、イメージを簡単に配布する仕組みがあります。そのうち最もポピュラーなのは、コンテナレジストリを使う方法でしょう。Docker はレジストリに対してイメージをアップロードしたり、公開されているイメージをダウンロードする機能があり、難しい手順を踏まなくても、レジストリ経由でイメージを共有できます。また実行中のコンテナから、tar 形式のファイルとしてイメージを出力する方法もあります。こちらはイメージを共有するというより、主にコンテナを保存する目的で使用されます。

2.6.2 Docker のデメリット

学習コスト

Docker を使用するには、コンテナについての一般的な知識にくわえ、Docker 特有の作法を知っておく必要があります。例えば Docker は、ひとつのコンテナ内では、ひとつのアプリケーションのみを実行するのがベストプラクティスです。ですがアプリケーションによっては、Web サーバーとデータベースサーバーなど、複数のプロセスを必要とする場合もあるでしょう。この場合、Web サーバーコンテナとデータベースコンテナを別々に起動し、ネットワークを通じて連携させるのが一般的ですが、Docker コンテナへの理解が足りないと、ひとつのコンテナの中で、複数のプロセスを起動する方法を模索してしまうかもしれません。

Dockerfile の書き方についても学ぶ必要があります。よくあるのがキャッシュの利用効率です。Docker は Dockerfile に記述された命令単位でレイヤーを作成し、キャッシュします。そして Dockerfile の該当行に変更がない限り、その命令は再実行されず、作成されたレイヤーキャッシュが使い回されます。つまり、変更の少ない順に命令を記述することで、キャッシュを効率的に活用し、イメージ作成の時間を短縮できます。逆に実行の度に結果が変わることを期待している命令があっても、古いキャッシュが利用されてしまったがために、古い情報がコンテナに埋め込まれてしまうケースもあります。このように、特有の仕様を理解した上で Dockerfile を記述しないと、イメージのサイズが肥大化したり、イメージの内容が意図しないものになる可能性があります。

これはほんの一例であり、Docker を正しく効率的に使用するには、コンテナ、OS、Docker 特有の作法について、さらに深い理解が必要です。

Docker は非常に強力なツールであり、DevOps を実践する上では避けては通れないツールのひとつです。ですが従来のオンプレミスと比較すると、環境に対する考え方そのものを変える必要もあり、その学習コストは決して低くありません。これが DevOps 実践のハードルとなってしまう可能性は否定できません。

2.7 Kubernetes とは

単一のマシン、具体的には 1 台のサーバーや開発者の手元の PC 上で、単にコンテナを実行したいだけであれば、Docker は素晴しい選択肢であると言えるでしょう。例えば開発環境をコンテナ化すれば、コンテナの環境再現性を利用して、チーム全員の開発環境を簡単に揃えることができます。また環境を手軽に構築・破棄できるため、一時的なテスト環境を用意することも容易です。同一のイメージから複数のコンテナを起動し、並列で動かすこともできるため、今使っている環境を止めることなく、別の環境を実行することもできます。

では、運用面はどうでしょうか。本番環境では、ひとつのコンテナを起動して終わりということはなく、複数のコンテナが協調して動作するのが一般的です。またコンテナが動作するホストも、耐障害性、高可用性、負荷分散などを考慮し、複数台用意するのが基本です。コンテナは何を、どれだけ起動したらよいのでしょう。オペレーターは複数のサーバーにデプロイされたコンテナを管理し、ロールア

ウトやロールバックに対して責任を持たなくてはなりません。それぞれのコンテナには監視や負荷分散の設定も必要です。コンテナ同士の通信は、どのように管理したらよいでしょう。上手に負荷分散するには？　ログを集約するには？　コンテナが障害で停止したら？　考えることは山積しています。そしてこうした運用を、人間が手動で、24 時間 365 日体制で行うのは現実的ではありません。

そこで、ソフトウェアによってコンテナの運用管理を自動化するためのツールが開発されました。こうしたツールを**コンテナオーケストレーションツール**と呼びます。コンテナオーケストレーションツールを使用すれば、複数のサーバーに分散してデプロイされた複数のコンテナの、効率のよい運用管理を実現できます。そしてコンテナオーケストレーションツールの分野において、デファクトスタンダードとなっているのが **Kubernetes** です。

Kubernetes の元となるソフトウェアは、Google によって開発されました。基本概念としては、もともと Google の社内で使用されていたクラスタマネージャの **Borg** と、その後継モデルである **Omega** の設計を引き継いでおり、そこに Google が日々行っているデプロイから学んだ、様々なノウハウが組み込まれています。Kubernetes はオープンソースソフトウェア (以下 OSS) として公開されているので誰でも無料で使用できます。また、バグなどを発見した場合は誰でもフィードバックを送信し貢献できます。

Kubernetes がどのように生まれたのかについては、

- Google から世界へ: Kubernetes 誕生の物語[*8]
- Kubernetes: The Documentary [PART 1][*9]
- Kubernetes: The Documentary [PART 2][*10]

といった資料が参考になります。

2.7.1 オーケストレーションツールのデファクトスタンダード

コンテナオーケストレーションツールはいくつか存在しますが、前述の通り、現在は Kubernetes がデファクトスタンダードとなっています。デファクトスタンダードとなった背景には、コミュニティの成長や、主要なクラウドプロバイダーの対応などが考えられます。

2.7.2 拡大するコミュニティ

Kubernetes は元々、Google を中心としたオープンソースプロジェクトでした。ですが現在では Google の手を離れ、Cloud Native Computing Foundation（CNCF）[*11]によってホストされています。CNCF には著名な開発者をはじめ、大手クラウドプロバイダーなど、多くの人や企業が参加しています。現在は 8,012 もの企業、77,115 ものコントリビューターが Kubernetes の開発に参加しています（2023 年 6 月現在）[*12]。

2.7.3 クラウドプロバイダーの対応

主要なクラウドプロバイダーは、Kubernetes をマネージドサービスとして提供しています。例えば Google Cloud であれば Google Kubernetes Engine（GKE）[*13]、Azure であれば Azure Kubernetes Service（AKS）[*14]、AWS であれば Amazon EKS[*15]がそれに該当します。これにより、構築運用の決

[*8] https://cloudplatform-jp.googleblog.com/2016/08/google-kubernetes.html
[*9] https://www.youtube.com/watch?v=BE77h7dmoQU
[*10] https://www.youtube.com/watch?v=318elIq37PE
[*11] https://www.cncf.io/
[*12] https://www.cncf.io/reports/kubernetes-project-journey-report-jp/
[*13] https://cloud.google.com/kubernetes-engine?hl=ja
[*14] https://azure.microsoft.com/ja-jp/products/kubernetes-service
[*15] https://aws.amazon.com/jp/eks/

して少なくない部分をクラウドプロバイダーに任せることが可能になり、ユーザーは Kubernetes を簡単に利用しやすくなりました。

2.8 Kubernetes の基礎知識

Kubernetes を理解するには、まず Kubernetes 特有の用語や、仕組みを理解しておく必要があります。ただし、Kubernetes はあまりにも巨大で複雑なシステムなため、その全容を解説することは容易ではありません。そこで本書では、使用を開始するにあたって困らない程度の、最低限の用語と仕組みに留めて説明します。

2.8.1 kubectl

`kubectl` は、Kubernetes を操作するためのコマンドラインツールです。アプリケーションのデプロイからリソースの管理、ログの表示といった作業は、すべてこのコマンドを通じて行います。

2.8.2 マニフェスト

Kubernetes では、どんなオブジェクトを作成するかを、YAML フォーマットのファイルで定義します。この YAML フォーマットの定義書を**マニフェスト**と呼びます。マニフェストには、コンテナをどのように稼動させるか、必要な CPU やメモリなどのリソース量、どのように振る舞うか、再起動が必要なのかなど、「システムがあるべき状態」を記述します。そして kubectl を通じて、マニフェストを Kubernetes に伝えると、Kubernetes がシステムをその状態へ収束させてくれます。これを**宣言的な構成管理**などと呼びます。

2.8.3 ポッド

ポッド（Pod）とは、簡単に言ってしまえばコンテナの集まりです。ひとつのポッドには、ひとつ以上のコンテナが含まれ、これが Kubernetes における、デプロイの最小単位となります。一般的なコンテナ運用においては、プロセス単位で別のコンテナを起動します。ですが複数のプロセス（コンテナ）を密結合させ、セットで運用したいケースも存在します。例えばコンテナ間でストレージやネットワークを共有したい場合や、アプリケーションに足りない機能を追加したい場合などです。また複数のコンテナをグループ化することで、必要なコンテナ群を効率よくスケールアウトすることが可能になります。

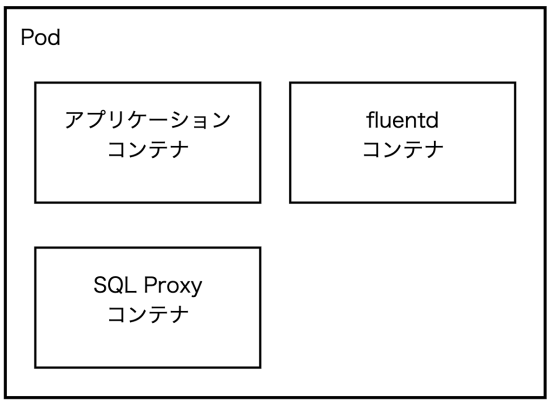

図: コンテナとポッド

2.8.4 コントロールプレーン

コントロールプレーン（Control Plane）は、Kubernetes の頭脳にあたる部分です。コントロールプレーンを実行しているサーバー群を、**マスターノード**とも呼びます。後述するワーカーノードやポッドを管理し、クラスターに関する決定や、クラスターイベントの検知および応答をします。例えばポッドの新規作成を検知すると、どのノードで実行するのが最適かを計算してデプロイ先を決定し、ポッドの作成を指示します。コントロールプレーン自身でもポッドのホストは可能ですが、構成をシンプルにするため、コントロールプレーンにはコントロールプレーン以外の仕事はさせないのが一般的です。

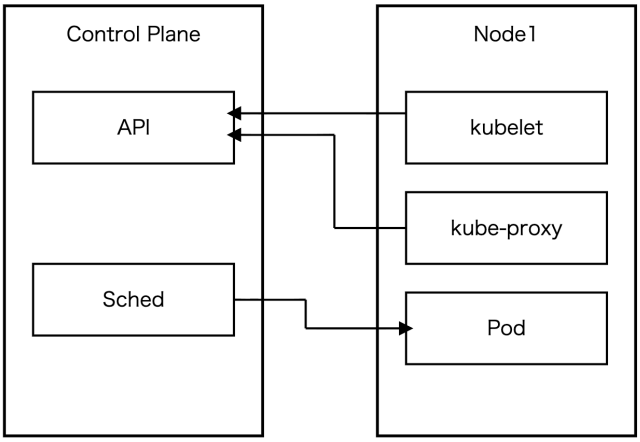

図: コントロールプレーンとワーカーノード

2.8.5 ノード

ノード（Node）はポッドをホストするサーバーです。**ワーカーノード**とも呼ばれています。Kubernetes クラスターには、ポッドを実行するためのノードが、最低 1 台以上必要です。ノードでは実行されているポッドの状態を監視し、必要に応じてコントロールプレーンに状態を通知します。また、クラスター内部または外部からのトラフィックを目的のコンテナに転送する処理もノード内で行われています。ワーカーノードの集まりは、コントロールプレーンに対して**データプレーン**とも呼ばれます。

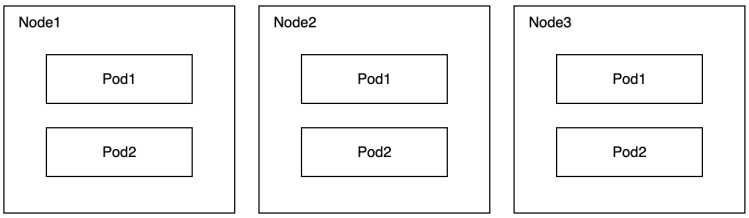

図: ワーカーノードとデータプレーン

2.8.6 サービスディスカバリー

コンテナは、デプロイする度に IP アドレスが変化する可能性があります。そこでコンテナの IP アドレスが変化してもアクセスを継続できるよう、Kubernetes はコンテナに DNS レコードを付与しま

す。またコンテナを冗長化している場合には、DNS レコードに複数の IP アドレスを登録することで、DNS ラウンドロビンによる負荷分散が可能になっています。そして DNS レコードへの IP アドレスの登録は、Kubernetes が自動的に行います。

2.8.7 自動的なロールアウト

あらかじめコンテナのあるべき状態を Kubernetes に通知しておくことで、Kubernetes はその状態を保つよう、コンテナを調整します。例えば実行するコンテナの数を Kubernetes に通知しておけば、仮に希望する台数と、実際に起動されたコンテナの数に差が出てしまった場合でも、希望台数と同じになるように、自動でコンテナの実行と停止が行われます。

2.8.8 デプロイ先ノードの決定

Kubernetes は、データプレーンを構成するそれぞれのワーカーノードの空きリソースと、起動すべきコンテナが要求しているリソースから、ノードのリソースを最大限効率的に活用できるコンテナの配置方法を自動で計算します。

2.8.9 自動修復

Kubernetes は、ヘルスチェックに応答しないコンテナを、強制終了させます。こうして終了させられたコンテナや、処理が失敗してクラッシュしたコンテナは、再起動、あるいは新しいコンテナへ入れ替えられます。これらは Kubernetes が自動的に行います。また Kubernetes は、コンテナが正常に起動し、処理の準備が整うまで、クライアントからのリクエストをルーティングしません。これにより、起動中のコンテナにリクエストが送られ、サービスが応答しなくなるということを防げます。

2.8.10 機密情報の管理

パスワードやトークンなどの機密情報を、コンテナイメージに直接埋め込むのはセキュリティリスクであり、絶対に行ってはいけません。ですが何らかの方法で、コンテナ内に機密情報を送り込む必要は出てきます。そのため Kubernetes には、機密情報を安全に保持し、管理する機能が用意されています。機密情報をコンテナイメージから分離できるため、コンテナイメージを再作成せずとも、機密情報のみを更新できます。

2.9 Kubernetes のメリット・デメリット

Kubernetes は非常に強力で便利なツールですが、機能が豊富で自由度が高い上、分散システム特有の複雑さもあるため、一朝一夕で使いこなせるようなツールでもありません。きっちりと使いこなすためには、相応の学習が求められます。そのため、本当に Kubernetes を導入するメリットがあるのかについては、熟慮が必要でしょう。ここでは Kubernetes のメリット・デメリットについて説明します。

2.9.1 メリット

コスト

Kubernetes でコンテナを運用することで、サーバー台数を削減できる可能性があります。Kubernetes は、空きリソースのあるノードを積極的に使用します。空きリソースのあるノードにポッドを詰め込むことで、サーバーの集約率を上げられます。つまり、少ないノードで最大限のポッドを稼動させられるため、結果的にサーバー台数を削減できる可能性があります。

従来のアプリケーション運用であれば、それぞれのサーバーや仮想マシンを監視し、問題が起きれば人の手で修復していました。そのため 24 時間 365 日無停止で動かすのならそれだけ人員も必要になり

ます。対して Kubernetes は、コンテナの運用を自動で行います。そのためコンテナ管理にかかるコストを削減できる可能性もあります。

移植性

Kubernetes はオープンソースソフトウェアであるため、ベンダーロックインされにくい傾向があります。例えば、オンプレミスの Kubernetes から Amazon EKS のように、運用のプラットフォームそのものを変更したい場合もあるでしょう。こうした場合でも、内部的な Kubernetes のバージョンが同じであれば、マニフェストを修正することなく、そのまま移植できる可能性は非常に高くなります。これは、クラウドベンダーをまたいだ乗り換えであっても同様です。

2.9.2 デメリット

サーバー台数の増加

メリットの説明で「Kubernetes はサーバー台数を削減できる可能性がある」と説明したばかりですが、耐障害性や高可用性を考慮した構成を取った場合、逆にサーバー台数が増加する可能性があります。例えば可用性を高めるため、3 つのアベイラビリティゾーンにまたがってシステムを運用することを考えてみましょう。アベイラビリティゾーンの障害を考慮して、コントロールプレーンとノードをそれぞれ 3 つのアベイラビリティゾーンに置きます。これだけですでに 6 台のサーバーが必要になってしまいます。物理マシンや仮想マシンであれば、アプリケーションを動作させるサーバーだけあればアプリケーションを動かすことができますが、Kubernetes はコントロールプレーンが必要である都合上、最小構成であっても多くのサーバーが必要になってしまいます。

頻繁なアップデート

Kubernetes は平均して 4 ヶ月に 1 回、つまり 1 年に 3 回ほど新しいバージョンがリリースされます。そしてサポート期間は、リリースから 12 ヶ月間です。高頻度にバージョンアップが行われるため、積極的に新しい機能を使いたい人にとってはメリットでもありますが、その都度新機能の学習や、廃止される API の確認にかかるコストが重いとも言えます。

また Kubernetes は、複数のバージョンをとばしたアップデートが保証されていません。通常、1 回のアップデートで上げられるバージョンは、現在使用中のバージョンの、次のバージョンだけです。例えば 1.26 から 1.28 へアップデートする場合は 1.26 → 1.27 → 1.28 のように、1 つずつアップデートしていく必要があります。離れたバージョンへ、一気にアップデートしたい場合は、メンテナンス作業にかかる時間も長くなるでしょう。

かかるコストとリターン

Kubernetes はコンテナの運用を自動化し、今まで人の手で行われていた作業の大半を削減してくれます。繰り返しになりますが、非常に強力で便利な反面、簡単に使えるツールではなく、すべての機能を使いこなすには相当な学習が必要となります。また、小規模なアプリケーションではオーバースペックとなりがちです。そのため導入や学習にかかるコストの重さをペイできず、別のツールやサービスを検討した方がよい可能性も十分にあります。AWS であれば Amazon ECS[*16]のように、より手軽にコンテナをホストできるサービスの利用を検討してもいいでしょう。これは Kubernete に限った話ではありませんが、アプリケーションの規模に応じて、最適なツールを使い分けることが肝心です。

2.10 実行環境の選択肢

ここまでで、コンテナを使った開発や、運用に使用できるツールについて説明してきました。では実際に運用をしていくにあたり、どのような環境で運用していけばよいのでしょうか。DevOps を活用して高速に開発サイクルを回していくにはどのような実行環境が適しているのか説明します。

[*16] https://aws.amazon.com/jp/ecs/

実行環境とは、アプリケーション本体や、その動作に必要な RDB、NoSQL といったソフトウェアを動かす環境のことです。実行環境には大きく、オンプレミスとパブリッククラウドに分けられます。

2.10.1 オンプレミスとは

インターネットなどのネットワーク越しに、クラウドベンダーが用意したサーバーやストレージといった IT リソースを利用できるクラウドに対し、サーバーやネットワーク機器などのハードウェア、ソフトウェアを自社で保有し、管理運用する形態を**オンプレミス**と呼びます。

クラウド登場以前は、IT システムを運用するには、必然的にこの形態しかありませんでした。そのため特に名前はつけられていなかったのですが、クラウドが登場したことにより、区別のため「オンプレミス」と命名されたといういきさつがあります。

なおオンプレミス環境上に構築する、**プライベートクラウド**というものも存在します。これは自社のサーバー上にクラウドと同等の環境を構築するものですが、ここではプライベートクラウドについては触れず、従来型のオンプレミスについてメリット・デメリットを説明します。

2.10.2 オンプレミスのメリット

カスタマイズ

オンプレミスは自社で構築を行うため、ハードウェアの選定レベルから、自社の要望にあったカスタマイズが可能です。また自社内にサーバーがあるため、アプリケーションの連携をはじめ、思い通りの設定ができます。

セキュリティ

自社で管理しているハードウェアですから、自社のセキュリティポリシーを適用できます。外部からオンプレミス環境に、一切アクセスさせないような設定も可能です。

2.10.3 オンプレミスのデメリット

運用コスト

オンプレミスでは、ハードウェアを自社で購入し、設置や設定をしなければなりません。サーバーを設置するには、データセンターや専用のサーバールームのように、セキュリティが担保され、空調のある環境が必要になります。電源やネットワークなども考慮する必要があるでしょう。不審者が侵入しないようなセキュリティ体制の確保も必要ですし、停電や火災への備えも大切です。当然ですが、トラブル時の対処も自社で行う必要があります。こういった点で、初期費用や運用保守を行なう人員の確保にコストがかかってきます。

スケールのしにくさ

サーバーのハードウェア選定においては、そのサーバーに求められる性能を明確にしなくてはなりません。そのためには、実行したいアプリケーションに必要なリソースを正確に見積る必要があります。仮に見積りよりもリソースを消費するようであれば、追加のサーバーの購入が必要となるでしょう。逆に見積りよりもリソースを消費しなければ、購入したサーバーのリソースが無駄になります。サーバーの代金は、使用されず遊んでいるリソース分にもかかっているため、できるだけ無駄なく使い切りたいところです。ですが本当に無駄なく使い切ってしまうと、突発的な負荷に耐えられなくなってしまいます。オンプレミスのサーバーの性能は固定されているため、「余力を持つ」と「無駄を省く」を両立することは、本質的に不可能です。

それでも実際にサーバーを発注するとして、実際に稼動を始められるのはいつでしょうか。サーバーは今日明日で調達できるものではありません。早くても数週間、場合によっては数ヶ月のオーダーで時

間がかかってしまうでしょう。機材調達にかかる時間を考えると、気軽にサーバー追加やスペックアップができません。

2.10.4 クラウドのメリット

クラウドのメリット・デメリットは、おおむねオンプレミスのメリット・デメリットと表裏一体となっています。

コスト

クラウドはサーバーやその他のサービスを使った分だけ料金を支払います。マシンリソースが足りない時や多すぎる場合にはマシンスペックを変更するなどして調整できます。新しくサーバーを追加することも容易です。サーバー調達が容易になったため使い捨ての様に使用できます。

自社にサーバーを置く必要がなくなったため、サーバールームの様な設備や、それをメンテナンスする人員も不要になります。人や設備にかかっていたコストをサーバー代金に当てることができます。

機材調達のスピード

クラウドは欲しいリソースを即座に用意できます。オンプレミスであれば数週間以上の時間をかけてサーバーを調達しますが、クラウドはクリック一つでサーバーを用意できます。

2.10.5 クラウドのデメリット

コスト増加の可能性

クラウドは使った分だけ料金を支払います。これはメリットであると同時に、デメリットでもあります。クラウドは必要になった時に、オンデマンドにリソースを調達することができます。そのためオンプレミスのように、不要かもしれないサーバーをあらかじめ購入しておく必要はありません。オンプレミスと異なり、実際に運用をはじめてから、負荷を見つつサーバー構成を変更することも容易です。これが使った分だけの従量課金と非常に相性がよく、無駄を省いたコストの最適化がしやすいのです。

ただし一般的なクラウドは、使えば使うほど料金がかかる青天井方式です。誤って大きすぎるスペックのサーバーを用意してしまったり、クラウドに最適化されていない構成を作ってしまい、無駄な料金を発生させてしまう可能性もあります。使い終わった不要なリソースを消し忘れて、無駄に料金を払い続けてしまったというケースもよくあります。また不正アクセスにより、IT リソースを大量に作成されてしまったといったインシデントも、よく耳にします。

コストの動向は常に監視し、見積り以上のコストが発生した場合はアラートを出すといった対策も必要です。

学習コスト

これはどのツールを使っても同じことが言えるのですが、クラウドを扱うにも学習が必要です。よくある例として、クラウドをオンプレミスのように使用しているケースがあります。たしかにクラウド化するだけでも、運用コストを削減できるというメリットはあります。ですがオンプレミスの構成を、そのままクラウド上にコピーしただけでは、クラウドを使う旨味がありません。クラウドのメリットを最大限に引き出すためには、クラウド特有の仕様や、その操作方法などに習熟することは不可欠です。

2.11 クラウドネイティブという考え方

近年、**クラウドネイティブ**という言葉をよく耳にします。クラウドネイティブとは、**パブリッククラウドやプライベートクラウドで運用することを前提としたシステムや、その考え方**を指します[*17]。

[*17] https://github.com/cncf/toc/blob/main/DEFINITION.md#%E6%97%A5%E6%9C%AC%E8%AA%9E%E7%89%88

なぜ今クラウドネイティブが注目されているのでしょうか。クラウドネイティブが注目されている背景について説明します。

2.11.1 販売形態の変化

現代のビジネスでは、なによりも顧客のニーズに対応していく速度が重要視されています。これにはサービスの販売形態の変化が関係しています。

近年、サービスの主流は従来の買い切り型から、サブスクリプション方式に変化しつつあります。従来の買い切り型は、一度リリースしてしまえば、新機能の追加などは行われません。新機能は次のバージョンに含まれ、顧客はそれを楽しみに待っている状態です。フィードバックに対し、リアルタイムで応える必要はありません。

一方でサブスクリプション方式では、日々顧客からフィードバックを受け、新機能の追加や不具合の修正が行われます。これは継続的に料金を支払ってもらうためにも欠かせません。長期間に渡って不具合や欠点が放置されているようなサービスを、誰が来月以降も使い続けようと考えるでしょうか。

またサブスクリプション方式では、サービスを利用する顧客数が流動的です。ある新機能のリリースで爆発的に人気が出ることもあれば、ほんのささいなきっかけで、別のサービスに移ってしまうこともありえます。そのためにも、常に顧客のニーズに目を光らせ、迅速に対応してく必要があるのです。

2.11.2 アジャイルと DevOps

顧客から受けたフィードバックや、顧客数の増減に素早く対応するためには、それを実現するための開発体制が必要不可欠です。そこで従来のウォーターフォール開発にかわって注目されているのが、**アジャイル開発**や DevOps です。

アジャイル開発では優先度の高い機能から開発し、できたものからリリースします。高頻度にリリースを繰り返すことで、顧客に早く機能を提供し、フィードバックの機会を増やすのが狙いです。そして高頻度にリリースを繰り返すためには、テストの自動化やデプロイの安定性が重要となってきます。そこで利用されるのが DevOps の仕組みというわけです。

DevOps を実践することで、効率化や自動化、デプロイの安定性が実現できます。そしてこうした DevOps を支えている技術がコンテナであり、Infrastructure as Code(IaC) による環境構築の自動化です。

2.11.3 クラウドの必要性

アジャイル開発や DevOps は、あくまでも開発運用をしていくための考え方に過ぎません。ですがこれらの考え方は、オンプレミスよりもクラウドとの相性がよいとされています。

前述の通り、オンプレミスはサーバーの調達に時間がかかります。つまり本質的に、スケールアウトや構成変更が気軽にできないという問題を抱えているわけです。そのため必然的に、一度作った環境をサービスが終了するまで、もしくはハードウェアが故障するまでなど、長期間使い続けることになります。対してクラウドは、必要な時に必要なだけ即座にサーバーを調達できます。そのため気軽に構成変更ができますし、同じ環境一式のコピーを、別途用意することも簡単です。

サービスの販売形態が変化しつつあることで、従来のウォーターフォール開発では顧客のニーズに**即座に**応えることができません。繰り返しになりますが、顧客のニーズを迅速にキャッチアップするためには、アジャイル開発が向いています。そしてアジャイル開発を実践するためには DevOps 的な考え方が必要で、その DevOps を実践するためには、柔軟にインフラを構成できるクラウドが必要不可欠というわけです。

もちろん、オンプレミスにはオンプレミスのいいところがあります。クラウドからオンプレミスに回帰したサービスも、世の中には数多く存在します。どういったサービスを運用するのか、誰が使用するのかという点を考慮し、実行環境を選択するのが重要です。

第 3 章　バージョン管理システムと開発環境

現代のソフトウェア開発では、複数人による並行開発と高速なリリースサイクルを実現するため、バージョン管理システムと適切な開発環境が不可欠です。この章では、Git を中心としたバージョン管理の基本から、統合開発環境やコンテナ技術を活用した効率的な開発ツール群まで、DevOps 実践の土台となる技術基盤について解説します。

3.1 アプリケーション開発に求められるツール

DevOps に限らず、アプリケーションを開発していくために必要不可欠なツールの 1 つに、**バージョン管理システム**があります。

アプリケーションを手戻りなく、開発した順番にリリースできれば理想的ですが、そんなことは現実的にあり得ません。システムやアプリケーションへのニーズは多様で変化が激しく、そしてより高速にリリースすることが求められています。そのため開発現場においては、急に方針転換を求められたり、不具合や緊急対応といった理由で、開発中の機能に割り込んで取り組むタスクが発生したりすることも珍しくありません。こうした変化にも、臨機応変に対応することが求められます。

アプリケーションはソースコードから構成されています。**アプリケーション開発とは、突き詰めればコードを新規に書き足したり、あるいは問題のあるコードを修正したりする作業の連続**に他なりません。そして前述の通り開発現場では、方針転換や、緊急に実装するタスクの発生は日常茶飯事です。こうしたコードの変更作業を、複数人が、無秩序に行ったらどうなるでしょうか。ソースコードが、誰も全貌を把握できない魔窟になるまでに、それほど長い時間は必要ないでしょう。

こうしたカオスからソースコードを守り、秩序ある状態を保つために必須なのが、バージョン管理システムです。バージョン管理システムを導入することで、安全に複数人による並行開発を進められます。

ソースコードはバージョン管理システムによって管理されますが、実際にコードを書くのは、開発者の手元にある開発ツールです。そして当然ですが、書いたコードは実行しなければなりません。そのために必要となるのが実行環境です。

3.2 バージョン管理システムとは

ファイルを「誰が」「いつ」「どのように」変更したのかという情報を記録し、後から辿れるようにするのがバージョン管理システムの役割です。ファイルをバージョン管理下に置くことで、変更履歴の可視化や、過去の任意の状態への復元が、簡単かつ確実に行えるようになります。

もしもファイルがバージョン管理されていなかったり、あるいはバージョン管理を人力に頼っていると、「どの状態が最新なのかわからない」「なぜこの変更が加えられいるのかわからない」「ファイルを壊してしまったので復元したいけれど、元の正しい状態がわからない」といったことになりがちです。これがアプリケーションのソースコードであれば、「バグのある古いバージョンをリリースしてしまった」「バグを直したつもりが直っていなかった」といったトラブルにも繋がります。

バージョン管理システムには様々な実装が存在しますが、現在もっとも広く利用されているのが **Git** です。少なくとも Web 開発やオープンソースソフトウェア開発といった領域では、ほぼ Git 一択と言ってよいほど、支配的な存在となっています。

3.3 バージョン管理システムの分類

バージョン管理システムは、その設計上、大きく**集中型**と**分散型**に二分されます。

3.3.1 集中型

バージョン管理システムにおいて、ソースコードを保存する領域を**リポジトリ**と呼びます。集中型とは、ソースコードをホスティングするサーバーを用意し、そこを唯一のリポジトリとする、いわゆるクライアント・サーバー方式です。ユーザーはネットワークを介して中央リポジトリと通信し、ファイルの取得や、変更の送信を行います。

集中型のバージョン管理システムの代表格として挙げられるのが、**Subversion（SVN）**です。実際に Git 登場以前は、Subversion が広く利用されていました。Subversion は構造がシンプルでわかりやすいものの、集中型バージョン管理システムの宿命として、常に中央リポジトリと通信しないと作業が行えないというデメリットがあります。そのためこの欠点を克服し、作業する時と場所を選ばない、Git を代表する分散型のアーキテクチャへのシフトが進みました。例えば GitHub は当初、Subversion プロトコルもサポートしていましたが、2024 年現在、このサポートは終了しています。

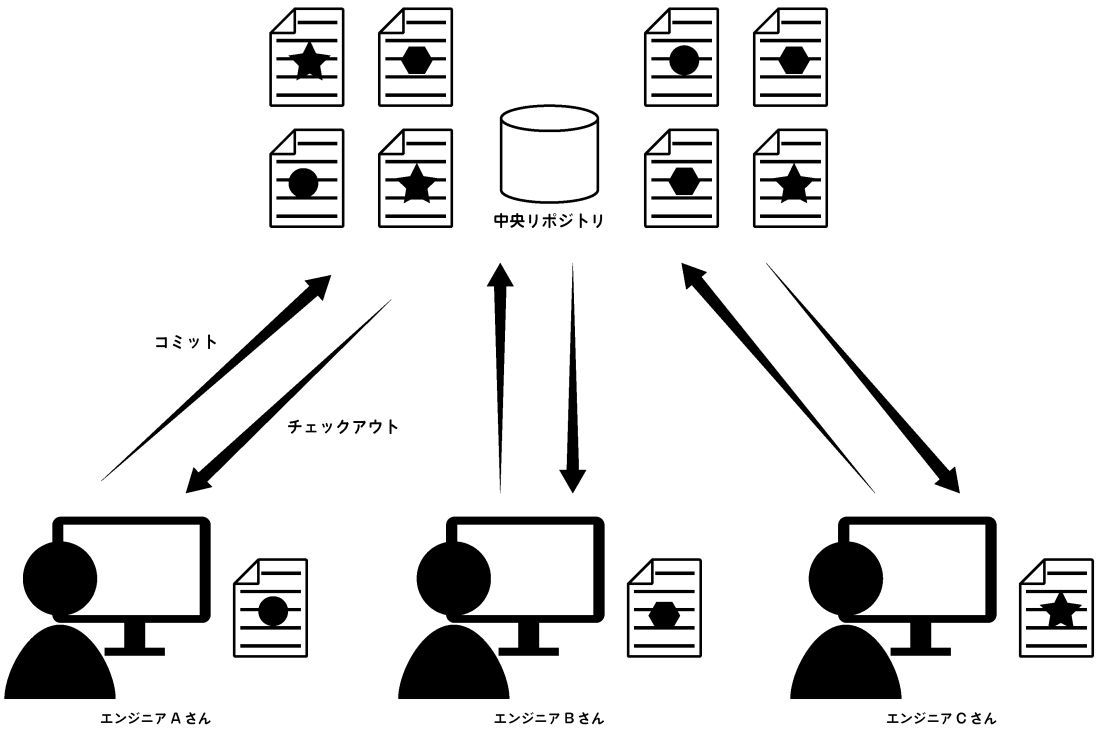

図 3.1: 集中型のバージョン管理

3.3.2 分散型

中央のサーバーが唯一のリポジトリを持つ集中型に対し、各クライアントがリポジトリの完全なコピーを持つ方式を分散型と呼びます。ネットワークを介して開発メンバーと変更を共有するためのリポジトリをリモートリポジトリと呼び、開発メンバーごとのローカルマシン上に構築されるのはローカルリポジトリです。

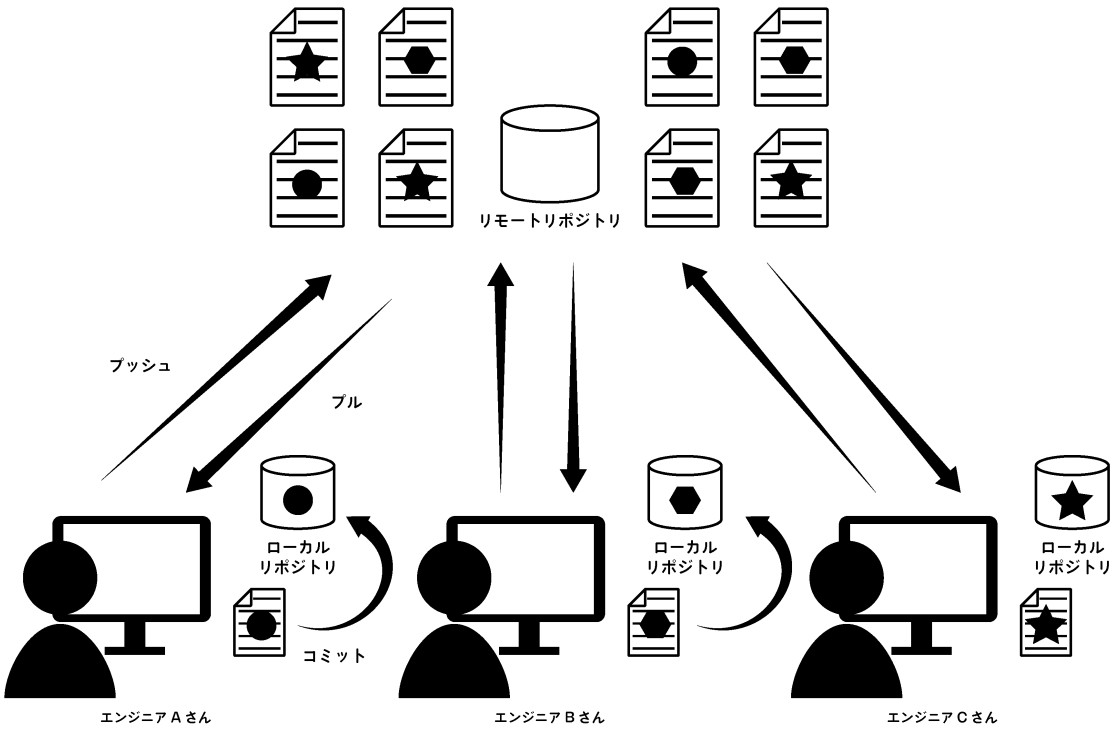

図 3.2: 分散型のバージョン管理

分散型であってもリモートリポジトリを経由して開発メンバーと変更を共有するため、ネットワークに依存することは変わりません。しかし、ローカルリポジトリがあることで、リモートリポジトリに接続できない状態であったとしても、バージョン管理が行えます。またローカルリポジトリで行った変更は、明示的にリモートリポジトリに送り込まない限り、自分以外のユーザーに影響を与えることはありません。そのため**変更を手元で試してみて、上手く行ったら共有する。失敗したら削除して、なかったことにする**という作業を、他の開発者に見せることなく行えます。このように破壊的な変更も気軽に試せるのが、分散型のメリットです。

そして分散型バージョン管理システムの代表格と言えるのが、この章で紹介する Git というわけです。

3.4 Git を使ったチーム開発

ここからは分散型バージョン管理システムの代表格である Git を使って、実際にチーム開発をしてくために必要なことについて学んでいきましょう。Git は、主にソースコードの変更履歴を管理する分散バージョン管理システムです。もともと Linux カーネルのソースコードを管理することを目的として、Linux の生みの親でもある、リーナス・トーバルズ氏によって開発されました。

Git を扱うためには、まず Git のクライアントツールをインストールする必要がありますが、それだけでは足りません。分散型は各々が手元にリポジトリのコピーを持つと述べましたが、チーム内で変更内容を共有するためには、開発の中心となるリモートリポジトリが必要になるためです。この点に限って言えば、分散型も集中型も同じであると言えます。

リモートリポジトリは、ネットワーク経由でアクセスできるバージョン管理サーバー上に構築します。サーバーは独自に構築してもよいですが、初期開発コストや運用コストを考えると、最初はソー

スコードホスティング用の SaaS を使うことをお勧めします。そして世界的に広く利用されている Git ベースのソースコードホスティングサービスが、**GitHub**[*1]です。

3.4.1 GitHub とは

GitHub は、Git をベースとしたソースコードホスティングサービスです。GitHub は買収により、現在では Microsoft 傘下のサービスとなっています。

GitHub は単なる Git のリモートリポジトリではありません。例えばリモートリポジトリに変更を取り込む前に、変更された内容をレビューできる**プルリクエスト**をはじめとした便利な機能が、数多く用意されています。これからの機能を活用することで、クリーンなコードを維持しつつ、効率よく開発を進められます。

プルリクエスト

プルリクエストは GitHub が持つ機能の中でも、特に重要なもののひとつです。GitHub を使用してチームで開発している人であれば、一度は使ったことがあるのではないでしょうか。プルリクエストを活用することで、別ブランチで行った修正を、リリースブランチにマージすることなく、安全に「提案」することができます。これによってリリースブランチに意図しない変更が行われることを防止しつつ、効率よくレビューを行うことができます。変更を提案する側としても、副作用を恐れず、気軽に提案を行えるようになります。

ソースコード単位でのコメント

プルリクエストに対するレビューでは、ソースコードの行単位で、直接コメントをつけることができます。行ごとのパーマリンクを別の場所に貼り付けて議論するよりも、GitHub 上で作業が完結するため、見通しがよく、レビューの効率が向上します。

承認

レビュアーが承認していないプルリクエストのマージをブロックできます。この機能を活用することで、正しいレビュープロセスを経由していない変更が、勝手にマージされることを防げます。こうすることで、マージされる変更は、必ず誰かのレビューを通していることを担保できます。

変更の提案

ソースコードの問題のある箇所にレビュアーがコメントをつけ、修正そのものはレビュイーがやり直すのが一般的なレビューでしょう。これに対し、レビュアーが最終的なソースコードの修正案を記述し、レビュイーに提案ができる機能もあります。コメントだけでは思ったように意図が伝わらず、何度も修正のやりとりを繰り返すといった手間を減らせます。

ここで挙げた機能以外にも GitHub には様々な便利機能があります。詳しくは公式のドキュメント[*2]を参照しつつ、実際に使ってみましょう。

3.5 Git を便利に使うためのツール

Git は CLI のツールです。ですが、ターミナルにコマンドを入力することが、あまり得意ではない方もいるのではないでしょうか。また公式の Git コマンドは非常にプリミティブであり、使うには少々慣れが必要です。そこでより直感的に使えるグラフィカルなツールを使うのもお勧めです。

[*1] https://github.com
[*2] https://docs.github.com/ja

Git のドキュメントサイト[*3]にそれぞれの OS ごとにサポートされているサードパーティ製のツールがまとめて紹介されていますので、気になる方はそちらをご覧ください。その中からよく知られているツールを 2 つピックアップして簡単にご紹介します。

GitHub Desktop

GitHub Desktop[*4]は、GitHub 社が提供している公式のデスクトップツールです。GitHub の提供ということもあり、GitHub との連携の相性は抜群です。

SourceTree

SourceTree[*5]は、アトラシアン（Atlassian）社が提供しているデスクトップツールです。GitHub Desktop に比べて、機能は豊富な印象があります。とはいえ、ツールは必ずしも大は小を兼ねません。シンプルなツールの方がよいケースもありますので、実際に使ってみて、使い勝手が良さそうな方を選択するのがよいでしょう。

3.6 Git の基本操作

ここからは Git をまだ触ったことがない方向けに、基本的なコマンド操作について解説します。ここでは、Git コマンドのインストール方法については省略しています。また実行環境は macOS を前提に解説します。Windows/Linux 環境を使う場合は、適宜読み替えてください。

3.6.1 Git を使った開発の始め方

Git を使った開発の始め方は、大きく以下の 2 パターンに分けられます。

1. GitHub などに既に存在するリモートリポジトリから、プロジェクトをクローンする
2. ローカルマシン上で Git プロジェクトを新規に作成する

それぞれのパターンごとに、具体的な手順を解説しましょう。

3.6.2 既存のリモートリポジトリから、プロジェクトをクローンする

既に存在するプロジェクトに、途中から参加する場合はこのパターンになります。この場合、まずプロジェクトの責任者や他の開発メンバーから、リポジトリにアクセスするための情報を教えてもらいましょう。例えば GitHub 上にあるリポジトリであれば、リポジトリの URL が必要です。またプライベートリポジトリの場合は、別途アクセス権限の付与も必要となるでしょう。その上で、以下のコマンドを実行します。

```
$ git clone <リポジトリ URL>
$ cd <リポジトリ名>
```

`git clone` コマンドを実行すると、リポジトリと同名のディレクトリが作成され、その中にリポジトリの内容が複製されます。ディレクトリが作成されたら、、cd コマンドでカレントディレクトリを変更しましょう。これで開発の準備は終了です。

なおリポジトリの情報は、プロジェクトのディレクトリ内の**.git** ディレクトリに集約されています。このディレクトリは、手でいじったり、削除しないようにしましょう。

[*3] https://git-scm.com/downloads/guis/
[*4] https://desktop.github.com/
[*5] https://www.sourcetreeapp.com/

3.6.3 ローカルマシン上で Git プロジェクトを新規に作成する

新規プロジェクトを立ち上げる場合は、このパターンになります。任意のディレクトリを用意してから、Git の初期化コマンドを実行します。

```
$ mkdir git-sandbox && cd git-sandbox
$ git init
```

`git init` コマンドを実行すると、このディレクトリがバージョン管理下に置かれます。これで、ローカルマシン上に閉じているとはいえ、Git の機能が利用できるようになりました。

最初から最後まで、自分のローカルマシン上だけで開発を行うということはないでしょう。他のメンバーとファイルを共有する場合はもちろん、個人開発プロジェクトであっても、万が一に備えてリモートのサーバーに中央リポジトリを用意するべきです。そこでローカルで初期化したリポジトリに、リモートリポジトリを追加しましょう。ここではリモートリポジトリが、GitHub 上に事前に用意できているものとします。

```
$ git remote add origin <リポジトリ URL>
```

リモートリポジトリが正しく追加できたかを確認するためには、次のコマンドを実行してください。

```
$ git remote -vv
```

正しく追加されている場合には、リモートリポジトリの内容が表示されます。

3.6.4 リモートリポジトリに変更を反映する

前述の 1 と 2 のどちらのパターンであっても、開発が進んだら、その内容を中央リポジトリへ送り込む必要があります。それでは実際にソースコードをバージョン管理下に追加してから、リモートリポジトリへ変更を反映してみましょう。ここではサンプルとして、テキストファイルを用意します。

```
$ echo “console.log(‘Hello, World!!‘)“ > hello.txt
```

変更した内容は**コミット**する必要があります。Git において変更内容をコミットする際には、まず**ステージング領域**へ、コミットしたいファイルを追加する必要があります。このあたりは Git 特有のお作法のため、仕組みが理解できるまでは少し難しく感じるかもしれません。

次のコマンドを実行すると、カレントディレクトリ以下の変更すべてを、ステージング領域へ追加します。

```
$ git add .
```

ステージング領域に変更を追加できたら、あらためてコミットを実行します。なおコミットごとに**コミットメッセージ**と呼ばれる覚え書きを記述する必要があります。`git commit` コマンドを実行すると、デフォルトのテキストエディタが起動し、コミットメッセージの記入を促されますが、ここでは「-m」オプションで、直接コミットメッセージを指定して、その手続きを省略しています。ちなみにコミットメッセージには**なぜそのような変更を行ったのか**という、変更の意図を記述するよう心がけてください[*6]。

[*6] ちなみに例に挙げた「hello.txt を追加した」というようなコミットメッセージは本来書く意味はなく、悪い例となります。なぜならバージョン管理システムのコミットの差分を見れば自明なためです。

```
$ git commit -m 'hello.txt を追加'
```

コミットができたら次のコマンドを実行して、リモートリポジトリに変更内容を反映します。

```
$ git push
```

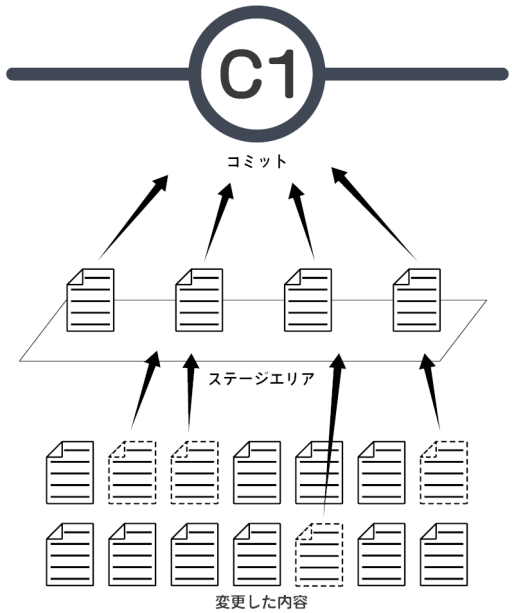

図 3.3: ステージングとコミットのイメージ

3.6.5 リモートリポジトリから変更内容を取得

チームで開発を行っていると、他のメンバーが行った変更が、どんどんリモートリポジトリに追加されていきます。それらの変更は、明示的に取得操作をしない限り、自分のローカルリポジトリには反映されません。つまり自分のローカルリポジトリが、どんどん時代遅れになってしまうわけです。実際のリモートリポジトリとの差異が大きい状態で、新しい変更を加えるのは得策ではありません。ファイルの内容に矛盾が生じ、コンフリクトが発生してしまう可能性が大きくなります。そこでローカルリポジトリは、常に最新のリモートリポジトリに追従するべきです。

リモートリポジトリの変更を、ローカルリポジトリに反映させるためには、次のコマンドを実行します。

```
$ git pull
```

自分一人で開発を行っている場合は、当然他の変更が追加されることがないため、実行しても意味がわかりにくいコマンドかもしれません。もし友人や同僚など、開発を手伝ってくれる方がいる場合は、お互いに変更をリモートリポジトリにプッシュして、相手が送り込んだ変更が取得できるか、試してみてください。

これが Git の基本的なフローになります。もちろん Git にはこれ以外にも沢山の機能があるため、実践を通して色々なテクニックを身につけてください。

3.7 開発サイクルを回していく中で役立つ開発ツール

3.7.1 開発サイクルとは

企画立案してから要件定義、設計、実装、リリースまでの流れを継続的に繰り返していく一連の流れを**開発サイクル**と呼びます。ビジネス環境やユーザニーズは、絶えず変化しています。そのため開発も、アプリケーションを 1 回リリースしただけで終わることは稀で、こうした変化をキャッチアップし、素早くシステムに適用していくことが求められています。

最近は**不確実性時代（VUCA）**とも呼ばれるようになり、開発スピードの高速化や、変化への柔軟性がより高いレベルで求められるようになってきています。このような時代だからこそ、開発の効率化や省人化が重要です。開発サイクルの中でも特に、**実装**、**テスト**、**リリース**は、非常に多くの時間と労力を費やすところであり、ボトルネックになりやすい箇所でもあります。開発効率を高めていくためには、開発を支援するツールの導入と、その使いこなしが重要です。どのように開発ツールを活用していくとよいのかについて、基本から学んでいきましょう。

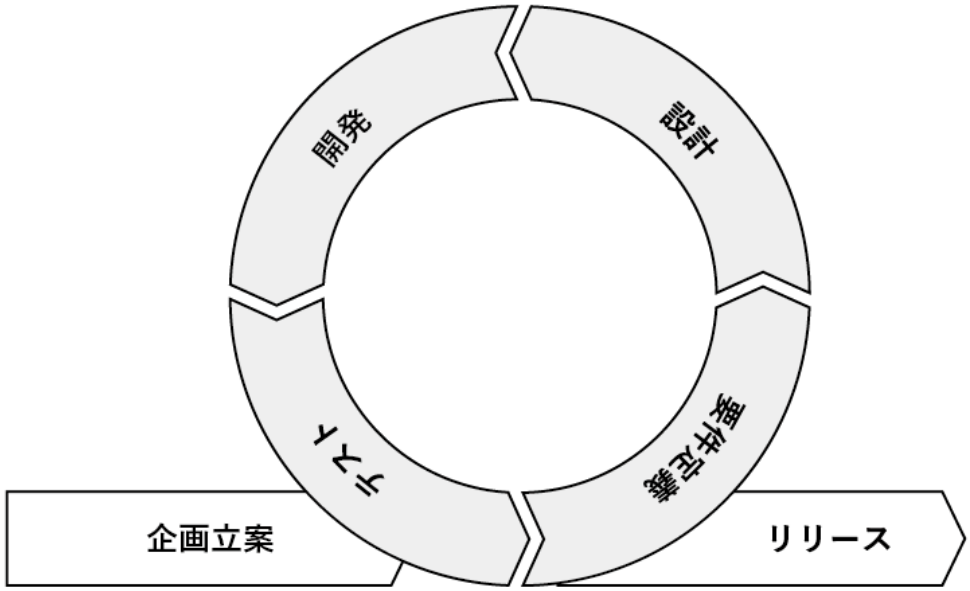

図: 開発サイクル

3.7.2 開発ツールとは

開発ツールとは、開発を助け、効率化するためのツールの総称です。そのためコードの入力を補完するものから、テストや静的解析を自動化するものまで様々なものが存在します。例えば前述の Git はもちろん、サービスとしての GitHub も、広義には開発ツールの 1 つです。そして開発においては、こうした数々のツールを、用途や状況に応じて使い分ける必要があります。ですがそれは非効率な上、数多くのツールをインストールしなくてはならないため、開発環境のセットアップにも時間がかかります。そこで開発に必要なツールをひとまとめにした、**統合開発環境（IDE）**もよく使われています。

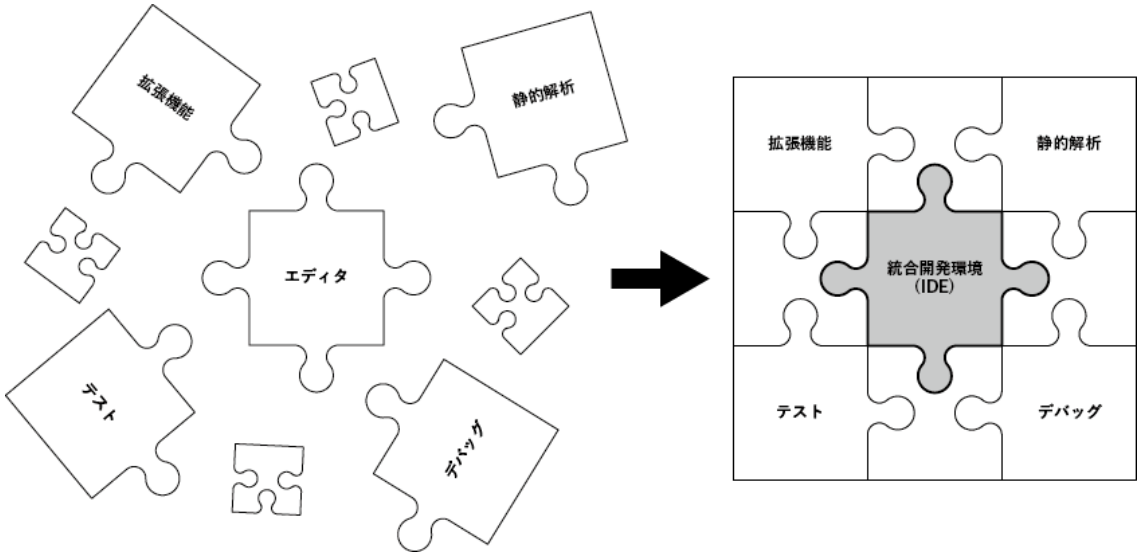

図: 様々な開発ツールの機能をひとまとめにしたのが統合開発環境

ここではよく使われる、主要な開発ツールを紹介します。

3.7.3 コードエディタ

テキストエディタ

エディタとは、なにかを編集するためのソフトウェア全般を指す言葉です。エディタと呼ばれるソフトウェアは、非常に多岐にわたります。例えば動画を編集するアプリは動画エディタですし、画像を編集するアプリは画像エディタです。プログラムのコードを編集するアプリは、**コードエディタ**とも呼ばれます。

そしてコードエディタとしてよく利用されるのが、テキスト編集に特化したエディタである**テキストエディタ**です。テキストエディタには、Windows のメモ帳のように、グラフィカルインターフェース（GUI）を持つものの他に、CLI で主に使うものや、その両方のインターフェイスを持つものがあります。例えば、非常に長い歴史と人気を持つ **vi** や **Emacs** といったテキストエディタは、CLI ベースで使うことも、GUI でマウスとキーボードから操作することもできるエディタの代表格と言えるでしょう。

メモ帳を見てもわかる通り、テキストエディタはテキスト入力に特化していることから、その多くがシンプルかつ軽量なソフトウェアです。学習コストが低く、軽量であることは大きなメリットではありますが、プログラミングの観点から見ると、機能的に不足している部分があることも否定できません。開発現場では、複数のファイルを同時に参照し、クラス化や関数化、ファイル分割などを行いながら、膨大な量のコードを書くことを求められることも珍しくありません。このように複雑な近年の開発スタイルにおいては、後述する統合開発環境を利用するケースも増えてきています。

とはいえ、統合開発環境があれば、テキストエディタが不要というわけではありません。運用の現場では、CLI でサーバーにリモートログインし、そこで設定ファイルを書き換えなければならないといった事態はよくあります。こうした時は、CLI で使える軽量なテキストエデイタが、何よりも頼りになります。そのため vi のようなテキストエディタの基本的な操作は、**エンジニアの嗜み**として身に付けておくべきです。

エディタは有名なものだけに絞っても、膨大な種類が存在します。またプログラミング向けのエディタは、プラグインを追加して自由にカスタマイズできるものも多く、非常に個人の好みが強く出るソフトウェアでもあります。ぜひ様々なエディタを試し、1 番しっくりくるエディタを探してみましょう。

また自分向けにカスタマイズされたエディタ環境は、よく研がれたナイフのようなものです。開発において、非常に頼れる相棒となってくれることでしょう。

オンラインエディタ

テキストエディタの中には、Web ブラウザ上で動作するタイプのものも存在します。こうしたエディタは**オンラインエディタ**とも呼ばれます。例えば **AWS Cloud9**[*7]や、**CodePen**[*8]、**Visual Studio Code for the Web**[*9]などが表的なオンラインエディタです。オンラインエディタはブラウザ上で動作するため、インストールや設定などの手間がなく、すぐに利用できることが特徴です。またオンラインエディタはネットワークを介して通信するため、通信環境に依存します。ネットワークに接続できない環境では利用できないことはもちろん、通信環境によっては動作が重くなることもあります。

オンラインエディタはどうしても、機能面や速度面で見劣りすることは否めません。ですがセキュリティポリシー上の理由で、ローカル環境に好みのエディタのインストールが許されないといったケースでは、こうしたオンラインエディタが活用できるかもしれません。

統合開発環境（IDE）

統合開発環境（Integrated Development Environment: IDE）とは、開発サイクルを回していく中で必要なツールをひとまとめにしたソフトウェアのことです。ツールにはテキストエディタを中心に、プロジェクト内のファイルにアクセスするための**ファイラー**、コマンドを実行するための**ターミナル**、プログラムをデバッグする**デバッガー**、テストを行う**テストツール**などが、文字通り**統合**されています。IDE には機能を拡張するための、プラグインや拡張といった仕組みが備えられているため、必要に応じて機能を追加できます。最近では SaaS との連携が可能なプラグインも増えてきており、多様化・複雑化によって必要なツールが増え続ける傾向にある近年の開発スタイルにおいて、ツールの集約先としても注目され始めています。

3.7.4 静的解析ツール

静的解析ツールとは、ソースコードを解析して問題を発見するツールのことです。実際にアプリケーションは実行せず、ソースコードのみから解析を行うため、**静的**と呼ばれます。静的解析から得られるメリットは様々ですが、特に**問題発見までの時間短縮**、**セキュリティ**、**保守性**の 3 つの観点から考えてみましょう。

問題発見までの時間短縮

プログラミング言語のソースコードとは、プログラミングをする人間にとって扱いやすいように作られています。これは言いかえると、コンピューターにとっては、必ずしも都合がよいわけではないということです。コンピューターはソースコードをそのまま解釈して実行することはできず、コンパイルやビルドといった工程を経て、コンピューターが実行可能な形式に変換します。

使用しているプログラミング言語や、プログラムの規模によっては、このコンパイルやビルドに時間を要することもあります。ですがコードを修正する度に、時間をかけてビルドを行って確認していたのでは、デバッグにいくら時間があっても足りません。そこで静的解析ツールを用い、ビルド前に可能な限りの問題を解消しておけば、デバッグの効率を上げることができます。

セキュリティ

セキュリティという観点でも様々なツールが提供されています。セキュリティの世界は高度で複雑であるため、このツールだけ使用していれば完璧というものではなく、様々な観点から対策する必要があります。

[*7] https://aws.amazon.com/jp/cloud9/
[*8] https://codepen.io/
[*9] https://code.visualstudio.com/docs/setup/vscode-web

現在、アプリケーションの機能をすべて自前で実装することは、ほぼないでしょう。通常は第三者によって提供されている、様々なライブラリを利用することになります。ですが利用しているライブラリに脆弱性があると、当然アプリケーションに脆弱性を埋め込んでしまうことになります。そこで依存しているライブラリを調べ、脆弱性がないかをスキャンすることは非常に重要です。例えば、GitHub が提供する **Dependabot**[*10]は、GitHub ユーザーであれば導入しやすいツールの 1 つです。

保守性

保守性とは、前述のふたつとは異なり、システムやサービスに直接致命的な問題を引き起こすようなものでありません。ですが継続的に開発を進めていく上で、徐々に効いいてくるジャブのようなものです。

保守性に影響を与える具体的な例を挙げると、ソースコードのフォーマットがあります。ソースコードを各々の開発者が好き勝手に書いてしまうと、プロジェクト全体としてソースコードの可読性を大きく損い、最終的には保守性の低下を招いてしまいます。可読性の低いコードは、バグが生まれる温床ともなってしまうでしょう。そこで「1 行は 80 文字以内にする」「変数名はスネークケース、メソッド名はパスカルケースに統一する」「インデントは半角スペース 4 文字とする」など、コードを書く時にはルールを徹底するのが基本です。これを**コーディング規約**と呼びます。静的解析ツールを使えば、そのコードが規約に沿っているかを自動的にチェックできます。

3.7.5 テストツール

システムの開発において、バグの発生は避けては通れません。当然ですが、サービスインの後にバグが発生すると、ユーザーに影響が出てしまいます。バグの影響範囲によっては、ビジネス全体に影響が出る可能性もあります。

事前にバグを発見するために、重要なのが**テスト**です。テストには、手動で行うテストと自動で行うテストがあります。手動で行うテストは、人間が実際に手で操作するため、実行頻度が少なく、テストの網羅性が低くなりやすく、特にリグレッションテストに関しては疎かになりやすいです。これではテストとしての意味をなしません。そこで活用したいのが、テストツールによるテスト自動化です。

テストツールとは、アプリケーションのコード内にテストコードを記述して、テストを自動化するためのツールです。テストコードが記述されていれば、例えばコーディング中であっても、ファイルの保存をトリガーに、自動でリグレッションテスト実行することもできます。こうすれば、ソースコードに何か問題があったとしても、その場で気づくことができるでしょう。QA などの後工程でバグが発覚した場合、バグレポートを受けてから、該当箇所を調査するという、大きな手戻りが発生します。ですが実装中に、プログラマー自身がその場で問題に気づけることで、調査や解析にかかる手間を大きく減らすことに繋がります。

テストツールは、テストの種類によって様々です。そしてテストツールをどのように選択してどこから始めていくのかは、とても奥が深い世界です。数あるテストの中でも、まずは単体テストと受入テストについて考えてみましょう。

単体テスト

単体テストは、プログラム内の関数やモジュールなど、機能ごとの動作を確認するためのテストです。ユニットテストとも呼ばれます。ソースコード内にテスト用のコードを記述し、テストコードの実行結果を確認することで、プログラムが仕様通りに動いていることを確認します。

ソースコード内に記述する都合上、プログラミング言語ごとにテストツールは異なります。例えば TypeScript で開発をしている際には、Jest[*11]と呼ばれるテストフレームワークを使用します。

[*10] https://docs.github.com/ja/code-security/dependabot/working-with-dependabot
[*11] https://jestjs.io/

受入テスト

受入テストは、システムやアプリケーション開発を依頼した側が、要求した通りに実装されているかを確認するテストです。画面があるアプリケーションであれば、実際に画面を操作しながらテストをすることになります。ですが実際にテストを行う際には、入力やクリックなど単調な操作が多く、作業に時間がかかったり、繰り返し同じ操作をし続けることで、人的なミスが発生することもあります。そこで自動化ツールを活用し、効率化を進めていくことになります。特に最近では、RPA など自動化ツールの多様化により、選択肢が増えてきています。

受入テストツールは大きく、プログラミング型とローコード/ノーコード型に分けられます。プログラミング型は、通常のプログラミングと同じように、実装さえ行えばあらゆる自動テストを実現できます。しかし、実装にはプログラミングの知識が必要になるため、テストコードを書けるエンジニアを確保する必要があります。代表的なツールとして、Selenium、Cypress などがあります。

一方、ローコード/ノーコード型は、画面上でシナリオを入力したり、ブロック上の命令を積み木のように組み上げながらテストを記述できます。プログラミングの知識が高くない、非エンジニアの人員でもツールを使ったテスト自動化ができます。代表的なものに UiPath[*12]や Autify[*13]などがあります。

3.7.6 デバッグツール

前述の通り、システムの開発において、バグの発生は避けては通れません。テストによってバグに気づいた後は、そのバグを修正する必要があります。この作業を**デバッグ**と呼びます。

よくある原始的なデバッグ手法のひとつが、コード内にログを出力する処理を記述して、実行状況や変数の内容をなどを確認する方法です。これを **Print デバッグ**などと呼びます。ロジックの間違いなど、シンプルなバグであれば、これだけで原因の特定と解決ができるでしょう。ですがバグの中には、こうした手法では対処できない、やっかいなバグも存在します。そういう場合に便利なのが、専用の**デバッグツール**です。特に統合開発環境（IDE）では、デバッグツールとの連携が標準的にサポートされており、より効率的にバグの原因を特定できます。

デバッグツールとは、ソフトウェアやアプリケーションの実行状況や変数の内容などを確認しながら、バグの原因を特定するためのツールのことです。デバッグツールを活用すれば、1 行ずつプログラムの処理を進めながら、その瞬間の変数の値や分岐処理の方向などを確認し、プログラムが意図した通りに処理が進んでいるかを詳細に追跡するようなことも可能になります。

3.8 整えるべき拡張機能

現在もっとも人気のある統合開発環境が、**Visual Studio Code（VS Code）**です。VS Code をこれから使い始める方向けに、まずは使ってみて欲しい、おすすめの拡張機能を紹介します。

Code Spell Checker

Code Spell Checker[*14]は人気の高い拡張機能のひとつで、コード内のスペルミスを検出してくれます。変数の名前などは、コード内で一貫性が保たれていれば、辞書的には間違ったスペルであっても、プログラムは動作します。ですがスペルミスが存在していると、単語で検索した際などに正しく検出できなくなったりなど、些細ですが可読性に影響します。これは保守性の低下にも繋がります。また単純なスペルミスの指摘は、レビュー工数の無駄な増加にもつながります。機械的にセルフチェックできる問題は、このようなツールを活用して解消することをお勧めします。

GitHub Pull Requests and Issues

GitHub Pull Requests and Issues[*15]は、GitHub が公式で提供している拡張機能です。プルリ

[*12] https://www.uipath.com/ja
[*13] https://autify.jp/
[*14] https://marketplace.visualstudio.com/items?itemName=streetsidesoftware.code-spell-checker
[*15] https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github

クエストの作成やレビューコメントからイシューの作成などの操作を、GitHub の Web ページにアクセスすることなく、VS Code に統合して操作を可能にします。特にプルリクエストは、行単位にコメントができ、コメントを確認する際は VS Code から確認できます。画面を行き来することなく確認と修正ができるため、地味ですが開発効率を確実に上げてくれるツールです。

GitHub Actions

最近の開発では、CI の導入が当たり前になってきています。GitHub Actions[*16]は、GitHub が提供している汎用ワークフローツールで、主に CI/CD の実行に利用されています。この拡張機能をインストールすると、CI の実行状況やログの確認などを、GitHub Actions の Web ページにアクセスせず、VS Code 上から確認できるようになります。また、失敗した処理の再実行も、VS Code 上から行えます。GitHub Actions を使用している場合には、この拡張機能をインストールしておくとよいでしょう。

3.9 クリーンな実行環境を目指す

開発サイクルを回していく中で、中心的な役割を果たすのは**実行環境**です。実行環境は、コーディングしたプログラムを実際に動かす環境ですが、その構築にはいくつかの方法が考えられます。

一番シンプルなのが、開発者のマシン上に直接構築する方法です。ですがシンプルな反面、ローカル環境を汚しやすいという問題があります。具体的には、日々の試行錯誤の末、消し忘れた不要な設定や他プログラミング言語のツールやライブラリが悪影響を与えてしまうような状態です。ライブラリをシステムワイドにインストールしてしまった結果、そのライブラリの別バージョンを要求する別のプロジェクトに影響が出てしまうなどは、よくある例です。

そこで最近では、仮想化技術やコンテナを活用して、プロジェクトごとに独立した環境を作るのが主流となっています。ローカル環境とは隔離された場所に環境を構築できるため、必要に応じて簡単に環境を破棄して元の綺麗な状態に戻すことができます。また、依存関係ファイルやキャッシュファイルなどの一時的なファイルが残ってしまうことを心配する必要もなくなります。そのような環境のことを**クリーンな環境**と呼ぶこともあります。

3.9.1 実行環境の種類と違い

それぞれの実行環境のメリットとデメリットを見ていきましょう。

ローカルマシン

実行環境の定番といえば、前述の通りローカルマシン上に直接インストールする方法です。この方法は、クラウドサービスや仮想化技術が普及する前から使われていました。セットアップ自体は簡単ですが、開発者のローカル環境と、実際にアプリケーションを実行する本番環境との差異（例えば OS の違い、バージョンの違い、ライブラリーの構成）によって動作しないという危険性を常にはらんでいます。また最近では、開発に使用する OS には Mac や Windows だけでなく、Linux を使用する人も増えてきています。そのため、自分が使用している OS と異なる端末を使用して開発しているメンバーのサポートができるとも限りません。使用する OS によっては、実行環境の構築に時間を要する場合もあります。

システムやサービスを開発するには、それなりのマシンスペックが求められます。参加するプロジェクトの方針や考え方によって、どのような端末が支給されるかは異なります。しかし、開発に使用するマシンスペックが不足していると、アプリケーションを動かすことができなかったり、1 つ 1 つの処理が遅くなり開発者の待っている時間が増えてしまうことにつながります。そのことが発覚したとしても、支給されたハードウェアの交換が、容易にできない場合もよくあります。

開発者それぞれがローカルマシン上に実行環境を構築することで、設定方法や構築テクニックなどが属人化してしまうこともあります。そのため、構築時に詰まったポイントやノウハウを共有することが

[*16] https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-github-actions

難しくなり、新しく参加したメンバーが毎回同じところに時間と労力をかけることになり、組織全体としては非効率的になってしまいます。

仮想化技術

仮想化技術を用いることで、本番に近い環境でアプリケーションを動かすことができます。例えばWindows を使い、Ubuntu 上で動かすアプリケーションを開発したいケースを考えてみましょう。このようなケースでは、Windows 上に VirtualBox[*17]のような仮想化ソフトウェアを導入して、仮想マシンを作ります。その中に Ubuntu をインストールし、アプリケーションの実行環境を作ることで、本番同様の環境を実現できます。その他にも、VMware vSphere[*18]や OpenStack[*19]のようなプライベートクラウド上で環境を作ってテストしたり、後述のクラウドサービスを用いた方法もあります。

仮想化技術を使えば、1 台の PC 上で、複数の異なる OS を同時に動かすこともできます。仮想マシンはソフトウェア的にエミュレートされた環境のため、パフォーマンス面では多少不利な面もあります。ですが使い勝手に関して言えば、仮想マシン内の OS は実マシンにインストールした時とほぼ同じ挙動をするため、他の方法と比べると学習コストが小さくて済むメリットがあります。

一方で、この方式では仮想マシン内に OS と実行環境一式をセットアップする必要があるため、それだけの手間がかかるというデメリットがあります。仮に **Ansible**[*20]のようなプロビジョニングツールを使って効率化したとしても、コンテナ技術に比べると、大きな手間がかかってしまいます。

コンテナ技術

開発者個人の実行環境をはじめ、検証環境、ステージング環境、そして本番環境と、アプリケーションを実行する環境は、用途ごとに同じものを複数用意するのが一般的です。ここで問題になるのが、環境ごとの違いです。特に環境を手作業で構築していると、長期間運用している間に、環境ごとの差異が大きくなり、開発者の手元では動くのに、本番にデプロイすると動かないといった問題を起こしがちになります。

そこで複数の環境で同じ状態を再現するため、近年採用されるケースが増えてきているのがコンテナです。コンテナは開発環境への導入も増えています。

コンテナ技術を用いれば、環境の作成手順そのものをコード化できます。これは環境の構築方法や設定を共有しやすくなるほか、環境の差異による数々の問題の発生を抑制でき、結果として開発の効率を底上げできます。そしてコード化されたファイルは、バージョン管理ツールの管理下に置くことで、環境の変更を正確に把握できるようにもなります。

コンテナ技術を活用すると、開発しているアプリケーションのバージョンアップや、新しいツール・ライブラリーの導入をしやすくなるというメリットもあります。

バージョンアップやツールの追加インストールというのは、本質的に不可逆な操作です。そのためローカルマシン上に直接構築した環境でこうした操作を行うと、ローカルマシンの環境が「変更されたという状態」を持ってしまいます。これが前述の、**環境によって動いたり動かなかったりする問題**の原因となります。ですがコンテナを使えば、「新しいバージョンのコンテナをビルドしてテストし、上手くいったらそのイメージを配布する」「上手くいかなかったら、古いバージョンのコンテナにロールバックする」といった作業を、他の環境への影響なしに行えます。

アプリケーションや使用するプログラミング言語、依存するライブラリーのバージョンアップは避けては通れません。ですが従来であれば、バージョンアップに伴う副作用を嫌い、いわゆる**塩漬け**を選択するケースが多く見られました。バージョンアップを行わない塩漬けは、非常に大きなセキュリティリスクを生みます。コンテナを利用すれば、低コストで、かつ実行中の環境への副作用を避けながら、バージョンアップを行いやすくなります。これはセキュリティの向上はもちろん、アプリケーションに挑戦的な機能を実装しやすくなるというメリットにも繋がります。

[*17] https://www.virtualbox.org/
[*18] https://www.vmware.com/products/cloud-infrastructure/vsphere
[*19] https://www.openstack.org/
[*20] https://ansible.com/

クラウドサービス

クラウドサービスを利用すれば、開発者のローカルマシンのスペックに束縛されず、アプリケーションの開発に必要なスペックを確保することができます。また環境がインターネット上に存在するため、環境の共有もしやすくなります。

クラウドサービスの特徴のひとつが、環境そのものを API で制御できるという点です。そのため環境の構築や設定変更、削除といった操作も、コードとソフトウェアで制御できるのです。コードによるインフラの定義は **Infrastructure as Code（IaC）**と呼ばれています。汎用の IaC ツールとしては、**Terraform**[*21]が有名です。また AWS における **CDK**[*22]や **CloudFormation**[*23]のように、クラウドサービス独自の IaC ツールが存在する場合もあります。IaC を活用することで、複雑な環境であっても、簡単に同一のものを構築できます。

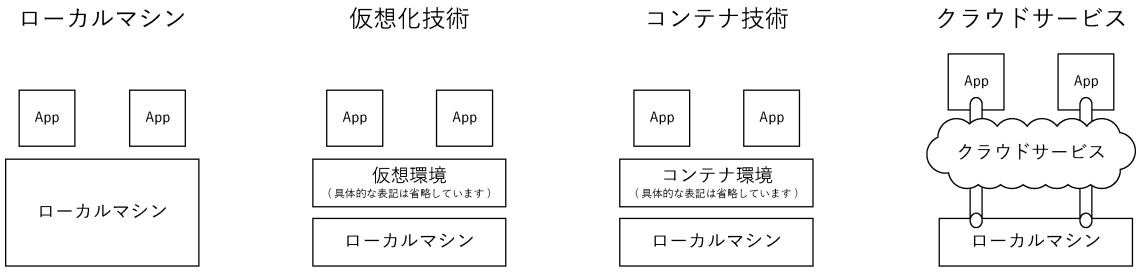

図: 様々な実行環境

3.9.2 開発環境におけるコンテナ

コンテナの、開発環境における活用について考えてみましょう。開発環境におけるコンテナ活用は、大きく 2 つに分けられます。1 つ目は、アプリケーションの実行環境です。前述の通りコンテナ技術を用いることで、本番環境との差異をなくすことができます。

2 つ目は、ツールの実行環境です。コンテナには、Web サーバーのように常時立ち上げ続ける使い方と、必要な処理が終わったら、その都度コンテナを破棄するような使い方があります。

たとえば、フロントエンドとバックエンドが分離されているアプリケーションで、バックエンドとの通信に REST を使った開発をしているとしましょう。フロントエンド側では、バックエンドと通信するための、API クライアントを生成する必要があります。その際に、バックエンド側で公開している OpenAPI の定義ファイルなどを用いて、自動生成する事があります。ここで使用するジェネレートツールを、コンテナ内で実行するといったケースです。ツールをコンテナ内に閉じ込めることで、ローカル環境を汚さずにツールを実行できます。前述のコンテナのメリットの通り、バージョンアップが早いクライアントツールを運用する際にも、コンテナによるカプセル化は非常に便利です。

3.9.3 主なコンテナ実行ツール

コンテナを実行するためのツールにも、様々な実装が存在します。

Docker Desktop

コンテナ実行環境の中でも、最もよく知られているものが Docker です。そして Docker を操作しやすくするためのグラフィカルインターフェースとして、**Docker Desktop**[*24]があります。Dokcer Desktop はグラフィカルインターフェイスだけでなく、Docker エンジンそのものを

[*21] https://developer.hashicorp.com/terraform
[*22] https://aws.amazon.com/jp/cdk/
[*23] https://aws.amazon.com/jp/cloudformation/
[*24] https://www.docker.com/products/docker-desktop/

内包しています。そのため特定のバージョンの Docker Desktop をインストールするだけで、Windows/Mac/Linux が混在しているような環境であっても、まったく同じ Dokcer 実行環境を、オールインワンで準備できるというメリットがあります。これは大人数で開発を行うようなチームで、非常に大きな効果を発揮するプロダクトです。なお Docker Desktop は、2021 年 9 月から一部ユーザーを対象に有料化されました。これから導入を検討される際には、有料プランの契約対象になるかどうかを事前に確認してください。Docker Desktop は利用者が多く、問題が発生した際も解決するための情報を得やすいため、有料プランに該当しない場合には、基本的に Docker Desktop を利用することを筆者はおすすめします。

Rancher Desktop

Docker Desktop の有料化の話が出た際に、代替ツール候補として各所で紹介したものの 1 つが、**Rancher Desktop**[*25]です。Docker Desktop と全く同じとまではいきませんが、基本的な機能は揃っています。そのため、Docker Desktop からの移行先としての最有力候補に名前が上がっていました。

Podman Desktop

Podman[*26]は、ツールとして Docker の置き換えとなることを目指して開発されている、Docker と互換性のあるオープンソースソフトウェアです。そして Podman Desktop は、Podman で Docker Desktop のような管理を可能とする、GUI のソフトウェアです。現在では VSCode の Docker 拡張や、Dev Containers 拡張を使った開発も実行可能です。

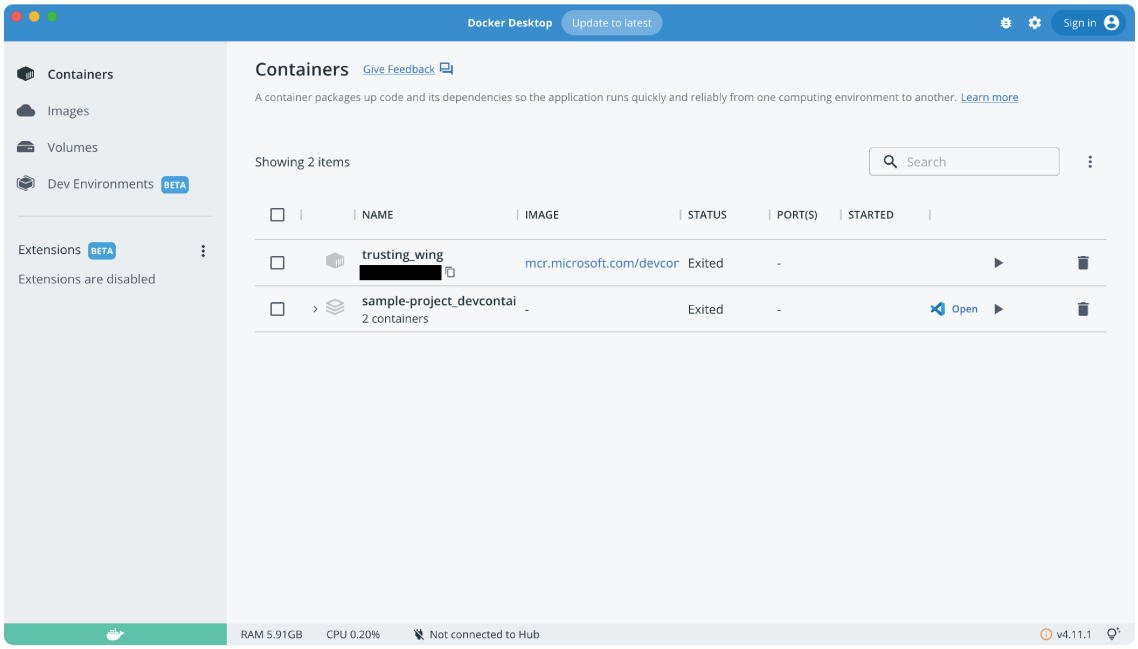

図: Docker Desktop

[*25] https://rancherdesktop.io/
[*26] https://podman.io/

第 4 章　ソフトウェアを素早く確実にリリースする

DevOps のサイクルを回す上で重要なのが、アプリケーションのみならず、インフラ構成、テスト、デプロイ手順、モニタリング、といったあらゆる要素をコードによって管理することと、これらのプロセスを自動化することです。この章ではアプリケーションのビルド、テスト、デリバリーなどのプロセスを自動化する CI/CD パイプラインについて解説します。

4.1 CI/CD とは

CI とは Continuous Integration の略で、日本語では「継続的インテグレーション」と呼ばれます。インテグレーションという表現が抽象的なため、具体的にイメージしづらいかもしれませんが、これは簡単に言えば**コードをリリースブランチへマージ可能な状態に保つ**ことです。そしてリリースブランチにマージ可能な状態とは、コードに問題がないことが確認できた状態と言えます。**必要なテストをすべてパスできた状態**と言いかえてもいいでしょう。CI はこの確認を自動化し、継続的に行うプロセスです。例えば「コミットやプルリクエストの発行をトリガーとして、自動的にコードの静的解析やテストを行う」などは、典型的な CI の一例です。CI によって自動的にテストが行われることで、追加したコードが意図通りに動いていることや、既存のコードにリグレッションが発生していないことが担保されます。

CD とは Continuous Delivery の略で、日本語では「継続的デリバリー」と呼ばれます。デリバリーとは、アプリケーションを正しくデプロイできるよう、その準備を整えるプロセスを指します。具体的にはコンテナイメージのビルドや、コンテナレジストリへのプッシュといった作業がデリバリーに含まれます。そして CD は、このデリバリーに必要な作業を自動化し、継続的に行います。CI によってアプリケーションのコードに問題がないことが確認されると、その後段階として CD が行われるのが一般的です。

CD はあくまでデプロイの準備である**デリバリー**までを自動化するもので、実際のデプロイは行いません。そのためデリバリーの完了後、担当者による承認などを挟んだ上で、別途デプロイ作業が必要になります。ただしデプロイまでを完全に自動化するケースもあり、その場合のプロセスは **Continuous Deploy（継続的デプロイ）**と呼ばれます。どちらも略称が同じ「CD」のため、取り違えないよう注意してください。こちらの継続的デプロイについては後述します。

その性質上、CI と CD は連続して行われることが多いため、このふたつはセットにして **CI/CD** と呼ばれます。また実際の CI/CD において、具体的にどのようなツールが、どのような処理を行うかを定義したものを **CI/CD パイプライン**と呼びます。

従来、アプリケーションへの新規機能の実装とリリースは、バグ混入のリスクや既存機能への影響といった面からコストが大きく、これがリリース頻度の低下の一因となっていました。ですがこれでは、激しく変化する市場の要求に対応できません。CI/CD によるテストとデリバリーの自動化は、品質を担保しつつ、同時に開発サイクルを高速化します。アジャイルな開発を実現するためにも、CI/CD は絶対に欠かすことのできない要素だと言ってよいでしょう。

4.2 CI/CD パイプラインの構成要素

CI/CD パイプラインは、テストやビルドといった、連続、あるいは並列に実行される、複数の要素が集まって構成されています。このパイプラインを構成する要素を**ジョブ**と呼ぶこともあります。

一口に CI/CD パイプラインと言っても、具体的にどのようなジョブが実行されるかは、対象となるアプリケーションの要求やシステムの仕様によって異なるため、その構成は一様ではありません。とはいえ細かな違いはあれども、一般的な CI/CD パイプラインにおいてはおおむね以下で紹介するようなジョブが、共通して実行されています。

なおツールの内部において、CI のジョブと CD のジョブは、必ずしも区別されません。前述の通り、CI とはコードをマージ可能にするための確認作業であり、CD とはデプロイの準備を整えることです。ですがこれは概念上の区別であり、ツールから見ればどちらも、コードが変更された際に実行するジョブのひとつでしかないためです。そのため単一のパイプライン内で、CI と CD の両方を連続して行うことは一般的ですし、場合によってはデプロイまでを一気通貫で実行することも珍しくありません。

4.2.1 コードのチェックアウト

なにはなくとも、対象となるコードをリポジトリから入手しないことには始まりません。そのため CI/CD パイプラインはまず最初に、コードをチェックアウトするジョブを実行するのが一般的です。そしてチェックアウトのような普遍的な処理は、簡単に何度でも実行できるよう、通常はパッケージ化して提供されています。具体的な例を挙げると、GitHub Actions では **Action** という、ジョブをパッケージ化して提供する仕組みが用意されており、以下のジョブ設定だけでコードをチェックアウトできます。

```
jobs:
  test:
    steps:
      - uses: actions/checkout@v4
```

4.2.2 コードの静的解析

どのようなジョブをどのような順に実行するかは、パイプライン実装者の自由です。ですが一般論として、**実行にかかるコストが低く、依存するジョブの少ないものから優先的に行うのが定石**となっています。その中でも静的解析は、コードのみがあればよく、ビルド等の作業が必要ないため、最初に行うのに適したジョブと言えるでしょう。

静的解析は、主に文法の間違いや、コーディング規約違反などをチェックします。コードの複雑さなど、潜在的なバグとなりうる箇所を見つけてくれる解析ツールも存在します。またこの段階で、インデントなどを揃える**コード整形**を実行することもあります。

4.2.3 アプリケーションのビルド

使用しているプログラミング言語にもよりますが、テストの前にコンパイルなどのビルド作業が必要となる場合があります。そうしたケースでは、テストジョブを実行する前段階で、ビルドジョブを実行します。また実行時に依存しているライブラリのインストールを行う場合もあります。

4.2.4 テスト環境へのデプロイ

後述するテストのうちのいくつかは、実際にアプリケーションをデプロイしないと確認できません。そのためこの段階で、テスト環境にアプリケーションをデプロイする場合があります。「継続的デリバリーでは、デプロイまでは行わない」と述べましたが、これはあくまで本番環境に対しての話です。テスト環境へのデプロイは、インテグレーションとデリバリーの一環として、実施されることもあります。

4.2.5 アプリケーションのテスト

アプリケーションに対してテストを行います。テストにも様々な種類がありますが、もっとも基本的なのが、プログラミング言語ごとに用意されたテストフレームワークなどを利用した単体テストです。

実際にどのようなテストを行うかは、アプリケーションごとに異なるでしょう。ですがコードを書く以上、どのようなアプリケーションであっても、単体テストは絶対に行うべきです。

それ以外では、データベース等の外部リソースとの連携を確認する**インテグレーションテスト**や、動作中のアプリケーションに負荷をかける**パフォーマンステスト**などがあります。また専用のツールを用いて、ユーザーの操作を模倣した、エンドツーエンドのテストを行う場合もあります。これらのテストはアプリケーションが動作していないと行えないため、実施前にはテスト環境へのデプロイジョブが実行されます。

4.2.6 アプリケーションのデリバリー

ここまでが主に CI に含まれる内容となります。これらをパスしたら、本番環境へのデプロイの準備、すなわちデリバリーを行います。

デリバリーの具体的な内容は、アプリケーションのデプロイ方法によって異なるため、一概にこうするとは言えません。ですが近年ではコンテナを使ってデプロイするケースが多く、その場合はコンテナイメージのビルドと、コンテナレジストリへのプッシュを行うことになるでしょう。GitHub Actions では、これらもコードのチェックアウトと同様に、再利用可能なアクションが用意されています。

4.3 CI/CD を実現するツール

CI/CD を実現するためには、専用のツールを用いるのが一般的です。

DevOps の中核を成すのが、ソースコード管理システムです。CI/CD はソースコードへの変更をトリガーとして、ソースコードや、そこから生成される成果物に対して行われます。この性質から、CI/CD ツールとソースコード管理システムの間には、密接な関係があります。そのため、ソースコード管理システムの一機能として、CI/CD ツールが提供されていることも一般的です。

例えば、全世界で 1 億人以上が利用するソースコードホスティングサービスである GitHub には、CI/CD のワークフローを自動化する GitHub Actions という機能が用意されています。また同様のサービスである **GitLab**[*1]にも、**GitLab CI/CD**[*2]という機能があります。これらはどちらも、CI/CD パイプラインが行うタスクを設定したファイルをリポジトリ内に含めるだけで、CI/CD を動かすことができます。ソースコード管理システムが標準で備えている機能のため、既にこれらのサービスを利用しているのであれば、非常に導入しやすいのが特徴です。

ソースコード管理システムとは独立した、単独の CI/CD サービスも存在します。その中でも老舗で有名なのが、**CircleCI**[*3]や **Travis CI**[*4]です。わざわざ独立したサービスを選択するメリットは、対応しているソースコード管理システムとであれば、自由な組み合わせを実現できるという点です。ソースコード管理システムが備えている CI/CD 機能よりも、機能面で優れている点も多いため、要件や好みに応じてこうした外部サービスを選択するのもよい考えです。

社内のセキュリティポリシーや予算といった理由で、こうしたクラウドサービスが利用できないこともあるでしょう。そうした場合は、オンプレミスでホスト可能な CI/CD サービスの利用を検討してみましょう。**Jenkins**[*5]は、CI/CD の自動化を支援するツールとして、広く利用されているオープンソースソフトウェアです。また GitHub Actions や GitLab CI/CD では、CI/CD パイプライン自体はクラウドサービス上で動かしつつも、実際のジョブは手元のサーバーで動かせる **Self Hosted Runner**[*6]をサポートしています。場合によっては、こうしたハイブリッドな構成を検討してみてもよいでしょう。

[*1] https://about.gitlab.com/
[*2] https://about.gitlab.com/solutions/continuous-integration/
[*3] https://circleci.com/
[*4] https://www.travis-ci.com/
[*5] https://www.jenkins.io/
[*6] https://docs.github.com/ja/actions/concepts/runners/self-hosted-runners

幸いなことに、ソースコード管理システム、特に GitHub や GitLab を利用しているのであれば、CI/CD ツールの導入はそれほど難しくはありません。もちろん CD までを完全に自動化するには、ワークフローの見直しや、インフラ側の整備といった作業が必要になることもあるでしょう。しかしソースコードの解析やテストなど、単体で完結する CI 部分に関しては、今日からでも取り組めるはずです。CI/CD の第一歩として、まずは現在手動で行っているテストを自動化する所から始めてみるのがお勧めです。

4.4 リリースとデプロイ

CI/CD が完了したら、いよいよ**デプロイ**を行います。そこで最初に、デプロイとは何かをしっかり理解しておきましょう。デプロイ（Deploy）とは、「展開する」や「配備する」といった意味を持つ英単語です。IT の文脈ではアプリケーションをサーバーに展開し、実際に利用可能な状態にすることをデプロイと呼んでいます。

デプロイと似た用語に**リリース**があります。ですが「アプリケーションをデプロイする」と「サービスをリリースする」には、明確な違いがあります。デプロイはあくまで、サーバー上でアプリケーションを動く状態にすることを指します。対してリリースは、エンドユーザーに対してサービスを解放することを指します。つまりデプロイが完了し、アプリケーションが起動していたとしても、リリースしない限り、ユーザーはサービスを利用できません。具体的な作業を例に挙げると、デプロイは `docker pull` コマンドでコンテナイメージをダウンロードしたり、`kubectl apply` コマンドで Kubernetes にマニフェストを適用したりといった作業が該当します。対してリリースは、ロードバランサーや DNS レコードの向き先を切り替え、デプロイしたアプリケーションにトラフィックをルーティングする作業などが該当します。

デプロイとリリースは同時に行われることも多いため、敢えて区別する必要がない場合も多く、そのせいで混同している人もいるでしょう。ですがこのふたつは意味合いが異なるため、区別して考えられるようにしましょう。デリバリー、デプロイ、リリースの違いを、以下の表にまとめました。

表 4.1: デリバリー、デプロイ、リリースの違い

作業	実施する内容	具体的な作業
デリバリー	デプロイの準備を整える	docker build、docker push など
デプロイ	アプリケーションをサーバー上に展開する	kubectl apply など
リリース	ユーザーがサービスを利用可能にする	ロードバランサーの切り替えなど

4.5 CIOps と GitOps

それでは実際のデプロイ作業について見て行きましょう。DevOps においては、デプロイに必要な設定情報はすべてコード化され、Git によって管理されているのが基本です。そしてツールを使ってこの設定を適用することで、実際にデプロイを行うわけです。例えばコンテナオーケストレーションとして Kubernetes を採用している環境であれば、アプリケーションを動かすためのマニフェストを作成し、`kubectl apply` コマンドを実行してデプロイを行います。

このデプロイ作業には大きく、**Push 型**と **Pull 型**のふたつのアプローチが存在します。

まずひとつが CI ツールを使い、CI/CD パイプラインの延長上でデプロイを行う、Push 型のアプローチです。コードの変更をトリガーとし、CI/CD のワークフローの最後に、承認を挟んでデプロイが行われます[*7]。この方式を特に **CIOps** と呼びます。以下の図は、典型的な CIOps の流れを表したものです。見ての通り、非常にシンプルで直感的な構成になっています。

[*7] 前述の通り、継続的デリバリーでは自動的なデプロイまでは行わないため、デプロイ作業前に担当者の承認を挟むといったフローが一般的です。

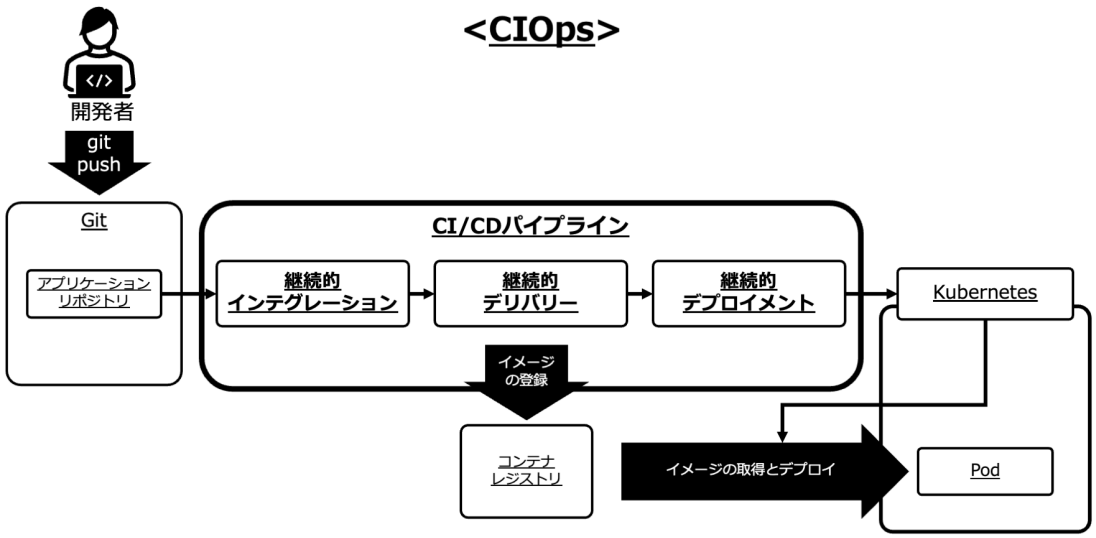

図 4.1: CIOps の模式図

ですが CIOps はシンプルな反面、デメリットもあります。それは CI/CD パイプラインが強力な権限を持ちすぎてしまうという点、そして CI とデプロイを分割できないという点です。

デプロイは本番サーバー上にアプリケーションを展開する作業です。そのため CIOps を行う際には、本番環境を操作する権限を、CI/CD パイプラインに持たせる必要があります。外部のツールにこうした権限を持たせるのは、それだけでセキュリティリスクとなる可能性があります。また CI/CD パイプライン内のジョブでは任意のコマンドを実行できるため、パイプラインの実装ミスによって、本番に障害が起きるといった可能性も否定できません。

CIOps では、CI からデプロイまでをひとつの流れとして実行します。つまり CI とデプロイを分割できないのです。これは、開発とデプロイを別の組織が行いたい場合に問題となります。具体的な例を挙げてみましょう。あるプロジェクトでは、アプリケーションは協力会社が作成し、成果物はコンテナイメージをレジストリに push する形で納品されます。そしてデプロイは別途、自社の都合のよいタイミングで行います。こういうケースでは、開発時に繰り返し行う CI と、最終的なデプロイが単一のパイプラインで構成されていると、非常に扱いづらいものになってしまうのです。

また CIOps は Push 型であるため、いわば**投げっぱなしのデプロイ**を行います。CI ツールはデプロイコマンドを実行しますが、その結果やデプロイ後のアプリの状態については関与しないのです。そこで最近注目されているのが、専用のデプロイエージェントを用いた Pull 型のデプロイです。その基本的な流れは以下のようになります。

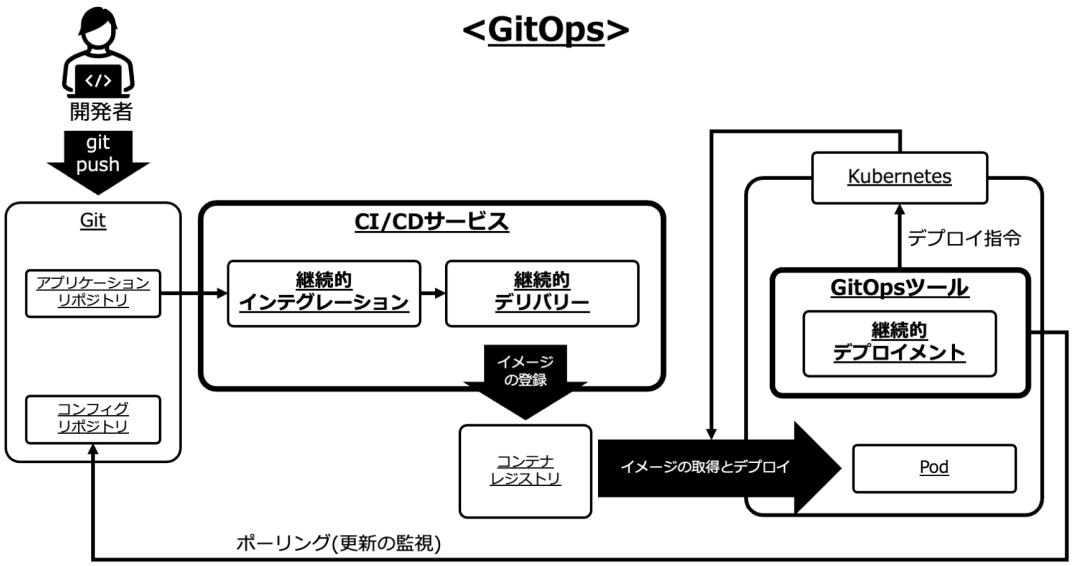

図 4.2: GitOps の模式図

こうしたデプロイ方式は **GitOps** と呼ばれています。GitOps は 2017 年に、英 Weaveworks 社が提唱した[*8]デプロイ手法です。GitOps では、システム全体のコンフィグは宣言的に記述され、すべてが Git の管理下に置かれます。そしてこの Git リポジトリを、唯一の信頼できるソースとして扱うことが、名前の由来となっています。

GitOps ではデプロイのためのコンフィグを、アプリケーションのコードとは別リポジトリに格納するのが一般的です。本番環境内に構築された**デプロイエージェント**がこのリポジトリの変更を監視しており、変更がマージされると、自動でデプロイを行います。これが「Pull 型」の所以です。CIOps と違って外部の CI ツールが本番環境にアクセスする必要がないため、CI/CD パイプラインに強い権限を持たせる必要がなくなります。また見ての通り、従来の CI/CD パイプラインとデプロイ処理は別のツールによって行われるため、責任分界点をはっきりさせることができます。さらにデプロイエージェントが本番環境内にいるため、デプロイ後のアプリケーションの状態までも管理できます。

DevOps をまず始めるのであれば、最初はシンプルな CIOps でも問題はないでしょう。ですが**プロジェクトがスケールするに従い、CI とデプロイの分離が必要になるタイミングが必ず訪れます**。こうした理由から、GitOps が注目され始めているというわけです。

4.6 GitOps を実現するツール

GitOps ワークフローを実現するためには、Git リポジトリを監視し、デプロイを行うエージェントをセットアップする必要があります。こうした GitOps ツールとしてメジャーなのが、**Argo CD**[*9]と**Flux**[*10]です。

Argo CD は、それ自体が Kubernetes クラスター内にデプロイされるアプリケーションです。デプロイしたいアプリケーションのソースとなる Git リポジトリと、デプロイ先を登録することで、GitOps 的なデプロイを可能にします。以下は Argo CD のアプリケーション登録画面の例です。Argo CD は

[*8] https://speakerdeck.com/errordeveloper/gitops-operations-by-pull-request
[*9] https://argo-cd.readthedocs.io/en/stable/
[*10] https://fluxcd.io/

Helm チャートで提供されているアプリケーションであれば、ソースとなる Git リポジトリと、デプロイ先のクラスターと Namespace を設定するだけです。

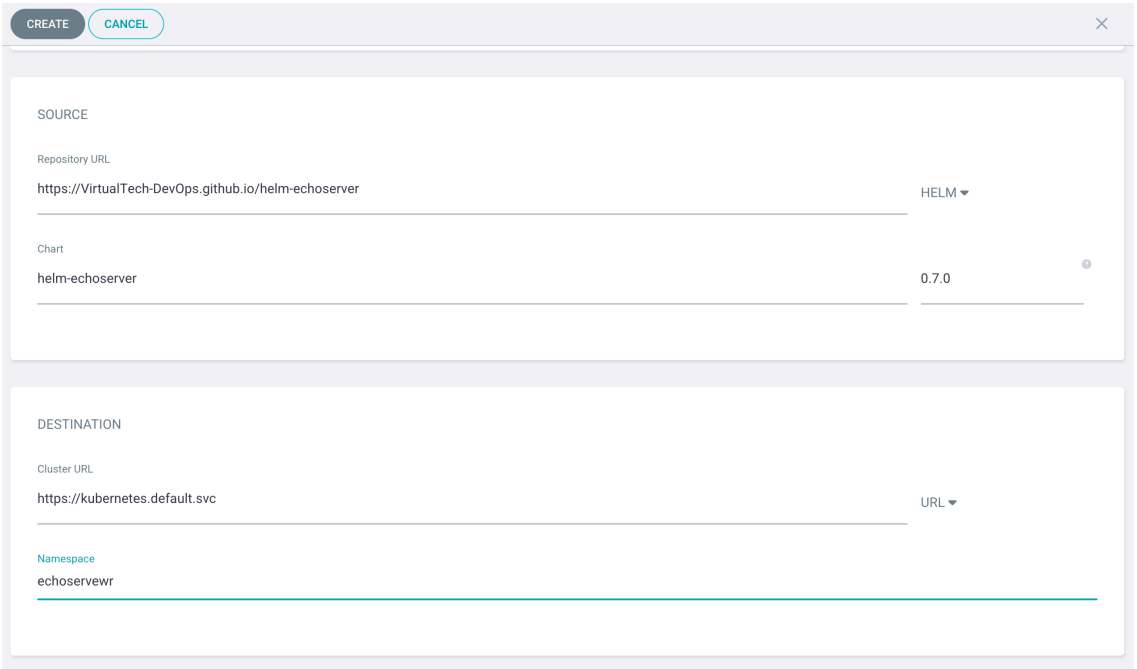

図 4.3: Argo CD のアプリケーション登録画面

実際にアプリケーションをデプロイさせると、以下のようになります。ソースやバージョンをはじめ、アプリケーションとデプロイに関する様々な情報が表示されているのがわかります。

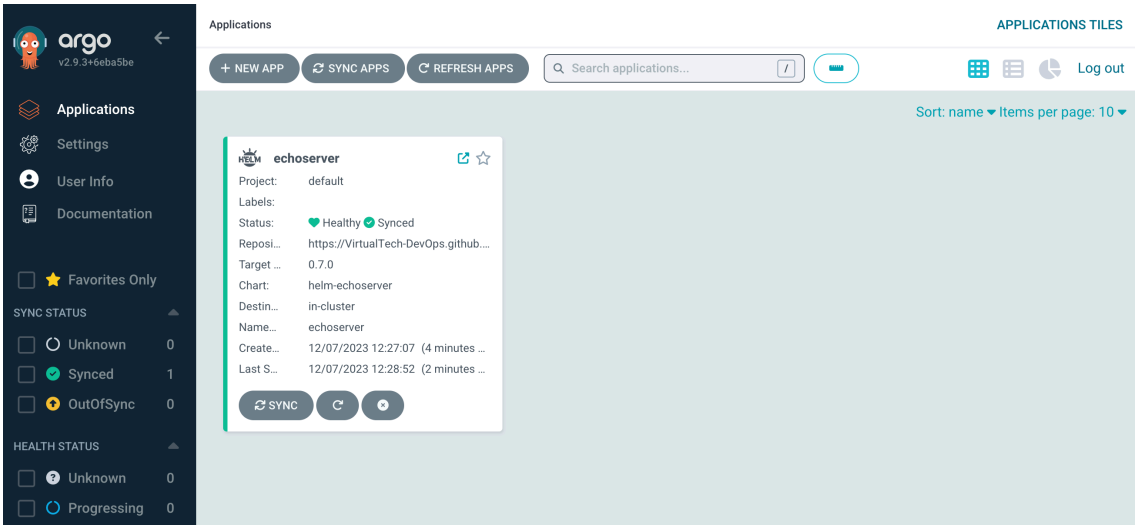

図 4.4: 実際にアプリケーションをデプロイさせた状態

Argo CD 自体がクラスター内にいるため、アプリケーションを構成する様々なリソースの状態も管理できます。これは Push 型のデプロイでは実現できないメリットのひとつです。

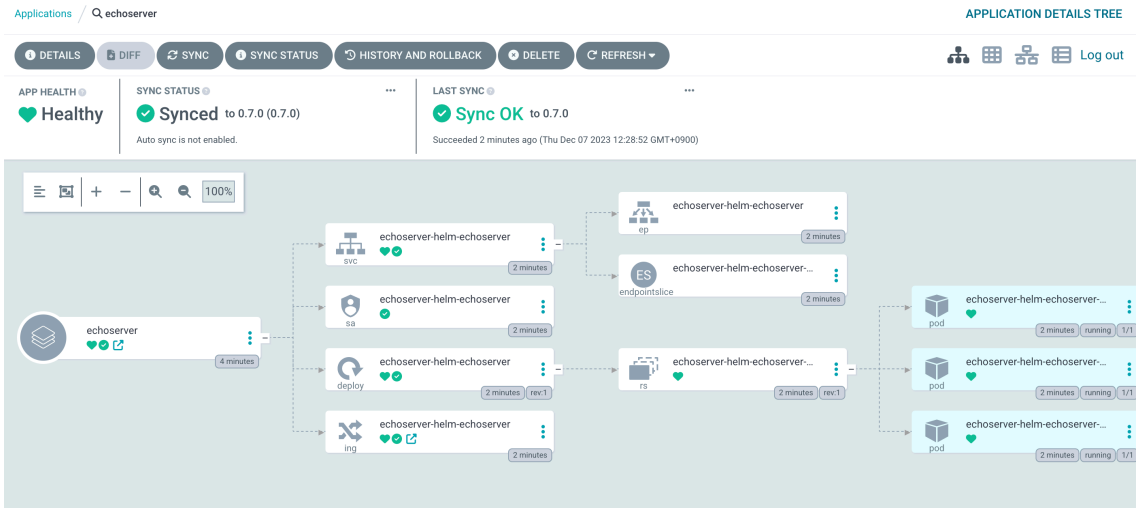

図 4.5: クラスター内のリソースの管理

ドキュメント[*11]を見るとわかる通り、Argo CD は Kubernetes クラスターにマニフェストを適用するだけで、簡単に使いはじめることができます。CLI のクライアントはもちろん、Web ベースの GUI インターフェイスも備えているため、はじめてでも使いやすい GitOps ツールと言えるでしょう。

Flux も Argo CD と同様に、Kubernetes クラスター上で動作する GitOps ツールです。もともと GitOps の提唱元である英 Weaveworks 社が開発したという出自を持つため、言ってみれば GitOps の本家本元と言ってもよいツールでしょう。Flux には大きくバージョン 1 とバージョン 2 が存在し、現在ではゼロから再設計された **Flux v2**[*12]が使われています。

GitOps を採用することで、クラスター内にアプリケーションを効率よく、自動的にデプロイできます。また従来の CIOps によるデプロイとは異なり、システムの状態が、宣言的に記述されたコードと一致することが保証されます。まずはこれらのツールを使い、GitOps によるデプロイを体験してみるとよいでしょう。

[*11] https://argo-cd.readthedocs.io/en/stable/getting_started/
[*12] https://github.com/fluxcd/flux2

第 5 章　実行環境を効率的に管理・運用する

ここまで、DevOps で開発効率を上げるためのツールや、その考え方について解説してきました。CI/CD や Git、IDE などはどちらかというと Dev の作業を加速させるためのツールです。そこで次は、Ops 側の作業を加速させるツールについて考えみましょう。

5.1 IaC とは

繰り返しになりますが、DevOps の基本はコード化と自動化です。そして Ops がメインに行っている作業は、環境構築や運用です。つまり環境の構築方法や運用手順をコードとして表し、自動的に実行できるツールが必要になります。インフラストラクチャの構成をコードとして定義し、その構築や運用をツールを用いて自動化しようという考え方を、**Infrastructure as Code（IaC）**と呼びます。そして IaC の考えに基いて実際の構築・運用を担当するツールを、**構成管理ツール**と呼びます。

まずは構成管理について理解しておきましょう。構成管理とは、ハードウェア、ソフトウェア、ネットワークなど、**システムを構成している要素を把握し、管理すること**です。

きちんと構成管理を行わず、手作業で構築しっぱなしの環境を想像してみてください。システムがどのように構成されているかがわからないため、同一構成のシステムを構築し直すことは困難でしょう。将来的にメンテナンスが必要になった場合はどうでしょうか。システムの構成要素がわからないと、変更を加えた場合の副作用も読み切れません。もしかしたら、メンテナンス作業が原因で、サービスがダウンしてしまうかもしれません。とてもではないですが恐しくて、そんなシステムに手を入れることはできないでしょう。メンテ不能な、負の遺産の完成です。ですが正しく構成管理を行えば、システム構成や現在の状態を明確にすることができます。そして状態を把握できていれば、構成変更を行うことも容易になるというわけです。システムを健全に維持するためには、構成管理は絶対に欠かせません。

実際に構成管理を行うのが、構成管理ツールです。具体的には、構成に必要な要素や構築手順などをコードとして記述し、ツールがそれを実行します。コードと呼ばれていますが、プログラミング言語のようにロジックが記述されているわけではなく、YAML や JSON、あるいは専用の DSL を用いて、構成を定義するのが一般的です。コードはテキストファイルであるため、インフラ構成をバージョン管理ツールで管理しやすいのもメリットです。変更の追跡や構成のレビューといったバージョン管理ツールの恩恵を、アプリケーションプログラムと同様に享受することができます。

5.2 コードの種類

構成管理ツールのコードには**手続き型**と**宣言型**と呼ばれる 2 つの記述方法があります。

5.2.1 手続き型

システムが「あるべき状態」に到達できるように、そこまでの手順をコードとして記述する方式です。構成管理ツールは、ターゲットシステム上で手順を積み重ねることで、システムをあるべき状態へ向かわせます。この方式は、いわばシェルスクリプトなどを利用した、手順の自動化と同じアプローチです。

例えばサーバーを 2 台起動するとしましょう。コードには現在のサーバー台数を取得し、足りなければ追加、超過していれば削除するような処理を記述します。これは構築手順書をコード化したようなものなので、手作業による構築に慣れ親しんだ方であれば簡単に理解できるでしょう。普段からこのような作業をしている方なら、手続き型は学習コストが低いと言えます。

手続き型の欠点は、あくまで実行する手順のみが記述されているだけであり、最終的なあるべき状態を定義できない点です。先の例ではサーバーを 2 台起動するために 2 台未満か超過かで条件分岐をし、手順を実行しました。結果的にサーバーが 2 台起動することを期待していますが、その保証はどこにも

ありません。タイミングによって、サーバーの起動に失敗してしまう可能性もあるでしょう。その場合のエラーハンドリングもきちんと行わなければ、最終的なサーバーの台数は不足してしまうかもしれません。この方式の問題点は、最終的にシステムがどういう状態になるのか、コードを実行してみるまでわからない点です。手続き型の主なツールとしては、**Chef**[*1]、**Ansible**[*2]などが有名です。

5.2.2 宣言型

システム構成の「最終的なあるべき状態」を記述する方式です。手続き型では、あるべき状態へ向かうための手順を記述しましたが、宣言型ではあるべき状態そのものを記述し、そこに到達するまでの具体的な手順は、構成管理ツールに一任します。

手続き型の時と同じく、2 台のサーバーを起動するとしましょう。手続き型では構築手順を記述したのに対し、宣言型は「サーバーを 2 台起動する」と宣言するのみです。サーバーを減らすのか、あるいは増やすのかは、ツールが勝手に判断して実行します。システムが最終的にどのような構成になるのかを、コードから直接読み取れるため、コードから構成を把握しやすいのもメリットです。

宣言型の欠点は、ツールのバージョンにシビアである点です。宣言型の構成管理ツールは、ツールが構築の具体的な手順を考えるため、異なるバージョンのツールを使うと、想定とは異なる挙動を示す可能性があります。例えば実行される手順が異なっていたり、API の仕様が変更されていたりといった具合です。そのため宣言型のツールを使用するのであれば、構成管理ツールのバージョン自体も固定し、ツールのアップデートは慎重に行う必要があるでしょう。宣言型の主なツールとしては、**Terraform**[*3]、**Puppet**[*4]などが有名です。

5.3 なぜ IaC が必要なのか

IaC を導入することで、属人化を解消し、また再現性を担保できるようになります。しかし IaC の導入を検討したときに真っ先に思い浮かぶのは、コード化のコストがペイできるかではないでしょうか。一度構築してそれっきりなシステムでは、コードを書くコストが増えるだけで、特に旨味がないようにも思えるでしょう。従来通り、慣れたエンジニアが手作業で構築した方が、一見コストパフォーマンスがよいようにも思えます。

ですがそうしたシステムであっても、IaC は絶対に導入すべきです。ここでは IaC を導入するべき、いくつかの理由について考えてみましょう。

5.3.1 属人化の解消

手順書を見ながら手作業で構築を行う場合、構成への理解や、インフラについて深い知識が要求されます。プロダクト内の Ops チーム全員が、一定の知識を保有していればいいのですが、なかなかそうもいきません。結果的にインフラに明るい方や、問題解決能力の高い方に依存してしまいます。

IaC はコードを実行するだけで、誰でも望む構成を構築できます。つまりインフラの構築に明るくない方でも、コードの実行方法がわかれば同じ構成を構築できます。構築するために必要な要素は、コードとしてチーム内で共有されるため、誰でも一定の知識を有することが可能です。また構成がコードとして明示されることで、インフラに明るい方や問題解決能力の高い方に、コードレビューや構成の問題点を確認できるようになるため、品質の向上も期待できます。

[*1] https://www.chef.io/
[*2] https://ansible.com/
[*3] https://developer.hashicorp.com/terraform
[*4] https://www.puppet.com/

5.3.2 人的ミスの解消

手作業で構築を行っていると、手順の間違いや環境の取り違えといった単純なミスが、どうしても発生します。その時の体調や作業する環境など、人的ミスが発生する要因は様々です。よくダブルチェックという言葉を耳にしますが、複数名で指差し確認をしていたとしても、その担当者両名が見落としている可能性も否定できません。では 3 人でトリプルチェックすれば安心でしょうか。言うまでもなく、人数をいくら増やしても意味はありません。人間は必ずミスをするものなのです。

人間によるミスをなくす方法は、たったひとつしかありません。それは作業から人間を排除することです。構築を自動化すれば、人間が行う作業は「コードを書く」「それを実行する」の 2 つとなるため、この作業以外で人的ミスが起こることがありません。もちろんコードにバグを埋め込んでしまう可能性もありますが、属人化の解消で説明した通り、知識のある方にレビューをしてもらったり、事前にテストを行うことで、問題の発生を抑制できます。

5.3.3 冪等性の担保

手動構築された環境が他の環境と全く同じかは、どのように保証するとよいのでしょうか。簡単な方法としては、チェック表を作ってチェックしながら構築する方法が思いつきます。しかしこのチェック表が最新の環境に追従しているのかはわかりませんし、項目も多数になるでしょうから間違えてチェックをつけてしまうこともあるでしょう。人間が行った作業は、本質的に信用できないため、この方法では他の環境と全く同じかという保証はできません。

しかし IaC でコード化しているとどうでしょうか。コードを使用して自動構築をしている場合、同一のコードを使っている限り、成果物も同一になります。冪等性を担保する上でも、IaC は重要です。

5.3.4 管理の効率化

IaC を使用しない場合、サーバーの管理はスプレッドシート等のドキュメントベースで管理することになるでしょう。何かを変更する度にドキュメントを修正し、手順の精査をすることになるでしょう。なぜ変更したのかエビデンスを残す必要もあります。ですがこうした面倒な作業は往々にして忘れられがちです。結果として実際の構成とドキュメントに差が出てしまい、構成管理が破綻している環境をよく目にします。

コードを実行して構築された環境では、コードと環境が必ず同一の状態となります。構築手順のような、無駄なドキュメントを作成する必要もなくなります。ドキュメントにはコードの実行に必要な情報のみを記載すればよくなり、ドキュメントの更新頻度はグッと下がります。

コードはテキストファイルであるため、バージョン管理システムの管理下に置くのがよいでしょう。なぜ変更したのか、そのエビデンスを残すことができ、変更の追跡も容易になります。また、コードレビューや問題の追跡にも役立ちます。

5.4 どこまでコード化するか

IaC においてよくある議論のひとつが、**どこまでコード化するべきか**です。「どこまで」というのは、変更頻度が少ない、一度しか実行しないような作業までコード化するべきかという意味です。これは関わるプロダクトの方針によるところもあるため、どちらがベストとは言いづらい部分ではありますが、基本的には全てをコード化しておくのがベストだと考えます。

仮に、もう二度とやらないような作業があったとしましょう。二度とやらない想定なので、わざわざ再利用可能なコードに起こすのは、一見無駄に思えます。ですが一度でも行った作業については、「なぜそうしたのか」「現在どういう状態になっているのか」というエビデンスを残す必要があります。前述の通り、このエビデンスを単なるメモとして残してしまうと、将来その環境を触る必要が出てきた時に「メモは見つかったが、果たしてこのメモが正しいのかわからない」という状態に陥ります。その時点で、そのメモにもはや意味はありません。

多少のコストには目を瞑って、あらゆる作業はコード化しておくべきでしょう。

5.5 IaC を導入したら気をつけること

IaC で環境を構築した以上、手動による作業の実施は御法度です。常にコードが正であり、コードと環境はイコールであるべきだからです。もしも IaC による構成管理下の環境を手でいじってしまうと、コードと実環境に差分が発生し、構成管理が破綻します。

例えばコードから自動構築した環境に、手作業で何らかのプラグインを追加したとしましょう。このプラグインはコードで管理されていないため、構成管理の外にあるツールとなります。構成管理ツールでこの環境を削除しようとした時に、未知のプラグインを検出し、正しく削除できなくなるといった可能性も考えられます。削除できないだけならまだいい方で、アップデート作業をしたつもりが構成管理ツールが思わぬ動作をしてしまい、環境の作り直しが発生してしまう可能性もあります。これによって、もしも本番のデータベースが作り直されてしまったら大惨事です。

とはいえ現実ではコード化のコストを嫌ったり、構成管理ツールを使いこなせないチームメンバーが気軽に手作業を行ってしまいがちです。IaC の導入においては、**こうしたメンバーへの啓蒙**もまた、重要なファクターとなるでしょう。

第 6 章　システムの稼動状態をチェックする

残念なことに、IT システムはいつか必ず壊れます。しかし壊れたままにしておくわけにはいきません。障害の予兆をしっかりとキャッチして事前に対策を打ったり、万が一にも障害が起きてしまったら、迅速に復旧対応を行うのもインフラエンジニアのお仕事です。そのために欠かせない監視とはなにか、そして監視はどのように行うべきかについて解説します。

6.1 監視とは

システムは構築を完了し、リリースしたらおしまいというわけにはいきません。むしろリリースはスタートラインに過ぎず、その後の運用こそが本番と言ってもよいでしょう。そしてシステムを安定して稼動させ続けるためには、継続的な**監視**が必要不可欠です。そして IT システムにおける監視とは、**システムの挙動や状態をチェックし続ける行為**を指します。

システム障害はいつか絶対に起こるものです。もしも障害が発生したら、ただちに復旧させなければなりません。そのためにはまず、障害が発生したという事実に気づける必要があります。サービスが停止しているのに誰も気づかず、ユーザーからのクレームで初めて障害を知るというのは最悪のパターンでしょう。また発生した障害に対応しているだけではなく、将来起こり得る障害を予測し、未然に防ぐことも重要です。Web サーバーのパフォーマンスは安定しているでしょうか。データベースサーバーのストレージが枯渇しそうになっていないでしょうか。ネットワークトラフィックはバーストしていませんか。こうしたリソース量の変化も常に意識し、障害に繋がりそうな予兆を事前に見つけ、先回りして対策を行うことも大切です。

システムを健全な状態に保つためには、こうした様々な部分に対して、常に気を配らなくてはなりません。そのために必要不可欠なのが監視なのです。ただし一口にシステムを監視すると言っても、実際にチェックしなければならない項目は膨大な数に上ります。また監視はその性質上、24 時間 365 日にわたって停止が許されません。これを人力で行うのは、現実的ではないでしょう。そこでシステム監視には、専用の**監視ツール**が利用されます。

6.1.1 メトリクス監視

それでは具体的な監視の手法や、使用するツールについて解説しましょう。システムの情報を定期的に取得し、グループごとにまとめたデータを**メトリクス**と呼びます。メトリクス自体は単なる数値に過ぎないため、そのままでは扱いづらいです。そこで一般的には値を時系列に並べ、グラフなどにプロットして可視化します。サーバーの CPU 使用率やストレージ容量、ネットワークトラフィック量を表したグラフなどは、メトリクスの典型的な例と言えるでしょう。システムのパフォーマンスを計測する上で、メトリクスは非常に有用です。メトリクスを活用すれば、負荷の急激な上昇を検出したり、メモリやストレージ容量が枯渇する前にアラートを上げるといったこともできます。そのためメトリクスは、様々な監視ツールで普遍的に利用されています。

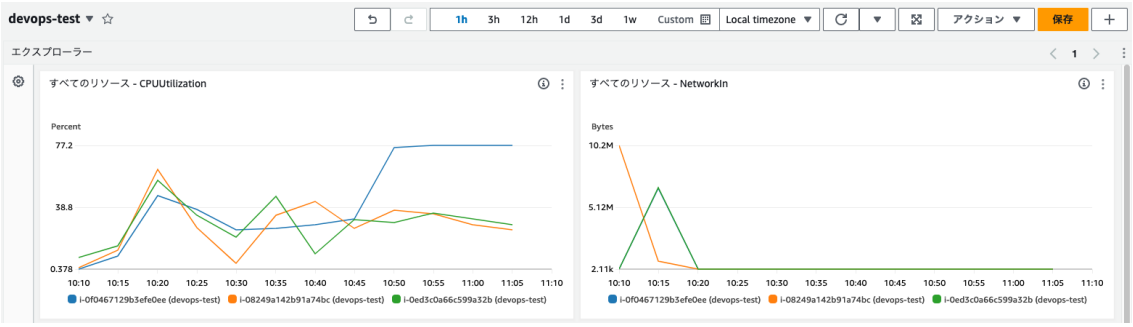

図 6.1: AWS の CloudWatch でメトリクスを表示した例

6.1.2 ログ監視

稼働中の OS やアプリケーション内では、様々なイベントやエラーが発生します。このイベントを記録として残したものを**ログ**と呼びます。ログには、どのアプリケーションにいつ何が起きたのかといった情報が記録されているため、問題の早期発見や障害時の調査において、非常に重要な手がかりとなります。またアクセスログや監査（audit）ログは、セキュリティ面からもしっかりと記録し、一定の期間、保全することが求められています。

ログには色々な形式がありますが、イベントの内容や付随データにタイムスタンプをつけ、テキストで出力するのが一般的です。従来では 1 行に 1 イベントを列挙した、テキスト形式のログが一般的でした。しかしこれは人間にとっては比較的読みやすいものの、機械がパースしづらいという問題があります。そこで最近では、JSON などのフォーマットを利用して、構造化されたログもよく利用されています。またテキストではなく、独自のバイナリ形式でログを記録するシステムも存在します。近年の Linux システムで広く利用されている **journald** は、内部的にはバイナリでログを記録しています。多くの監視ツールでは、ログを監視する機能を備えています。例えばアプリケーションのログに対して正規表現でマッチングをかけ、特定のエラーメッセージが発生したらアラートを上げるといったことができます。

図 6.2: AWS の CloudWatch Logs の例

6.1.3 外形監視

前述のメトリクス監視やログ監視は、言わばシステムの内部情報をチェックする監視です。こうしたシステム視点での監視を**内部監視**と呼びます。ですが内部監視だけでは、監視としては不十分です。例えばインターネットへの経路上でネットワーク障害が発生すると、当然ユーザーはサービスにアクセス

できなくなってしまいます。しかしこうした障害は、内部からサーバーのプロセスや負荷を監視していても、検出できないのです。そこでユーザーと同じ立場から、システムが正常に応答しているかどうかを監視することが重要になってきます。具体的には、「サーバーに Ping を送ってネットワークの疎通を確認する」、「サーバーのポートに接続可能か確認する」、「実際に API をコールして応答を確認する」などです。これらはシステムの外部から行う監視であるため、**外形監視**とも呼ばれます。

システムが動作していることと、実際にユーザーがサービスを利用できることはイコールではありません。そのため外形監視を、システムとは物理的に異なるロケーションから行うことは、非常に重要です。以下は監視 SaaS である **Datadog**[*1] による外形監視の例です。指定した URL に対して、世界中の AWS リージョンから Ping を送信し、ネットワークの疎通とレイテンシをチェックしています。

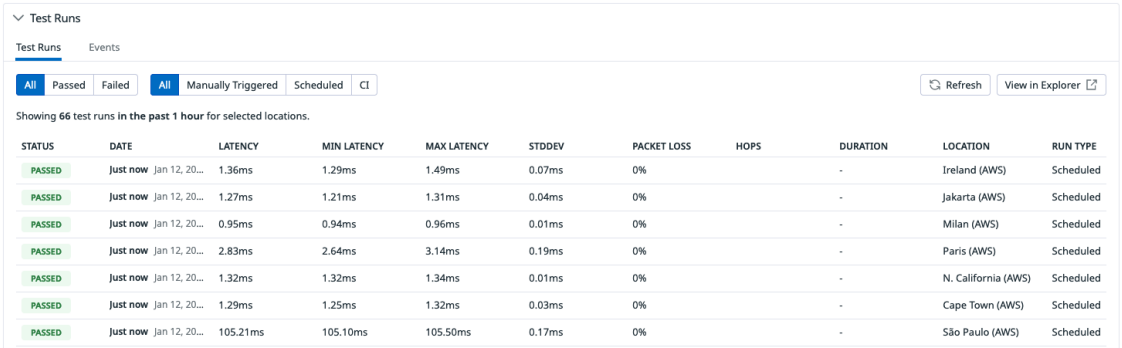

図 6.3: Datadog による外形監視の例

6.1.4 アラート

監視ツールはシステムの状態をチェックし続けますが、単に見ているだけでは意味がありません。システムが正常ではないと判断した場合には、すみやかにアラートを上げ、担当者に知らせる必要があります。そのため一般的な監視システムは、事前に設定した条件に従ってアラートを発生させる機能と、担当者への通知機能を備えています。

アラートは、監視項目に対して条件をつけることで設定します。条件の例を具体的に挙げると、「サーバーの CPU 使用率が 90% を越える」「Web サーバーが一定時間 Ping に応答しない」「アプリケーションログに致命的なエラーが出力された」といった具合です。ツールは監視によって収集したデータを常にこの条件と付き合わせており、二値的に正常/異常を判断します。そしていざアラート発生となった場合には、メール、SMS、Slack などのチャット、メッセンジャーツール、電話などを利用して（あるいは併用して）、担当者に通知します。特にメールよりも即時性が高く、かつ複数人の目に留まりやすく、電話ほど相手を拘束しないチャットは、アラートの一次通知先として便利です。

[*1] https://www.datadoghq.com/ja/

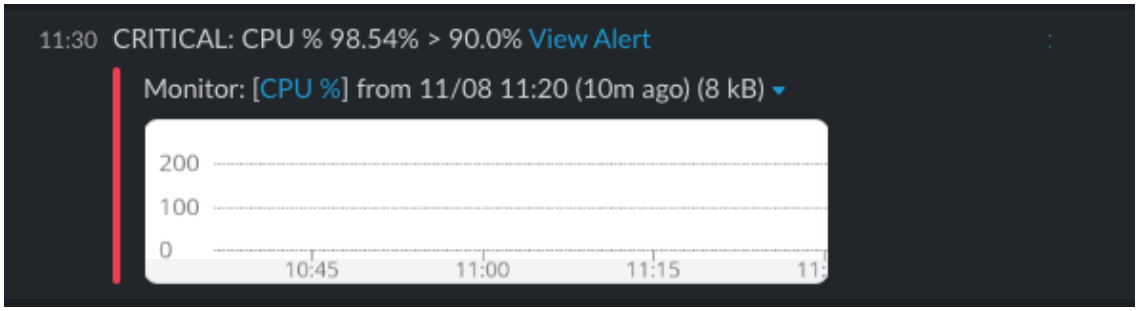

図 6.4: Slack にアラートが通知された例

こうなったら確実にアラートであると判断できるものについては、この方法はおおむね上手く動作します。ですが問題もあります。それは、アラートとする条件（閾値）を事前に、明確に設定しなければならない点です。この例で言えば、アラートとする CPU 使用率の閾値は 90% で妥当なのでしょうか。もしかすると 95% くらいまでは頑張れるのではないでしょうか。あるいは 80% でも危険かもしれません。ですが、この閾値を正確に導出する方法は、おそらく存在しないでしょう。最初は過去の知見に基いて、「ざっくり」設定するしかないはずです。そのためシステムの運用監視直後は、不適切な条件設定により、アラートの誤報が起きることも珍しくありません。

監視システムの運用には、**こうした誤報を乗り越えて、少しずつ最適な条件を追い込んでいくチューニング作業**も必要となってきます。

6.2 監視に使うツール

監視ツールには様々なオープンソースの実装や、クラウドサービスが存在します。どれも機能的にはおおむね似通っていますが、細かい違いもあるため、要件やコストに合わせてツールを選定することが大切です。

まずクラウド上にインフラを構築しており、そのクラウド事業者が専用の監視ツールを提供している場合は、その利用を第一に検討すべきです。というのもクラウドはその仕様上、クラウド事業者が提供している監視ツールでしか監視できない項目が少なからず存在するためです。例えば AWS であれば **CloudWatch**[*2]、Google Cloud であれば**オペレーション (旧称: Stackdriver)**[*3]などが利用できないかをまず検討してみましょう。

利用しているプラットフォーム向けの監視ツールが存在しない場合は、別途ツールを調達しなくてはなりません。その場合にお勧めしたいのは、監視 SaaS の利用です。なぜならば、監視はどれだけ頑張ったとしても、サービスの価値そのものに直接寄与しないためです[*4]。そのため必要な要件を満たせるのであれば、監視ツール自体の運用にかかる負荷は、可能な限り下げるのが望ましいと言えます。監視 SaaS を利用すれば、監視データを集約する監視サーバーや、データを閲覧するフロントエンド部分の運用を、SaaS の事業者に任せられます。つまり監視ツールをセルフホストする場合に比べて、インフラエンジニアの負担を大きく軽減できるのです。有名な監視 SaaS としては前述の Datadog、**NewRelic**[*5]、国産の **Mackerel**[*6]などがあります。

前述の条件に当てはまらない場合は、監視ツールのセルフホストを検討することになるでしょう。

[*2] https://aws.amazon.com/jp/cloudwatch/
[*3] https://cloud.google.com/products/operations?hl=ja
[*4] もちろんシステムの安定性が上がることで、間接的にユーザーの満足度が上がるということはあります。
[*5] https://newrelic.com/jp
[*6] https://ja.mackerel.io/

オープンソースの監視ツールとしては、**Nagios**[*7]、**Zabbix**[*8]、**Prometheus**[*9]などが有名です。ですが監視ツールのセルフホストは可能な限り避け、あくまで最後の手段とするべきだと筆者は考えています。理由としては、前述の通り監視ツールそのものに注力するビジネス的なインセンティブが低いこと。そして監視ツールの運用自体が、無視できない負担となる点です。例えば、監視ツールそのものの障害には、どう備えたらよいでしょうか。監視ツールを監視する別の監視ツールを用意するべきでしょうか。これではキリがありません。

なお監視ツールは、どれかひとつに絞らなければならないという決まりはありません。ツールにはそれぞれ得手不得手や、使い勝手、かかるコストなどに違いがあります。そのためツールの弱点を補う目的で、複数のツールを併用するのはよい考えです。例えば基本的な監視は SaaS を利用しつつ、それで監視できない項目はクラウドサービス独自の監視ツールで補う、といった使い方は珍しくありません。ツールごとの特性をよく理解した上で、効率よく組み合わせるようにするとよいでしょう。

6.3 監視のアンチパターン

大抵の監視ツールは、導入するだけで主要なメトリクスが取得されるようになっています。多くのグラフが整然と並び、リアルタイムに更新されていくダッシュボードを眺めるのは気持ちがいいものです。しかし、そのメトリクスは本当に監視する意味があるのでしょうか。意味のある監視を実現するためには、本当に監視しなければならない要素を選別し、放置できない異常を確実にキャッチできるよう、適切なアラートを設定しなくてはなりません。

一般的なセオリーに則って構築された IT システムであれば、本番環境を一切監視していないということは、さすがにないでしょう。しかし同時に、「それっぽい項目を監視して、メトリクスを眺めているだけ」で、適切な監視項目の選定やハンドリングが行われていない監視が多いのも、残念なことに事実です。ここでは監視にありがちなアンチパターンについて解説します。

6.3.1 見ている項目が多すぎる

大抵の監視ツールは、導入するだけで主要なメトリクスが取得されるようになっています。またデフォルトで充実したダッシュボードが提供されている場合も珍しくありません。ですがそれらのメトリクスのうち、本当に監視する価値のあるものは、どれだけあるのでしょうか。監視の現場で、よくありがちな会話を例に挙げてみましょう。

> Alice「このシステムは、ダッシュボードのここと、このメトリクスに注意してね」
> Bob「こっちのグラフは何を表しているんですか？」
> Alice「えーっと、よくわかんないから見なくていいよ」

サーバーの CPU 負荷、メモリ使用量、ストレージの空き容量、ディスク I/O、ネットワークトラフィック等々、ダッシュボードに所狭しと並んだメトリクスを眺めているのは楽しいですし、まさに監視をしている気分を味わうことができます。しかしダッシュボードの見栄えこそよくなるものの、不要な情報はノイズにしかなりません。もちろん、多くのメトリクスを取得することに意味がないとは言いません。ですがダッシュボードには、本当に必要な情報のみを登録すべきでしょう。そのグラフは、本当に必要なのでしょうか。

6.3.2 誤報が多すぎる

監視に誤報はつきものです。アラートの条件となる閾値を正確に導出することは難しいため、ある程度の誤報は監視の宿命として、受け入れなければならない側面もあります。しかし仕方ないと諦めるの

[*7] https://www.nagios.org/
[*8] https://www.zabbix.com/jp
[*9] https://prometheus.io/

ではなく、誤報をなくすよう少しずつ閾値をチューニングし、監視の精度を上げていく努力をすべきです。

> Bob「あれ、なんかアラートが上がっているっぽいですよ？」
> Alice「ああそれね、いつも鳴るんだ。無視していいやつだから」

見なくていいことが解っているのであれば、そのアラートは見直す必要があるでしょう。記録に残す必要があるイベントであっても、少なくともメールやチャットを使って、担当者に通知を送ることはやめるべきです。こうしたアラートを放置し続けると、アラートはいずれ狼少年になってしまい、最終的には誰もアラート自体を見なくなってしまうでしょう。

また監視初心者は、せっかく監視しているのだからと、すべてのメトリクスに閾値とアラートを設定しまいがちです。万が一にも障害を見逃さないよう、万全の体勢で望みたいという気持ちは理解できますが、これも大抵の場合は意味がないでしょう。詳しくは後述しますが、システム全体の稼動に影響を与えないイベントに対して、ひとつひとつアラートを上げる必要はありません。無駄なアラートは運用担当者を疲弊させるだけなのです。

6.3.3 イベントの詳細を追跡できていない

アラートによって障害の発生に気づき、復旧対応ができることは大前提です。ですが監視の役割は、それだけではありません。同じような障害が再発しないよう、原因の究明と解決もまた、監視の重要な役目です。メトリクスを監視していれば、リソースの変化に気づくことができます。ですがメトリクスは単なる数値でしかありませんから、何が起きたのかまでは読み取れません。

> Bob「昨夜、サービスに接続できない障害があったようですが、何があったんですか？」
> Alice「同じ時刻に、サーバーの CPU 使用率が 100% になってるね。過負荷によってコネクションを受け入れられなかったみたい」
> Bob「CPU を使い切ったのは、具体的にどのプロセスが原因なんでしょうか？」
> Alice「えっ……」

こうした例は非常によくありがちです。メトリクスだけでは不十分なため、ログや**トレース**といった情報も合わせて記録し、付き合わせて調査できる必要があります。例えば監視ツールの中には、アラートの発生時にプロセスリストをはじめとするシステムの情報一式を取得し、付随情報として記録するものもあります。こうしたツールを利用してみるのもよいでしょう。

6.4 本当に知りたいものは何かを考える

繰り返しますが監視は手段に過ぎず、あくまで目的はシステムを健全な状態に保つことです。そのためにはまず、システムがどのような振舞いをしていれば健全な状態と言えるのか、しっかりと定義するところから始めなくてはなりません。例えば一般的な Web アプリであれば、「リクエストを受け付け、許容される時間内に、想定通りのレスポンスを返せること」のように定義できるでしょう。そしてこの場合、許容される時間がいわゆる **SLO（Service Level Objectives、サービスレベル目標）**に該当します。

こうしたケースでは外形監視が有効です。ですが、Ping のようにネットワークレベルの導通を確認するものでは不十分です。例えば「Ping に応答はあるし、Web サーバーへの接続もできていたが、実はアプリケーションエラーで真っ白なページが返っていた」といった障害もありがちです。こうした問題を検出するためには、実際に API をコールし、ログインなどアプリの機能を利用する、高レベルな監視が必要になってきます。

そしてアプリが SLO を満たせているのであれば、内部のメトリクスが多少変動したからといって、それは些細な問題に過ぎません。「サーバーの CPU 使用率が 90% を越えたらアラート」などは、非常にありがちなアラートの例です。ですがアプリが時間内に正しく応答できているのであれば、このア

ラートを上げる意味はあるのでしょうか。もちろん、想定よりも負荷が上がっているのであれば、それは事実として把握しておくべきです。しかしまだ障害が起きておらず、システムが稼動しているのであれば、こうした些細な問題はアラートとして上げるべきではありません。

同様にサーバーを死活監視し、サーバーダウン時にアラートを上げるのもありがちなパターンですが、これも考えものです。一昔前のオンプレミスであればいざ知らず、近年のクラウドを利用したシステムであれば、ロードバランサーと複数インスタンスを利用した高可用性の構成は当たり前になっています。こうした構成では、サーバーが 1 台停止したからといって、ただちにシステムが機能不全を起こしはしません。またヘルスチェックに失敗したサーバーは自動的に排除され、自動起動設定により、減った分のサーバーは自動的に補充されるのが一般的でしょう。こうした自律的な復旧が可能なシステムでは、サーバーが 1 台ダウンした程度で、いちいちアラートを上げる意味はありません。もちろん想定しないサーバーのダウンは問題ですから、イベント自体の記録は残すべきですが、運用担当者を寝不足にする理由としては不十分でしょう。

6.5 インシデント管理の必要性

しつこく繰り返しますが、監視は手段に過ぎず、あくまで目的はシステムを健全な状態に保つことです。これはアラートについても同じことが言えます。障害を検出し、アラートを送信すればそれで終わりではありません。

ソフトウェア開発において、発生したバグは、**バグトラッキングシステム**を使って追跡管理するのが一般的です。なぜなら、バグをシステムで管理しないと、バグ修正に担当者がアサインされず宙に浮いてしまったり、作業状態が把握できなかったり、バグの修正がリリースから漏れてしまったりといった問題が起こる可能性があるためです。そして BTS に登録されたバグには優先度が決められ[*10]、決められた作業フローに沿って修正、テスト、リリースが行われます。

運用時における障害も、これと同じだと言えます。いつ、どのような障害が発生したのか。それはただちに対応する必要があるのか。ならば対応を担当するのは誰なのか。現在の復旧作業はどのような状態になっているのか。そして、いつ障害が解消したのか。発生した障害に場当たり的に対処するのではなく、こうした情報をきちんと管理し、あらかじめ決められたワークフローに従うべきです。これを**インシデント管理**と呼びます。

インシデント管理はバグ管理と同様に、専用のツールを用いて行います。インシデント管理のための SaaS としては、**PagerDuty**[*11]が非常に有名です。またインシデント管理について理解を深めたいのであれば、PagerDuty が公開しているドキュメント[*12]を一読しておくとよいでしょう。

6.6 よりよい監視を実現するために

まず、意味のない監視を減らしましょう。アラートをチューニングして精度を上げましょう。そもそも無駄なアラートは止めてしまいましょう。インフラは可能な限り自律的に復旧できるような構成を目指し、そもそものアラートの原因を減らすことも大切です。

また、アラートにはエスカレーションを設定することをお勧めします。例えばアラートの第一報はオンコール担当者にのみ送信し、一定時間が経過してもアラートが解消されない場合に限り、他のメンバーに第二報を送るといった具合です。適切なエスカレーションを設定することで、オンコールを担当していないメンバーの睡眠時間を守ることができます。

DevOps の柱のひとつに、自動化があります。インフラ構築、テスト、ビルド、デプロイなど、様々な場面において、コード化と自動化が推奨されています。しかし監視と障害対応は、どうしても最終的には人間の手に頼らざるを得ない部分があります。これはどうしても避けられないため、可能な限り人間に負担をかけない監視を目指しましょう。

[*10] これをバグトリアージと呼びます。
[*11] https://www.pagerduty.com/
[*12] https://response.pagerduty.com/#being-on-call

第 7 章　IaC をはじめよう

ここからは実践編として、DevOps で使用するツールの使い方や、コードの書き方について解説していきます。この章では宣言型の IaC ツール Terraform について、そもそも Terraform とは何なのか、基本的な使い方や覚えておくと便利なコマンドについて解説します。

7.1 Terraform とは

Terraform[*1]は、HashiCorp 社によって開発・保守されている、宣言型の構成管理ツールです。**HashiCorp configuration language（HCL）**[*2]フォーマットで記述されたコードによって、インフラの構成を宣言的に管理できます。

Terraform は、管理対象となるサービスが公開している API を通じて、具体的な操作を行います。そしてこうした API とのやりとりを実装しているのが**プロバイダー**です。具体的な API のコール方法はプロバイダーによって抽象化され、ユーザーは HCL のコードとして、管理リソースを記述できるようになるわけです。プロバイダーは **Terraform Registry**[*3]と呼ばれるレジストリで管理されており、**Official**、**Partner**、**Community**、**Archived** という、4 つの Tier に分けられています。例えば AWS を操作する場合であれば、**AWS プロバイダー**[*4]を利用します。AWS プロバイダーは Official に分類されています。

表 7.1: Terraform のプロバイダーの Tier

階層	説明
Official	Hashicorp によって開発・保守・検証・公開されています。
Partner	Hashicorp テクノロジーパートナープログラムによって開発・保守・検証・公開されています。
Community	コミュニティメンバーによって開発・保守・検証・公開されています。
Archived	API が非推奨になったり、関心が低かったりの理由でアーカイブされています。

特別な理由がない限り Official や Partner の使用を検討しましょう。Community プロバイダーを使用するならソースコードに問題がないことを確認しましょう。Community は誰でもプロバイダーを公開できるため悪意のあるコードを含んでいる可能性があるからです。Archived はすでにメンテナンスが終了しているため使用すべきではありません。Terraform Registry でプロバイダーを探す際には、サイドバーにあるフィルターを使い、Official や Partner に絞り込むのがお勧めです。

[*1] https://developer.hashicorp.com/terraform
[*2] https://github.com/hashicorp/hcl
[*3] https://registry.terraform.io/
[*4] https://registry.terraform.io/providers/hashicorp/aws/latest

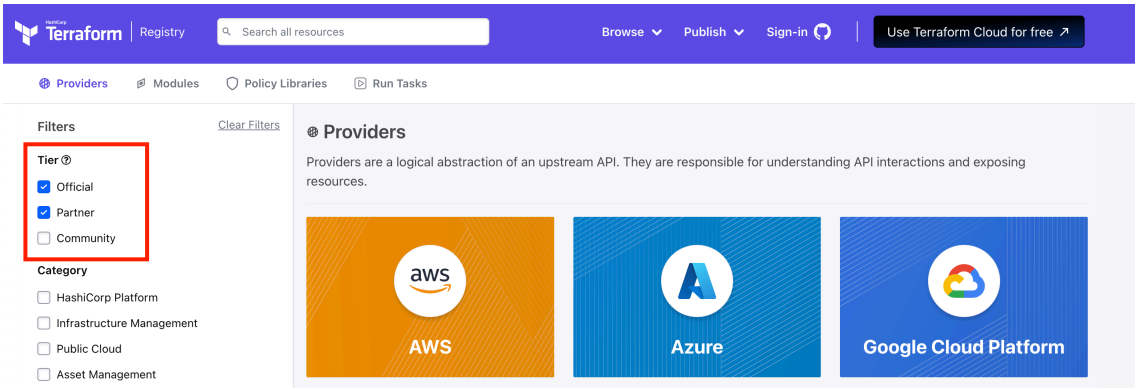

図 7.1: Terraform Registry で検索する階層を絞り込む

7.1.1 Terraform と OpenTofu

Terraform はバージョン 1.6.0 のリリース時に、そのライセンスをオープンソースライセンスである MPL 2.0 から、商用利用に制限のある BSL 1.1 へと変更しました。この変更により Terraform の**ソースコードを**商業的に利用して、競合サービスを運営できなくなりました。詳しくは HashiCorp 社のブログエントリー「HashiCorp adopts Business Source License[*5]」を参照してください。

オープンソースライセンスでなくなることに対し、多くの開発者から不満の声が上がりました。そしてこうした不満を持った開発者によって、ライセンス変更前のソースコードからフォーク（派生）したプロジェクトが **OpenTofu**[*6]です。現在 OpenTofu は Linux Foundation によりホストされています。OpenTofu は Terraform のバージョン 1.5 からのフォークですから、このバージョンの Terraform と OpenTofu には互換性があります。Terraform バージョン 1.5 系を想定したコードはそのまま OpenTofu でも実行できますし、Terraform から OpenTofu への移行方法については、公式にドキュメントも用意されています[*7]。

Terraform のバージョン 1.6.0 以降に実装された機能については、ライセンスの都合上、OpenTofu にそのまま取り込むことができません。そこで OpenTofu 側でも同じ機能を独自に実装しています。例えば test コマンドがそれに該当します。コミット履歴を見てみると Terraform にあった古い実装が削除され[*8]、新しく実装し直されている[*9]のがわかります。

また両者の `test.go` を見比べてみる[*10][*11]と、独自に実装されていることがわかります。

こうした事情により、今後 OpenTofu が、機能面で Terraform との互換性を保ち続けられるかはわかりません。そのことを十分理解した上で OpenTofu に乗り換えるか、Terraform を使い続けるかを検討するのがよいでしょう。

7.1.2 Terraform を使用するために理解しておくこと

Terraform は**ステートファイル**と呼ばれるファイルに、コードの実行結果を保存します。コードの実行結果とは、**対象のクラウドやサーバーに対して変更を加え、その結果リソースがどうなったのか**という情報です。Terraform はコードの実行時にステートファイルとコードの差分を算出し、変更があれば

*5 https://www.hashicorp.com/blog/hashicorp-adopts-business-source-license
*6 https://opentofu.org/
*7 https://opentofu.org/docs/intro/migration/
*8 https://github.com/opentofu/opentofu/commit/48c818927cb62ed19b1b944b2d35acfd9d594200
*9 https://github.com/opentofu/opentofu/commit/dfc26c2ac4acf99cef1e5fbe6383425502f91b20
*10 https://github.com/hashicorp/terraform/blob/main/internal/command/test.go
*11 https://github.com/opentofu/opentofu/blob/main/internal/command/test.go

対象に向けて処理を実行します。つまり、ステートファイルと実際のリソースの状態は、常に一致していなくてはなりません。ステートファイルとリソースの状態がズレていた場合は問題です。

例えば Terraform によって作成されたリソースを、クラウドのコントロールパネルから手作業で削除したとしましょう。ステートファイル上にはそのリソースが存在し続けていますが、クラウド上からはすでに削除された状態です。このリソースを、コードからも削除したとしましょう。Terraform はステートファイルにある通り、そのリソースがまだ存在するものとして扱います。そのためコードを実行すると、リソースの削除を試みますが、実際のリソースは削除済みのため、ここでエラーが発生します。この状態を解決するには、ステートファイルからそのエントリーを手動で削除するか、同じリソースをもう一度手で作りなおし、矛盾した状態を解消しなければなりません。逆もまたしかりです。ステートファイルにはまだ存在しないはずのリソースが手動で作成されていては、そのリソースを作成する段階でエラーが発生します。Terraform はリソースの有無をコードが実行されるまで知ることができないため、テストなどをしたとしてもこのエラーに気づくことができません。

このようにステートファイルはとても重要なファイルなのですが、デフォルトの状態では Terraform を実行したローカル環境上に保存されてしまいます。間違って削除してしまう恐れもありますし、クラウドを複数人で管理している場合は、ユーザーごとに異なるステートファイルを持つことになり、矛盾した状態となってしまいます。そのためステートファイルは共有する必要があるのですが、問題はその共有方法です。まず最初に思いつくのは、Git などのバージョン管理ツールの使用です。ですがこれはアンチパターンです。なぜならステートファイルには、パスワード等の機密情報も平文で保存されてしまうためです。ステートファイルをバージョン管理ツールの管理下に置くのは、**公式に非推奨**となっています。幸いなことに、Terraform にはこれらの問題を解決する **backend** と呼ばれる設定があります。backend を設定することによりステートファイルを外部のストレージにおくことができます。例えば AWS を利用する場合であれば、S3 を backend にするのが定石となっています。

7.2 Terraform の使い方

具体的な Terraform の使い方を解説します。

7.2.1 必須コマンド

まずは Terraform を使用する上で、必須となるコマンドです。

terraform init

Terraform を使い始める前には、まず初期化処理が必要です。この初期化処理を行うのが `terraform init` です。このコマンドの実行により、ステートファイルの格納先の設定や、プロバイダーやモジュールのダウンロードが行われます。Terraform のコードと同じディレクトリに移動するか、Terraform コマンドに`-chdir=<working-directory>`オプションを指定して、`terraform init` コマンドを実行してください。

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Installing hashicorp/aws v5.42.0...
- Installed hashicorp/aws v5.42.0 (signed by HashiCorp)

Terraform has been successfully initialized!
```

初期化が成功すると`.terraform` ディレクトリと`.terraform.lock.hcl` ファイルが作成されます。

```
.
├── .terraform
├── .terraform.lock.hcl
└── main.tf
```

.terraform ディレクトリには、コード内で使用されているプロバイダーがダウンロードされます。また、ステートファイルの保存先であるバックエンドの設定をしていた場合、.terraform/terraform.tfstate ファイルが出力されます。.terraform ディレクトリと同じ階層に作られる terraform.tfstate は、リソースの状態を管理しているステートファイルそのものですが、.terraform/terraform.tfstate は、backend の設定を含む、Terraform の状態を管理しています。同じ名前のファイルですが、管理対象は別ですので混同しないように注意してください。

.terraform.lock.hcl ファイルには使用しているプロバイダーのバージョンとそのハッシュ値が記録されます。.terraform.lock.hcl は Git などで管理することが公式で推奨されています。このファイルを固定しておくことで、プロバイダーやモジュールのバージョンを固定することができるためです。またこのファイルに変更があるということは、使用するプロバイダーのバージョンに変更があることを意味するため、レビューを通すべきだからです。

初期化処理はプロバイダーの追加やアップデート時にも必要になります。アップグレードが必要な時は-upgrade オプションを指定して init コマンドを実行します。

ステートファイルの格納先を変更した場合にも再初期化が必要です。変更を適用するには-migrate-state オプションか-reconfigure オプションを指定します。多くの場合-migrate-state オプションを使用していれば問題ありません。-reconfigure オプションの使い所については、Reconfigure flag in terraform init[*12]を参照してください。簡単に説明すると、何らかの問題でステートファイルのマイグレーションができなくなってしまった場合に、とりあえず設定だけリモートからローカルに切り替えるといった用途で使用します。

terraform plan

実際にそのコードを実行した時に、対象に対してどのような変更が入るのか、その実行計画を出力するコマンドが terraform plan です。出力の先頭に + がついている項目は、コードが実行された際に作成されるリソースを表しています。同様に-がついている項目は削除されるリソースを、~がついている項目は変更されるリソースを表しています。

```
$ terraform plan

data.aws_caller_identity.current: Reading...
data.aws_caller_identity.current: Read complete after 0s [id=123456789012]

Terraform used the selected providers to generate the following execution plan. Resource actions are i
following symbols:
  + create

Terraform will perform the following actions:

  # aws_s3_bucket.this will be created
  + resource "aws_s3_bucket" "this" {
      + acceleration_status         = (known after apply)
      + acl                         = (known after apply)
      + arn                         = (known after apply)
      + bucket                      = "poc-dummy-bucket"
      + bucket_domain_name          = (known after apply)
      + bucket_prefix               = (known after apply)
      + bucket_regional_domain_name = (known after apply)
      + force_destroy               = false
      + hosted_zone_id              = (known after apply)
```

[*12] https://github.com/hashicorp/terraform/issues/15463#issuecomment-312745775

```
      + id                          = (known after apply)
      + object_lock_enabled         = (known after apply)
      + policy                      = (known after apply)
      + region                      = (known after apply)
      + request_payer               = (known after apply)
      + tags_all                    = (known after apply)
      + website_domain              = (known after apply)
      + website_endpoint            = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

──────────────────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────────────────
──────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly t
run "terraform apply" now.
```

terraform plan の結果を見れば、何が作られて、何が削除されるのかが一目瞭然です。コードレビューをするときは、実際の terraform plan の内容も共有できると、期待通りの動作なのかを確認しやすいでしょう。

terraform apply

実際にコードを環境に適用するのが terraform apply コマンドです。terraform plan とほぼ同様の実行計画が出力された後に、コード適用の最終確認が入ります。ここで yes と入力することで、実際の処理が行われます。もしもこの段階で中止したければ、yes 以外を入力するか、もしくは Ctrl+C などで中止してください。

```
$ terraform apply

...

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_s3_bucket.this: Creating...
aws_s3_bucket.this: Creation complete after 3s [id=poc-dummy-bucket]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Apply complete!と表示されれば成功です。ただし状況によっては、terraform plan までは問題なく通るものの、実際に terraform apply するとエラーになることがあります。なぜならば terraform plan はステートファイルとコードの差分のみを出力しており、対象のクラウドやサーバーの実情を把握していないためです。そのため対象のリソースに予期しない値を設定していたり、作成しようとしていたリソースがすでに存在していたりすると、机上の terraform plan はうまく行くものの、terraform apply は失敗してしまうということが起こります。こうした事態に直面したら、エラー内容をよく読み、冷静に対処しましょう。

terraform destroy

terraform destroy は、Terraform で作成したリソースを削除します。terraform apply の時と同じように続行するか聞かれるので yes で進めてください。

```
$ terraform destroy

data.aws_caller_identity.current: Reading...
data.aws_caller_identity.current: Read complete after 0s [id=123456789012]

No changes. No objects need to be destroyed.

Either you have not created any objects yet or the existing objects were already deleted outside of Te

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

7.2.2 覚えておくと便利なコマンド

ここからはオペレーションに必須ではないものの、覚えておくときっと役立つコマンドを紹介します。

terraform fmt

Terraform には、コードのインデント等のフォーマットを整えるフォーマッターが付属しています。コードと同じディレクトリで `terraform fmt` コマンドを実行すると、拡張子が `tf` や `tfvars` のファイルに対してフォーマッターが動作します。標準出力にはフォーマットが行われた（つまりフォーマッターによって書き換えられた）ファイル名が出力されます。

```
$ terraform fmt

main.tf
terraform.tfvars
...
```

フォーマッターが適用済みかチェックするだけの`-check` オプションも用意されています。CI などで実行する場合には、コードを書き換えて自動でコミットを積むより、チェックだけを実行し通知する方がよいでしょう。標準出力にはフォーマットが適用される（つまりフォーマットが崩れている）ファイル名が出力されます。

```
$ terraform fmt -check

terraform.tfvars
...
```

terraform validate

`terraform validate` コマンドは、コード検証するためのバリデーターです。こちらも CI などで実行すると便利です。

```
$ terraform validate

╷
│ Error: Unsupported argument
│
│   on main.tf line 8, in resource "aws_s3_bucket" "this":
│    8:   backet = "dummy-${data.aws_caller_identity.current.account_id}"
```

```
An argument named "backet" is not expected here. Did you mean "bucket"?
```

このコードは、本来 bucket と記述すべきところを backet と typo しているため、バリデーターがそれを指摘しています。また単にエラーを出力するだけでなく、「もしかして"bucket"?」と提案もしてくれています。

terraform validate コマンドはコードを検証するのみで、リモートの状態や API を通じた検証は行いません。つまり、terraform apply コマンドを実行するまで気付けない問題には対応できません。より詳しい検証を行いたければ、**tflint**[*13]などの使用をおすすめします。

terraform console

Terraform には対話型のインターフェイスが用意されています。コードと同じディレクトリで terraform console コマンドを実行すると、対話的にコードが評価されます。例えば以下のコードが記述された main.tf があるとします。

```
data "aws_caller_identity" "current" {}
```

main.tf と同じディレクトリで、terraform console コマンドを実行してみましょう。

```
$ terraform console

> data.aws_caller_identity.current
{
  "account_id" = "123456789012"
  "arn" = "arn:aws:iam::123456789012:user/tf-user"
  "id" = "123456789012"
  "user_id" = "ABCDEFGHIJK1234567890"
}
>
```

対話的コンソールに data.aws_caller_identity.current と入力することで、この data リソースで取得できる値を表示できました。コンソールを終了するときは Ctrl+C か Ctrl+D を押します。

また terraform console コマンドは、標準入力からコードを受け取って実行もできます。

```
$ echo 'data.aws_caller_identity.current' | terraform console

{
  "account_id" = "123456789012"
  "arn" = "arn:aws:iam::123456789012:user/tf-user"
  "id" = "123456789012"
  "user_id" = "ABCDEFGHIJK1234567890"
}
```

7.3 Terraform のコード

Terraform のコードは HashiCorp Configuration Language（HCL）というフォーマットで記述していきます。HCL は HashiCorp によって作らた Terraform 専用の独自言語です。構文の詳しい説明については公式ドキュメントである Configuration Syntax[*14]や、Style Guide[*15]をご覧ください。

[*13] https://github.com/terraform-linters/tflint
[*14] https://developer.hashicorp.com/terraform/language/syntax/configuration
[*15] https://developer.hashicorp.com/terraform/language/style

Google も Terraform を使用するためのベストプラクティスというガイド[*16]を公開しています。

7.3.1 HCL の構文

HCL は**ブロック**という単位で構成され、1 つのブロックに対し 1 つのリソースの状態を記述していきます。

```
resource "aws_instance" "this" {
  ...
}
```

記述方法は左から順に、ブロックタイプ（上の例では `resource`）と、ブロックタイプに応じたいくつかのラベルを設定します。ブロックタイプが `resource` の場合、リソースタイプ（上の例では `aws_instance`）、リソース名（上の例では `this`）と続きます。

リソースタイプに指定できる値はプロバイダーのドキュメントページ左側にあるツリーから確認できます。

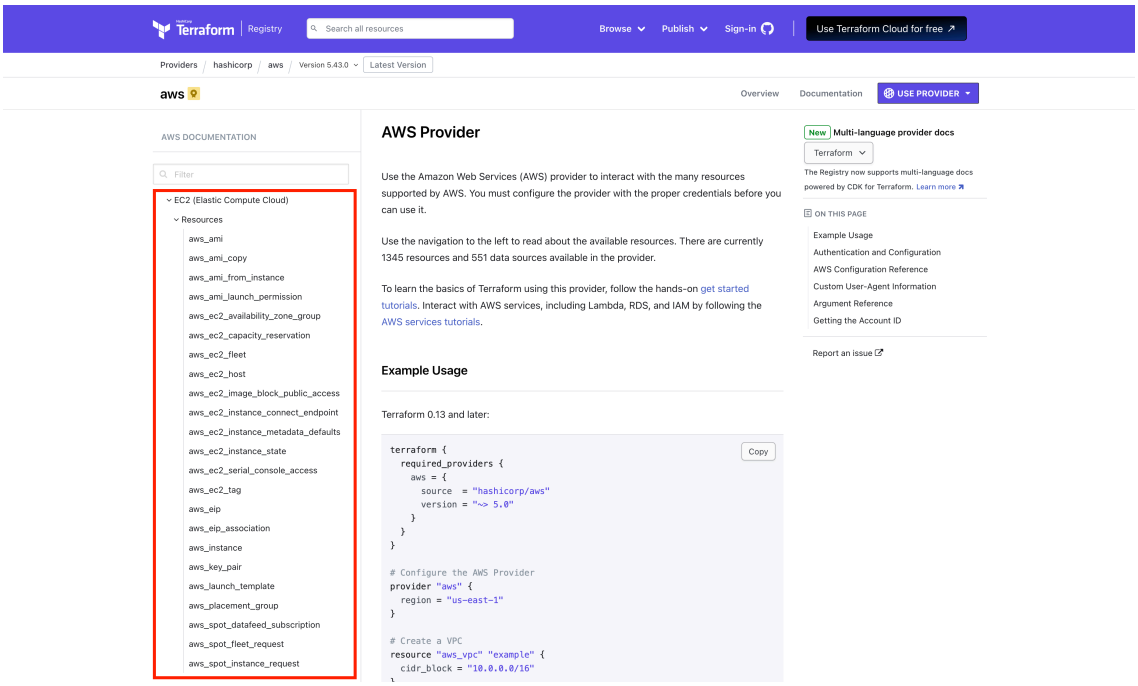

図 7.2: 使用可能なリソースタイプ一覧

使用したいリソースタイプを選択すると、そのリソースで設定できる値などが確認できます。コード化する時はこのドキュメントを確認しながら進めることになるので、使い方を覚えておきましょう。

7.3.2 リソース名

リソース名は Terraform が内部的にリソースを管理するために必要な値です。実際に構築したリソースにつけられる、サービス上での名前ではありません。リソースタイプとリソース名はセットで管

[*16] https://cloud.google.com/docs/terraform/best-practices-for-terraform?hl=ja

理されます。

リソース名の重複

同じリソースタイプを複数管理することもあるため、リソース名はユニークである必要があります。例えば以下の例では、web という名前で aws_instance リソースを 2 つ定義していますが、これはエラーとなります。

```
resource "aws_instance" "web" {}
resource "aws_instance" "web" {}
```

同じリソースを複数作成する場合は、以下のように異なる名前をつけてください。

```
resource "aws_instance" "web_1" {}
resource "aws_instance" "web_2" {}
```

Style Guide や、Google によるベストプラクティスでは、さらにいくつかのルールも提唱されています。

リソース名は単数形にする

以下の例では aws_subnet リソースに subnets と複数形の名前とつけていますが、これは非推奨です。

```
resource "aws_subnet" "subnets" {}
```

リソース名は以下のように単数形にすることが推奨されています。

```
resource "aws_subnet" "subnet" {}
```

リソースタイプを繰り返さない

以下の例では aws_subnet リソースに public_subnet という名前とつけています。ですがリソース名にリソースタイプを繰り返して含むのは非推奨です。

```
resource "aws_subnet" "public_subnet" {}
```

この場合は subnet を省略し、public だけにしましょう。

```
resource "aws_subnet" "public" {}
```

1 つしかないリソースタイプには main とつける

以下の例では aws_vpc リソースに vpc という名前とつけています。多くの場合、システムに含まれる VPC は 1 つだけでしょう。

```
resource "aws_vpc" "vpc" {}
```

このように 1 つしか存在しないリソースには、main と命名することが推奨されています。

```
resource "aws_vpc" "main" {}
```

なおこの非推奨例では、前述の「リソースタイプを繰り返さない」にも違反しています。無理に名付けるより main や this などで簡略化しましょう。

リソース名を区切る時はアンダースコアを使う

以下の例では web-server と、リソース名の単語の区切りに「-（ハイフン）」を使っています。

```
resource "aws_instance" "web-server" {}
```

ですがリソース名の区切りには、「_（アンダースコア）」を使うことが推奨されています。

```
resource "aws_instance" "web_server" {}
```

7.3.3 値の参照

HCL では、他のブロックで作成したリソースの値を参照できます。参照するにはブロックタイプ、リソースタイプ、リソース名を .（ドット）で連結します。ただし、ブロックタイプが resource の場合はブロックタイプを省略できます。

```
data "aws_ami" "al2023" {
  ...
}

resource "aws_instance" "this" {
  ami = data.aws_ami.al2023.image_id
}

output "instance_id" {
  value = aws_instance.this.id    # or resource.aws_instance.this.id
}
```

上の例では aws_ami の値を別のブロックから取得しています。この場合 data.aws_ami.al2023 は resource.aws_instance.this より先に解決される必要がありますが、ブロック間の依存関係は Terraform が自動で解決するため、コードの書き方や実行方法で意識する必要はありません。

参照できる値はリソースタイプのドキュメントに記述されている、Attribute Reference の項目で説明されています。

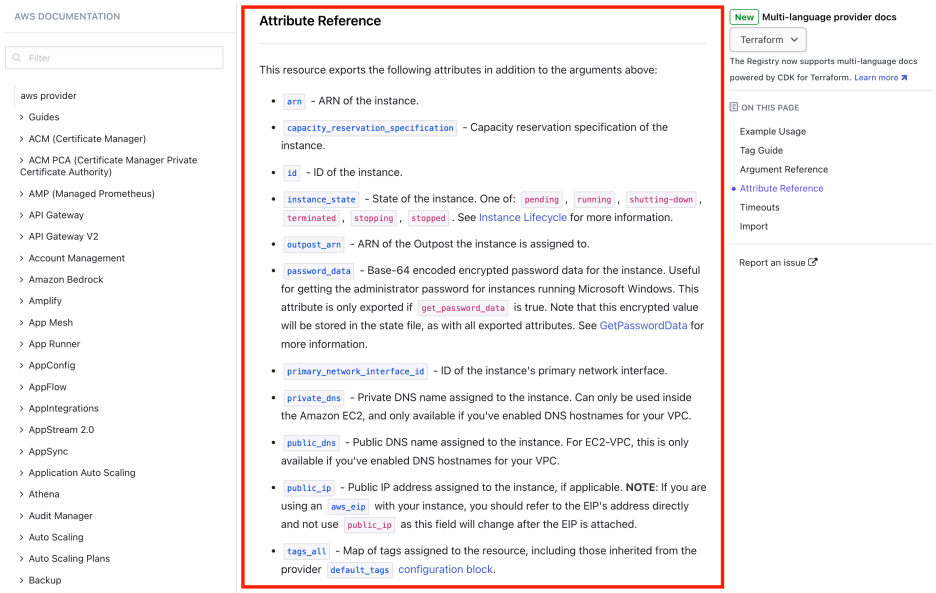

図 7.3: 参照できる値の一覧

こちらもコード化する時に必ず確認するドキュメントとなりますので、使い方を覚えておきましょう。

7.3.4 変数定義

HCL ではローカル変数とグローバル変数を定義できます。以下はローカル変数の例です。

```
locals {
    変数名 = 値
}
```

以下はグローバル変数の例です。

```
variable "変数名" {}
変数名 = 値
```

ローカル変数は、外部から値を変更できません。変数の定義は.tf 拡張子のファイルで行います。ローカル変数を参照するには local.変数名を使用します。

グローバル変数は外部から値を変更できます。.tf 拡張子のファイルで variable ブロックを定義し、.tfvars 拡張子のファイルで変数に代入します。また、特別なファイルを使用せず環境変数から値を渡すことも可能です。その場合は TF_VAR_変数名という環境変数に値を代入します。グローバル変数に設定された値を参照するには var.変数名を使用します。変数に設定できる型については Types and Values ドキュメント[*17]で詳しく説明されています。

[*17] https://developer.hashicorp.com/terraform/language/expressions/types

7.3.5 ループ

HCL はループを使って複数のリソースや値を作成できます。HCL で記述できるループは count、for_each、for の 3 つです。それぞれ使い方や動作が異なるため違いを意識して使用できるようになりましょう。

count

count は指定された回数分処理を繰り返します。以下の例では instances というローカル変数にインスタンス名を list でセットし、length() 関数で list の要素数を count にセットしています。count.index で現在のインデックス番号を取得し、Name タグにインスタンス名をセットしています。

```
locals {
    instances = ["instance1", "instance2", "instance3"]
}

data "aws_ami" "al2023" {
    ...
}

resource "aws_instance" "this" {
    count = length(local.instances)

    ami = data.aws_ami.al2023.image_id
    tags = {
        Name = local.instances[count.index]
    }
}
```

このコードが展開されると、以下のようになります。

```
resource "aws_instance" "this" {
    ami = data.aws_ami.al2023.image_id
    tags = {
        Name = "instance1"
    }
}

resource "aws_instance" "this" {
    ami = data.aws_ami.al2023.image_id

    tags = {
        Name = "instance2"
    }
}

resource "aws_instance" "this" {
    ami = data.aws_ami.al2023.image_id

    tags = {
        Name = "instance3"
    }
}
```

リソースを参照するにはインデックス番号を指定して aws_instance.this[0] や aws_instance.this.1 でアクセスできます。

count を使用する際には、1 つ注意が必要です。それはインデックスが変更された時の動作です。たとえば、ローカル変数のリストからインデックス番号 1 の値 instance2 を削除したとしましょう。動作として以下の状態を期待します。

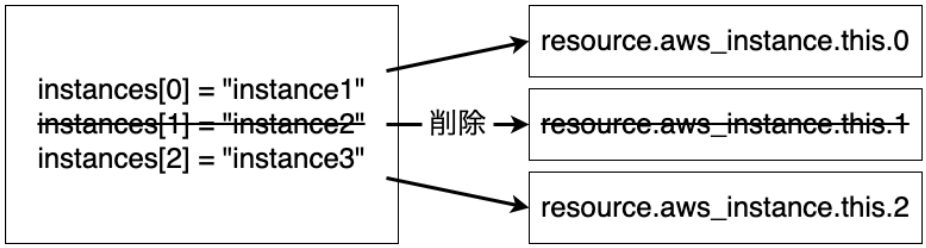

図 7.4: 期待する動作

しかし、実際の動作はこのようになります。

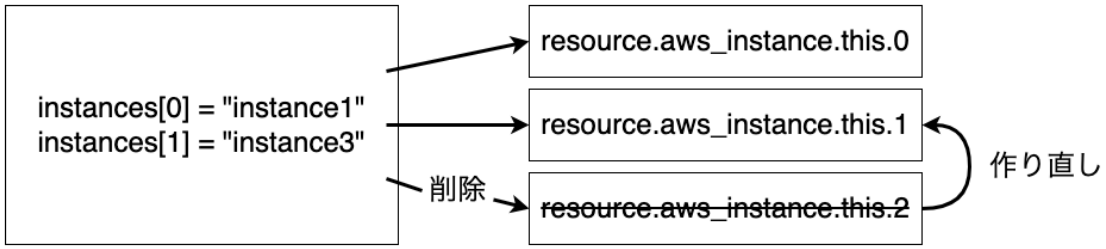

図 7.5: 実際の動作

list は途中のインデックスを削除すると、後続の値が繰り上がってきます。リソースを管理しているインデックスが変更となるため、リソースの再作成が発生します。

for_each

for_each は渡されたオブジェクトの数だけ処理を繰り返します。for_each は set 型か map 型のみを受け取るため、以下の例では list の値を toset() 関数で set 型に変更しています。set 型は重複する値を削除しソートしてくれます。ループ中、キーや値には each.key や each.value でアクセスできます。

コードを展開すると count と同様になりますが、リソースへのアクセス方法が変わります。for_each で作ったリソースへは aws_instance.this["instance1"] でアクセスできます。count はキーがインデックス番号だったのに対し、for_each は連想配列のような構造になってます。そのため list は set 型に変換し重複を削除する必要があったのです。この構造のおかげで一部の値が削除されたとしても count の時のようにインデックスが切り詰められることはありません。

```
locals {
    instances = ["instance1", "instance2", "instance3"]
}
```

```
data "aws_ami" "al2023" {
    ...
}

resource "aws_instance" "this" {
    for_each = toset(local.instances)

    ami = data.aws_ami.al2023.image_id

    tags = {
        Name = each.value
    }
}
```

for

for は count や for_each と少し毛色が違います。count や for_each はリソースを繰り返し作成していたのに対し、for はなんらかの値を加工するのに使います。以下の例では作成したリソースを for ループにかけて、それぞれのリソースからインスタンスの ID を取得するコードです。

```
locals {
    instances = ["instance1", "instance2", "instance3"]
}

data "aws_ami" "al2023" {
    ...
}

resource "aws_instance" "this" {
    for_each = toset(local.instances)

    ami = data.aws_ami.al2023.image_id

    tags = {
        Name = each.value
    }
}

output "instance_id" {
    value = [for x in aws_instance.this : x.id]
}
```

count や for_each の特殊な使い方

また count や for_each を処理の分岐に使用する方法もあります。

```
locals {
    create_instance = false
}

resource "aws_instance" "count" {
    count = local.create_instance ? 1 : 0
    ...
}
```

```
reousrce "aws_instance" "for_each" {
    for_each = local.create_instance ? [1] : []
}
```

上の例では、create_instance の値が true だったら count に 1 をセットし、false の場合は count に 0 がセットされます。for_each の場合は要素が 1 つのリストか空のリストが渡されます。どちら場合も 0 か空のリストであれば処理が実行されません。つまり、リソースの作成を変数でコントロールできるようになります。公開モジュールなどを見てみると、このような実装が至る所で確認できます。

7.3.6 サンプルコード

ここまでのコードをまとめると以下のようになります。

```
provider "aws" {
    region = "ap-northeast-1"
}

locals {
    instances = ["instance1", "instance2", "instance3"]
}

data "aws_ami" "al2023" {
    most_recent = true
    owners      = ["amazon"]

    filter {
        name   = "architecture"
        values = ["x86_64"]
    }

    filter {
        name   = "name"
        values = ["al2023-ami-2023.*"]
    }

    filter {
        name   = "virtualization-type"
        values = ["hvm"]
    }
}

resource "aws_instance" "this" {
    for_each = toset(local.instances)

    ami           = data.aws_ami.al2023.id
    instance_type = "t3.micro"

    tags = {
        Name = each.value
    }
}
```

```
output "instance_id" {
    value = [for x in aws_instance.this : x.id]
}
```

このコードを main.tf として保存し terraform apply を実行すると、AWS の ap-northeast-1 リージョンに、t3.micro の EC2 インスタンスが 3 つ起動します。実行に成功すると、起動した EC2 の instance_id が標準出力されます。

コードを見るとわかる通り、宣言型特有の望む構成が記述されているのみです。手続型のような構築手順などの記載が一切ないため、どんな構成が構築されるかコードから簡単に読み取れます。

7.4 ステートファイルの分割

多くのリソースを管理するようなコードを書き始めるのであれば、ステートファイルの分割粒度を考えておく必要があります。ステートファイルの分割粒度とは、どの粒度でリソースをまとめて作成するか、ということです。Terraform のステートファイルはディレクトリ単位で分割されます。つまり、1 つのディレクトリ以下ですべてのリソースを管理するか、ある程度の粒度でディレクトリを分割し管理するかを考える必要があります。

前者は全く分割しない方法です。これは依存関係の全てを Terraform が解決し、1 度の apply で必要なリソースを全て作成します。環境構築をするための特別な知識を必要としません。一方で、一部のリソースにだけ変更を適用しようと考えた時には Terraform やコード、クラウドの知識が求められます。意図しないリソースの再作成や環境を破壊してしまっても問題のない環境で使用する分には問題ありませんが、停止できないリソースがある場合は、そのリソースを別のコードで作成した方がよいでしょう。

後者はリソースごとのグループで分割する方法です。ここで重要なのはリソースごとの**グループ**で分割するという点です。例えば全てのリソースで分割したことを考えてみましょう。リソース間の依存関係は自分で解決する必要があり、プロビジョニングの順番が重要となってきます。これでは Terraform を使用している旨みが減ってしまいます。そこで筆者がおすすめするのはグループで分割する方法です。具体的には、それぞれの分割されたリソースは疎結合を保ちつつ、密結合となるリソースをまとめてグループ化する方法です。これは 1 つの例ですが、AWS 上に Web アプリをデプロイするインフラを構築する場合、以下のような粒度での分割が考えられるでしょう。

- 基盤となる VPC
- LB とエンドポイントの設定に DNS、SSL 証明書の発行、バックエンドインスタンス
- RDB
- NoSQL
- CDN とオリジン

基盤となる VPC には Subnet や Route Table も含まれます。なぜなら、これらのリソースは密結合しており、わざわざ分割して管理する必要がないからです。他も同じように必要となる設定はまとめてしまいます。それぞれ分割されたリソースには VPC ID などが必要となってしまいますが、それ以外はお互いをあまり必要としていない状況です。このように疎結合にしておけば LB だけ作り直す、RDB だけ更新するなど個別に変更を加えていけるので管理が楽になります。また、これくらいの粒度であればプロビジョニングする順番が重要になったとしても人間が解決できる分量でしょう。

第 8 章　CI/CD をはじめよう

4 章では CI/CD パイプラインとは何か、そして CI/CD を実現するツールについて解説しました。CI/CD がなぜ必要なのか、どういう効果があるのかはご理解いただけたのではないでしょうか。この章では、4 章でも紹介した CI/CD ツール **GitHub Actions** の使い方を解説します。

8.1 GitHub Actions とは

GitHub Actions は、その名の通り GitHub が提供している CI/CD プラットフォームです。Jenkins や GitLab のようにセルフホストする必要はなく、CircleCI や Travis CI のように、ソースコードリポジトリとは別のプラットフォームを契約する必要もありません。GitHub にアカウントがあれば誰でも簡単に使い始められます。

GitHub Actions を実行するには、その内容を定義した**ワークフローファイル**が必要です。ワークフローファイルの中身は、YAML フォーマットで記述されたテキストファイルです。ワークフローファイルをリポジトリに含めるだけで、ファイル内で定義された**ジョブ**を、任意の OS 上で実行できます。指定できる OS は、**Linux**、**Windows**、**macOS** です。ジョブにはこれらの OS 上で実行できる、任意のコマンドを含められます。

8.2 Action とは

GitHub Actions においては、すべてのジョブをスクラッチで実装する必要はありません。GitHub Actions には **Action** と呼ばれる概念があります。これはプログラミング言語で言うサブルーチンに相当するもので、再利用可能な処理の塊です。「コードをチェックアウトする」「ビルドしたコンテナをレジストリにプッシュする」といったよく使う処理は、あらかじめ Action が用意されています。そのため典型的なワークフローであれば、こうしたありものの Action を並べるだけで、効率よくワークフローを組立てられるのです。

便利な Action ですが、使用においては注意点もあります。それは誰もが自由に Action を作成し、Marketplace で公開できるという点です。企業や著名なエンジニアが公開している Action であれば、比較的安心して利用できるでしょう。ですが誰が作成したのかわからない Action には、悪意のあるコードが含まれているかもしれません。少しでも安全に Action を使用するには、Marketplace で Action を検索する際に **Verified Creator**[*1] を指定しましょう。Verified Creator とは、Action を公開しているパブリッシャーの身元と信頼性を示すバッジです。このバッジがついている Action は、GitHub によって検証された企業や個人によって公開されています。

ただし、GitHub が Action の中身まで分析しているわけではない点には留意する必要があるでしょう。Verified Creator はあくまでも、パブリッシャーの身元を保証しているだけであり、コードの品質を担保しているわけではありません。Action を使用する時は、Action の ChangeLog や Issue などにも目を通し、慎重に検討してください。

8.3 GitHub Actions の料金設定

CI/CD サービスは人間の目に見えない所で自動的に、かつ繰り返し起動する性質上、設定を間違えていると、想定外の請求がなされることも珍しくありません。GitHub Actions を使う際には、料金プランを理解し、適切に利用するよう心がけましょう。GitHub Actions の利用料金は、使用するリポジ

[*1] https://docs.github.com/ja/apps/github-marketplace/github-marketplace-overview/about-marketplace-badges

トリによって変わります。具体的には、パブリックリポジトリで利用する場合は無料です。そのためオープンソースプロジェクトなどにおいて、CI/CD を導入しやすくなっています。プライベートリポジトリで使用する場合は、以下の表のように、アカウントのプランごとに無料利用枠が設けられています。

表 8.1: GitHub Actions の無料利用枠

プラン	ストレージ	分/月
GitHub Free	500 MB	2,000
GitHub Pro	1 GB	3,000
GitHub Free for organizations	500 MB	2,000
GitHub Team	2 GB	3,000
GitHub Enterprise Cloud	50 GB	50,000

例えば GitHub Free アカウント[*2]であれば、一ヶ月あたり 2,000 分までは、無料で利用できます (2025 年 10 月現在)。詳しくは GitHub Actions の課金[*3]を参照してください。

GitHub Actions では、実際にジョブを実行するマシンを**ランナー**と呼びます。使用するランナーの OS によって、実行時間に倍率がかかります。

表 8.2: ランナー OS ごとの倍率

ランナーの OS	倍率
Linux	1
Windows	2
macOS	10

ランナーの OS が Linux の場合、倍率は 1 倍です。つまり実行に 1 分かかるジョブを実行すると、そのまま無料利用枠から 1 分の利用時間が消費されます。対して Windows の場合は、倍率が 2 倍です。そのため同じく 1 分のジョブを実行したとしても、そこに 2 倍の倍率がかかり、**無料利用枠からは 2 分の利用時間が消費される**のです。そして macOS の場合は、10 倍の倍率が適用されます。

先に述べた通り、GitHub Free アカウントであれば、一ヶ月あたり 2,000 分の無料利用枠があります。ですが Windows のランナーを使用すると、実時間としては 1,000 分しかワークフローを実行できません。macOS であれば 200 分です。ワークフローの実行時間や実行頻度にくわえ、使用するランナーの OS も考慮に入れて、最適なプランを選択しましょう。そして時間単位で課金されることから、キャッシュ等を活用した実行時間の短縮といったワークフローの最適化も、コスト面から重要になってきます。

8.4 GitHub Actions の使用方法

ここからは実際に、GitHub Actions を動かしてみましょう。GitHub Actions を使用するには、当然ですが GitHub アカウントとリポジトリが必要になります。ドキュメントを参考にアカウント[*4]とリ

[*2] 一般的な GitHub ユーザーがこれでしょう。

[*3] https://docs.github.com/ja/billing/concepts/product-billing/github-actions

[*4] https://docs.github.com/ja/get-started/start-your-journey/creating-an-account-on-github

ポジトリ[*5]を作成し、手元のマシンにクローン[*6]しておいてください。

リポジトリのルートディレクトリ直下に、.github/workflows ディレクトリを作成します。そしてこのディレクトリ内に、ワークフローを定義する YAML ファイルを作成して、リポジトリにコミットします。

```
$ mkdir -p .github/workflows
$ touch .github/workflows/actions.yaml
```

上記の例では、ワークフローを定義するファイル名を actions.yaml としました。ワークフローファイルは拡張子が.yml か.yaml であれば、任意のファイル名をつけて構いません。GitHub Actions はファイル単位でワークフローを定義・実行するため、どのようなワークフローなのか、判断しやすい名前をつけることを心がけましょう。

作成した YAML ファイルに、下記の内容を記述してください。なおこの例では、解説用に最小限の処理しか記述していません。ワークフローに設定できる内容を、より詳しくしりたい場合はワークフロー構文[*7]を参照してください。

```
# ワークフロー名を「example workflow」に設定
name: example workflow

# リポジトリへの Push をトリガーに、Actions を実行する
on: push

jobs:
  # ジョブ名を「echo」に設定
  echo:
    # ランナー OS として最新の Ubuntu を使用する
    runs-on: ubuntu-latest
    steps:
      # ここにジョブ内で実行する処理を列挙する
      # uses: は公開されている Action を呼び出す
      # ここでは actions/checkout を実行して、ランナー OS 上にリポジトリをクローンしている
      - uses: actions/checkout@v4
      # run: はシェルコマンドを実行する
      # ここでは hello world を表示するだけ
      - run: echo 'hello world'
```

コードをコミットし、GitHub へプッシュします。

```
$ git add .github/workflows/actions.yaml
$ git commit -m 'add actions'
$ git push origin main
```

GitHub Actions が実行されたか見てみましょう。リポジトリの上部にある「Actions」タブから確認できます。

[*5] https://docs.github.com/ja/repositories/creating-and-managing-repositories/creating-a-new-repository

[*6] https://docs.github.com/ja/repositories/creating-and-managing-repositories/cloning-a-repository

[*7] https://docs.github.com/ja/actions/using-workflows/workflow-syntax-for-github-actions

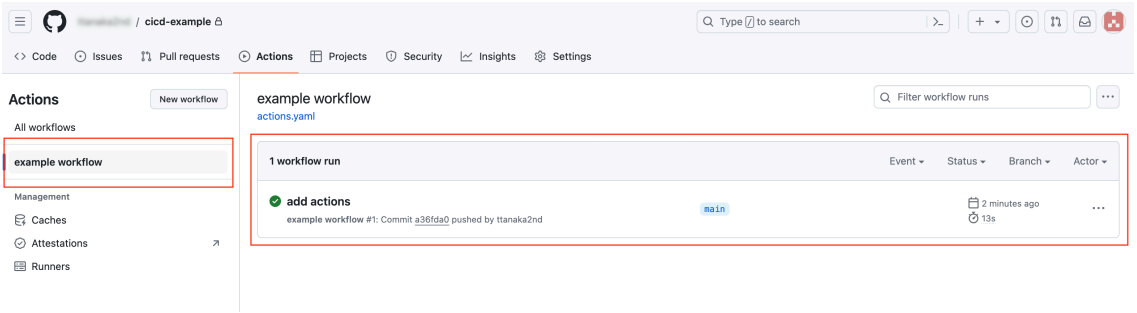

図 8.1: Actions の画面

左側の赤枠で囲われている部分を選択するとワークフローの中身が確認できます。`example workflow` となってる部分がワークフロー名です。ワークフローファイルの上部で設定した `name: example workflow` と一致します。新たに別のワークフローが追加されると項目が増えていくので、ワークフローにはわかりやすい名前をつけておきましょう。

中央の赤枠で囲われている部分が実行されたワークフローです。`add actions` となっている部分がコミットメッセージになります。このワークフローを選択するとワークフローで何が実行され、どういう結果になったのか確認できます。

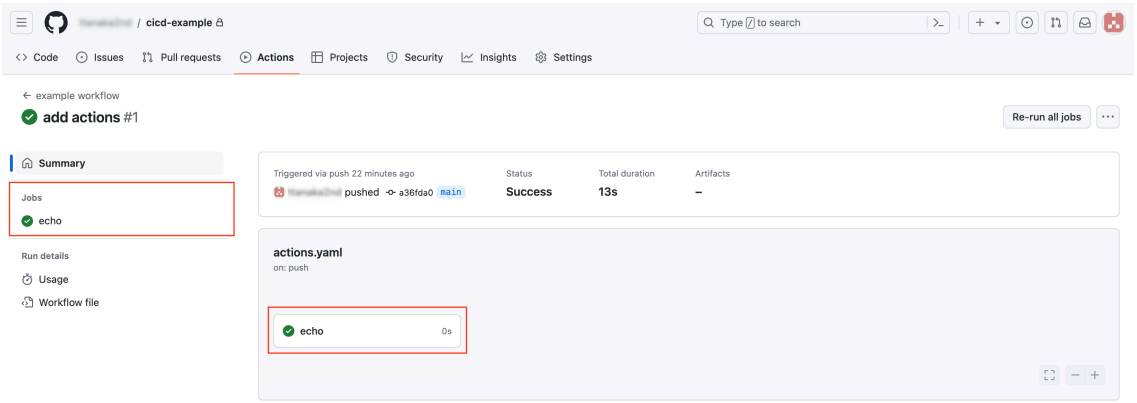

図 8.2: ワークフローの画面

赤枠で囲われている部分がジョブです。ジョブの左側にステータスが表示されていて、緑のチェックマークは成功を表しています。ジョブが失敗したら赤の×マークになるため、どのジョブが失敗しているのかここから判断できます。ジョブを選択するとジョブの中身を確認できます。

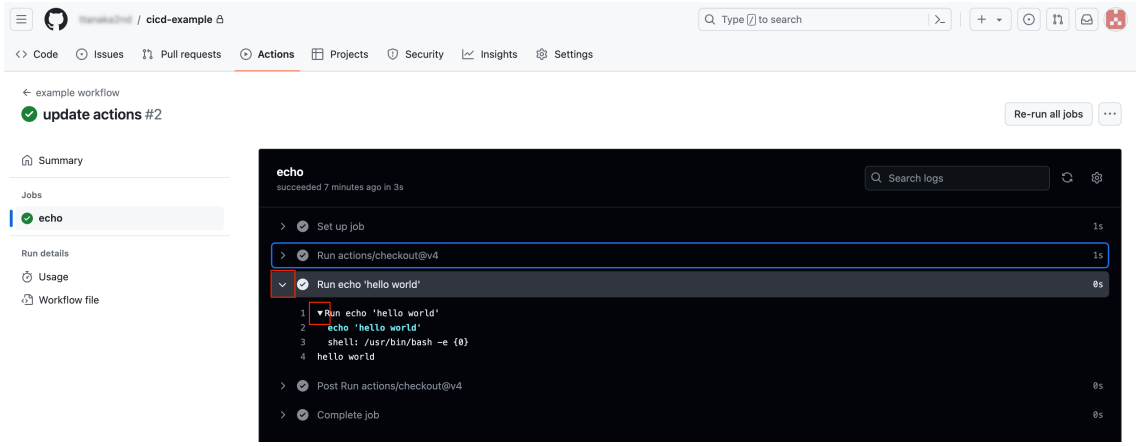

図 8.3: ジョブの実行画面

中央の黒い画面がジョブの実行履歴です。上から順に処理されています。通常は>マークの横に実行したジョブが表示されているだけですが、赤枠で囲われている部分をクリックすることで詳細な情報が展開されます。ジョブが失敗した時などに展開して詳細情報を確認するといいでしょう。

8.5 GitHub Actions のデバッグ

プログラミングと同じく、実装したワークフローが一度で正しく動作するとは限りません。想定外のエラーなどが起きた場合は、デバッグを行い、ワークフローを修正する必要があります。例として、わざとエラーが発生する以下のジョブをリポジトリにコミットしてみます。`.github/workflows/actions.yaml` を以下のように書き換えてください。

```
name: example workflow

on: push

jobs:
  echo:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          repository: ${{ github.repository }}-not-found
      - run: echo 'hello world'
```

このコードでは `actions/checkout` に対してパラメータを設定しています。`actions/checkout` は `repository` というパラメータを指定することで、指定されたリポジトリをランナー上にクローンします。`${{ github.repository }}`は今回使用しているリポジトリ名にアクセスするための変数です。末尾に`-not-found` という文字列を追加して存在しないリポジトリを指定しました。これでチェックアウトの Action は失敗します。

修正したコードをリポジトリに Push します。

```
$ git add .github/workflows/actions.yaml
$ git commit -m 'add fail job'
$ git push origin main
```

Actions タブからワークフローを見てみましょう。ジョブが失敗しているのがわかります。

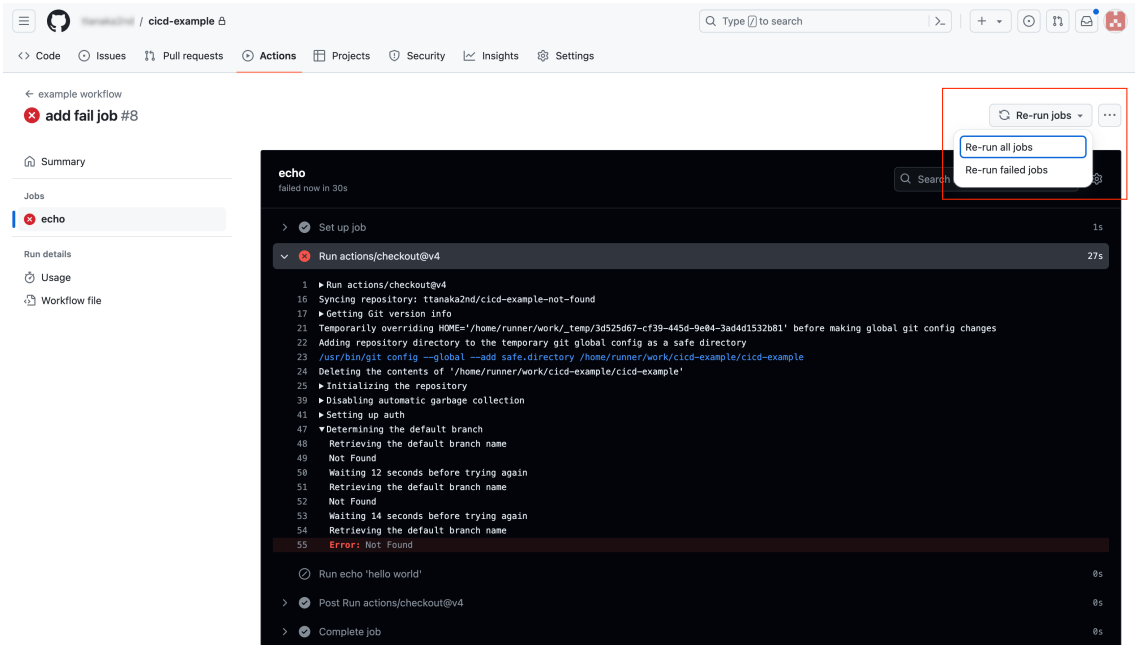

図 8.4: 失敗したジョブの実行画面

今回の例であれば、エラーの原因は明確なため、それほど問題にはなりません。ですが場合によっては、ここからエラーの原因が読み取れないこともあるでしょう。そこでデバッグログを有効にすることをお勧めします。

赤枠内の `Re-run jobs` をクリックしジョブを再実行してみましょう。ジョブの再実行には `Re-run all jobs` と `Re-run failed jobs` の 2 つの選択肢があります。前者は全てのジョブを再実行するのに対し、後者は失敗したジョブだけを再実行します。ここではどちらを選んでも問題ありません。ジョブが複数あった場合、最初から順に実行していかないといけない場合は前者を、失敗したジョブだけで実行できるのであれば後者を選択するのがいいでしょう。

どちらかを選択すると再実行画面が出力されます。

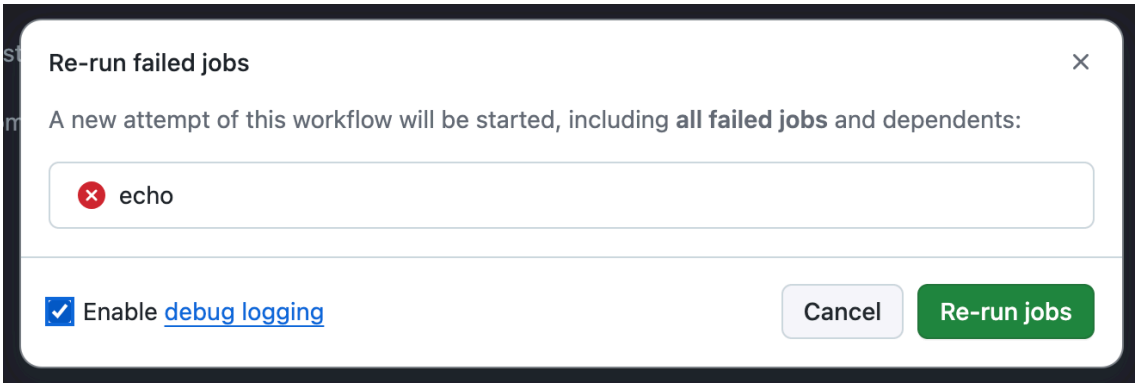

図 8.5: 再実行画面

この画面の左下にある `Enable debug logging` にチェックを入れることでデバッグログを出力でき

ます。

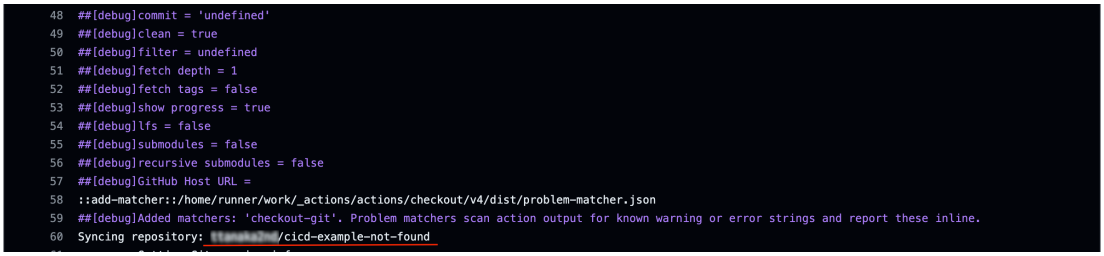

図 8.6: デバッグログを有効化した実行画面

紫色で##[debug]と書かれている部分がデバッグログになります。デバッグログを有効にする前ではわからなかった情報が表示されるようになりました。なぜ期待通りに動かないのか悩んだら、デバッグログを有効にしてみましょう。

これが GitHub Actions の基本的な使い方になります。GitHub Actions は非常に高機能なため、ここですべてを解説することはできません。公式ドキュメント[*8]が充実しているため、ぜひそちらを熟読してください。ワークフロー同士、ジョブ同士の連携ができたり、プルリクエストを処理できたりなど、GitHub を使った開発を楽にしてくれる設定や、面白い使い方などが解説されています。

[*8] https://docs.github.com/ja/actions

第 9 章　監視をはじめよう

6 章では、システム運用における監視の重要性について解説しました。この章では実践編として、AWS の監視サービスである CloudWatch を利用して、監視と異常時のアラームを設定する例を紹介します。内容自体はあくまで簡単な設定例となりますが、具体的な監視の設定をはじめる際の参考にしてください。

9.1 EC2 インスタンスの起動

今回は AWS を使い、Amazon Linux の EC2 インスタンスを対象に監視を設定します。VPC やインターネットゲートウェイ等のリソースは、既に作成済みのものとします。このあたりの設定については、本記事の範囲外となるため省略します。

図 9.1: テスト用のインスタンスを起動する

EC2 のインスタンスが起動したら、インスタンス一覧から該当のインスタンスを選択し、ウィンドウ下部に表示されている**モニタリング**タブをクリックしてください。

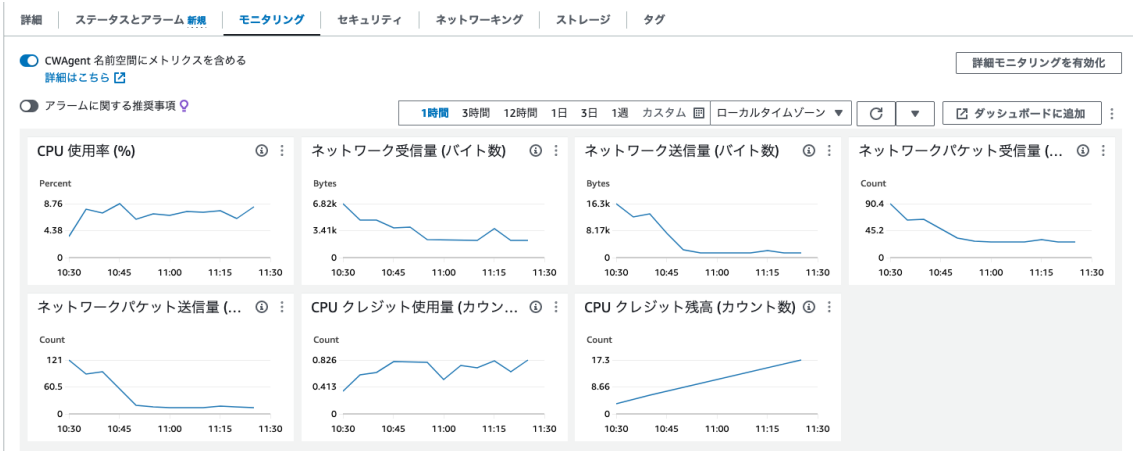

図 9.2: EC2 の簡易的なモニタリング

CPU 使用率、ネットワーク送受信量、CPU クレジット残高といった、ごくごく基本的な情報であれば、これだけで監視ができていることがわかります。

9.2 CloudWatch でメトリクスを監視する

EC2 のダッシュボードから確認できる情報は、ごくごく基本的なメトリクスのみでした。ですが**CloudWatch**[*1]を利用すれば、もう少し詳細にメトリクスを確認できます。まだ何も監視の設定をしていないよ、と思うかもしれませんが、ご安心ください。EC2 では CPU 使用率といった基本的なメトリクスは、インスタンスを起動するだけで自動的に収集されるようになっています。

CloudWatch を開き、**すべてのメトリクス**をクリックしてください。続いて **EC2 →インスタンス別メトリクス**の順にクリックします。

図 9.3: CloudWatch のメトリクス

インスタンス別メトリクスを開くと、EC2 インスンタンスのメトリクスが、インスタンス別にリストアップされます。ここから表示させたいメトリクスにチェックを入れると、ひとつのグラフとして表示できます。

図 9.4: EBS のリードとライトの状況をグラフに重ねて表示した例

最近の多くの Web アプリは、**ダッシュボード**と呼ばれる機能を備えています。本来は車の計器盤を指す言葉で、速度計や回転計、燃料計、水温計などが配置されており、ドライバーが車の状態を瞬時に把握するための装置です。Web アプリのダッシュボードも同様に、アプリの現在のデータを、わかり

[*1] https://aws.amazon.com/jp/cloudwatch/

やすく整理して可視化するためのインターフェイスです。ダッシュボードには様々なウィジェットやメトリクスを配置でき、ユーザーが自由にカスタマイズできるのが一般的です。

CloudWatch もダッシュボードを備えています。まず左ペインの**ダッシュボード**」→**ダッシュボードの作成**から新しいカスタムダッシュボードを作成しておきましょう。そして前述のようにメトリクスからグラフを作成したら、**アクション→ダッシュボードに追加**で、ダッシュボードにグラフを配置できます。以下は **CPU クレジットの残量と使用状況、ネットワークの送受信トラフィック、EBS のリードとライト状況**を表す 3 つのグラフを作成し、ダッシュボードに配置してみた例です。

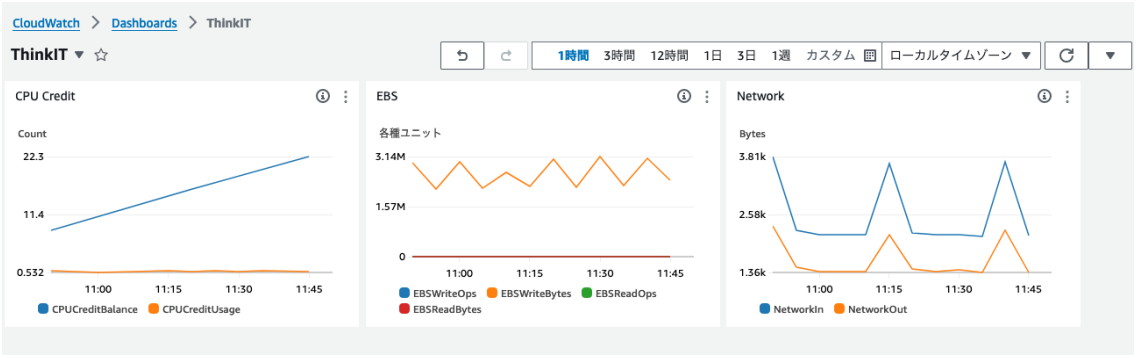

図 9.5: 自分にとって必要な情報だけを見やすく配置する

CloudWatch は多くのメトリクスを取得しているため、都度必要なメトリクスを探すのは手間ですし、時間もかかります。システムの状況を把握しやすくするためにも、必要な情報だけを見やすく整理することを心がけましょう。なおカスタムダッシュボードは別途料金がかかりますので、その点には注意してください。

9.3 より詳しい監視を行うには

一般的な監視ツールの利用経験がある方は、CloudWatch のメトリクスを見て、必要なメトリクスが足りないと思ったのではないでしょうか。具体的にはサーバー監視でよくある**メモリ使用量**や**ディスク使用量**といったメトリクスが存在しません。先ほど EC2 を起動しただけで自動的に監視が始まると述べましたが、実は標準で監視される項目は、非常に限定されています。より詳細なメトリクスを収集したい場合は、EC2 インスタンスに **CloudWatch エージェント**をインストールする必要があります。

まず前準備として、CloudWatch へ情報を書き込める権限を、EC2 インスタンスに付与します。ドキュメント[*2]を参考に、**CloudWatchAgentServerPolicy** ポリシーがアタッチされたロールを作成してください。

[*2] https://docs.aws.amazon.com/ja_jp/AmazonCloudWatch/latest/monitoring/create-iam-roles-for-cloudwatch-agent.html

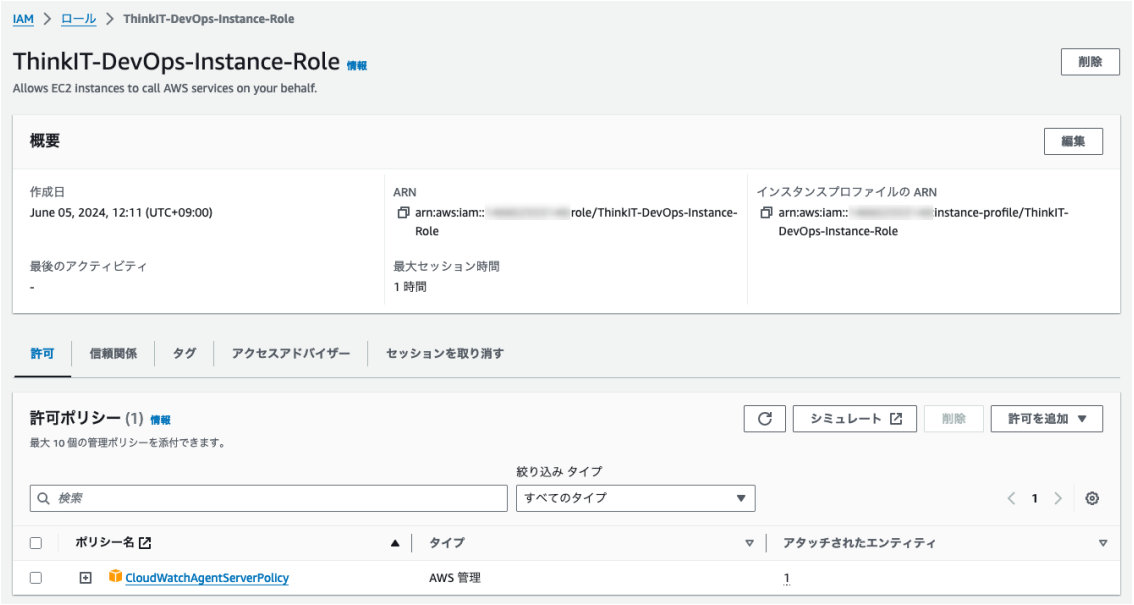

図 9.6: 作成したロール

EC2 インスタンスの一覧でインスタンスを選択し、**アクション→セキュリティ→ IAM ロールを変更**をクリックします。アタッチする IAM ロールを選択するページに遷移しますので、先ほど作成したロールを選択してください。

図 9.7: CloudWatch に書き込めるロールをインスタンスにアタッチする

続いてインスタンスにログインし、CloudWatch エージェントと、将来的にカスタムメトリクスを収集できる用に、**collectd** をインストールしておきます。以下のコマンドを実行してください。

```
$ sudo dnf install amazon-cloudwatch-agent collectd
$ sudo systemctl enable --now collectd.service
```

CloudWatch エージェントの設定は、対話的なウィザードを使って行います。以下のコマンドを実行

してください。

```
$ sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-config-wizard
```

ほとんどの設定はデフォルトのままでよいのですが、今回はログ監視を行わず、System Manager のパラメータストア[*3]も使用しないように設定しました。収集するメトリクスセットとしては、もっとも詳細な **Advanced** を選択しています。ウィザードについて、詳しくはドキュメント[*4]を参照してください。

ウィザードによって作成された設定の雛形は、`/opt/aws/amazon-cloudwatch-agent/bin/config.json` に保存されています。ですが実際に CloudWatch エージェントのプロセスは、`/opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.d/`以下から設定を読み込みます。そこで以下のコマンドを実行してください。設定を反映した後に、CloudWatch エージェントのサービスを起動します。

```
$ sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a fetch-config -m ec2 -c file
$ sudo systemctl enable --now amazon-cloudwatch-agent.service
```

しばらく待つと、CloudWatch のメトリクスに **CWAgent** というカスタム名前空間が表示されます。CloudWatch エージェントが収集したメトリクスは、その中に集められています。

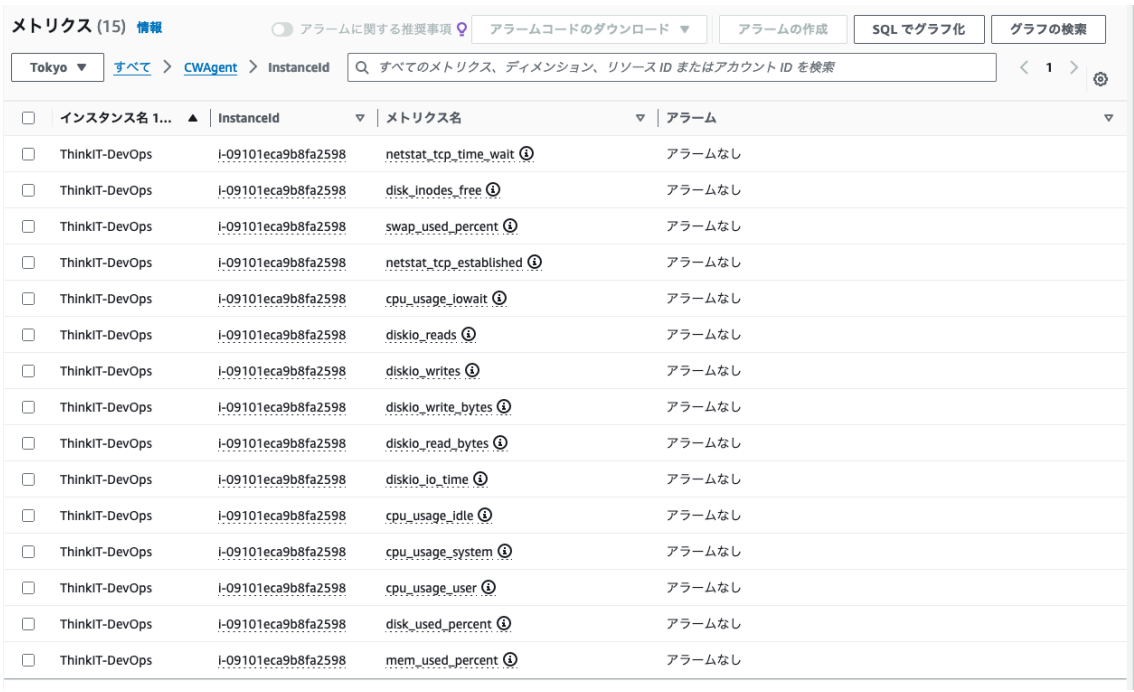

図 9.8: インスタンス ID ごとに収集されたシステムレベルのメトリクス

[*3] https://docs.aws.amazon.com/ja_jp/systems-manager/latest/userguide/systems-manager-parameter-store.html

[*4] https://docs.aws.amazon.com/ja_jp/AmazonCloudWatch/latest/monitoring/create-cloudwatch-agent-configuration-file-wizard.html

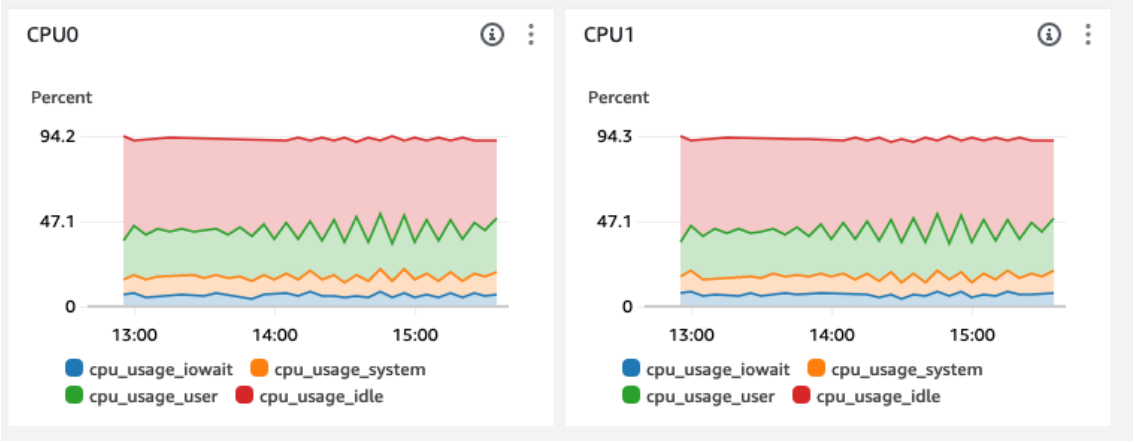

図 9.9: コアごとの CPU の利用状況を詳細なグラフにした例

9.4 アラームの設定

メトリクスは見られるようになりましたが、それだけでは不十分です。異常が発生した時に気づけるよう、監視したい項目には**アラーム**を設定しましょう。ここでは例として、データベースサーバーなどでありがちな、サーバーのストレージの枯渇に対してアラームを設定してみます。まず CloudWatch の左ペインから**すべてのアラーム**を選択し、**アラームの作成**をクリックします。対話的なアラームの設定がはじまりますので、まずはアラームのトリガーとなるメトリクスを選択します。**メトリクスの選択**をクリックし、CloudWatch で収集しているメトリクスを選択してください。

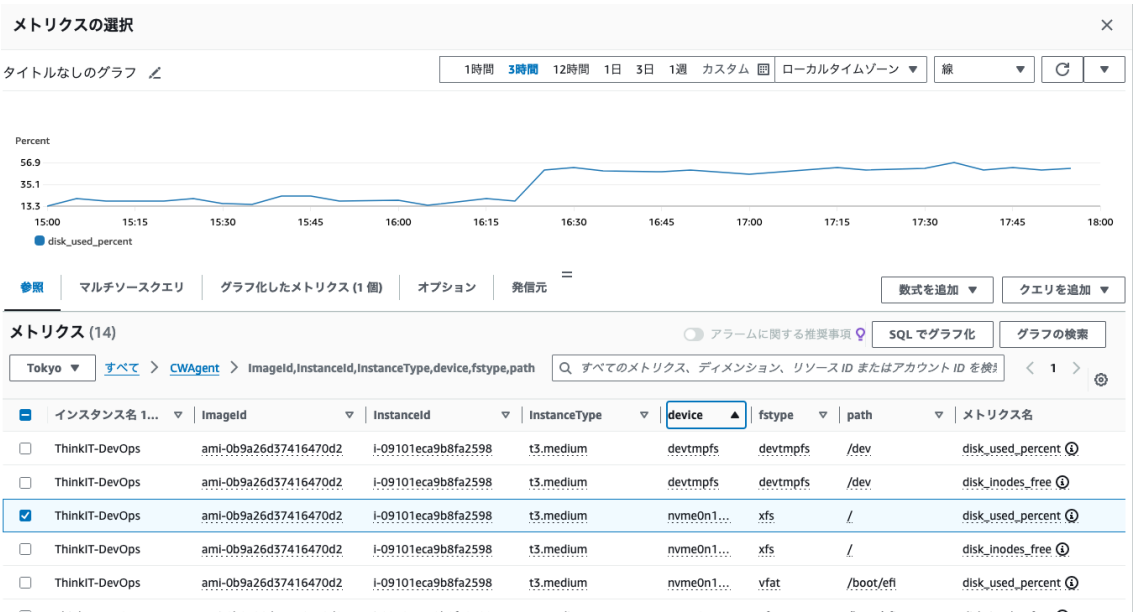

図 9.10: サーバーのルートディレクトリの disk_used_percent を選択した

続いてアラームが発生する条件を設定します。

図 9.11: しきい値を 80（%）より大きいとした

通知先を設定します。通知には AWS の **SNS** を利用するため、SNS トピックが必要です。今回は特定のメールアドレス宛てに通知する、新規トピックを作成しました。

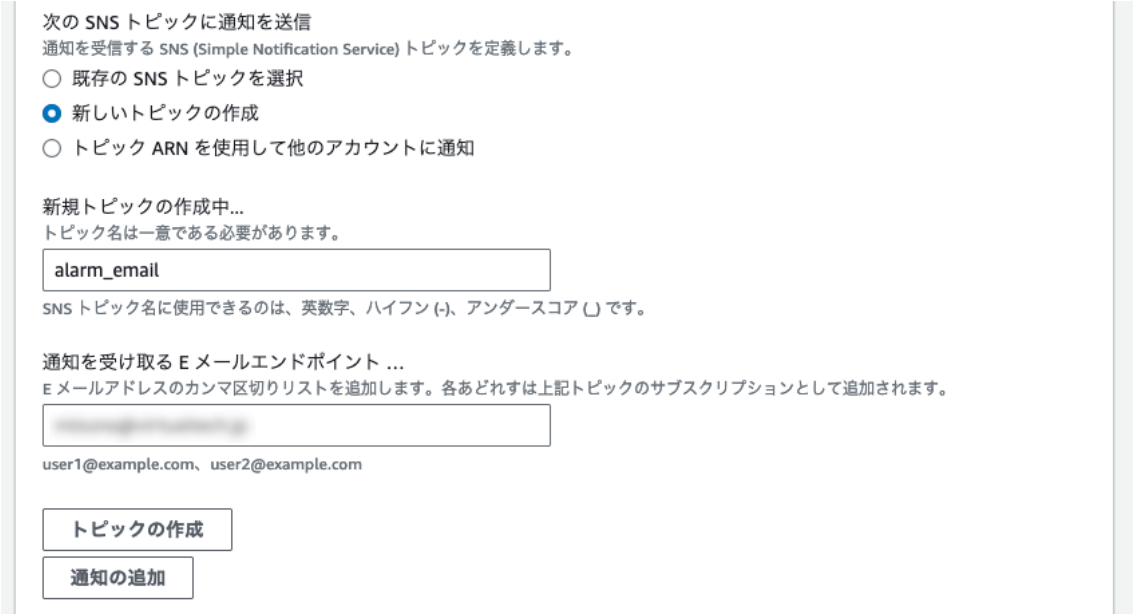

図 9.12: トピックの作成

トピックの名前と通知を受け取るメールアドレスを入力し、**トピックの作成**をクリックします。このメールアドレス宛てに確認のメールが送信されていますので、指示にしたがって **confirm** のリンクを

クリックしておいてください。もちろん既存のトピックが存在するのであれば、それを使うことも可能です。

最後にアラームに名前をつけて完了です。

図 9.13: アラームにはわかりやすい名前をつける

これでディスク使用量が 80% を越えた時にアラームが発生します。以下は急激にサーバーのストレージが消費されはじめた例です。このままではストレージの枯渇によってサーバーが停止してしまう可能性もありますが、80% を越えた時点でアラームが発報されるため、異常に気づくことができます。

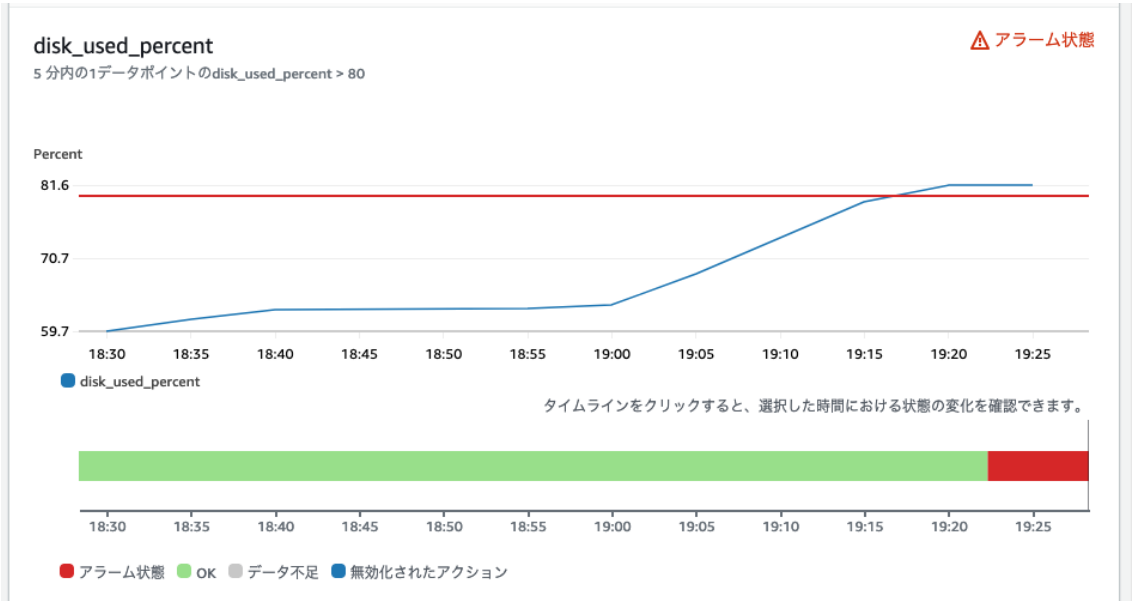

図 9.14: アラームが発生した時のグラフ

以上が、AWS における基本的なメトリクス監視の方法となります。メトリクスを収集し、アラームを設定するだけであれば、AWS の基本的な機能だけで簡単に行えることが理解できたのではないで

しょうか。

この章で紹介した内容は、EC2 インスタンスの情報を集めて監視する、いわゆる内部監視です。ですが実際にサービスを運用する場合は、ユーザーに対して正しく応答を返せているのかを監視する、外部監視が必須となるでしょう。実は CloudWatch には、スクリプトを実行して API を外部から監視する機能が用意されています。この機能は **CloudWatch Synthetics**[*5]と呼ばれています。CloudWatch Synthetics については解説しきれないため、ここでは名前の紹介に留めます。詳しくはドキュメントを読んでみてください。

また CloudWatch エージェントはメトリクス以外にも、ログファイルや、**AWS X-Ray**[*6]のトレースを収集することもできます。オブザーバビリティ的な観点から見ると、収集するデータはメトリクスだけでは不十分です。こうした機能もあるということを念頭に置いて、より高度な監視システムを考えてみてください。

[*5] https://docs.aws.amazon.com/ja_jp/AmazonCloudWatch/latest/monitoring/CloudWatch_Synthetics_Canaries.html

[*6] https://aws.amazon.com/jp/blogs/news/using-the-unified-cloudwatch-agent-to-send-traces-to-aws-x-ray-jp

あとがき

本書を最後まで読んでいただき、ありがとうございました。DevOps を支える技術要素について、駆け足で解説してきましたが、いかがでしたでしょうか。

本書の執筆にあたって筆者が重視したのは、「DevOps は決して難しいものではない」ということを実感していただくことでした。確かに DevOps には多くの技術要素が含まれており、一見すると覚えることが膨大に思えるかもしれません。しかしそれぞれの技術は独立したものではなく、また DevOps を構成する個々のツールも、決して複雑なものではありません。それらが相互に関連し合いながら、より良い開発・運用体験を実現するために設計されています。

本書でも紹介した Docker、Git、CI/CD、IaC といった技術は、DevOps という大きな枠組みの中で活用することで、さらなる真価を発揮します。本書を通じて、それぞれの技術がどのように連携し、どのような価値を生み出すのかを理解していただけたなら幸いです。また実践編では、IaC、CI/CD、監視について、実際の開発現場でもそのまま応用できるよう、具体的な動かし方を紹介しました。まずはこれらのツールを実際に触ってみて、その効果を体感してください。小さな自動化でも、積み重ねることで大きな効率化につながります。

DevOps の実践において重要なのは、完璧を求めすぎないことです。最初からすべてを自動化しようとせず、手作業で行っている作業の中から、繰り返し行うものを見つけて、一つずつ自動化していけば十分です。コードの品質チェックから始めても良いですし、デプロイの自動化から始めても良いでしょう。重要なのは、継続的に改善していく姿勢です。

また本書では技術面に焦点を当てましたが、DevOps の真の価値は、技術による効率化を通じて、チーム全体がより創造的で価値の高い仕事に集中できるようになることです。単調な作業から解放され、Dev と Ops が協力して、新しい機能の開発やユーザー体験の向上により多くの時間を割けるようになったとき、DevOps の本当の意味を実感していただけるはずです。

技術は日進月歩で進化しています。本書で紹介したツールも、将来的にはより良いものに置き換わっていくかもしれません。しかし DevOps の根底にある「自動化」「継続的改善」「コラボレーション」といった考え方は、どれだけ使う技術が置き換わっても、決して変わることはないでしょう。

最後に、DevOps は一人で実践するものではありません。チームのメンバーと知識を共有し、協力しながら取り組むことで、その効果は何倍にも増幅されます。本書で学んだ内容を、ぜひ周囲の方々とも共有していただければと思います。

それでは、みなさんの DevOps 実践が成功されることを心より願っております。

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
日本仮想化技術株式会社、技術部所属。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 たけのこ派。

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。

今日からはじめる **DevOps** 実践ガイド

2025 年 10 月 20 日　発行
　発行所　VirtualTech Japan Inc.
　連絡先　VirtualTech Japan Inc. (https://virtualtech.jp/)

Get started with

DevOps today

Virtualtech Inc.