IT技術者のための入門教材

クラウドネイティブ教科書

Ver. 1.0.0

open your NEXT future
LPI-JAPAN
Linux Professional Certification
https://lpi.or.jp

クラウドネイティブ教科書

1.0.0
LPI-Japan 発行

クラウドネイティブ教科書

まえがき

このたび、特定非営利活動法人エルピーアイジャパンは、クラウドネイティブ技術者教育に利用していただくことを目的とした教材「クラウドネイティブ教科書」（以下、本教科書）を開発し、インターネット上にて公開し、提供することとなりました。

本教科書は、多くの教育機関や技術者の皆さまから、クラウドネイティブの原理と実践を「基礎」から学習するための教材や学習環境の整備に対するご要望があり、開発したものです。

公開にあたっては、本教科書に添付されたライセンス（クリエイティブ・コモンズ・ライセンス）の下に公開されています。

本教科書は、最新の技術動向に対応するため、随時アップデートを行っていきます。

本教科書の最新情報は以下の Web ページをご参照ください。

`https://linuc.org/textbooks/cloudnative/`

本教科書の目的

本教科書の目的は、LinuC レベル 2 相当以上を前提とする技術者が、クラウドネイティブの構造と原理を理解し、実践を通して学習することにあります。

Kubernetes やコンテナの操作手順を覚えることではなく、なぜその設計が必要になったのかを構造として捉えることを重視しています。

本教科書でカバーされない範囲

実習手順の記述を簡潔にするため、LinuC レベル 1 の学習範囲に含まれる Linux の基本的な操作やシステム管理についての記述は必要最低限になっています。また、自分で調べることも学習において重要であるため、あえて調べてみる余地を残しています。分からない部分は別途調べて理解を深めてください。

想定している学習環境

本教科書は一人で独学自習できることを想定しています。実習は第 2 部に集中しており、第 1 部および第 3 部以降は読み物中心です。操作手順の詳細は第 2 部を参照してください。

第 2 部のハンズオン（標準）

第 2 部では、**Killercoda** を標準の実習環境としています。Killercoda は、ブラウザ上で Kubernetes などを学習できるハンズオン環境です。Kubernetes クラスタが起動済みの状態から、`kubectl` で YAML を適用する形式で実習を進めます。Google アカウントなどを利用して開始でき、ローカル PC へのインストールや環境構築を行わなくても利用できます。詳細は第 2 部第 2 章を参照してください。

必要な前提

実習を行うネットワークは、インターネットに接続できることを前提としています。Killercoda の利用や、本教科書の Web ページの参照に必要です。

第 2 部では **kubectl** を使いますが、操作を暗記することより、クラスタの状態を観察することを目的としています。Linux の基本的な操作（シェル、ファイル操作など）は、LinuC レベル 1 相当の知識を前提としています。

ローカル環境（任意）

Minikube、Kind、k3s、Docker Desktop などを用いて、ローカル PC 上で Kubernetes を動かすこともできます。環境によって挙動や前提が異なるため、本教科書では Killercoda を標準としています。ローカル環境へ展開する際の注意点は、第 2 部第 4 章で整理しています。

教室での利用

教室などで複数の受講者が学習する場合でも、基本的に各受講者がブラウザから Killercoda を利用して個別に実習を行うことを想定しています。ネットワーク上で Killercoda へのアクセスが制限されている場合は、事前に接続可否を確認してください。講師が共有の Kubernetes 環境を用意する場合は、第 2 部の手順がその環境でそのまま使えるか、事前に確認してください。

全体の流れ

本教科書では、以下の通りに学習を進めます。

序章 クラウドネイティブとは何か

本教科書の使い方と、クラウドネイティブを学ぶ意味を整理します。

第 **1** 部 基礎ステップ

コンテナ、Kubernetes、クラウドとオンプレの違いなど、原理を理解します。

第 **2** 部 実践ステップ

Killercoda などの環境でハンズオンを行い、Kubernetes の振る舞いを体感します。

第 **3** 部 応用ステップ

マイクロサービス、継続的デリバリー、サービス間通信など、アプリケーションを届けてつなぎます。

第 **4** 部 運用ステップ

モニタリング、トレーシング、セキュリティなど、見える化して守ります。

第 **5** 部 展開ステップ

社会・ビジネスへの展開、FinOps、AI との接続など、技術を価値に変えます。

第 6 部 総括

これまでの内容を構造として総括します。

付録 実践への扉

KCNA などの資格と学習ロードマップを示します。

執筆者・制作者紹介

本教科書は、オープンなプロジェクト形式で開発を行っています。企画段階から意見交換を行い、事前の技術的な調査、執筆、レビューなどをプロジェクトのメンバーで分担して行っています。

田染 寿之 (株式会社シャインソフト)

本教科書は、クラウドネイティブ技術をこれから学ぶ皆さんと、その学習を支援する方々の一助となることを願い作成いたしました。クラウドネイティブは、多くの技術や OSS によって構成されているため、個々のツールや手順の理解だけでは全体像を捉えにくい領域でもあります。本書では、特定の製品や実装方法だけではなく、「なぜその仕組みが必要になったのか」「どのような課題を解決するために設計されたのか」という構造理解を重視しました。本書が、クラウドネイティブを学ぶための入口として役立てば幸いです。

開発協力者紹介

勝村 友博 (株式会社野村総合研究所)

クラウドネイティブ技術はその領域の広さやプロダクトの多さから学習を始めるまでの心理的ハードルが高いように感じます。しかしながら昨今のビジネス環境において競争力を得るには避けては通れない領域でもあります。本書がクラウドネイティブ技術を学ぶ皆様の一助となることを願っています。

古藤 寛大 (株式会社野村総合研究所)

私自身、日頃の業務で Kubernetes をはじめとしたクラウドネイティブ技術に触れる機会はありましたが、「クラウドネイティブとは何か」を体系的に説明することには難しさを感じていました。本書の執筆を通じて、その概念や技術のつながりを改めて整理する貴重な機会となりました。本書が、読者の皆さまの理解や学習の一助となれば幸いです。

高棹 大樹 (株式会社野村総合研究所)

自分自身が Kubernetes をはじめとしたクラウドネイティブ技術を学んだ際は、全体を見渡す様な教材が無く手探りで学習しました。今回、技術全体を俯瞰した教科書を作成する本プロジェクトにお声掛け頂き、その苦労の軽減にお役に立てればと思い参加させて頂きました。本書を通して、クラウドネイティブ技術に触れる良いきっかけとなれば幸いです。

著作権

使用に関する権利

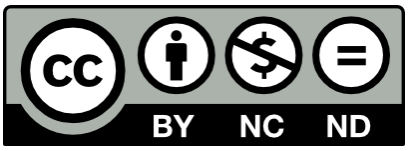


図 1: CC BY-NC-ND 4.0

表示

本教科書は、特定非営利活動法人エルピーアイジャパンに著作権が帰属するものであることを表示してください。

非営利

本教科書は、非営利目的で教材として自由に利用することができます。

商業上の利得や金銭的報酬を主な目的とした営利目的での利用は、特定非営利活動法人エルピーアイジャパンによる許諾が必要です。ただし、本教科書を利用した教育において、本教科書自体の対価を請求しない場合は、営利目的の教育であっても基本的に使用できます。その場合も含め、LPI-Japan 事務局までお気軽にお問い合わせください。

＊営利目的の利用とは以下のとおり規定しております。営利企業または非営利団体において、商業上の利得や金銭的報酬を目的に本教科書の印刷実費以上の対価を受講生に請求して本教科書の複製を用いた研修や講義を行うこと。

改変禁止

本教科書は、改変せず使用してください。本教科書に対する改変は、特定非営利活動法人エルピーアイジャパンまたは特定非営利活動法人エルピーアイジャパンが認める団体により行われています。

フィードバック

フィードバックは誰でも参加できる Slack で受け付けていますので、積極的にご参加ください。Slack 参加の詳細は以下の本教科書の Web ページを参照してください。

`https://linuc.org/textbooks/cloudnative/`

図 2: https://linuc.org/textbooks/cloudnative/

本教科書の使用に関するお問合せ先

特定非営利活動法人エルピーアイジャパン（LPI-Japan）事務局

お問い合わせ：https://lpij.tayori.com/f/textbookinfo/

図 3: https://lpij.tayori.com/f/textbookinfo/

目次

序章：クラウドネイティブとは何か **1**
この教科書の使い方〜正しく理解するために〜 . . . 1
1. なぜ今なのか . . . 1
2. この教科書の構造 . . . 1
3. 構造で理解するとは何か . . . 1
4. 読み方 . . . 2
5. 再現性 . . . 2
クラウドネイティブとは何か . . . 2
1. 問いの提示 . . . 2
2. 背景：技術進化ではなく「前提の変化」として . . . 2
3. 本質：静的最適化から動的安定へ . . . 3
4. CNCF の定義 . . . 4
5. 学ぶことで見える世界 . . . 4
6. まとめ . . . 5

第 1 部：基礎ステップ〜実行単位が変わったという話〜 **6**
導入 . . . 6
第 1 章：コンテナの基礎と Docker 〜実行単位が変わったという話〜 . . . 6
1. なぜ「実行単位」を見直す必要があったのか . . . 6
2. 仮想化（VM）が変えたこと . . . 7
3. コンテナという実行単位 . . . 7
4. 単位として扱うという発想 . . . 8
5. Docker の位置づけ . . . 8
6. コンテナとマイクロサービスは別の話 . . . 8
7. 次に残る問い . . . 9
第 2 章：Kubernetes の全体像と主要コンポーネント〜分散した実行単位を成立させ続けるという話〜 9
1. コンテナが増えた世界で何が問題になったのか . . . 9
2. Kubernetes は何を引き受けるのか . . . 9
3. 状態を扱い、差分を埋め続ける . . . 9
4. 状態を扱う側と実行する側 . . . 9
5. 責務として分割された構造 . . . 10
6. Kubernetes は OS ではない . . . 10
7. 次に残る問い . . . 11
第 3 章：クラウドとオンプレの特徴〜どこで動かすかを、構造として考える〜 . . . 11
1. なぜ議論が噛み合わないのか . . . 11
2. クラウドという前提 . . . 12
3. オンプレミスという前提 . . . 12
4. 何が違うのか . . . 12
5. 前提を選ぶということ . . . 13
6. 次に残る問い . . . 13
第 4 章：Linux / ネットワークの基本整理〜すべての前提になっている足場を言葉にする〜 . . . 14
1. なぜ今、基盤に戻るのか . . . 14
2. Linux という前提 . . . 14
3. 資源は安定しているとは限らない . . . 14
4. ネットワークという前提 . . . 14
5. 失敗は前提である . . . 15
6. 誰が前提を引き受けるのか . . . 15

7. 次に残る問い 15
第 1 部 まとめ〜原理を理解し、構造として捉える〜 15

第 2 部：実践ステップ〜前提を体感する〜 17
導入 17
第 1 章：実践環境の選択肢〜Kubernetes はどこで動かすのか〜 17
1. なぜ環境の話から始めるのか 17
2. 前提を揃えるということ 18
3. この部での扱い 19
第 2 章：実践環境の統一〜前提を揃えて観察する〜 19
1. なぜ環境を揃えるのか 19
2. この部での扱い 19
3. 環境の位置づけ 20
4. 実践に入る前の確認 20
第 3 章：ハンズオン課題〜Kubernetes の振る舞いを体感する〜 20
1. この章の目的 20
2. この章での見方 21
3. この章で登場する主なリソース 21
4. この章のまとめ 23
3-1 ハンズオン：Pod / Deployment / Service 〜壊して、戻って、任せる〜 23
0. 事前確認（現在の状態を見る） 23
1. Pod を作成する（単体の実行単位） 23
2. Pod を削除する（壊してみる） 24
3. Deployment を使って Pod を管理する 25
4. レプリカ数を変更（状態を変える） 26
5. Service を通じて Pod にアクセスする 26
6. まとめ（このハンズオンで確認したこと） 27
3-2 ハンズオン：ConfigMap / Secret / Volume 〜設定と実行を切り離す〜 27
0. 前提確認 27
1. ConfigMap を作成する（設定を Pod の外に置く） 28
2. ConfigMap を Pod から参照する 28
3. Pod を削除する（設定は残るか？） 29
4. ConfigMap を変更する（設定はどこに効くか） 30
5. Volume を使う（データの置き場所を考える） 30
6. Pod を削除する（データは残るか？） 31
7. Secret について（ここでは体験しない） 31
8. まとめ（このハンズオンで確認したこと） 31
3-3 ハンズオン：Gateway API とアクセス確認〜外から入る責務はどこにあるのか〜 32
0. 前提確認 32
1. Service の役割を確認する（内部向けの接続点） 32
2. Gateway API の前提（CRD と Controller）を確認する 33
3. Gateway を作成する（外部からの入口を定義する） 33
4. HTTPRoute を作成する（流れを定義する） 34
5. Gateway 経由でアクセスする 35
6. Pod を削除する（構造は崩れるか） 35
7. 責務整理 35
8. 何を「やっていないか」が重要 36
9. まとめ：このハンズオンで確認したこと 36
3-4 kubectl コマンドの基本整理〜操作ではなく「状態を見る」ための道具〜 37

1. kubectl とは何か 37
2. kubectl get 37
3. kubectl describe 38
4. kubectl apply 38
5. kubectl delete 38
6. kubectl scale 39
7. kubectl expose 39
8. kubectl logs / exec 39
9. まとめ 39
第 4 章：ローカル環境へ展開する場合の注意点 ～自由が増えるということは、責任が増えるということ～ 40
1. 学習環境とローカル環境の違い 40
2. 環境差が表に出るポイント 40
3. 自由と責任 41
4. 第 1 部の前提との接続 42
5. まとめ 42
第 2 部 まとめ～体感したものを、設計につなげる～ 42

第 3 部：応用ステップ～アプリケーションを届け、つなぐ～ 43
導入 43
第 1 章：マイクロサービスとアプリケーション構成～分けることで、何が起きるのか～ 43
0. 前提の置き方 43
1. なぜ分割が現れるのか 43
2. 分けると何が起きるのか 44
3. 複雑さは移動する 44
4. 責務としての分割 45
5. 分割と実行環境 46
6. 選択であって正解ではない 46
7. 次に残る問い 46
第 2 章：継続的デリバリーと GitOps ～変更を届け続けるということ～ 47
1.「一度デプロイすれば終わり」ではなくなった理由 47
2. 継続的デリバリーは「自動化」ではない 47
3. なぜ「人が直接デプロイしない」構造になるのか 47
4. GitOps という考え方 48
5. Kubernetes と GitOps の関係 48
6. 継続的デリバリーの意味 49
7. 次に残る問い 49
第 3 章：サービス間通信制御～複雑さを、どこで引き受けるか～ 49
1. 分割された瞬間に、通信は「設計課題」になる 49
2. 通信制御は「どこかで必ず行われている」 50
3. 複雑さは「消える」のではなく「移動する」 50
4.「導入するか」ではなく「引き受けられるか」 50
5. 次に残る問い 51
第 4 章：通信の振る舞いをどう扱うか～リトライ・タイムアウト・可観測性～ 51
1. 通信は「結果として現れる」 51
2. 振る舞いを決めていない通信は不安定になる 52
3. 振る舞いは「ルール」として定義される 52
4. リトライとタイムアウト 52
5. 見えないものは制御できない 52
6. 可観測性という考え方 53

7. 振る舞いはどこで引き受けるのか . . . 53
8. まとめ . . . 53
第 3 部まとめ〜応用ステップで扱ってきた「判断」とは何だったのか〜 . . . 54
技術は「答え」ではなく「責務の引き受け先」である . . . 54
それでも残る問い . . . 55

第 4 部：運用ステップ〜見える化して、守る〜 **56**
第 1 章：モニタリングと可観測性〜見えるだけでは、判断できない〜 . . . 56
1. 分散した瞬間に、「全体が見えなくなる」 . . . 56
2.「見えているのに分からない」状態 . . . 57
3. 観測できても、判断できない . . . 57
4.「見る」から「判断する」へ . . . 58
5. 可観測性とは何か . . . 58
6. モニタリングとの違い . . . 58
7. 次に残る問い . . . 59
第 2 章：トレーシングと分散システムの可視化 . . . 59
1. 分散システムで起きる問題 . . . 59
2. 分散トレーシングという考え方 . . . 60
3. トレースデータの構造 . . . 61
4. 代表的なツールの役割 . . . 62
5. 次に残る問い . . . 62
第 3 章：セキュリティとアクセス制御〜壊さないために、何を制限するのか〜 . . . 62
1. 分散した瞬間に、「壊せる場所」が増える . . . 62
2.「壊せる」というリスク . . . 62
3. アクセス制御（RBAC） . . . 63
4. ポリシー管理 . . . 63
5. Secrets 管理 . . . 63
6. 次に残る問い . . . 64
第 4 章：運用とトラブルシューティング〜判断し続けるための設計〜 . . . 64
1. 運用とは「継続させること」である . . . 64
2. 障害対応とは何をしているのか . . . 64
3. なぜツールが必要になるのか . . . 65
4. 改善し続けるという前提 . . . 65
5. 信頼性という設計（SRE） . . . 66
6. まとめ . . . 66
第 4 部まとめ . . . 66

第 5 部：展開ステップ〜技術を価値に変える〜 **68**
第 1 章：社会的課題とクラウドネイティブの解決力 . . . 68
1. 社会の前提が変わったという事実 . . . 68
2.「作ること」より「成立させ続けること」が難しくなった . . . 68
3. なぜ構造が必要になるのか . . . 69
4. クラウドネイティブが解いている問題 . . . 69
5. 技術ではなく構造として捉える . . . 70
6. 次に残る問い . . . 70
第 2 章：SES から価値提供型への転換 . . . 70
1. 分解は責任の分け方を変える . . . 70
2. 価値は「どこまで責任を持つか」で決まる . . . 71
3. 作業では扱えなくなる理由 . . . 71

4.「何をするか」から「何を成立させるか」へ . . . 71
5. 構造が価値を決める . . . 72
6. 次に残る問い . . . 72
第 3 章：FinOps ～コストは構造で決まる～ . . . 72
1. なぜコストの問題が現れるのか . . . 72
2. コストは観測できなければ制御できない . . . 72
3. FinOps が扱っているもの . . . 73
4. コストは設計に現れる . . . 73
5. コストは設計を制約する . . . 73
6. この章が示していること . . . 73
7. 次に残る問い . . . 74
第 4 章：AI 時代における構造の再定義 . . . 74
0. クラウドネイティブと AI は同じ流れの中にある . . . 74
1. AI もまた責任の分け方を変える技術だ . . . 74
2.「使う側」と「成立させる側」 . . . 74
3. AI を「使うだけ」では成立しない . . . 75
4. 観測できるかどうかが分岐になる . . . 75
5. それでも変わらないもの . . . 75
6. この章が示していること . . . 75
第 5 部まとめ . . . 76
構造は社会とつながっている . . . 76
それでも残る問い . . . 77

第 6 部：総括 **78**
1. 構造を扱うということ . . . 78
2. それでも残る問い . . . 79

付録：実践への扉（資格と学習ロードマップ） **80**
この付録の位置づけ . . . 80
1. Linux 技術者認定「LinuC（リナック）」のご紹介 . . . 80
LinuC レベル 1 . . . 80
LinuC レベル 2 . . . 81
LinuC レベル 3 プラットフォームスペシャリスト・セキュリティスペシャリスト . . . 81
LinuC レベル 4 システムアーキテクト . . . 81
2. KCNA（Kubernetes and Cloud Native Associate） . . . 82
KCNA とは . . . 82
試験概要 . . . 82
KCNA が扱う領域 . . . 82
本書との対応 . . . 83
3. CKA / CKAD / CKS（中級～上級資格）との違いと進め方 . . . 83
中級資格では「責任範囲」が変わる . . . 83
CKA（Certified Kubernetes Administrator） . . . 83
CKAD（Certified Kubernetes Application Developer） . . . 84
CKS（Certified Kubernetes Security Specialist） . . . 84
どこから進むべきか . . . 85
4. 学習プラン例 . . . 85
1：段階的に学びたい場合 . . . 85
2：運用寄りに進みたい場合 . . . 86
3：開発・設計寄りに進みたい場合 . . . 86

４：セキュリティ寄りに進みたい場合 87
まとめ 87

序章：クラウドネイティブとは何か

この教科書の使い方〜正しく理解するために〜

この教科書は、操作や手順を覚えるためのものではない。
変化する環境の中で、システムをどう成立させるかを理解するためのものである。

1. なぜ今なのか

システムを取り巻く前提は変わった。

変化は例外ではなくなり、
分散は避けられなくなり、
不確実性は前提になった。

この状況の中で、「作ること」より
「成立させ続けること」が難しくなっている。

クラウドネイティブは、この変化に対する構造的な回答である。
新しい技術だから学ぶのではない。
従来の前提では成立しなくなったために、必要になった。

2. この教科書の構造

この教科書は、次の流れで構成されている。

- 序章で視点を渡し、
- 第 1 部で原理を理解し、
- 第 2 部で体感し、
- 第 3 部で構造として扱い、
- 第 4 部で運用として判断し、
- 第 5 部で社会と接続し、
- 第 6 部で総括としている

この順番には意味がある。
前の部で整理した前提の上に、次の部が成り立つ。

3. 構造で理解するとは何か

この教科書では、ツールや手順よりも先に、
「なぜその形になっているのか」を扱う。

Kubernetes を覚えることではなく、
なぜそれが必要になったのかを見る。

コマンドを覚えることではなく、
何が起きているのかを捉える。

構造で理解するとは、技術そのものではなく、
その技術が引き受けている責任を見ることである。

4. 読み方

すぐに分からなくても構わない。

この教科書は「理解した気になる説明」を意図的に避けている。
構造として捉えながら読み進めてほしい。

分からない箇所は、先に進んでから戻ると見え方が変わる。
前の部で扱った内容が、別の形で現れる。

この往復の中で、理解は積み重なっていく。

5. 再現性

この教科書で扱う構造は、特定の製品や環境に依存しない。

AWS、GCP、Azure、オンプレミス。
環境が変わっても、判断の軸は変わらない。

何を成立させるのか。
どこまで責任を持つのか。
どこで境界を引くのか。

この問いは変わらない。

クラウドネイティブとは何か

1. 問いの提示

「クラウドネイティブ」という言葉は、クラウドサービスやコンテナ技術の総称ではない。

クラウドとは、変化・拡張・自動化を前提とした環境構造である。
ネイティブとは、その環境に適応し、全体として成立し続けることである。

では、クラウドネイティブとは何か。

2. 背景：技術進化ではなく「前提の変化」として

モノリスは、固定された前提のもとで成立していた。
仮想化は、環境を分離したが、前提は変わらなかった。
コンテナは、単位を小さくしたが、全体は依然として人が管理していた。

そしてクラウドネイティブでは、前提そのものが変わった。

これは技術の進化ではなく、
システムが置かれる前提条件の変化である。

従来のシステムは、環境が大きく変わらないことを前提にしていた。
構成は固定され、変更や障害は例外として扱われる。

しかし現在では、サービスは継続的に変化し、
構成は分散し、環境は一定ではなくなった。

変化・分散・不確実性が前提になったことで、
従来の前提ではシステムは成立しなくなっている。

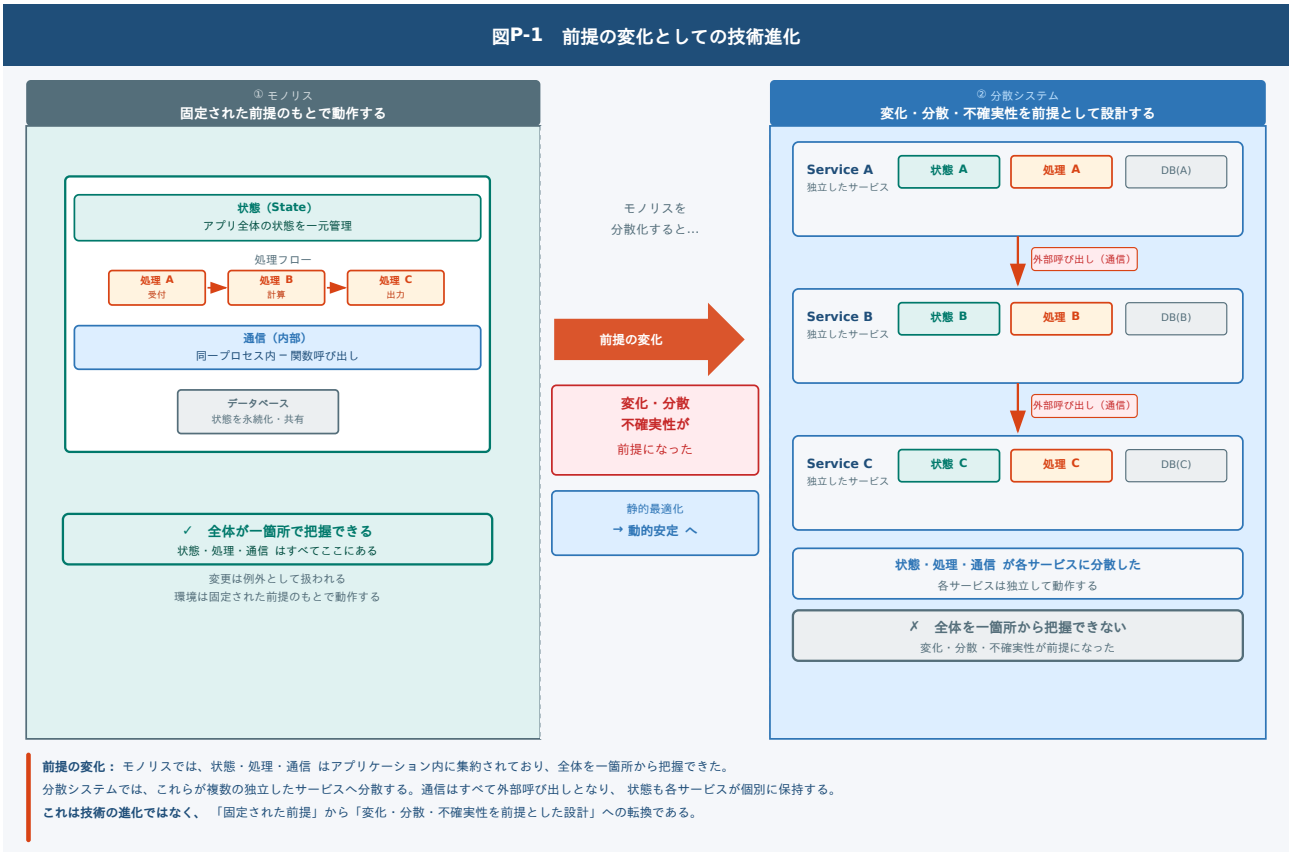


図 P-1 技術進化ではなく前提の変化

3. 本質：静的最適化から動的安定へ

従来の設計は、変化しないことを前提に最適化されていた。
構成は固定され、障害や変更は例外として扱われる。

しかしクラウドネイティブでは、前提が逆転する。

変化は常に起こるものとして扱われ、
構成は動き続けることを前提に設計される。

ここで一つ、前提を明確にしておく必要がある。

クラウドネイティブの設計は、壊れないことを前提としていない。
完全に制御できることも前提としていない。

一部が壊れる、状態がずれる、環境が変わる。
これらは異常ではなく、前提として扱われる。

その中で、それでも全体として成立し続けることが求められる。
止めないことを目指すのではなく、止まっても全体が成立するようにする。

運用で支えるのではなく、仕組みで吸収する。
変更を避けるのではなく、変更を前提として扱う。

この前提のもとでは、人が都度判断して対応する運用では追いつかない。

そのため、成立を維持するための判断は、
人ではなく仕組みが担うように設計される。

4. CNCF の定義

クラウドネイティブについては、CNCF による定義が広く参照されている。

そこでは、コンテナやマイクロサービス、宣言的 API、可観測性などが例として挙げられ、
疎結合・自動化・拡張性・回復性といった特性が重視されている。

ただし、これはひとつの整理に過ぎない。

本書では、これらを個別の技術としてではなく、
変化を前提とした構造として捉える。

これらは個別の技術ではない。

コンテナは実行単位を分離するための手段であり、
マイクロサービスは責任を分割するための構成であり、
宣言的 API は状態を扱うためのインターフェースであり、
可観測性は状態を把握し判断するための仕組みである。

これらはすべて、変化を前提とした環境の中で、
全体を成立させ続けるための要素として位置づけられる。

5. 学ぶことで見える世界

クラウドネイティブをこの視点で捉えると、
システムの見え方は大きく変わる。

個々の技術ではなく、
どこが変化し、どこが成立を支えているのかが見えるようになる。

「動かすこと」ではなく、
「成立し続けること」が設計の中心になる。

そして判断は、
機能ではなく構造に基づいて行われるようになる。

この変化は、単なるスキルの追加ではない。
システムに対する前提そのものが変わることを意味する。

例えば、サービスが遅いという問題が発生したとき、
どこに原因があるのかをすぐに特定できないことがある。

アプリケーションの問題なのか、
ネットワークの問題なのか、
それとも別のサービスとの連携に問題があるのか。

分散した構造の中では、
一つの原因が全体に影響する。

このとき重要なのは、
個々の技術の知識ではない。

どこで変化が起きているのか、
どこが成立を支えているのかを捉えることだ。

6. まとめ

ここまでで扱ってきたのは、
クラウドネイティブという言葉の説明ではない。

システムをどのような前提で捉えるか、という視点である。

この視点に立ったとき、システムの見え方や、設計の考え方は大きく変わる。

では、その構造はどのように実現されるのか。
この問いを持ったまま、次に進んでほしい。

第 1 部：基礎ステップ〜実行単位が変わったという話〜

導入

クラウドネイティブの技術は、すでに広く使われている。

コンテナ、Kubernetes、クラウド
これらを触ったことがある人も多いだろう。

しかし次のような状態に陥ることがある。

- 動かせるが、なぜそうなるのか分からない
- 構成を真似できるが、変更すると崩れる
- 問題が起きても、どこを見ればよいか分からない

ここで問題になっているのは、技術ではない。
全体を構造として捉えられていないことである。

この部では、クラウドネイティブを個別の技術としてではなく、
分離された構造として扱う。

扱うのは次の 4 つである。

- 実行単位（どこで区切るのか）
- 制御（どう成立させるのか）
- 実行環境（どこで動かすのか）
- 前提条件（何に依存しているのか）

これらはそれぞれ別の問題である。

重要なのは、すべてを理解することではない。
どこで何が起きているのかを切り分けられることである。

この視点を持った上で、次の章に進んでほしい。

第 1 章：コンテナの基礎と Docker 〜実行単位が変わったという話〜

1. なぜ「実行単位」を見直す必要があったのか

かつて、アプリケーションは特定の環境の上で動くものとして扱われていた。

ここでいう環境とは、OS や依存ライブラリ、ランタイムなど、
アプリケーションが動作するための前提条件の集まりを指す。

環境が増え、変更が増え、同じものを同じように
動かす必要が高まるにつれ、その前提は崩れ始める。

問題はアプリケーションそのものではない。
どの前提で動くのかが環境に依存していたためである。

例えば、開発環境では動いていたものが、
本番環境では動かないということが起きる。

- ライブラリのバージョンが違う
- 設定が違う
- 依存関係が揃っていない

このとき問題なのは、コードではない。

どの環境を前提にしているのかが一致していないことである。
ここで問われたのは、アプリケーションをどの単位で扱うのかという点である。

2. 仮想化（VM）が変えたこと

仮想化によって、環境は物理サーバーから切り離された。

環境は複製でき、移動でき、戻すことができるようになった。
環境は固定である必要がなくなった。

しかし、アプリケーションの扱い方は変わっていない。
実行単位は依然として環境に近いままだった。

環境は分離されたが、
アプリケーションは依然として環境に依存している。

この状態では、
「どこで動かすか」は解決できても、
「どう扱うか」は変わっていない。

そのため、扱う単位そのものを変える必要があった。

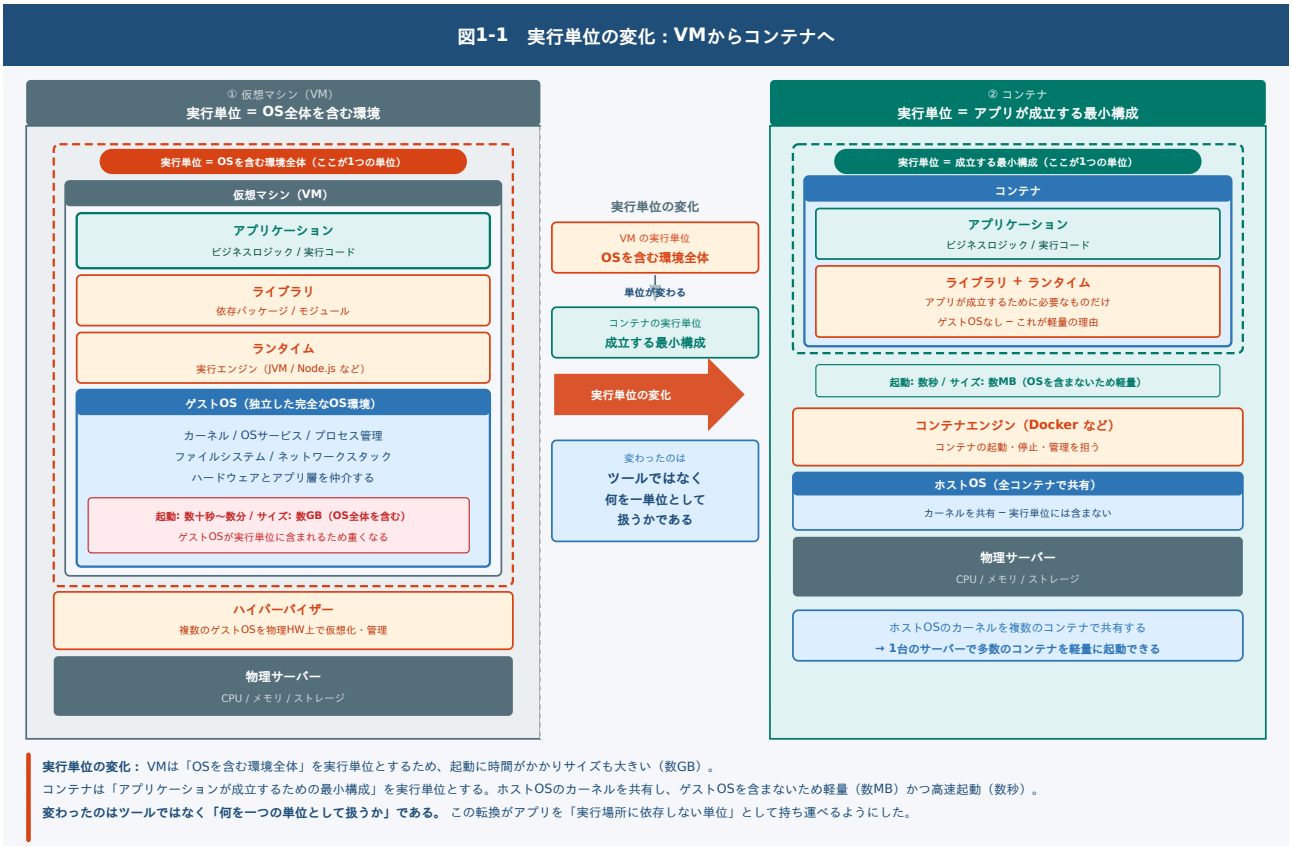


図 1-1 実行単位の変化：VM からコンテナへ

3. コンテナという実行単位

コンテナは、実行単位を環境から切り離す。

扱うのは OS 全体ではなく、アプリケーションが成立するための最小構成である。
この違いによって、アプリケーションは実行場所に依存しない単位として扱われる。

例えば、同じコンテナをそのまま別の環境に持っていけば、
同じように動くことが期待できる。

環境ごとに設定を調整するのではなく、
成立する状態を単位として持ち運ぶという発想になる。

必要になれば作られ、不要になれば破棄される。
実体を維持するのではなく、入れ替えることが前提になる。

4. 単位として扱うという発想

コンテナによって変わったのは、ツールではない。

アプリケーションは「環境の上で動くもの」ではなく、
成立条件を含んだ単位として扱われるようになる。

環境を維持するのではなく、成立する単位を入れ替え続ける。
この違いによって、安定の意味が変わる。

安定とは、状態を維持することではない。
置き換え続けた結果として成立する状態である。

5. Docker の位置づけ

コンテナという仕組み自体は、Linux の機能を活用して実現されている。
Docker は、その仕組みを誰でも扱えるように整えた代表的なツールだ。

統一された操作と再現可能な形式によって、コンテナという考え方を広く扱えるようにした。
統一された操作と再現可能な形式によって、コンテナは個人でも扱える単位になった。

Docker はクラウドネイティブそのものではない。

しかし、この単位を実際に扱えるようにしたことで、
「単位として扱う」という考え方が現実のものになった。

6. コンテナとマイクロサービスは別の話

コンテナが扱うのは、実行単位である。
マイクロサービスが扱うのは、分割と責任である。

単位をどう動かすかと、
単位をどう分けるかは異なる問題である。

この 2 つを混同すると、設計の意図が見えなくなる。

コンテナは「どう扱うか」、
マイクロサービスは「どう分けるか」。

扱う問題が違う。

7. 次に残る問い

実行単位は環境から切り離され、小さな単位として扱えるようになった。
その結果、単位は増え、分散し、変化と障害が前提になる。

では、この増え続ける単位を、どのように全体として成立させ続けるのか。

第 2 章：Kubernetes の全体像と主要コンポーネント～分散した実行単位を成立させ続けるという話～

1. コンテナが増えた世界で何が問題になったのか

コンテナによって、アプリケーションは小さな単位として扱えるようになった。
その結果、単位は増え、実行場所は分散し、障害は例外ではなく前提になる。

この状態では、一つ一つの単位を人が管理することは現実的ではない。
数が増えるほど、状態はばらつき、全体の把握は難しくなる。

この状態で問題になるのは、個々のアプリケーションの動作ではない。
増え続ける単位を、誰が、どのように、全体として成立させ続けるのか。

2. Kubernetes は何を引き受けるのか

Kubernetes は、この問題に対して用意された仕組みである。

扱うのは、個々のコンテナではない。
全体として成立している状態である。

人が個別に操作するのではなく、全体として成立している状態を保つことが前提になる。
どこで動いているかではなく、どういう状態であり続けるべきかを扱う。
その状態が保たれるように、配置・再起動・調整が行われ続ける。

3. 状態を扱い、差分を埋め続ける

Kubernetes は、現在の状態と、あるべき状態を分けて扱う。
「どうするか」ではなく、「どうなっていてほしいか」を定義する。

例えば、一つのコンテナが停止したとしても、
「動いているべき」という状態が定義されていれば、別の場所で起動される。

実際の状態は常に変化し続ける。
そのため、理想の状態との間には必ず差が生まれる。

Kubernetes はその差分を見続け、埋めるように動き続ける。
一度合わせるのではなく、ずれ続ける前提で戻し続ける。

4. 状態を扱う側と実行する側

Kubernetes はこの問題に対して、状態を管理する役割と
実際にアプリケーションを実行する役割を分けて設計している。

この仕組みを成立させるために、役割は大きく二つに分かれている。

状態を扱い、全体を成立させる側と、
実際にアプリケーションを動かす側である。

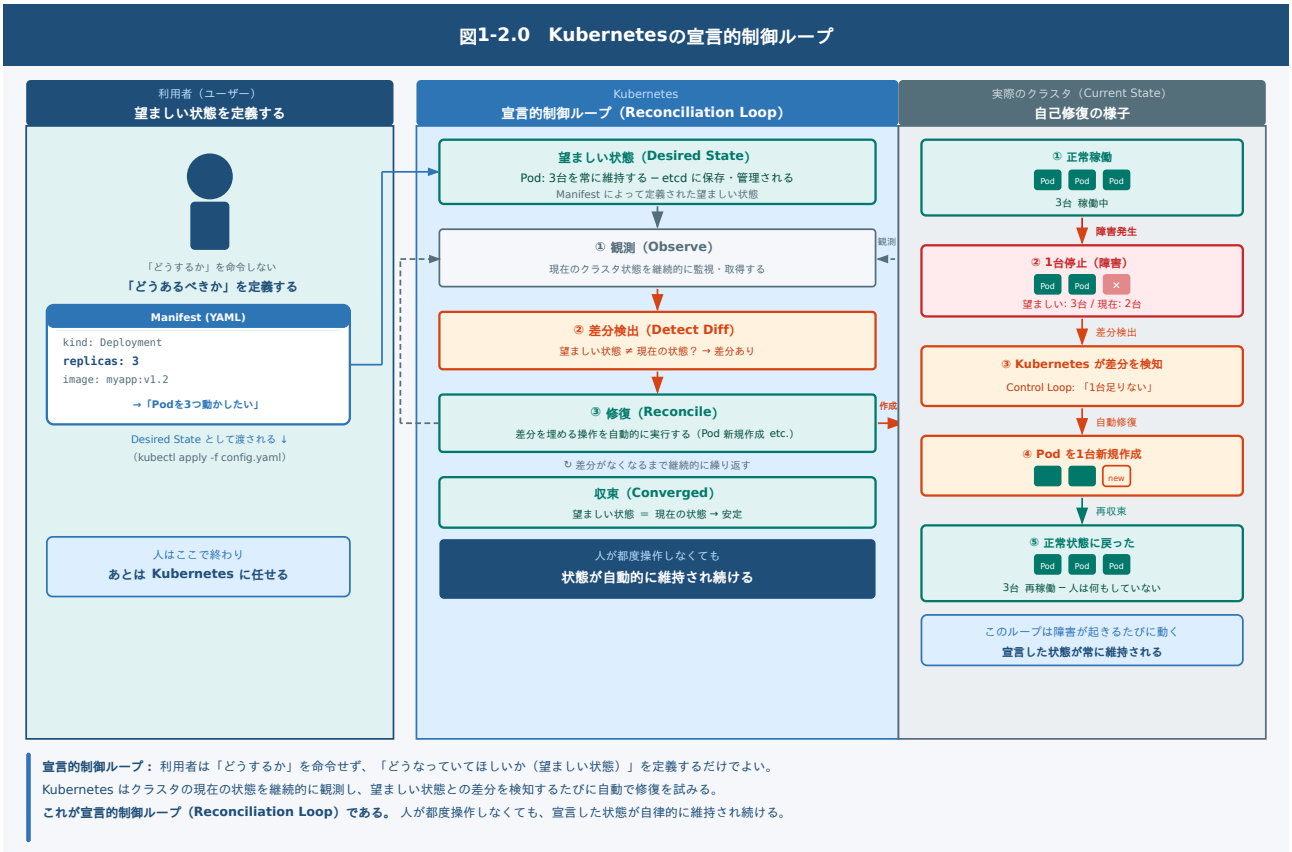


図 1-2.0 Kubernetes の宣言的制御ループ

前者は、あるべき状態を理解し、その状態が保たれるように判断し続ける。
後者は、その結果として指示された状態を受け取り、実際にコンテナを動かす。

この分離によって、一部が変化しても全体が崩れない構造が保たれる。

5. 責務として分割された構造

これは Kubernetes のアーキテクチャとしての設計判断である。

この構造の中では、それぞれの役割がさらに分割されている。

- 状態の入り口となるもの
- 実行場所を決めるもの
- ずれを検知して戻すもの
- 実際に状態を成立させるもの

それぞれが一つの責務だけを持ち、それ以上のことをしない。

この分離によって Kubernetes の一部に障害が発生しても、クラスタ全体は成立し続ける。

6. Kubernetes は OS ではない

Kubernetes は、OS ではない。

複数のマシンをまとめて扱うことができるため、
そのように見えることもある。

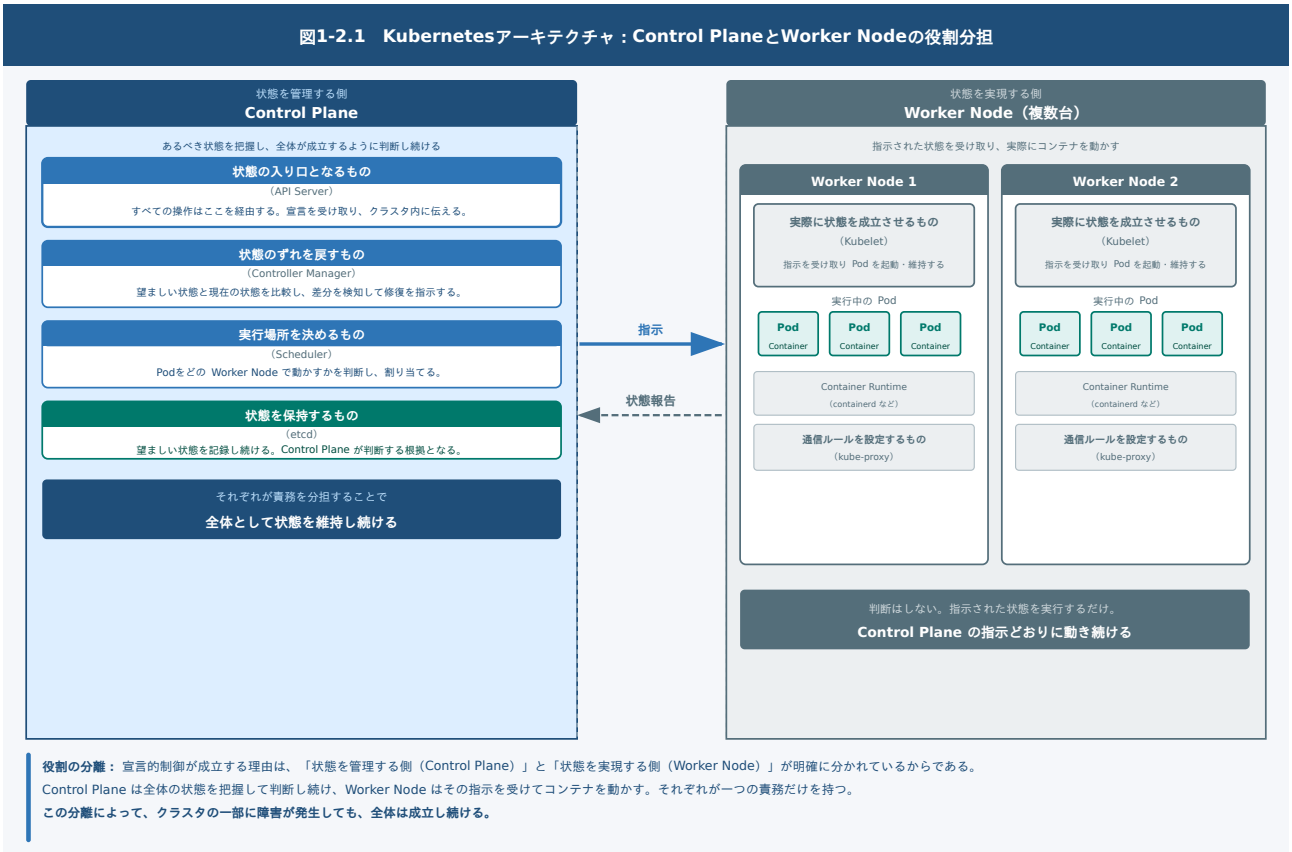


図 1-2.1 Control Plane と Worker Node の役割分担

しかし実際には、資源を直接管理するのではなく、
あるべき状態を保つための制御を担っている。

実行そのものは OS とその上の仕組みに委ねられている。

Kubernetes はそれらを置き換えるものではなく、
その上で状態を成立させ続けるための層である。

7. 次に残る問い

ここまでで、分散した単位を全体として成立させ続ける仕組みが見えてきた。

しかし、もう一つの前提が残っている。
その単位は、どこで動かすのか。

環境は固定されるのか、変化するのか。
どこまでを自分たちで制御し、どこを仕組みに任せるのか。

この前提によって、設計の意味は大きく変わる。

第 3 章：クラウドとオンプレの特徴～どこで動かすかを、構造として考える～

1. なぜ議論が噛み合わないのか

クラウドとオンプレミスの議論は、結論が出ないことが多い。

どちらが優れているかではなく、
前提が揃っていないまま比較されているためである。

同じシステムであっても、
変化を前提としているのか、
固定を前提としているのかによって、
評価の軸は大きく変わる。

前提が異なれば、最適な設計も、判断も変わる。

なお、ここで述べる特徴は一般的な傾向であり、クラウド上で固定的な構成を採用することも、
オンプレミス上で変化を前提とした設計を採用することも可能である。

2. クラウドという前提

なお、ここで扱うクラウドは実行基盤としての意味であり、
クラウドネイティブという設計思想とは区別して捉えてほしい。

クラウドは、特定のサービスや場所ではない。
変化・拡張・自動化を前提とした環境である。

環境は固定されず、
必要に応じて増減し、
障害は前提として扱われる。

例えば、負荷が増えればリソースは増え、不要になれば減る。

障害が発生しても、個々を修復するのではなく、
別の場所で再構成される。

環境を維持するのではなく、
変化を受け入れながら成立させ続ける。

3. オンプレミスという前提

オンプレミスは、クラウドの対極ではない。
自分たちで制御することを前提とした環境である。

どこで動くか、どの構成で動くか、その責任を自分たちで引き受ける。
構成は意図的に設計され、変更は管理されたものとして扱われる。

何を固定し何を変えるのかは、自分たちの判断に委ねられる。

4. 何が違うのか

違いは性能やコストではない。

- 何が変わる前提なのか
- 何を固定する前提なのか
- 誰が責任を引き受けるのか

この前提の置き方が異なる。

変化を前提とした場合、構成は動き続けるものとして扱われる。
状態は常に変化し、成立は維持され続ける必要がある。

一方で、固定を前提とした場合、構成は安定したものとして扱われる。
状態は維持され、変更は管理されたものとして扱われる。

図 1-3 クラウドとオンプレミスの前提の違い

どちらが正しいかではない。

前提が変わることで、成立のさせ方そのものが変わる。

5. 前提を選ぶということ

どこで動かすかを決めるということは、環境を選ぶことではない。
どの前提を採用するかを決めることである。

変化を受け入れるのか、
制御して固定するのか。

その選択によって、設計の意味は変わる。
何を前提にするかによって、扱う問題も変わる。

- 何を成立させるのか
- どこまでを仕組みに任せるのか
- どこを自分たちで引き受けるのか

その問いの置き方が変わる。

6. 次に残る問い

単位は切り出され、その成立は制御されるようになった。
そして、その前提となる環境も整理された。

では、その前提は、何によって支えられているのか。

第 4 章：Linux / ネットワークの基本整理～すべての前提になっている足場を言葉にする～

1. なぜ今、基盤に戻るのか

クラウドや Kubernetes は、その上で完結しているように見える。
しかし実際には、その動きはすべて下の層に依存している。

前提は消えたのではない。

見えなくなっているだけである。

抽象化によって扱いやすくなった分、
その前提を意識する機会は減っている。

2. Linux という前提

コンテナも Kubernetes も、その多くの仕組みは Linux の機能や考え方を土台としている。

プロセスは分離され、
資源（CPU・メモリ・ネットワークなどの計算資源）は分配され、
境界が保たれる。

この仕組みによって、一つのシステムの中で複数の処理を独立して扱うことができる。

コンテナは新しい仕組みではない。

既にあるこの前提を利用し、**実行単位として扱える形にしている。**
そのため、必要に応じて作られ不要になれば破棄される。

3. 資源は安定しているとは限らない

CPU やメモリは、常に同じように使えるとは限らない。

負荷が変われば振る舞いも変わり、
状況によっては処理は遅くなり、停止する。

これらは例外ではない。

有限な資源を扱っている以上、避けられない性質である。
抽象化された環境では、これらの制約は見えにくくなる。

しかし、見えなくなっているだけで、なくなっているわけではない。

4. ネットワークという前提

ネットワークは、常につながるとは限らない。

- 遅延する
- 順序が入れ替わる
- 突然切れる

例えば、通信が遅いとき原因がすぐに分からないことがある。

これらは例外ではない。

有限な環境の中で発生する、前提となる性質である。

5. 失敗は前提である

コンテナによって単位は分離され、Kubernetes によって制御されるようになった。
しかし、それらが動く基盤は常に変化し不確実である。

一部が正常で一部が失敗している状態は、特別なことではない。
この状態を排除することはできない。

必要なのは、**失敗した状態から戻れる構造である。**

6. 誰が前提を引き受けるのか

この前提はどこかが必ず引き受けている。

資源の制約やネットワークの不確実性は、そのまま扱うことはできない。

- アプリケーションが直接引き受けるのか
- Kubernetes のような制御の仕組みが肩代わりするのか
- クラウド環境が抽象化して見えなくするのか

どこかが必ず引き受けている。

どこに責任があるのかによって、
設計の意味は変わる。

7. 次に残る問い

ここまでで見てきたものは、新しい技術ではない。
すべては、既に存在していた前提の上に成り立っている。

では、その前提を前提として扱うために、
どのように状態を把握し判断するのか。

第 **1** 部 まとめ〜原理を理解し、構造として捉える〜

ここまでで扱ってきたのは、個別の技術ではない。
システムをどのように捉えるかという視点である。

最初に扱ったのは、単位である。

アプリケーションは、一つの塊としてではなく、
分離された単位として扱われるようになった。

しかし、単位が分かれるとそれだけでは成立しない。
その単位を、誰がどのように成立させ続けるのか。

ここで、制御が必要になる。
さらに、その制御はどこで動くのか。

変化を前提とするのか、
固定を前提とするのか。

前提によって、設計の意味は変わる。

そして、その前提の下には変えられない性質がある。

資源は有限であり、
ネットワークは不確実であり、
失敗は避けられない。

これらは、消えたのではない。
見えなくなっているだけである。

単位、制御、前提、性質。

この 4 つがつながったとき、初めて全体としての構造が見える。
重要なのは、個々の技術の詳細をすべて覚えることではない。

どこで何が起きているのかを、切り分けて捉えられることである。

そしてもう一つ重要なことがある。

その前提は、どこかが必ず引き受けている。

- アプリケーションか
- 制御の仕組みか
- それとも環境か

どこに責任があるのかによって、設計の意味は変わる。

ここまでで、第 1 部で扱ってきた構造はすべて揃った。

単位、制御、前提、性質。
そして、それを引き受ける責任。

この構造を前提として捉えられるとき、
個々の技術はばらばらの知識ではなく、
一つの流れとして見えるようになる。

では、この構造は実際にはどのように動くのか。

第 2 部：実践ステップ〜前提を体感する〜

導入

第 1 部では、クラウドネイティブを構造として捉えてきた。

単位は分離され、
制御によって成立し、
前提によって設計が変わり、
その下には変えられない性質がある。

しかし、この理解は、言葉だけでは完全には定着しない。

構造として「分かった」と思っていても、
実際に手を動かすと、思っていた通りに動かないことがある。

Pod を削除したら何かが起動した。
設定を変えたのに反映されない。
エラーが出たが、どこを見ればよいか分からない。

このとき問題なのは、操作ミスではない。

「何が起きているのか」を見る視点が、
まだ体感として持てていないことにある。

言葉で理解することと、動きとして捉えることは、別の回路である。

ここからは、その構造を実際の動きとして捉えていく。

目的は、操作を覚えることではない。

第 1 部で整理した前提が、
実際の挙動としてどう現れるのかを確認することだ。

ここからの進め方はシンプルである。

まず、どのような環境で Kubernetes を動かすことができるのかを知る。
その上で、この部で扱う環境の前提を確認する。

次に、基本的なリソースを使いながら、
実際にどのような挙動になるのかを観察する。

そして最後に、その理解を持ったまま、
実際の環境に戻ったときに何が変わるのかを整理する。

この視点を持った上で、次の章に進んでほしい。

第 1 章：実践環境の選択肢〜Kubernetes はどこで動かすのか〜

1. なぜ環境の話から始めるのか

Kubernetes を実践する環境は一つではない。

同じ操作をしても、環境が違えば結果は変わる。
手順通りにやったのに動かない。
書かれている通りなのに挙動が違う。

このとき問題なのは、Kubernetes ではない。
試している環境の前提が違うだけである。

環境の前提が異なると、何が自動で行われ、
何を自分で扱う必要があるのかが変わる。

その違いを理解しないまま進むと、「なぜ動かないのか」が分からなくなる。

2. 前提を揃えるということ

重要なのは、どの環境を使うかではない。

どんな前提の環境で試しているのかを 自覚しているかどうかである。
環境によって何が自動で用意されているのか、何を自分で引き受けるのかは大きく変わる。

例えば、ローカル環境では、
Minikube / Kind / k3s / Docker Desktop などを用いて、
セットアップや制御の一部を自分で引き受ける必要がある。

一方で、あらかじめ構成された環境として、
Killercoda のように、実行環境が用意されているものもある。

どちらが優れているかではない。

この部では、「挙動を観察すること」に集中するために、
環境構築や前提の差による影響をできるだけ排除する。

そのため、あらかじめ構成された環境を前提として扱う。

重要なのは、この選択が「便利だから」ではなく、
「観察に集中するため」であるという点である。

■ Kubernetes を動かす主な選択肢

※補足：この章で扱う実行環境について

Minikube / Kind / k3s / Docker Desktop は、
主に学習や検証を目的とした Kubernetes の実行環境である。

一方で、実際のサービス運用で使用される
パブリッククラウド（AWS / Azure / Google Cloud）などでは、
異なる前提の環境が使われることが多い。

ここで重要なのはどちらが優れているかではない。
前提が異なるという点である。

本章では、「本番ではどれを使うべきか」を決めることは目的としない。

Kubernetes を動かし、その挙動や前提を理解するための
学習環境としての選択肢を扱う。

ここでは、Kubernetes を試す際によく使われる代表的な環境を見ていく。
重要なのは、どれが一番良いかを決めることではない。

それぞれの選択肢が、
どのような目的で作られているのか、
どのような前提を持っているのかを捉えることである。

3. この部での扱い

この部では、環境差による影響を最小限にする。

実践に入る前に、環境の前提を意識することが重要になる。
その前提に立った上で、Kubernetes の挙動を観察していく。

目的は、操作を覚えることではない。
Kubernetes の挙動を観察することである。

同じ操作をしても、なぜその結果になるのか。
何が変わり、何が変わっていないのか。

その差を捉えることが重要になる。
すぐに理解できなくても構わない。

第 1 部で扱った前提が、どのように現れるのかを、
繰り返し確認してほしい。

第 2 章：実践環境の統一〜前提を揃えて観察する〜

1. なぜ環境を揃えるのか

Kubernetes の学習では、環境の違いによって挙動が変わる。

同じ操作でも結果が違うとき、問題は Kubernetes ではなく環境の前提にある。
そのまま進めると、何を見ているのか分からなくなる。

学習の本質と関係ないところで時間を消費する原因の多くは、
この前提のズレにある。

2. この部での扱い

この部では、環境差による影響を最小限にする。

目的は操作ではなく、挙動そのものを観察することである。
そのため、この部ではあらかじめ構成された環境を前提として扱う。

本書では、**Killercoda** の環境を使用する。

Killercoda は、ブラウザ上で Kubernetes や
クラウドネイティブ技術を学習できるハンズオン環境である。

Google アカウントなどを利用して簡単に開始でき、
ローカル PC へのインストールや環境構築を行わなくても、
あらかじめ用意された環境をすぐに利用できる。

本書では、環境構築を学ぶためではなく、
Kubernetes の挙動を観察するための土台として利用する。

クラウドネイティブの設計は、
壊れる前提や戻る前提の上に成り立っている。

主な特徴は次の通りである。

- ブラウザだけで利用できる
- 事前の環境構築が不要

- 初期状態が保証されている
- 失敗しても簡単にやり直せる

これにより、「何を学ぶか」以外の要素を
できるだけ排除した状態で実践に入ることができる。

3. 環境の位置づけ

ここで使う環境は、本番に近いものでも、高機能なものでもない。
学習の本質と関係ない差分を減らすためのものである。

壊れてもすぐに戻せること、同じ状態から何度でも試せること。
この性質によって、変化と挙動に集中できる。

ただし、実際の環境では前提が異なる。

- ネットワーク構成
- ストレージ
- リソース制限

これらは、環境によって大きく変わる。

ここで扱う内容は、それらをすべて再現するものではない。
あくまで、構造と挙動を理解するための土台である。

4. 実践に入る前の確認

ここで重要なのは、環境そのものではない。
どの前提で観察しているのかを自覚していることである。

この前提が揃っていない状態では、何を見ているのかが分からなくなる。

ここまでで、実践に入るための前提は揃った。
では、この環境の中で実際に何が起きるのか。

第 3 章：ハンズオン課題～Kubernetes の振る舞いを体感する～

1. この章の目的

この章では、実際に Kubernetes のリソースを扱いながら、
その挙動を観察する。

ここでの目的は、
操作を覚えることではない。

Kubernetes が何を自動で行い、
人が何をやらなくてよいのかを体感することである。

第 1 部で整理した前提と、
第 2 部で揃えた環境が、
実際にどのような動きとして現れるのかを見る。

2. この章での見方

ハンズオンでは、次の視点を意識する。

「自分が何をしたか」ではなく「何が勝手に起きたか」
「失敗しないこと」ではなく「壊れた後にどう戻るか」

Kubernetes は、人の操作を減らすための仕組みである。
この章では、あえて壊し、あえて戻ることで、その設計を確認する。

3. この章で登場する主なリソース

この章では、いくつかの Kubernetes リソースを扱う。
ここでは詳細を覚える必要はない。

まずは、それぞれがどのような役割を持っているのかだけを押さえてほしい。

リソース	役割
Pod	コンテナを実行する単位
Deployment	Pod の状態を維持する仕組み
Service	Pod への接続を安定して提供する仕組み
ConfigMap	設定情報を管理する仕組み
Secret	機密情報を管理する仕組み
Volume	データを保持する仕組み
Ingress / Gateway	外部との接続を扱う仕組み

これらは独立した機能ではない。

Kubernetes は、それぞれの役割を分離しながら、
組み合わせることでシステムを成立させている。

重要なのは名前を覚えることではない。

なぜ役割が分離されているのかを、実際の挙動を通して確認してほしい。

3-1. Pod / Deployment / Service ～実行単位と責務の分離～

■問い
単位を分けたとき、
それを誰が成立させるのか。

■操作（概要）

- Pod を作成する
- Pod を削除する
- Deployment を使う
- レプリカ数を変更する
- Service 経由でアクセスする

■観察

- Pod を削除したとき、何が起きたか
- 自分が指示していない処理は何か
- Deployment が何を肩代わりしているか

Pod は壊れる前提で扱われる。

Kubernetes は、壊れないようにするのではなく、
壊れたら戻す設計になっている。

3-2. ConfigMap / Secret / Volume　～設定と実行を分離する～

■問い
なぜ設定は、実行と分離されるのか。

■操作（概要）

- ConfigMap を作成する
- Pod から参照する
- Pod を削除する
- 再作成後の状態を確認する

■観察

- Pod が消えても何が残るか
- 設定はどこにあるか
- なぜ Pod に持たせないのか

Pod は使い捨てである。

Kubernetes では、長く動かすことではなく、
入れ替えられることが前提になる。

3-3. Ingress / Gateway ～境界はどこに置かれるのか～

■問い
外部との接続は、どこで扱うのか。

■観察

- Service と Ingress の違い
- Ingress がそのままでは動かない理由
- 実際に処理しているのは何か

Kubernetes は、通信の方法ではなく、
「どう扱いたいか」を定義させる仕組みである。

3-4. kubectl ～操作ではなく観察のために使う～

この章では、以下のコマンドを使う。

- get
- describe
- logs
- exec

重要なのは、本書では kubectl を操作のための道具としては使わないという点である。

- 状態を見る
- 変化を見る
- 差分を見る

そのために使う。

4. この章のまとめ

この章でやってきたことは、操作を覚えることではない。

Kubernetes が壊れる前提の中でどのように振る舞うのかを
実際の挙動として確認する。

- Pod は消える
- Deployment は戻す
- 設定は外に出る
- 責務は分離される
- 通信は分離される

これらはすべて、第 1 部で整理した前提の上に成り立っている。

kubectl は、その状態を見るための窓口である。

ここまでの体験は、Killercoda という環境の上で成立していた。
環境が変われば、引き受ける前提も変わる。

それを確認するために、次に進む。

3-1 ハンズオン：Pod / Deployment / Service ～壊して、戻って、任せる～

想定環境：Killercoda（Kubernetes クラスタ起動済み）
以降の操作はすべて kubectl を使用します。

0. 事前確認（現在の状態を見る）

まず、何も起きていない状態を確認します。

```
kubectl get pod
```

観察

- 何も作っていない状態では、何も存在しない
- **Kubernetes** は「何も勝手に起動しない」

何も定義していなければ、何も起きない

1. Pod を作成する（単体の実行単位）

Pod 定義ファイルを作成します。

```
cat <<EOF > pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  containers:
```

```
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
EOF
```

Pod を作成します。

```
kubectl apply -f pod.yaml
```

状態を確認します。

```
kubectl get pod
```

Pod の詳細を確認します。

```
kubectl describe pod sample-pod
```

観察

- Pod は「1 つの実行単位」
- 管理しているのは Kubernetes だが、守ってはいない

Pod は動くが、維持されない

※ 補足

Kubernetes は「動いているもの」を管理しているのではなく、
「どうあるべきか（状態）」をもとに動く仕組みである。

この Pod には、「存在し続けるべき」という定義がない。

そのため削除された場合でも、Kubernetes にとっては
「あるべき状態とのズレ」は発生していない。

守るべき状態が定義されていないものは、維持されない

2. Pod を削除する（壊してみる）

Pod を削除します。

```
kubectl delete pod sample-pod
```

再度状態を確認します。

```
kubectl get pod
```

観察

- Pod は消えたまま戻らない
- Kubernetes は「単体 Pod を守らない」

壊れたら終わる

問い

- これが本番だったらどうなるか？
- 毎回人が作り直すのか？

3. Deployment を使って Pod を管理する

Deployment 定義ファイルを作成します。

```
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sample
  template:
    metadata:
      labels:
        app: sample
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
EOF
```

Deployment を作成します。

```
kubectl apply -f deployment.yaml
```

Pod を確認します。

```
kubectl get pod
```

Pod を削除します。（もう一度壊す）

```
kubectl delete pod -l app=sample
```

すぐにもう一度確認します。

```
kubectl get pod
```

観察

- Pod が自動的に再作成される
- 自分は何も指示していない

状態が保たれる
守っているのは人ではなく仕組み

4. レプリカ数を変更（状態を変える）

Deployment のレプリカ数を変更します。

```
kubectl scale deployment sample-deployment --replicas=3
```

Pod を確認します。

```
kubectl get pod
```

観察

- Pod が 3 つ存在する
- 個体ではなく「数」だけを指定している

人は状態だけを指定する

5. Service を通じて **Pod** にアクセスする

Service を作成します。

```
kubectl expose deployment sample-deployment \
  --type=ClusterIP \
  --name=sample-service \
  --port=80
```

Service を確認します。

```
kubectl get service
```

Service 経由でアクセス確認します。

```
kubectl run curl --rm -it --image=curlimages/curl --restart=Never -- \
  curl http://sample-service
```

観察

- Pod を直接指定していない
- Service が間に入っている

実体ではなく入口を扱う

6. まとめ（このハンズオンで確認したこと）

このハンズオンで体験したのは、次の違いです。

Pod

- 実行単位
- 壊れる
- 守られない

Deployment

- 管理単位
- 状態を保つ
- 人の代わりに判断する

Service

- 接続点
- 実体を隠す
- 責務を分離する

Kubernetes は コンテナを動かす仕組みではない。

状態を保つ仕組みである。

ここまでで、壊す・戻る・任せるという流れを体験した。
では、その状態や設定はどこに置かれるのか。

3-2 ハンズオン：ConfigMap / Secret / Volume ～設定と実行を切り離す～

想定環境：Killercoda（Kubernetes クラスタ起動済み）
以降の操作はすべて kubectl を使用します。

0. 前提確認

前のハンズオンで作成した Deployment が存在していることを確認します。

```
kubectl get deployment
kubectl get pod
```

Pod が稼働していれば OK です。

1. ConfigMap を作成する（設定を **Pod** の外に置く）

ConfigMap を定義します。

```
cat <<EOF > configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  MESSAGE: "Hello from ConfigMap"
EOF
```

ConfigMap を作成します。

```
kubectl apply -f configmap.yaml
```

確認します。

```
kubectl get configmap
```

2. ConfigMap を **Pod** から参照する

ConfigMap を環境変数として読み込むようにします。

```
cat <<EOF > deployment-config.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sample
  template:
    metadata:
      labels:
        app: sample
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        env:
        - name: MESSAGE
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: MESSAGE
```

```
        ports:
        - containerPort: 80
EOF
```

適用します。

```
kubectl apply -f deployment-config.yaml
```

Pod を確認します。

```
kubectl get pod
```

環境変数を確認します。

```
kubectl exec -it $(kubectl get pod -l app=sample -o
    jsonpath='{.items[0].metadata.name}') -- env | grep MESSAGE
```

観察

- 設定は Pod の中に見える
- しかし定義は Pod の外にある

設定は **Pod** の外に定義される

3. Pod を削除する（設定は残るか？）

Pod を削除します。

```
kubectl delete pod -l app=sample
```

再度 **Pod** を確認します。

```
kubectl get pod
```

新しい **Pod** が起動したら、もう一度環境変数を確認します。

```
kubectl exec -it $(kubectl get pod -l app=sample -o
    jsonpath='{.items[0].metadata.name}') -- env | grep MESSAGE
```

観察

- Pod は入れ替わった
- 設定はそのまま使われている

Pod は設定を「所有していない」

4. ConfigMap を変更する（設定はどこに効くか）

ConfigMap を変更します。

```
kubectl edit configmap app-config
```

MESSAGE の内容を変更して保存します。

```
MESSAGE: "Hello updated ConfigMap"
```

Pod を再起動します。

```
kubectl delete pod -l app=sample
```

再度環境変数を確認します。

```
kubectl exec -it $(kubectl get pod -l app=sample -o
    jsonpath='{.items[0].metadata.name}') -- env | grep MESSAGE
```

観察

- 設定変更は Pod 再作成時に反映される
- 設定と実行が分離されている

設定を変更しても、**Pod** の定義は変わらない

5. Volume を使う（データの置き場所を考える）

Volume を使用する **Deployment** を作成します。

```
cat <<EOF > deployment-volume.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: volume-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: volume-sample
  template:
    metadata:
      labels:
        app: volume-sample
    spec:
      containers:
      - name: app
        image: busybox
        command: ["/bin/sh", "-c"]
```

```
        args: ["echo hello > /data/message.txt && sleep 3600"]
        volumeMounts:
        - name: data-volume
          mountPath: /data
      volumes:
      - name: data-volume
        emptyDir: {}
EOF
```

適用します。

```
kubectl apply -f deployment-volume.yaml
```

データを確認します。

```
kubectl exec -it $(kubectl get pod -l app=volume-sample -o
    jsonpath='{.items[0].metadata.name}') -- cat /data/message.txt
```

6. Pod を削除する（データは残るか？）

```
kubectl delete pod -l app=volume-sample
```

新しい **Pod** が起動したら、再度確認します。

```
kubectl exec -it $(kubectl get pod -l app=volume-sample -o
    jsonpath='{.items[0].metadata.name}') -- ls /data
```

観察

- データは消えている
- emptyDir は Pod と運命を共にする

データの寿命は、**Volume** の種類によって決まる

7. Secret について（ここでは体験しない）

Secret の概念は ConfigMap と同じである。
ただし用途が異なるため、ここでは操作は行わない。

重要なのは、「秘匿されているか」より「Pod の外にあるか」という点である。

8. まとめ（このハンズオンで確認したこと）

ConfigMap

- 設定を Pod の外に出す
- Pod が消えても設定は残る
- 設定変更は Pod の再作成で反映される

Secret

- 秘密情報を Pod の外に出す
- 扱い方は ConfigMap と同じ

Volume

- 状態の寿命を明確にする
- emptyDir は Pod と運命を共にする

このハンズオンで確認したのは、Pod は実行単位であり、
状態や設定の保管場所ではない。

では、外部との接続はどこで扱うのか。
Ingress / Gateway に進む。

3-3 ハンズオン：Gateway API とアクセス確認〜外から入る責務はどこにあるのか〜

想定環境：Killercoda（Kubernetes クラスタ起動済み）
以降の操作はすべて kubectl を使用します。

0. 前提確認

これまでのハンズオンで、

- Deployment（nginx）
- Service（ClusterIP）が存在している前提で進めます。

確認します。

```
kubectl get deployment
kubectl get service
```

sample-deployment と sample-service があれば OK です。

1. Service の役割を確認する（内部向けの接続点）

Service 経由でアクセスを確認します。

```
kubectl run curl --rm -it --image=curlimages/curl --restart=Never -- \
  curl http://sample-service
```

観察

- アクセス先は Pod ではない
- Service 名だけで通信できている
- Pod の数や入れ替わりは意識していない

Service は内部の接続点である

2. Gateway API の前提（CRD と Controller）を確認する

Gateway API も、リソース単体では動かない。
必ず、Gateway Controller（実装）が必要である。

CRD を確認します。

```
kubectl get crd | egrep
    "gateways.gateway.networking.k8s.io|httproutes.gateway.networking.k8s.io"
    || true
```

GatewayClass を確認します。

```
kubectl get gatewayclass
```

GatewayClass がない場合、Controller が未導入の可能性がある。
その場合は「3.〜5.」はスキップし、「7. 責務整理」を読めば OK です。

宣言と実行は分離されている

3. Gateway を作成する（外部からの入口を定義する）

GatewayClass を確認します。

```
kubectl get gatewayclass
```

利用可能なクラス名を確認し、を置き換えます。
（例：nginx / istio / envoy など）

Gateway を作成します。

```
cat <<EOF > gateway.yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: sample-gateway
spec:
  gatewayClassName: <GATEWAY_CLASS_NAME>
  listeners:
  - name: http
    protocol: HTTP
    port: 80
    hostname: sample.local
EOF
```

適用します。

```
kubectl apply -f gateway.yaml
```

Gateway を確認します。

```
kubectl get gateway
kubectl describe gateway sample-gateway
```

観察

- Gateway は入口の形を定義している
- どこへ流すかはまだ決まっていない

Gateway は定義であり、実行ではない

4. HTTPRoute を作成する（流れを定義する）

HTTPRoute リソースを定義します。

```
cat <<EOF > httproute.yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: sample-route
spec:
  parentRefs:
  - name: sample-gateway
  hostnames:
  - sample.local
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /
    backendRefs:
    - name: sample-service
      port: 80
EOF
```

適用します。

```
kubectl apply -f httproute.yaml
```

HTTPRoute を確認します。

```
kubectl get httproute
kubectl describe httproute sample-route
```

観察

- 流れは HTTPRoute 側で定義される
- Gateway と役割が分離されている

入口と振り分けは別である

5. Gateway 経由でアクセスする

Gateway に紐づく **Service** を探します。

```
kubectl get svc -A -l gateway.networking.k8s.io/gateway-name=sample-gateway ||
    true
```

ポートフォワードします。

```
kubectl -n <NAMESPACE> port-forward svc/<SERVICE_NAME> 8080:80
```

別ターミナルでアクセスします。

```
kubectl run curl --rm -it --image=curlimages/curl --restart=Never -- \
  curl -H "Host: sample.local" http://localhost:8080
```

nginx のレスポンスが返ってくれば成功です。

観察

- Service の裏にある Pod を意識していない
- Gateway → Route → Service の流れで処理される

※ Service が見つからない / port-forward できない場合、Gateway Controller が未導入の可能性があります。
その場合も「7. 責務整理」まで理解できれば、この節の目的は達成です。

6. Pod を削除する（構造は崩れるか）

Pod を削除します

```
kubectl delete pod -l app=sample
```

再度アクセスします。

```
kubectl run curl --rm -it --image=curlimages/curl --restart=Never -- \
  curl -H "Host: sample.local" http://localhost:8080
```

観察

- Pod は入れ替わる
- アクセス方法は変わらない

経路は維持される

7. 責務整理

ここで、各リソースの役割を整理する。

アプリ（**Pod**）

- 処理を行う
- 通信経路を知らない

Service

- 内部接続を担う

Gateway

- 外部との入口を定義する

HTTPRoute

- 流れを定義する

通信の責務は、アプリの外側にある。

8. 何を「やっていないか」が重要

このハンズオンでは、次を行っていない。

- アプリ側の設定変更
- IP アドレスの指定
- Pod 名の意識

それでも、外部からアクセスでき、Pod が変わっても動く状態が成立した。
何をしなくてよいかが、設計の意図を表している

9. まとめ：このハンズオンで確認したこと

Pod

- 処理に集中する

Service

- 内部接続を担う

Gateway

- 入口を定義する

HTTPRoute

- 流れを定義する

通信は分離される。
では、この構造のまま次に進む。

3-4 kubectl コマンドの基本整理～操作ではなく「状態を見る」ための道具～

この章のハンズオンでは、いくつかの kubectl コマンドを使う。
ここでは、すべてのオプションや使い方を覚える必要はない。

重要なのは、kubectl の役割を誤解しないことにある。

kubectl の役割は、リソースを直接操作して管理することではない。
kubectl には delete や scale など、状態を変更するコマンドも存在する。

しかし本質的には、

- いまクラスタがどういう状態にあるのか
- あるべき状態と比べて何が起きているのか

これらを確認し、理解するための道具である。

この章では、

- get
- describe
- logs

を中心に、状態を観察するという使い方を扱う。

1. kubectl とは何か

kubectl は、クラスタの状態を確認し、
状態を伝えるためのインターフェースである。

kubectl 自体が何かを判断したり、リソースを管理することはない。

- 判断するのは Kubernetes
- kubectl は「伝える」「見る」だけ

2. kubectl get

今の状態を一覧で見る

```
kubectl get pod
kubectl get deployment
kubectl get service
```

get は、「今、何が存在しているか」を一覧で確認するコマンドです。

ここで分かるのは、

- 存在しているかどうか
- 数
- 大まかな状態

詳細な理由は分かりません。
存在と状態を確認する

3. kubectl describe

詳細な状態を見る

```
kubectl describe pod sample-pod
```

describe は、「なぜ今この状態なのか」を確認するためのコマンドです。

- イベント
- エラー
- スケジューリングの結果

get がリソースの概要を一覧で確認するものだとすると、
describe は個々のリソースの詳細な状態を確認するための道具である。

経緯と理由を確認する

4. kubectl apply

あるべき状態を伝える

```
kubectl apply -f deployment.yaml
```

apply は、「この状態であってほしい」という宣言を Kubernetes に伝えるコマンドである。
ここで重要なのは、操作命令ではないという点である。

Kubernetes は、

- 現在の状態
- 宣言された状態

これらを比較し、その差分を埋めるように動く。
あるべき状態を宣言する

5. kubectl delete

状態を崩してみる

```
kubectl delete pod sample-pod
```

delete は、リソースを削除するコマンドです。
ただし Kubernetes では、削除 = 終わりとは限らない。

Deployment のような管理単位がある場合、

- 状態が崩れる
- 自動で戻される

という挙動になる。
状態を崩して挙動を見る

6. kubectl scale

数だけを変える

```
kubectl scale deployment sample-deployment --replicas=3
```

scale は、「いくつ存在してほしいか」だけを変更するコマンドである。

- どの Pod を増やすか
- どこに配置するか

これらは Kubernetes が判断します。

数だけを指定する

7. kubectl expose

接続点を作る

```
kubectl expose deployment sample-deployment --type=ClusterIP --port=80
```

expose は、リソースを Service 経由で公開するためのコマンドである。

ここでも、

- どの Pod に接続するか
- Pod が増減したときの処理

これらを人が考える必要はない。
Service がその責務を引き受ける。

接続点を定義する

8. kubectl logs / exec

中を「のぞく」ための道具

```
kubectl logs <pod名>
kubectl exec -it <pod名> -- /bin/sh
```

これらは、コンテナの中で何が起きているかを確認するためのコマンドである。

トラブル対応のために入り続けるためのものではない。
あくまで観察のための道具である。

中を観察する

9. まとめ

この時点で捉えておくべき対応関係は次の通りである。

コマンド	役割
get	今何が存在しているかを見る
describe	詳細な状態を見る

コマンド	役割
apply	あるべき状態を伝える
delete	状態を崩してみる
scale	数だけを変える
expose	接続点を作る
logs / exec	中を観察する

kubectl は Kubernetes を操作する道具ではない。

状態をやり取りするための窓口である。
この関係だけを捉えられていればよい。

では、次に進む。

第 4 章：ローカル環境へ展開する場合の注意点 ～自由が増えるということは、責任が増えるということ～

第 3 章までで、Killercoda を使った実践を通じて、

- Kubernetes の基本的な振る舞い
- 壊れる前提での設計
- 人が直接操作しない世界

これらを体感してきた。
ここから環境が変わる。

1. 学習環境とローカル環境の違い

Killercoda のような学習環境では、

- 初期状態が整っている
- 依存関係が揃っている
- 想定外の差分が少ない

という前提がある。

一方、ローカル環境では、

- OS
- コンテナ実行環境
- ネットワーク設定
- リソース制限

といった要素が、すべて環境依存になる。
これは、Kubernetes が難しくなったのではない。

前提を肩代わりしてくれる存在がいなくなっただけである。

2. 環境差が表に出るポイント

2-1. ネットワーク

ローカル環境では、

- ポートフォワードが必要になる
- 名前解決が期待通りに動かない
- 外部からアクセスできない

といった問題が起こる。

これは異常ではない。

ローカル環境では、ネットワーク境界や DNS 設定が
学習環境とは異なるために起こる。

2-2. リソース（CPU / メモリ）

ローカル環境では、

- メモリ不足
- CPU 競合
- プロセスの強制終了

これらが起こる。

コンテナが落ちる、Pod が再起動する。
これは Kubernetes の問題ではない。

多くの場合、**Linux** がリソースを守るためにプロセスを落としている結果である。
Kubernetes は、それを防ぐものではない。

それを前提として動く。

これらはローカル環境で起こりやすいが、リモート環境でも同様に発生しうる。
重要なのは、Kubernetes の問題ではなく基盤の性質として前提になっているという点だ。

2-3. ストレージ

ローカル環境では、

- Volume が期待通りに動かない
- データが残らない
- 権限で詰まる

といった問題が出る。

ここでも同じである。
Kubernetes がストレージを持っているわけではない。

実体は、

- OS
- ファイルシステム
- マウント

これらに依存している。

3. 自由と責任

ローカル環境では、

- 自由に設定できる
- 構成を自分で決められる

一方で、うまくいかなかった理由も自分で引き受ける必要がある。
これは難しさではない。

責任の所在が変わっただけである。

4. 第 1 部の前提との接続

ここまでの差分は、すべて第 1 部とつながっている。

- 環境は変わる
- ネットワークは切れる
- リソースは枯渇する
- 人はすべてを制御できない

ローカル環境でこれらが表に出るのは、
学習環境が肩代わりしていた前提が外れるからである。

Kubernetes は、それらを隠す仕組みではない。
それらを前提として、人の負担を減らす仕組みである。

5. まとめ

環境が変われば、責任の所在も変わる。
どの前提を、誰が引き受けているのか。

それを自覚しているかどうかが、設計を決める。

次は、この前提を持ったまま、設計に入る。

第 2 部 まとめ〜体感したものを、設計につなげる〜

この部で行ったのは、操作ではない。

挙動を観察することで、
前提がどのように現れるかを確認した。

単位は分離され、数は増える。

その結果、状態は固定されず、
変化し続けるものとして扱われる。

このとき、壊れることは例外ではなく、
戻ることが前提になる。

人が一つ一つを操作して成立させるのではなく、
あるべき状態を定義し、その成立は仕組みに任せる。

ここで重要になるのは、
何を任せ、何を自分で引き受けるのかという境界である。

どこまでを仕組みに委ね、どこに責任を残すのか。
その境界の置き方が、設計そのものになる。

では、この前提の上で、どこで責務を分けるのか。

第3部：応用ステップ〜アプリケーションを届け、つなぐ〜

導入

マイクロサービス環境では、
システムの中に流れるものが変わる。

一つは、**変更**である。
アプリケーションは更新され、継続的に届けられる。

もう一つは、**通信**である。
分割されたサービス同士が、つながり、やり取りを行う。

単位が分かれたことで、システムの中には新しい流れが生まれる。

ここで問題になるのは、技術ではない。
この流れを、どこで引き受けるのか。

まず、アプリケーションをどの単位で分けるのかを整理する。

その上で、分割されたアプリケーションに対して、変更をどのように届けるのか。
さらに、分割されたサービス同士の通信を、どこで制御するのか。

- **分割**
- **変更**
- **通信**

この三つを、個別の技術としてではなく、
一つの構造として捉える。

重要なのは、どのツールを使うかではない。
その複雑さを、どこに置く設計なのかである。

この視点を持ったまま、次に進む。

第1章：マイクロサービスとアプリケーション構成〜分けることで、何が起きるのか〜

0. 前提の置き方

この章では、特定の構成や正解は扱わない。

アプリケーションの分け方は、置かれている前提によって変わるためである。
ここで扱うのは、どの形が正しいかではない。

- **なぜ分割という発想が生まれるのか**
- **分けたときに何が起きるのか**

その構造である。

1. なぜ分割が現れるのか

ここで扱うのは、アプリケーションである。

利用者に機能を提供するための処理は、一つのまとまりとして構成される。
しかし、変化が増えると状況は変わる。

- 変更の回数が増える

- 同時に扱う機能が増える
- 処理のやり取りが増える

システムの中を流れるものが増えていく。

- **変更という流れ**
- **通信という流れ**

この状態では、一つの変更が全体に影響し、
変更のたびに全体を扱う必要がある。

どこか一部を直すだけでも、全体を止める必要があるかもしれない。
このとき、「一つのまとまりのままで扱い続けること」が現実的ではなくなる。

ここで生まれるのは、「どこで区切るのか」という問いである。

2. 分けると何が起きるのか

アプリケーションを分割すると、
それぞれを独立した単位として扱えるようになる。

- 一部だけを変更できる
- 一部だけを動かすことができる

しかし同時に、それぞれは単体では成立しなくなる。

一つの中にあった処理が分かれると、
分かれたもの同士をつなぐ必要が生まれる。

ある処理は別の処理を呼び出し、ある状態は別の場所に置かれ、
結果を返すまでに複数の単位をまたぐようになる。

つまり、分けた瞬間に**通信、状態、依存関係**が外に現れる。

分けることで扱いやすくなる部分はある。

しかしその一方で、分けたもの同士を
どう成立させるかという新しい問題が生まれる。

3. 複雑さは移動する

分割は、複雑さを減らす行為ではない。
複雑さの置き場所を変える行為である。

一つの中にあったものは、分割されることで外に現れる。

- 通信
- 変更
- 依存関係

これらは、分割された単位の外で扱われるようになる。

その結果として、これまで意識する必要がなかったものが、
扱う対象として現れる。

- 通信は遅延し、失敗する
- 状態はズレ、不整合が起きる

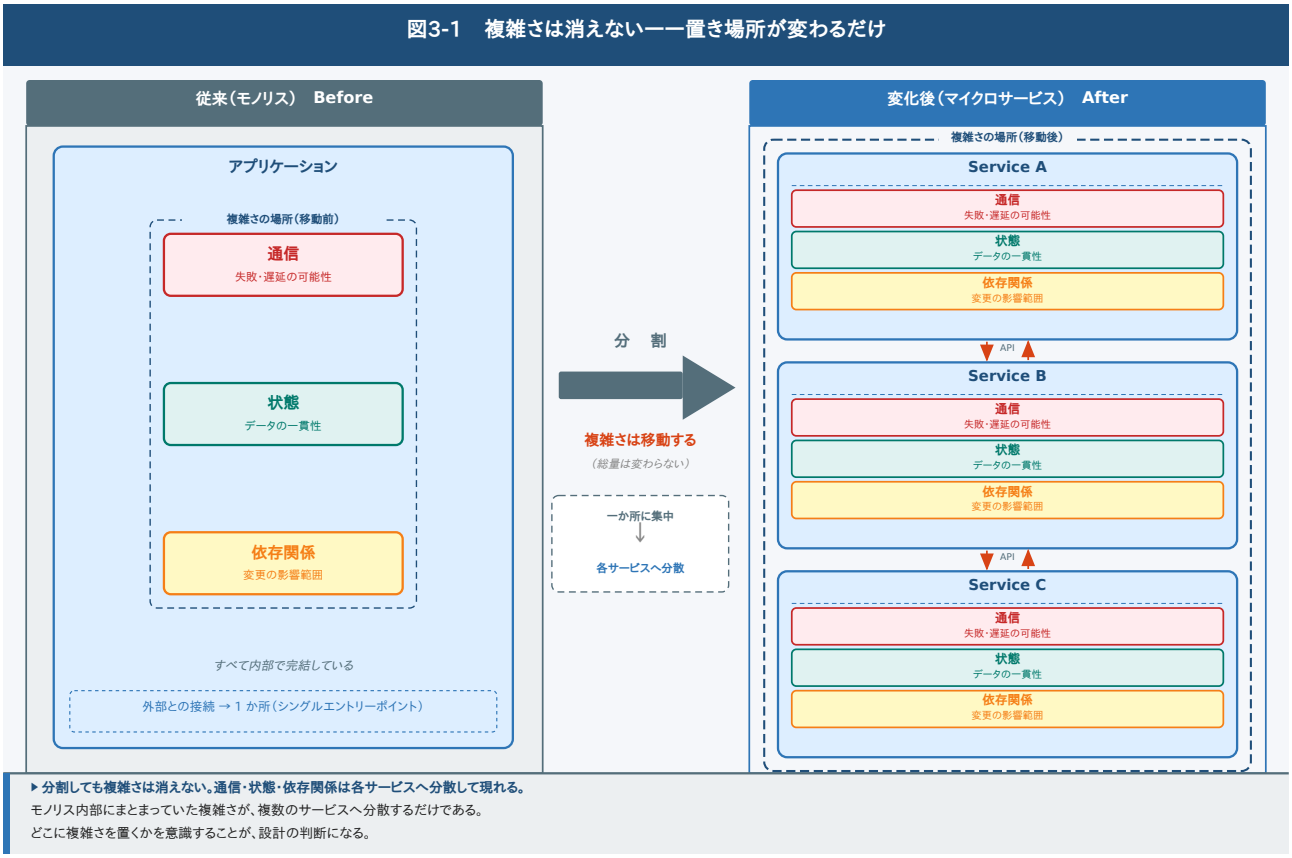


図 3-1 複雑さは消えない―置き場所が変わるだけ

- 依存関係は、影響範囲を広げる

これらは例外ではない。
分割によって前提が表に出る。

4. 責務としての分割

アプリケーションは、いくつかの責務の集合として捉えられる。

- 利用者とのやり取り
- 処理
- データの保持

この分け方そのものが重要なのではない。

どこまでを一つとして扱うのか、どこで責任を分けるのか

この切り分けによって、扱い方は変わる。
一つとして扱えば、変更はまとめて行われる。
分けて扱えば、一部だけを独立して扱うことができる。

しかしその一方で、分けたもの同士をつなぐ必要が生まれる。

切り方によって、扱う問題は変わる。
この判断が、そのまま設計になる。

5. 分割と実行環境

分割された単位は、そのままでは成立しない。
それぞれが別の場所で動き、それぞれが独立して変化する。

あるものは増え、
あるものは減り、
一部は失敗し、
別の場所で動き直す。

この状態を維持するためには、

- どこで動かすのか
- いつ動かすのか
- どのようにつなぐのか

といった判断が常に発生する。

これをすべて人が扱うとすると、変化のたびに状態を確認し、
必要な操作を都度行う必要がある。

この状態は、現実的ではない

そのため、これらを管理し続ける仕組みが必要になる。
ここで初めて、Kubernetes のような仕組みが意味を持つ。

分割によって、責任の置き場所が現れる

6. 選択であって正解ではない

ここで初めて、マイクロサービスという言葉が意味を持つ。

マイクロサービスは、目指すべき形ではない。
設計上の選択肢である。

どの前提を置くかによって、最適な形は変わる。
重要なのは、分割することではない。

分割した結果として、

- どの責務を引き受けるのか
- どの変化を許容するのか
- どこまでを制御するのか

その理解である。

7. 次に残る問い

分割によって、変更と通信という流れが生まれた。

変更は、一度届けて終わりではない。
繰り返し、安全に、継続的に届ける必要がある。

では、その変更はどのように届けられるのか。

第 2 章：継続的デリバリーと GitOps ～変更を届け続けるということ～

1.「一度デプロイすれば終わり」ではなくなった理由

従来のシステムでは、デプロイはイベントだった。
リリース日を決め、手順を確認し、反映する。

このやり方は、変更頻度が低く、影響範囲も限定的な環境では成立していた。
しかし、アプリケーションが分割され、変更の単位が小さくなると、この前提は崩れる。

小さな変更が頻繁に発生し、「誰が、いつ、何を変えたか」を追い続ける必要が生まれる。
変更が増えるということは、その分だけ、状態がズレる可能性が増えるということである。

- 意図しない変更が混ざる
- 反映されている状態が分からなくなる
- 元に戻せなくなる

変更は、常に正しく反映されるとは限らない。
変更もまた、不確実な流れである

2. 継続的デリバリーは「自動化」ではない

継続的デリバリーはしばしば「自動デプロイ」と同一視される。
しかし本質はそこではない。

問題は、変更が発生するたびに人が判断し、手順を確認し、
状態を合わせ続ける必要があるという点にある。

この状態では、変更の頻度が上がるほど運用は不安定になる。

人の操作は揺らぎ、
手順は微妙に変わり、
結果は再現できなくなる。

人が頑張るほど、不安定になる

継続的デリバリーが目指しているのは、

- 変更を、いつでも同じ手順で反映できること
- 状態を再現できること
- 問題が起きたときに戻せること

つまり、**人が毎回判断しなくても成立する状態を作ること**

3. なぜ「人が直接デプロイしない」構造になるのか

クラウドネイティブな環境では、
デプロイを人の操作に依存させない構造が一般的になる。

これは人が信用できないからではない。

変更が増えた環境では、

- 操作の記録が残らない
- 手順が揃わない
- 状態が再現できない

これらの問題が避けられないからである。

重要なのは「誰がやったか」ではなく「何が行われたか」

そのため、デプロイという行為そのものを人の手から切り離す設計が選ばれる。

4. GitOps という考え方

この課題に対する一つの整理が GitOps である。

GitOps では、環境のあるべき状態を Git に定義し、
実際の環境をその状態に近づけ続ける構造を取る。

ここで重要なのはツールではない。
変更がどのように扱われるかである。

- 変更はすべて Git に記録される
- 差分として比較できる
- 過去の状態に戻すことができる

変更という流れを追える状態になる

Git が使われるのは、変更を管理する構造として成立しているからである

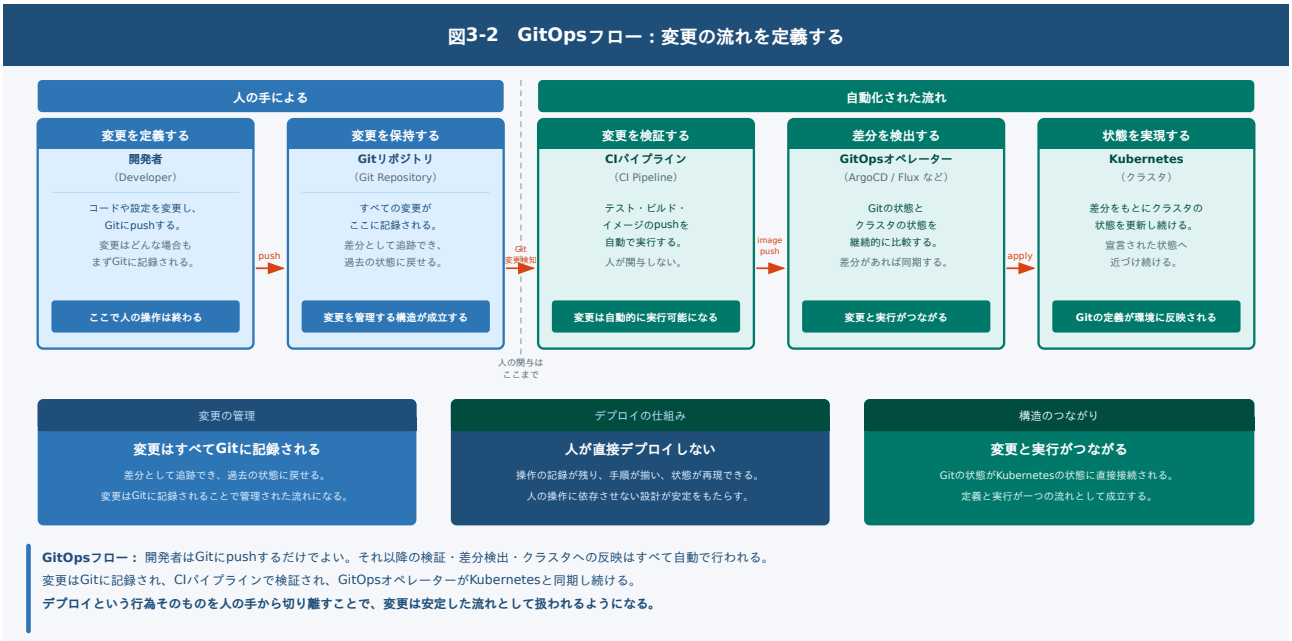


図 3-2 GitOps フロー：変更の流れを定義する

5. Kubernetes と GitOps の関係

Kubernetes は、あるべき状態を宣言的に定義できる基盤である。
望ましい状態を記述し、現在の状態との差分を埋め続ける。

この性質は、GitOps と一致する。

Git に定義された状態を、Kubernetes が実際の状態に近づけ続ける。
変更と実行がつながる

この構造は、実際にはツールとしても実装されている。

ArgoCD や Flux は、Git の状態とクラスタの状態を比較し、
差分を同期し続けることで、この構造を成立させている。

重要なのはツールの違いではない。
この構造が成立していること

6. 継続的デリバリーの意味

ここまで見てきたように、変更が増えた環境では、
人が都度判断し続けることは現実的ではない。

そのため、状態を定義し、変更の流れを固定して
反映の仕組みを持つことで、**判断を仕組みに移す**。

7. 次に残る問い

ここまでで扱ってきたのは、変更という流れである。

アプリケーションは更新され、
状態は書き換えられ、環境に反映される。

しかし、分割されたシステムには、もう一つの流れがある。
サービス同士が呼び出し合い、処理をやり取りする。

通信という流れである。

変更という流れは整理された。
では、この通信はどのように扱われるのか。

第 3 章：サービス間通信制御〜複雑さを、どこで引き受けるか〜

1. 分割された瞬間に、通信は「設計課題」になる

単一のアプリケーションの中では、通信はほとんど意識されない。
しかしアプリケーションを分割すると、この流れはネットワーク越しの通信に変わる。

ネットワーク通信には必ず次の前提がある。

- 到達しない可能性
- 遅延が発生する可能性
- 相手が応答しない可能性

通信は失敗する前提を持つ

この瞬間から、通信は実装の詳細ではなく設計として扱うべき対象になる。
通信の振る舞いを各サービスに任せた場合、挙動はばらつき、原因の特定が困難になる。

通信は、放置すると制御できなくなる

通信は、成功することもあれば、失敗することもあり、遅延することもある。
通信の振る舞いとは、成功・失敗・遅延といった状況に対して、どのように動くかというルールである。

2. 通信制御は「どこかで必ず行われている」

サービス間通信を「制御しない」という選択肢は存在しない。

タイムアウト、リトライ、エラーハンドリングといった判断は、
意識していなくても必ずどこかで行われている。

問題は、それをどこで引き受けているかだ。

各サービスのコードなのか、共通ライブラリなのか、基盤側なのか。
通信制御はすでに存在している。ただ、見えていないだけだ。

3. 複雑さは「消える」のではなく「移動する」

通信制御をアプリケーション側で引き受けると、振る舞いは分かりやすくなる。
しかし同じような実装が増え、振る舞いの統一が難しくなる。

通信制御を基盤側に寄せると、コードはシンプルになる。
しかし、設定の理解、振る舞いの可視化、障害時の切り分けといった別の形で複雑さが現れる。

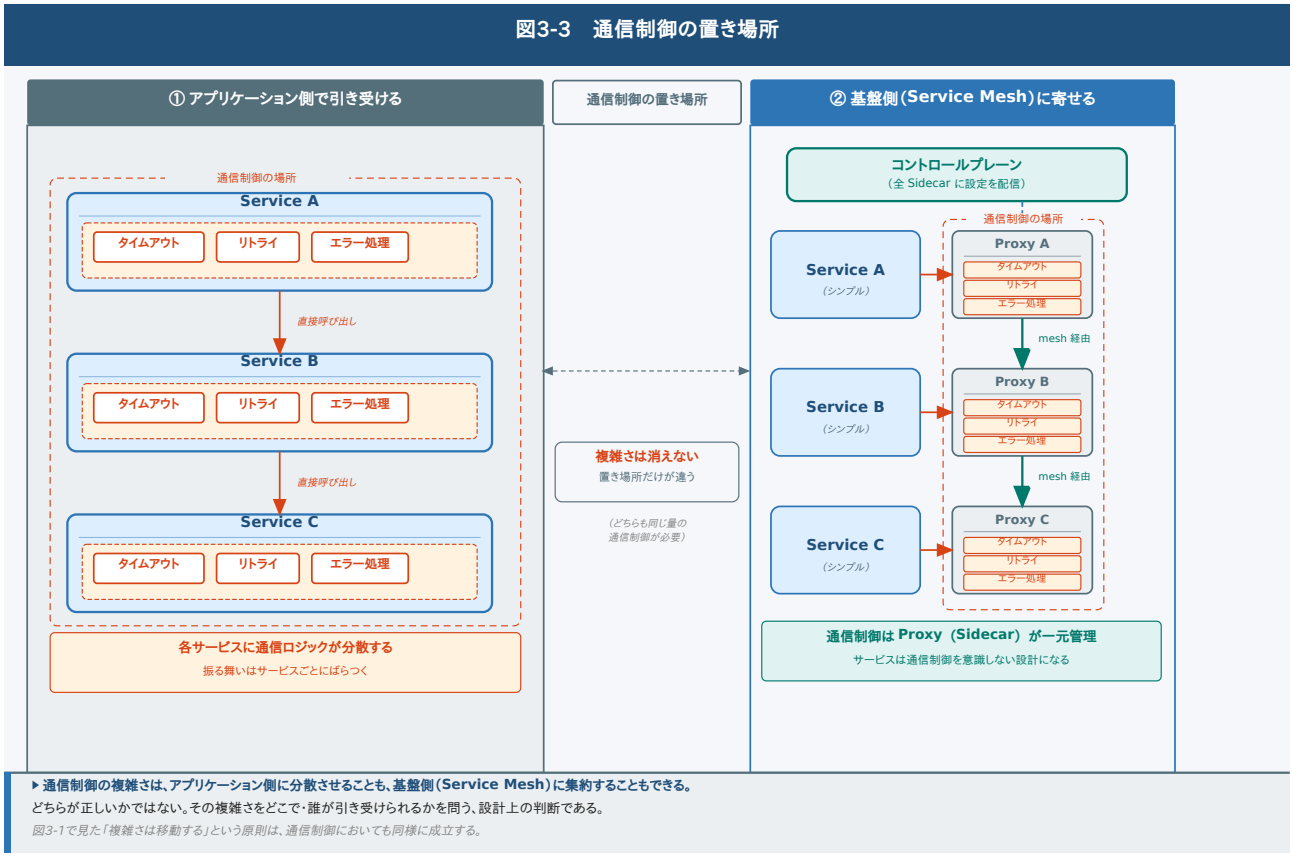


図 3-3 通信制御の置き場所

通信制御の設計とは、複雑さをなくすことではなく、どこに置くかを決めることだ。
この文脈で、**サービスメッシュ**という考え方が現れる。

4.「導入するか」ではなく「引き受けられるか」

この章の問いは「サービスメッシュを導入するかどうか」ではない。
本質的な問いは次の点にある。

- この通信制御の複雑さを誰が理解するのか
- 障害時にどこまで追えるのか
- 組織として引き受けきれるのか

技術的に可能であることと、
組織として引き受けられることは
必ずしも一致しない。

理解できないものは運用できない

この構造は、実際にはツールとしても実装されている。

Istio や Linkerd は、通信制御を基盤側に寄せるための実装である。
重要なのは機能の違いではない。

通信制御が構造として切り出されていること

5. 次に残る問い

ここまでで見てきたのは、通信という流れである。

サービスは分割され、
互いに呼び出し合い、
ネットワーク越しにやり取りを行う。

しかし、ここで扱っているのは
あくまで「どこで制御するか」という構造である。

**通信の振る舞いとは、成功・失敗・遅延といった状況に対して、
どのように動くかというルールである。**

どのように定義され、
どのように制御され、
どのように観測されるのか。

次は、この通信の振る舞いそのものを、
どのように扱うのかを見ていく。

第 4 章：通信の振る舞いをどう扱うか〜リトライ・タイムアウト・可観測性〜

1. 通信は「結果として現れる」

第 3 章では、通信をどこで制御するかを見てきた。
しかし実際に現れるのは、設定そのものではなくその結果である。

通信は、

- 成功する
- 失敗する
- 遅延する

といった形で現れる。

通信は結果としてしか観測できない

2. 振る舞いを決めていない通信は不安定になる

通信は失敗する前提を持つ。

しかし、失敗したときにどうするかを決めていなければ、挙動は一定にならない。

- あるサービスはリトライする
- あるサービスは即座に失敗する
- あるサービスは長時間待ち続ける

同じ失敗でも結果が変わる。
これは、通信が制御されていない状態である。

振る舞いを決めていない通信は不安定になる

3. 振る舞いは「ルール」として定義される

通信の振る舞いは、ルールとして定義される。

- どこまで待つのか
- 何回試すのか
- どの時点で諦めるのか

これらはすべて、失敗したときにどう動くかを決めている。
ここで重要なのは、成功時ではなく、失敗時の動きが設計を決める。

4. リトライとタイムアウト

通信の振る舞いの中で、最も基本となるのが次の 2 つである。

- リトライ
- タイムアウト

リトライは、失敗した通信を再試行する。
タイムアウトは、どこまで待つかを決める。

これらがなければ、

- 失敗は即座に広がる
- 待ち続けて処理が詰まる

システム全体が不安定になる

5. 見えないものは制御できない

通信は外に出た瞬間、見えなくなる。

- どこで遅いのか
- どこで失敗しているのか
- どの経路を通っているのか

これが分からなければ、振る舞いを調整することはできない。
見えないものは制御できない

図 3-4 遅延が上流へ伝播する

6. 可観測性という考え方

通信を扱うためには、状態を知り、流れを追って原因を特定する必要がある。

このための考え方が、可観測性である。
可観測性はツールではない。

振る舞いを理解するための前提である

7. 振る舞いはどこで引き受けるのか

これらの振る舞いは、

- アプリケーションで持つこともできる
- 基盤側で持つこともできる

どちらが正しいかではない。
どこで引き受けるかの設計である

8. まとめ

この章で見てきたのは、通信の振る舞いである。

通信は、

- 成功する
- 失敗する
- 遅延する

これらの結果として現れる。

重要なのはその結果に対して、
どう動くかを決めているかという点である。

ここまでで、

- 分割
- 変更
- 通信
- 振る舞い

という構造が出揃った。

では、この前提の上で、どのように全体を成立させるのか。
第 3 部のまとめで整理する。

第 3 部まとめ〜応用ステップで扱ってきた「判断」とは何だったのか〜

第 3 部では、クラウドネイティブなアプリケーションを構成する技術を、
手法や正解としてではなく判断の積み重ねとして見てきた。

マイクロサービス環境では、アプリケーションを分割することで
システムの中に二つの物流が生まれる。

変更という物流と、
通信という物流だ。

- どう分けるか
- どう届けるか
- どうつなぐか
- どう振る舞うか

いずれの章でも、特定の構成や技術を「選ぶべきもの」としては扱っていない。
それは、**何を選ぶかよりも、何を引き受けるかを自覚しているか**が重要だからだ。

技術は「答え」ではなく「責務の引き受け先」である

第 3 部で登場した技術や考え方は、いずれも問題を消し去るものではない。

- マイクロサービスは、複雑さを減らすものではなく**配置を変える選択**だ。
- GitOps は、判断をなくすものではなく**判断を構造として固定する考え方**だ。
- サービス間通信制御は、通信の難しさを避けるのではなく**どこで向き合うかを決める設計**だ。
- 通信の振る舞いは、その結果に対してどのように動くかという**ルールとしての設計**だ。

技術は判断を不要にするものではない。

判断をどこで引き受けるのかを構造として表現する手段だ。

それでも残る問い

分け、届け、つなぐ。

ここまで整えても、システムが自動的に安全になるわけではない。
判断を構造として外に出したことで、次の問いがより明確になる。

いま、システムの中で何が起きているのか。
その判断は、正しく機能しているのか。

では、それはどのように観測できるのか。

第4部：運用ステップ〜見える化して、守る〜

第3部では、アプリケーションを分け、届け、つなぐ構造を整理した。

しかし、構造として判断を外に出しただけでは、
システムが正しく機能しているかどうかは分からない。

分散したシステムでは、内部の状態は直接は見えない。
見えるのは、結果として現れる挙動だけである。

- レスポンスが遅い
- エラーが返る
- 処理が止まる

こうした現象は観測できるが、その原因がどこにあるのかはそのままでは分からない。

- アプリケーションの問題なのか
- ネットワークの問題なのか
- データベースの問題なのか

判断できなければ、対応は遅れる。
原因が分からなければ、同じ障害は繰り返される。

この状態では、安定した運用は成立しない。

構造は整っていても、状態が分からなければ全体は維持できない。
ここで必要になるのは、内部を直接見ることではない。

観測できる情報から、状態を判断するという視点である。

いま、システムの中で何が起きているのか。
そして、その判断はどこから来るのか。

第1章：モニタリングと可観測性〜見えるだけでは、判断できない〜

1. 分散した瞬間に、「全体が見えなくなる」

単一のアプリケーションでは、

- 状態は一箇所に集まっている
- ログを見れば分かる
- メトリクスを見れば把握できる

しかしこれは、すべてが一つのまとまりとして動いているから成立している。

アプリケーションを分割すると、

- 柔軟に変更できるようになる
- 一部だけを更新できる
- 独立してスケールできる

その一方で、状態は分散する。

複数のサービスが動き、それぞれが独立して状態を持つ。
ある処理は複数のサービスをまたぎ、結果はネットワーク越しに返される。

この瞬間から、システムの状態は一箇所では把握できなくなる。

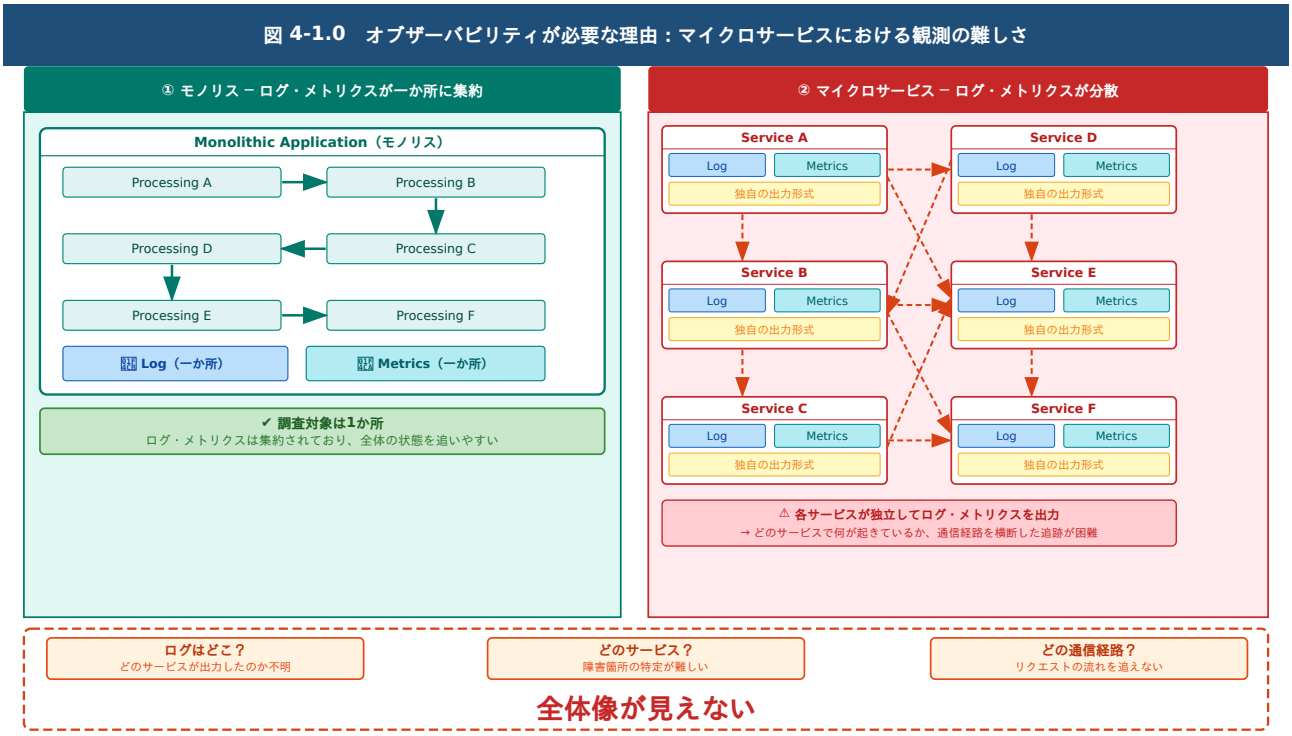


図 4-1.0 分散した瞬間に全体が見えなくなる

2.「見えているのに分からない」状態

分散したシステムでも、

- 情報は存在する
- ログは出ている
- メトリクスも取得できる

しかし、それらは分断されている。

それぞれのサービスが独立して動き、異なる場所で、
異なるタイミングで情報を出すためである。

- あるサービスではエラーが出ている
- 別のサービスでは遅延が発生している
- 別の場所ではリソースが逼迫している

それぞれは観測できるが、
それが一つの問題としてどう繋がっているのかは分からない。

3. 観測できても、判断できない

情報があることと、判断できることは別である。

ログが大量にあっても、原因は特定できない。
メトリクスが揃っていても、どこが問題かは分からない。

分断された情報は、そのままでは意味を持たない。

それらがどう関係しているのかを結びつけて初めて、全体の状態が見える。

4.「見る」から「判断する」へ

ここで必要になるのは、情報を増やすことではない。

重要なのは、観測した情報から状態を推測できるかどうかである。

- どこで遅延が発生しているのか
- どの処理がボトルネックになっているのか
- どの変化が影響を与えているのか

これらが判断できるようになることで、はじめて安定した運用が成立する。

5. 可観測性とは何か

可観測性とは、すべてを見ることではない。

ログ、メトリクス、トレースといった観測可能な情報から、
システムの内部状態を推測できるように構成されている性質である。

すべての情報を集めることではなく、判断に必要な情報を扱える状態にすること。

現実の運用では、観測にもコストがある。

- ログを増やしすぎると読めなくなる
- メトリクスを増やしすぎると管理できなくなる
- トレースを増やしすぎるとシステム負荷が増える

そのため可観測性とは、「**どこまで観測するか**」を決める設計でもある。

6. モニタリングとの違い

モニタリングは、あらかじめ決めた指標を監視する。

- CPU 使用率
- レスポンス時間
- エラー率

異常の兆候を捉えることはできる。
しかし、想定していない問題には対応できない。

可観測性は、観測した情報から状態を推測するための前提である。
ツールは、この観測を支える手段にすぎない。

- Prometheus はメトリクスを収集する
- Grafana はそれを可視化する
- Cilium は通信の状態を観測する

重要なのはツールではない。
何を観測するかという判断である。

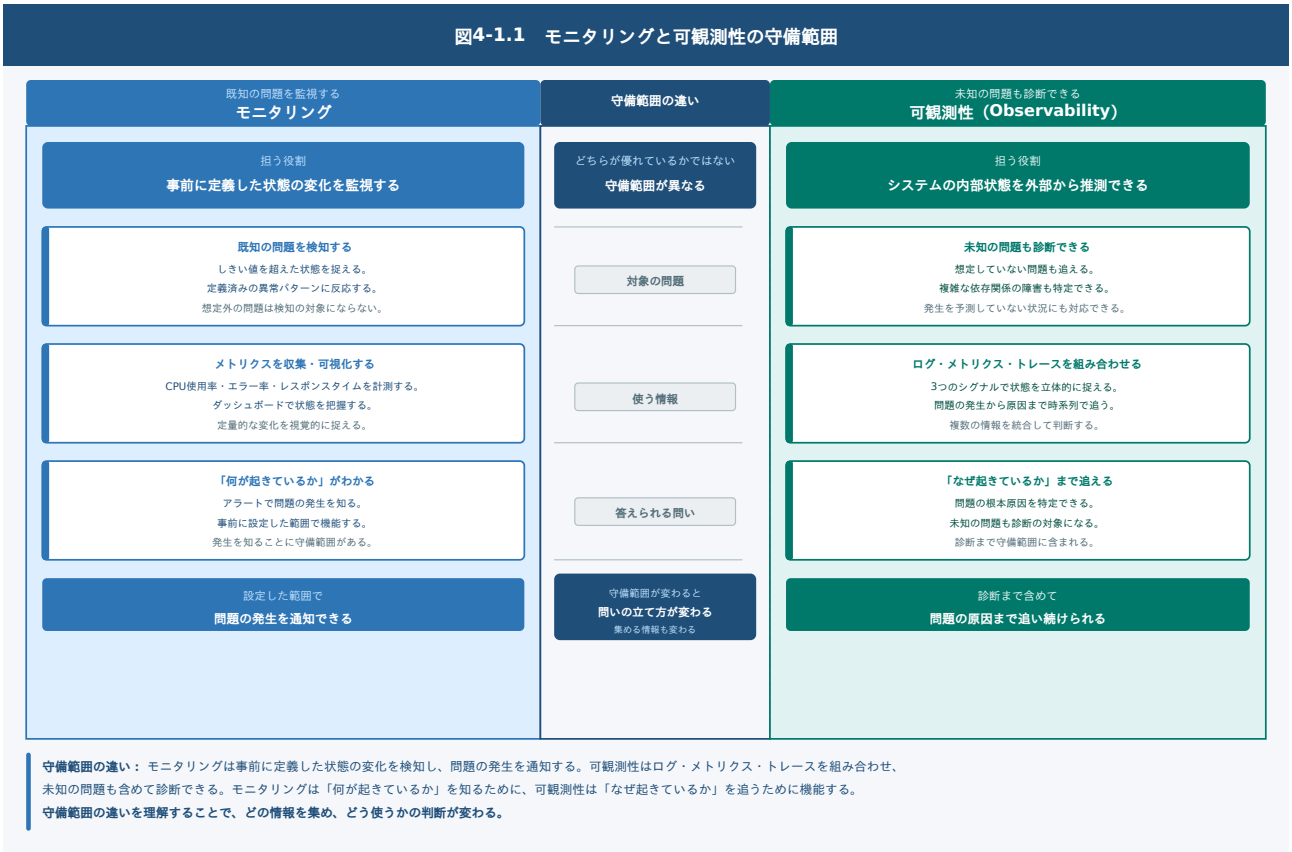


図 4-1.1 モニタリングと可観測性の守備範囲

7. 次に残る問い

観測できるようになれば、それで十分なのか。

- どの情報を見ればよいのか
- どのように関連付けるのか
- どのように異常を判断するのか

それらは、どのように一つの判断になるのか。

第 2 章：トレーシングと分散システムの可視化

1. 分散システムで起きる問題

ここで扱うのは、複数のサービスに分割されたアプリケーションである。
それぞれのサービスが独立して動作し、連携しながら一つの機能を成立させる。

マイクロサービス環境では、一つのリクエストが複数のサービスを経由して処理される。

- API サービスがリクエストを受け取り、
- 認証サービスが認証を行い、
- データサービスがデータベースにアクセスする。

例えば EC サイトでは、

- 注文サービスが注文情報を受ける

- 決済サービスが支払い処理を行う
- 通知サービスが利用者へ結果を通知する

利用者からは一つの購入操作に見えていても、内部では複数のサービスが連携しながら処理を進めている。

この構造では、処理は一箇所で完結しない。

- どこで処理が遅くなっているのか分からない
- どのサービスでエラーが発生しているのか分からない

それぞれのサービスは観測できる。

しかし、それが一つのリクエストとしてどう繋がっているのかは分からない。

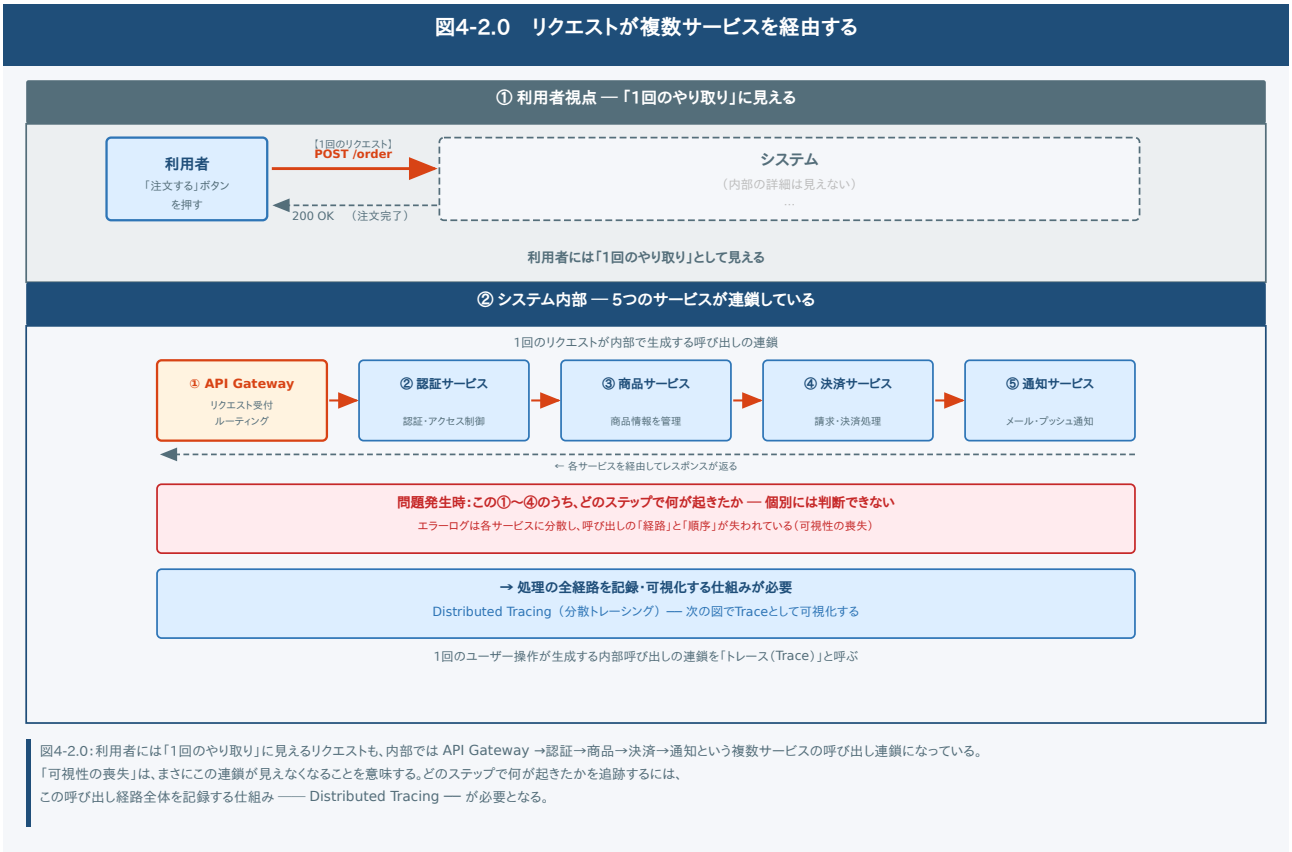


図 4-2.0 リクエストが複数サービスを経由する

この状態では、

- 問題が発生しても原因を特定できない
- 対処しても再現性が持てない
- 同じ障害が繰り返される

処理の流れが分からないままでは、システムは安定して運用できない。

2. 分散トレーシングという考え方

この問題に対応するのが分散トレーシングである。

一つのリクエストが、システムの中でどのように処理されたのかを追跡できるようにする。

- リクエストがどのサービスを通過したか
- 各サービスでどれくらい時間がかかったか

これらを一つの流れとして扱うことで、処理全体を可視化する。

- どこで遅延が発生しているのか
- どこでエラーが起きているのか

それを一つのリクエスト単位で特定できるようになる。
ただし、すべてのリクエストを追えばよいわけではない。

データ量は増え続け、ストレージコストが増大し、
システム負荷にも影響する。

トレーシングとは、すべてを追うことではなく、
どこまで追うかを決める設計である。

3. トレースデータの構造

分散トレーシングでは、リクエストの処理情報をトレースとして記録する。

Trace は一つのリクエスト全体の流れを表し、
Span は各サービスでの処理を表す。

各サービスで発生した処理を Span として記録し、
それらを一つの Trace として関連付けることで、

分断されていた処理が一つのリクエストの流れとして再構成される。

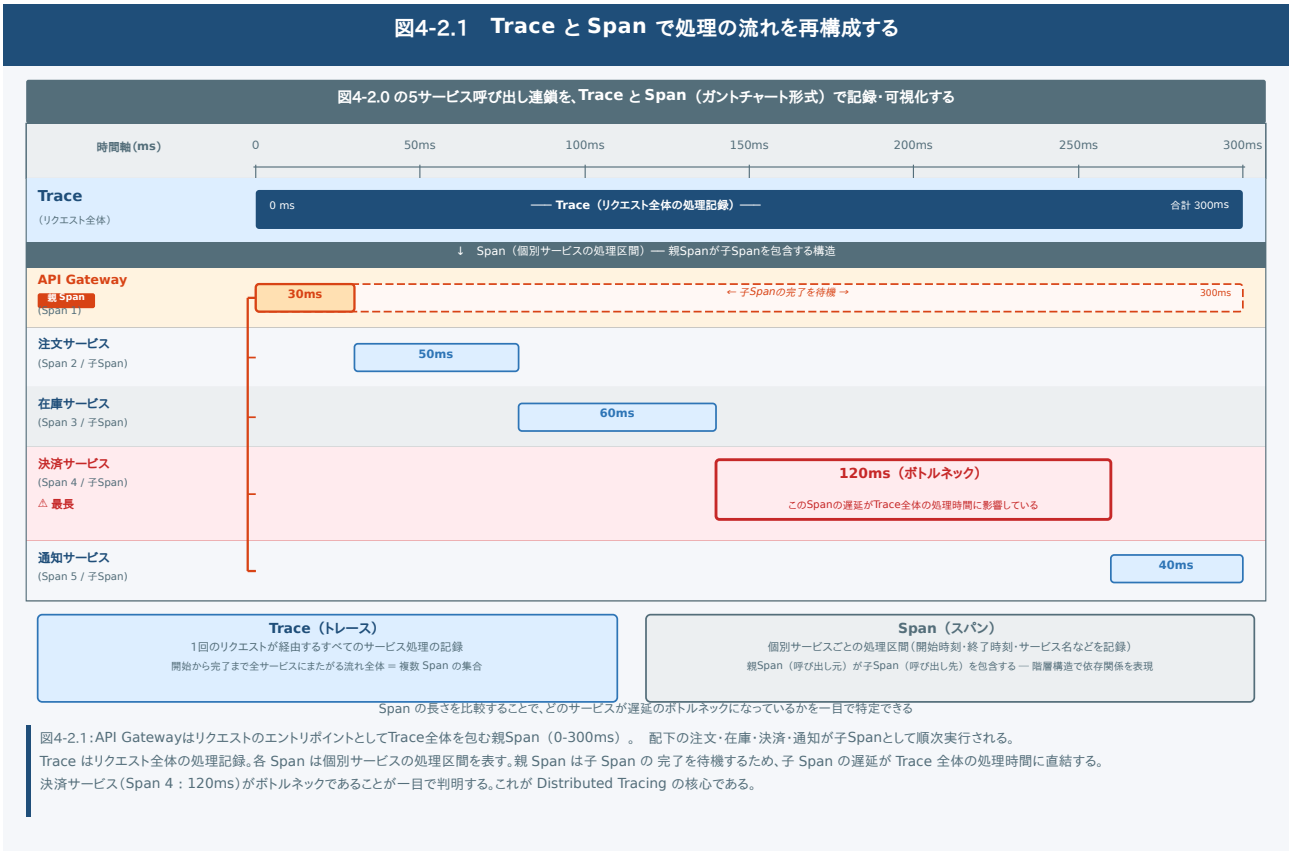


図 4-2.1 Trace と Span で処理の流れを再構成する

4. 代表的なツールの役割

■ Jaeger
トレースデータを収集・保存し、
処理の流れを可視化するためのツールである。

■ OpenTelemetry
ログ・メトリクス・トレースといった観測データを
統一的に扱うための標準仕様である。

重要なのはツールではない。

- どの処理を追跡するのか
- どの粒度で観測するのか

その判断である。

5. 次に残る問い

処理の流れを追えるようになれば、それで十分なのか。

- どのリクエストを追うのか
- どの粒度で記録するのか
- どの情報を残すのか

観測の設計は、どのように成立するのか。

第 3 章：セキュリティとアクセス制御～壊さないために、何を制限するのか～

1. 分散した瞬間に、「壊せる場所」が増える

ここで扱うのは、複数のサービスに分割されたシステムである。

ここでいうセキュリティとは、攻撃から守ることだけではない。
システムを壊さないために、操作を制御するという意味も含まれる。

アプリケーションを分割し、サービスが増え、API が増え、通信経路が増える。
この構造では、システムに対して操作できるポイントが増えていく。

- どのサービスにアクセスできるのか
- どのリソースを操作できるのか
- どの設定を変更できるのか

これらを適切に制御しなければ、意図しない操作が行われ、
本来触れてはいけない範囲に影響が及び、情報漏えいが発生する。

2.「壊せる」というリスク

クラウドネイティブ環境では、

- デプロイ
- 設定変更
- リソース操作

これらの操作が柔軟に行える。

これは強力な特徴である一方で、同時にシステムを壊せる能力でもある。

初学者であっても強い権限を扱えるため、意図しない変更や削除によって、
システム全体に影響が及ぶ可能性がある。

この状態では、誰がどこまで操作できるのかが曖昧になり、
一つの操作が全体に影響する

システムは安定して運用できない。

セキュリティは、攻撃から守るためだけではない。
事故を防ぎ、影響範囲を制御するための仕組みである。

3. アクセス制御（RBAC）

この問題に対して必要になるのがアクセス制御である。

Kubernetes では、RBAC（Role Based Access Control）によって
ユーザーやサービスに対して操作権限を定義する。

RBAC の目的は、権限を与えることではない。

どこまで操作できるのかを限定し、影響範囲を制御することにある。
「できること」を増やすのではなく、**「壊せる範囲」を制限する設計である。**

4. ポリシー管理

アクセス制御だけでは不十分である。

- どのような操作が許可されるのか
- どのような構成が許されるのか

これらを一貫したルールとして管理する必要がある。

このようなポリシー管理の仕組みとして、
OPA（Open Policy Agent）が利用される。

ポリシーの目的はルールを増やすことではない。

秩序のない操作を防ぎ、一貫した制御を維持することにある。
Kubernetes のような柔軟な環境では、統制がなければ簡単に破綻する。

5. Secrets 管理

機密情報の扱いも同様である。

- データベースのパスワード
- API キー
- 認証トークン

これらは単なる設定ではない。

漏えいすれば、システム全体に影響を及ぼす。

- どこから参照できるのか

- どこで使用できるのか

その範囲を制御しなければならない。

6. 次に残る問い

制御できるようになれば、それで十分なのか。

- どこまでを制限するのか
- どの粒度で管理するのか
- どの範囲を許可するのか

その設計は、どのように決まるのか。

第 4 章：運用とトラブルシューティング〜判断し続けるための設計〜

1. 運用とは「継続させること」である

システムは、一度動けば終わりではない。
継続的にサービスを提供し続ける必要がある。

しかしクラウドネイティブ環境では、
コンテナが動的に作成・削除され、
サービスが分散し、構成が継続的に変化する。

この状態では、

- すべてを把握することはできない
- すべてを追い続けることもできない

それでも、サービスは止められない。

- どこに問題があるのか
- どこまで影響しているのか
- どこまで対応すべきなのか

これらをその都度判断する必要がある。

運用とは、すべてを知ることではない。
限られた情報の中で、継続させるために判断し続けることである。

2. 障害対応とは何をしているのか

障害は防ぎきることができない。
重要なのは、発生したときに適切に対応できることである。

一般的に、障害対応は次の流れで行われる。

- 検知する
- 状況を把握する
- 原因を特定する
- 復旧する
- 再発を防ぐ

しかし実際の運用では、すべてを順番に行えるとは限らない。
まず必要なのは、サービスを復旧させることだ。

- 深く調査するのか
- 一時的に回避するのか

どこまで対応するかを判断する必要がある。

しかしこの判断は、情報が不完全な状態で行われる。

- 原因が分からない
- 再現できない
- 同じ障害が繰り返される

判断できなければ、運用は成立しない。

3. なぜツールが必要になるのか

この問題に対して、人が直接状態を追い続けることはできない。

- Pod に入って調査し続ける
- 手動で状態を変更する

こうした操作は一時的には解決できるが、
システムの構造を崩し、長期的に問題を引き起こす。

Kubernetes は宣言的に管理する仕組みである。

その場しのぎの操作は、本来の状態とズレを生み続ける。

さらに、

- 分散したサービスを横断的に把握する必要がある
- 過去の状態や変化を追う必要がある
- 複数の情報を組み合わせて判断する必要がある

kubectl だけでは、この状態に対応できない。

そのため、

- メトリクス
- ログ
- トレーシング

これらの情報を統合して扱う仕組みが必要になる。

重要なのはツールではない。
判断に必要な情報を扱える状態にすることである。

4. 改善し続けるという前提

障害対応は、復旧して終わりではない。

- なぜ発生したのか
- どうすれば防げるのか
- どの観測が不足していたのか

これらを見直す必要がある。

しかし重要なのは、すべてを完璧にすることではない。

- どこまで対応するのか
- どこで割り切るのか

その判断を繰り返すことで、システムの安定性は徐々に高まっていく。

5. 信頼性という設計（SRE）

この判断を支える考え方が SRE である。

SRE では、

- SLI（指標）
- SLO（目標）

これらを定義し、

- どこまでの品質を維持するのか
- どこまでの障害を許容するのか

これらを決める。

信頼性とは、品質を数値として可視化し、
どこまで対応するかを判断できる状態である。

すべてを守ることはできない。
だからこそ、どこを守るのかを決める必要がある。

6. まとめ

運用とは、すべてを把握することではない。

分散し、変化し続けるシステムの中で、
限られた情報をもとに判断し続けることである。

観測によって状態を知り、
制御によって影響を限定し、
その上で、どこまで対応するのかを決める。

すべてを防ぐことはできない。
すべてを理解することもできない。

だからこそ、

- どこまでを見るのか
- どこまでを制御するのか
- どこまでを許容するのか

その判断が必要になる。

技術は、その判断を支えるために存在する。

第 4 部まとめ

第 4 部では、クラウドネイティブシステムを見える化し、安全に運用するための考え方を整理した。

本質は一点だ。**クラウドネイティブ運用とは「判断の連続」である。**

- 可観測性は、すべてを観測することではなくどこまで観測するかを決めることだ。
- トレーシングは、すべてを追うことではなくどこまで追跡するかを決めることだ。
- セキュリティは、攻撃から守るためだけでなくシステムを壊さないための仕組みだ。
- 運用は、すべてを調べることではなくどこまで対応するかを決めることだ。

見る、追う、制御する、判断する。
これらは一連の流れとして繋がっている。

クラウドネイティブ環境では、システムは常に変化し続ける。
すべてを完璧に行うことはできない。

だからこそ、どこまでやるかを決める力が重要になる。

この判断とは、単なる選択ではない。

不完全な情報の中で、
どこまでを受け入れ、
どこまでを制御し、
どこまでを許容するのかを決める行為である。

そしてその判断の積み重ねが、システムの振る舞いを決め、
そのまま価値やコスト、さらにはビジネスの在り方に影響していく。

第 5 部：展開ステップ〜技術を価値に変える〜

第 1 部から第 4 部では、クラウドネイティブのシステムを、
技術の集まりとしてではなく、構造として見てきた。

単位は分離され、
状態は制御され、
変化は観測され、
責任の置き場所は整理された。

ここまでで見てきたのは、システムを成立させるための構造である。
しかし、システムは成立していれば十分なのではない。

その構造は、誰のために使われるのか。
何を支え、何を変え、どのような価値につながるのか。

ここで、新たな問いが生まれる。

この構造は、どのように価値を生み出すのか。

ここで扱う価値とは、収益に限らない。
システムが継続的に機能し続けることそのものも含まれる。

第 5 部では、これまで見てきたクラウドネイティブの構造が、
社会やビジネスの中でどのような意味を持つのかを整理する。

扱うのは、特定のツールや実装手法ではない。
構造が、どのように価値の提供と意思決定に接続されるのか。

この視点を持った上で、次に進む。

第 1 章：社会的課題とクラウドネイティブの解決力

1. 社会の前提が変わったという事実

クラウドネイティブは「便利な技術」だから広がったのではない。
従来の前提ではシステムが成立しなくなったことへの対応として生まれた。

かつてのシステムは、「安定していること」を前提に設計されていた。
一度作ったものを長く動かし、問題が起きたら人が対応する。

このやり方は、環境が大きく変わらず、障害が例外として扱える状況では成立していた。

しかし現在では、この前提が崩れている。

サービスは継続的に変化し、
利用状況は予測できず、
システムは複数の要素で構成され、
ネットワークを前提とした分散構造になる。

変化・分散・不確実性が前提になったということだ。

2.「作ること」より「成立させ続けること」が難しくなった

この変化によって、システムにおける難しさは大きく変わった。
以前は**「どう作るか」**が中心の問題だった。

現在では「どうやって成立させ続けるか」が中心の問題になっている。

- 一部が壊れる
- 状態がずれる
- 環境が変わる

これらは異常ではなく、前提条件だ。

この前提のもとでは、

- 手動での運用は追いつかない
- 固定された構成は維持できない
- 一体化されたシステムは変化に耐えられない

このままでは、システムは長く成立し続けることができない。

さらに現実の制約として、

- 人員は増やせない
- コストは制限される
- 対応できる時間も限られる

こういった条件が重なる。
成立しないだけでなく、維持し続けることもできない。

3. なぜ構造が必要になるのか

この状況に対して、人の対応だけで解決することはできない。

- 変化することを止められない
- 分散を避けられない
- 不確実性を排除できない

この条件のもとでは、安定させることではなく、
壊れながらも戻り続ける構造が必要になる。

これが、

- 状態管理
- 自動修復
- スケーリング
- 再配置

といった仕組みが必要になる理由である。

4. クラウドネイティブが解いている問題

クラウドネイティブは、「成立し続ける構造」を作ることに対する答えである。
第 1 部から第 4 部で見てきた内容は、すべてこのために存在している。

- 実行単位を分ける
- 制御を仕組みに任せる
- 環境の前提を選択する
- 不確実性を前提にする

これらは個別の技術ではない。

変化を受け入れ、分散を前提とし、不確実性の中で成立させ続けるための構造である。

5. 技術ではなく構造として捉える

クラウドネイティブをツールの集合として見ると、

- Kubernetes を覚える
- クラウドサービスを使う
- コンテナを扱う

といった理解に留まる。

しかし本質は、そうではない。

- 何を分離し、どこで責任を持つのか
- どこまでをアプリケーションが持つのか
- どこまでを基盤に任せるのか
- どこで状態を管理するのか

この切り分けこそが、すべての設計の基準になる。

クラウドネイティブは「より良い方法」ではない。
そうしなければ成立しない状況で選ばれた構造である。

6. 次に残る問い

構造によって、システムは成立し続けるようになった。

では、その構造は、

- どのように価値につながるのか
- どのように意思決定に影響するのか
- どのように社会の中で機能するのか

次は、この構造が価値としてどのように現れるのかを見ていく。

第 2 章：SES から価値提供型への転換

1. 分解は責任の分け方を変える

- コンテナによって、アプリケーションは実行単位として分離された。
- Kubernetes によって、その単位の制御は仕組みに委ねられるようになった。
- クラウドによって、インフラの一部は外部に任せられるようになった。

これは単なる技術の変化ではない。

システムを構成する単位が分解され、それぞれに責任が
割り当てられるようになったという変化である。

分解とは、構造を細かくすることではない。
責任の置き場所を再配置することである。

この変化によって、「誰がどこまでを扱うのか」という前提が変わる。
その結果、仕事のあり方そのものが変わっていく。

2. 価値は「どこまで責任を持つか」で決まる

同じ技術を扱っていても、

- 指示された作業を行う
- 環境を提供する
- 運用を引き受ける
- 結果に責任を持つ

提供する技術によって価値はまったく異なる。
この違いはスキルの差ではない。

どこまでを自分の責任範囲として扱っているかによって決まる。

3. 作業では扱えなくなる理由

変化・分散・不確実性が前提となる環境では、作業単位を切り出して
人に割り当てるだけでは 対応できなくなる。

なぜなら、分解された構造では、

- 一部だけを見ても全体は分からない
- 処理が分散し、原因と結果が切り離される
- 状態が変化し続け、前提が固定できない

こういった状況が常に発生するからである。

この状態では、

- 問題を特定できない
- 同じ障害が繰り返される
- 責任の所在が曖昧になる

結果として、**作業をこなすだけでは、システムは成立し続けない。**

4.「何をするか」から「何を成立させるか」へ

この状況の中で求められるのは、作業ではない。
状態を観測し、問題を特定し、改善を継続する。

つまり、**「何をするか」ではなく「何を成立させるか」**が問われる。

ここでいう「成立」とは、

- 状態が把握できること
- 問題が特定できること
- 責任の所在が明確であること

これらの条件が満たされている状態である。

第 4 部で扱った観測・制御・判断は、この成立を支えるための前提である。

5. 構造が価値を決める

構造が変わると、価値の出し方が変わる。

モノリシックな構造では、一体として扱うことが前提となる。

一方で分散された構造では、

- 分離された単位
- 仕組みによる制御
- 変化を前提とした設計

これらのもとで判断が行われる。

この中で、
どこまで責任を持つのか
どこまでを成立させるのか

この選択が、そのまま価値になる。

クラウドネイティブは「より良い方法」ではない。
責任の持ち方そのものを変える構造である。

6. 次に残る問い

責任の範囲が価値を決めるとしたとき、

- その責任はどのように定義されるのか
- どこまで引き受けるべきなのか
- どのように組織として成立させるのか

第 3 章：FinOps ～コストは構造で決まる～

1. なぜコストの問題が現れるのか

モノリシックな構造では、システムは一体として扱われる。
リソースの利用は比較的固定され、コストも予測可能だった。

しかしクラウドネイティブ環境では、
システムは分散され、
構成は動的に変化し、
リソースは増減し続ける。

この変化によって、**コストは固定されたものではなく、常に変化する対象になる。**

2. コストは観測できなければ制御できない

この状況で最初に問題になるのは、何にいくらかかっているのか分からないという状態である。

第 4 部で見たように、**観測できないものは制御できない。**
コストも同じである。

さらに、誰がそのコストに責任を持つのかが曖昧になると、

- 最適化は行われない
- 無駄が放置される

- コストは増え続ける

結果として、**コストは制御されずに増大し続ける。**

3. FinOps が扱っているもの

この問題に対する整理が FinOps である。
FinOps はツールでも手法でもない。

コストを観測し、責任を分け、継続的に最適化するための構造である。

- 可視化（何にいくらかかっているか）
- 責任分界（誰がそのコストを持つのか）
- 継続的な最適化（変化し続ける前提）

これはそのまま、第 1 部から第 4 部で見てきた構造と一致する。

4. コストは設計に現れる

コストは、設計の結果として現れる。

- どの単位で分割するのか
- どの程度スケールさせるのか
- どこまでを自動化するのか
- どの環境を選ぶのか

これらの選択が、そのままコストに反映される。
コストは「設計の影」である。

5. コストは設計を制約する

コストは単なる結果ではない。

- どの構成を選択できるのか
- どこまで冗長性を持たせるのか
- どの程度の可用性を許容するのか

これらの判断に影響を与える。

つまりコストは、**設計の結果であると同時に、設計の制約でもある。**

6. この章が示していること

この章で見てきたことは、コストをどう下げるかという話ではない。
コストは構造をそのまま反映するという事実である。

- 見えていないものは最適化できない
- 責任が曖昧だと改善されない
- 構造が変わるとコストも変わる

クラウドネイティブの世界では、コストは後から管理するものではない。
最初から設計に含まれるものになる。

またコストは、設計の結果として現れるだけでなく、
どの設計を選択するかを制約する要素でもある。

7. 次に残る問い

コストが構造によって決まるとしたとき、

- どの単位でコストを把握するのか
- どのように責任を分割するのか
- どのように継続的に最適化するのか

第 4 章：AI 時代における構造の再定義

0. クラウドネイティブと AI は同じ流れの中にある

ここまで見てきたのは、

- 実行単位の分離
- 制御の仕組み化
- 環境の抽象化
- 責任の再定義

これらの構造である。

クラウドネイティブは、変化・分散・不確実性という前提のもとで、
システムを成立させ続けるために生まれた。

AI もまた、この流れの上に現れている。

人が担っていた判断を外に出し、仕組みとして扱う対象に変える。
これはクラウドネイティブで見てきた構造と同じである。

1. AI もまた責任の分け方を変える技術だ

AI は「新しい機能」を追加しているように見えるが、本質はそこではない。
変わっているのは、**人が担っていた判断や処理の一部が外に出たこと**だ。

手動で行っていた作業が自動化され、判断の一部が仕組みに委ねられ、人が関与する範囲が変わる。

これはクラウドネイティブで見てきた構造の変化と同じである。

2.「使う側」と「成立させる側」

AI の普及によって、大きな分岐が生まれている。
AI を使って結果を得る側と、AI を使って仕組みを成立させる側だ。

この違いはスキルの差ではない。

どこまでを自分の責任として扱うかの違いである。

- 出てきた結果をそのまま使うのか
- その結果がどのように生成されたのかを理解するのか
- その仕組み自体を設計するのか

ここで価値の出し方が分かれる。

3. AI を「使うだけ」では成立しない

AI はなぜその結果になったのか、どこで誤りが生まれたのか、
どの範囲で使えるのかが見えにくい。

この状態で結果だけを使い続けると、

- 誤りに気づけない
- 再現性を持てない
- 問題が拡大する

結果として、**AI を使っているにもかかわらず、システムは成立し続けない。**

4. 観測できるかどうかが分岐になる

このとき重要になるのが、**観測できるかどうか**である。

- どの入力がどの結果につながったのか
- どこで判断が行われたのか
- どの範囲で信頼できるのか

これらを把握できなければ、AI は制御できない。

第 4 部で見てきたように、**観測できないものは制御できない。**
AI も例外ではない。

5. それでも変わらないもの

AI が入っても変わらないものがある。

- どこまで責任を持つのか
- 何を成立させるのか
- どこで境界を引くのか

これらは AI があっても変わらない。

むしろ、ここが曖昧なまま AI を使うと、問題は拡大する。

技術が変わり、役割が変わり、作業が変わる。
その中でも変わらないのは、**何を成立させるのかを定義できる力**である。

クラウドネイティブで見てきた構造は、そのまま AI 時代にも接続される。

- 観測できること
- 制御できること
- 責任が分かれていること

これらを扱える人が、次の時代でも価値を出し続ける。

6. この章が示していること

この章で見てきたことは、AI という新しい技術の扱い方ではない。
クラウドネイティブで見てきた構造が、そのまま別の領域にも適用されるという事実である。

- 実行単位の分離
- 制御の仕組み化

- 環境の抽象化
- 責任の再定義

この構造は、技術が変わっても変わらない。
AI もまた、この構造の上に成り立っている。

重要なのは、技術そのものではない。

何を成立させるのかを定義し、
その責任をどこに置くのかを決めること

この判断ができるかどうかが、価値を生み出せるかどうかを決める。

第 5 部まとめ

構造は社会とつながっている

第 5 部では、働き方・価値の出し方・コスト・AI といったテーマを扱ってきた。
一見するとばらばらのテーマに見えるが、実際にはすべて同じ構造の上に成り立っている。

第 1 部から第 4 部で見てきたのは、

- 実行単位を分けること
- 制御を仕組みに委ねること
- 環境を抽象化すること

これらはすべて、何をどこまで引き受けるのかという責任の分解だ。

第 5 部では、その責任の分け方が働き方・価値・コスト・AI にどのように影響するのかを見てきた。
同じ技術を扱っていても、どこまでを自分の責任として扱うかによって、提供している価値はまったく異なる。

コストは後から発生するものではなく、構造がそのまま反映される。
AI もまた責任の分け方を変える技術であり、構造の延長にある。

技術や手段は変わり続ける。しかし変わらないものがある。

- 何を成立させるのか、
- どこまで責任を持つのか、
- どこで境界を引くのか。

これらはどの時代でも変わらない。

この教科書が扱ってきたのは、ツールや手順ではない。

構造だ。

- 実行単位をどう分けるか
- 制御をどこに置くか
- 責任をどう分けるか
- 状態をどう観測するか

これらを理解することで、技術が変わっても応用できる土台が生まれる。

クラウドネイティブは技術の話だが、その先にあるのは**構造を扱う力**だ。

変化する環境の中で、不確実性を前提として、分散したシステムを成立させる。
この力は、技術だけでなく、ビジネスや社会にもそのまま接続される。

それでも残る問い

ここまで構造を見てきたとき、

- 自分はどこまでを引き受けるのか
- 何を成立させるのか
- どの境界に立つのか

その判断は、どのように行われるのか。

第 6 部：総括

1. 構造を扱うということ

この教科書で扱ってきたのは、クラウドネイティブという技術ではない。

変化・分散・不確実性という前提の中で、
システムをどのように成立させるのかという構造である。

■ 第 1 部では、前提を見た。

環境は変化し続け、システムは分散し、不確実性は避けられない。
この前提のもとでは、従来のやり方では成立しない。

■ 第 2 部では、それを体感した。

分割され、接続され、
状態がズレ、通信が失敗する。

システムは常に不安定な要素を含んでいる。

■ 第 3 部では、その状態をどう扱うかを見た。

- どう分けるか
- どう届けるか
- どうつなぐか
- どう振る舞うか

これらはすべて、設計としての判断である。

■ 第 4 部では、それを観測する方法を見た。

見えることではなく、判断できることが重要になる。

観測とは、状態を把握することではない。
判断を可能にするための情報を得ることである。

■ 第 5 部では、この構造が社会とどのようにつながるのかを見た。

- 働き方
- 価値の出し方
- コスト
- AI

これらはすべて、どこまで責任を持つのかという構造によって決まる。

ここまで見てきたことは、一貫している。

- 技術は変わる
- 手法も変わる
- 役割も変わる

しかし変わらないものがある。

- 何を成立させるのか。
- どこまで責任を持つのか。
- どこで境界を引くのか。

構造を理解するとは、これらを自分で定義できるようになることだ。
構造を扱えるようになると、技術に振り回されなくなる。

新しいツールが出ても、新しい手法が現れても、
それが何を解決し、何を引き受けるのかを判断できる。

構造を扱えるようになると、問題の見え方が変わる。
表面的な事象ではなく、その背後にある
責任の配置や前提のズレが見えるようになる。

構造を扱えるようになると、価値の出し方が変わる。
作業をこなすのではなく、成立させることに責任を持つようになる。

この教科書の目的は、知識を増やすことではない。
判断できる状態をつくることである。

2. それでも残る問い

ここまで構造を見てきたとき、

- 自分は何を成立させるのか
- どこまで責任を引き受けるのか
- どの構造を選ぶのか

その判断は、どのように行われるのか。

この問いは、誰かが用意してくれるものではない。

状況によって変わり続けるものであり、
常に自分で向き合い続ける必要がある。

ここまで見てきた構造は、そのためにある。

では、改めて問い直してほしい。

- **自分は何を成立させるのか**
- **どこまで責任を引き受けるのか**
- **その構造を、どのように選ぶのか**

付録：実践への扉（資格と学習ロードマップ）

この付録の位置づけ

本書では、クラウドネイティブを「技術の集合」としてではなく、
「変化を前提とした分散環境を、どのように観測し、制御し、運用するか」
という構造として整理してきました。

一方で、実際の学習や実務の現場では、こうした知識は資格体系や技術カテゴリとして整理されています。
この付録の目的は、試験対策そのものではありません。

本書で積み上げた構造理解が、

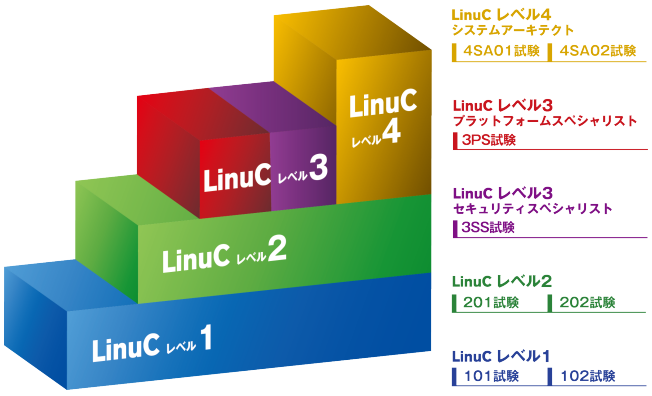
- どの知識体系へ接続されるのか
- どのような役割や責任へ繋がっていくのか
- 次に何を学ぶと理解が深まるのか

を整理し、次の学習への橋渡しとすることにあります。

1. Linux 技術者認定「LinuC（リナック）」のご紹介

Linux 技術者認定「LinuC（リナック）」とは、クラウド／ DX 時代の IT エンジニアに求められるシステム構築から運用管理に必要なスキルを証明できる技術者認定です。アーキテクチャ設計からシステム構築、運用管理までの技術領域を広くカバーしており、４つのレベルの認定取得を通じて一歩ずつ確実に求められるスキルを習得し、それを証明することができます。

LinuC の出題範囲策定や試験開発は、実際に現場で活躍しているハイレベルな IT エンジニアが参加するコミュニティによって行われています。そのため、グローバルで業界標準として利用されている技術領域をカバーし、システム開発や運用管理の現場で本当に必要とされる知識や実践的なスキルを問う内容になっています。その結果として従来型の Linux 領域にとどまった技術認定とは異なり、国内・海外を問わず活躍を目指す IT エンジニアにとっても十分役立つ技術者認定となりました。



LinuC の体系図

LinuC レベル 1

コンピュータシステムを理解し、仮想環境を含む Linux システムの基本操作とシステム管理が行える即戦力エンジニアの証明（ITSS レベル 1）

LinuC レベル 2

仮想環境を含む Linux のシステム設計・ネットワーク構築において、アーキテクチャに基づいた設計・導入・保守・問題解決ができるエンジニアの証明（ITSS レベル 2）

LinuC レベル 3 プラットフォームスペシャリスト・セキュリティスペシャリスト

仮想化・自動化などにより柔軟性と可用性を備えた Linux プラットフォームの構築運用や、セキュア Linux システムのための OS からミドルウェアまでの各層堅牢化や認証認可基盤の構築および攻撃対策を実現できるスペシャリストの証明（ITSS レベル 3）

LinuC レベル 4 システムアーキテクト

オンプレ／クラウド、物理／仮想化を含むシステムのライフサイクル全体を俯瞰して最適なアーキテクチャを設計・構築ができる上級エンジニアの証明（ITSS レベル 4）

LinuC の詳細については、以下の Web サイトを参照してください。

```
https://linuc.org/about/01.html
```

https://linuc.org/about/01.html

2. KCNA（Kubernetes and Cloud Native Associate）

KCNA とは

KCNA は、CNCF（Cloud Native Computing Foundation）が提供する認定資格です。
この資格が確認しているのは、Kubernetes の操作スキルそのものではありません。

- クラウドネイティブとはどのような考え方なのか。
- なぜ分散構造が必要になるのか。
- なぜ宣言的管理や可観測性が重要なのか。

こうした、クラウドネイティブ全体の構造理解を問う資格です。
その意味で本書で扱ってきた内容と接続しやすい知識体系といえます。

試験概要

項目	内容
試験形式	選択式
試験時間	90 分
問題数	60 問
合格ライン	非公開（おおよそ 75%前後とされる）
実施形式	オンライン試験
提供元	CNCF / The Linux Foundation

※試験内容や形式は変更される場合があります。受験時には公式情報を確認してください。

KCNA が扱う領域

KCNA は、クラウドネイティブの世界を大きく以下の領域に分類しています。

１：クラウドネイティブ基礎（すべての出発点は、ここにある。）

コンテナ、マイクロサービス、宣言的管理など、
クラウドネイティブの基本的な考え方を扱う領域です。

> 「なぜ、コンテナが必要になったのか？」

２：Kubernetes とコンテナオーケストレーション（分散を、管理可能にする。）

Pod・Node・Control Plane・Service といった基本構成要素と、
分散環境の管理構造を扱う領域です。

> 「なぜ、分散を管理する仕組みが生まれたのか？」

３：クラウドネイティブアーキテクチャ（変化を前提に、設計する。）

マイクロサービス設計、API、ステートレス構造など、
変化と拡張を前提とした設計の考え方を扱う領域です。

> 「なぜ、固定的な設計では限界が来るのか？」

４：**Observability**【可観測性】（見えないものを、見る技術。）

Monitoring・Logging・Tracing を通じて、
分散環境の内部状態をどのように把握するかを扱う領域です。

「なぜ、システムが分散するほど内部が見えなくなるのか？」

５：ネットワーク（つながりを、抽象化する。）

Service・DNS・Ingress など、
分散環境における接続と通信の仕組みを扱う領域です。

「なぜ、接続先を固定しないのか？」

本書との対応は、次の表で整理します。

本書との対応

本書の構成	KCNA の対応領域
序章・第 1 部：コンテナと Kubernetes の基本構造	Cloud Native Fundamentals / Kubernetes Fundamentals
第 2 部：Kubernetes 実践と基本操作	Kubernetes Fundamentals / Orchestration
第 3 部：GitOps・Service Mesh・分散設計	Cloud Native Architecture / Application Delivery
第 4 部：Observability と運用判断	Observability
第 5 部：継続的変化と価値・責任	Cloud Native Concepts（関連領域）

重要なのは、KCNA 用語を覚えることではありません。

本書で理解してきた構造に、
知識体系としての名前を接続することです。

3. CKA / CKAD / CKS（中級～上級資格）との違いと進め方

中級資格では「責任範囲」が変わる

KCNA がクラウドネイティブ全体の構造理解を扱う資格であるのに対し、
CKA・CKAD・CKS は、それぞれ異なる役割や責任範囲を扱う資格です。

重要なのは、「**どの資格が上か**」ではありません。
どの領域を深く扱い、どの責任を担うのか。

その違いによって、学習の方向性が変わっていきます。

CKA（Certified Kubernetes Administrator）

■分散環境を止めずに支える。

CKA は、Kubernetes クラスタの運用・管理を中心に扱う資格です。

ノード管理、クラスタ構成、ネットワーク設定、障害対応、スケーリングなどを通じて、
「**分散環境を安定して維持する**」ことを扱います。

システムを構築するだけではなく、
障害や変化が起き続ける環境の中で、
状態を把握しながら変化へ対応していく運用が求められます。

「なぜ、運用そのものが設計になるのか？」

CKAD（Certified Kubernetes Application Developer）

■変化し続ける前提で作る。

CKAD は、Kubernetes 上で動作するアプリケーションの設計・配置を扱う資格です。

Deployment、ConfigMap、Secret、Service、Helm、CI/CD などを通じて、
「変化を前提としたアプリケーション設計」を扱います。

アプリケーションを単体で完結させるのではなく、
分散環境の中でどのように接続し、
どのように継続的に変更していくか。

クラウドネイティブ時代のアプリケーション設計に関わる領域です。

「なぜ、アプリケーションも分散前提になるのか？」

CKS（Certified Kubernetes Security Specialist）

■変化を許容しながら、安全性を維持する。

CKS は、Kubernetes 環境におけるセキュリティを専門的に扱う資格です。

RBAC、NetworkPolicy、Supply-Chain-Security、Runtime-Security などを通じて、
「どこまでを許可し、どこで制御し、どのように安全性を維持するか」を扱います。

クラウドネイティブ環境では、システムが分散するほど接続点や責任分界点も増えていきます。
また、Kubernetes は単独で完結する仕組みではありません。

アプリケーション、コンテナ、OS、ネットワークなど、
複数の技術や OSS が組み合わさることでシステムは成立しています。

そのため、CKS では Kubernetes の設定だけではなく、
周辺技術や基盤レイヤーに対する理解も求められます。

セキュリティは特定のコンポーネントだけで成立するものではなく、
システム全体の境界や責任分界点を意識しながら考える必要があります。

その中で、
柔軟性を保ちながら、
どのように境界を設計し、
どのようにリスクを制御し、
安全性を維持するか。

それがこの領域の中心になります。

CKS は、Kubernetes のセキュリティ資格であると同時に、
クラウドネイティブ環境全体をどのように守るかを考える資格ともいえます。

「なぜ、分散するほど責任分界点が重要になるのか？」

[!WARNING]
CKS の受験には CKA 資格が必要

どこから進むべきか

学習の順番に絶対的な正解はありません。

重要なのは、「**自分がどの領域を深めたいのか**」です。

関心	次の選択肢
クラウドネイティブ全体像を整理したい	KCNA
Kubernetes の運用を深めたい	CKA
アプリケーション設計を深めたい	CKAD
セキュリティや境界制御を深めたい	CKS

※ KCNA は、本書で学んだ構造理解を知識体系として整理したい場合の入口になります。

資格はゴールではなく、理解を深めるための視点のひとつです。

本書で扱ってきた構造理解を土台にしながら、自分がどの責任や役割を深めたいのかを考え、
次の学習へ接続していくことが重要になります。

4. 学習プラン例

■学習に「唯一の正解」はない

クラウドネイティブの学習では、
ひとつの技術だけで完結することはほとんどありません。

- 運用
- 開発
- 設計
- セキュリティ
- 観測

それぞれが接続しながら、システム全体を構成しています。

そのため、重要なのは、「**何から始めるか**」だけではなく、
「**どの理解を深めたいのか**」です。

また、学習の進み方もひとつではありません。

全体像から段階的に広げていく方法もあれば、
特定の役割や責任から深めていく方法もあります。

ここでは、代表的な学習の進め方を例として紹介します。

１：段階的に学びたい場合

▼まず全体を理解し、その後に深める。

クラウドネイティブ全体の流れを把握しながら、少しずつ実践と専門領域へ進んでいく学習プランです。

フェーズ	学習内容
Step1	Linux / Docker 基礎
Step2	本書で構造理解
Step3	KCNA で知識体系整理
Step4	Kubernetes ハンズオン
Step5	CKA / CKAD / CKS へ分化

「なぜ、部分理解だけでは全体運用が難しくなるのか？」

２：運用寄りに進みたい場合

▼止めないために理解する。

Kubernetes の運用や障害対応、安定稼働を深めていく方向の学習プランです。

フェーズ	学習内容
Step1	本書 第 1〜4 部
Step2	Kubernetes 基本操作
Step3	Monitoring / Logging
Step4	CKA
Step5	実運用・障害対応

障害や変化が起き続ける環境の中で、どのように安定性を維持するか。
分散環境における運用判断や状態把握を深めていく流れになります。

「なぜ、分散環境では“運用”そのものが重要になるのか？」

３：開発・設計寄りに進みたい場合

▼変更され続ける前提で作る。

アプリケーション設計や継続的変更、分散環境に適応した設計を深めていく方向の学習プランです。

フェーズ	学習内容
Step1	本書 第 1〜3 部
Step2	Kubernetes 基本操作
Step3	GitOps / CI/CD
Step4	CKAD
Step5	マイクロサービス設計

アプリケーションを単体で完結させるのではなく、継続的に変更される環境の中で、
どのように接続し、どのように運用するかを学んでいきます。

「なぜ、設計も“変化”を前提にする必要があるのか？」

4：セキュリティ寄りに進みたい場合

▼変化を許容しながら守る。

クラウドネイティブ環境における、境界設計や責任分界点を深めていく方向の学習プランです。

フェーズ	学習内容
Step1	本書 第 4〜5 部
Step2	Kubernetes 基礎
Step3	RBAC / Policy 管理
Step4	CKS
Step5	Supply Chain Security

クラウドネイティブ環境では、システムが分散するほど接続点や責任分界点も増えていきます。
その中で、どこまでを許可し、どこで制御するのか。

柔軟性を保ちながら、境界をどのように設計するかを学んでいきます。

> 「なぜ、分散するほど“境界”が重要になるのか？」

■ 最後に

クラウドネイティブは、単一技術だけで完結する世界ではありません。

重要なのは、知識を増やすことだけではなく、
「どのような観測点でシステムを見るか」を広げていくことです。

学習プランそのものにも、絶対的な正解はありません。

クラウドネイティブ自体が変化を前提とした世界である以上、
学び方もまた、固定されるものではありません。

本書が、その入口となれば幸いです。

まとめ

本書では、クラウドネイティブを単なる技術用語や製品群としてではなく、
「変化を前提とした分散環境を、どのように観測し、制御し、運用するか」
という構造として整理してきました。

コンテナ、Kubernetes、GitOps、Service Mesh、Observability といった技術も、
それぞれを個別に覚えるためではなく、「なぜ、その構造が必要になったのか」
を理解するために扱っています。

クラウドネイティブの世界では、システムは常に変化し続けます。

構成は固定されず、役割も変わり続け、求められる責任や視点も増えていきます。
そのため、重要なのは、「正解を覚えること」ではなく、「変化の中で、どのように構造を捉えるか」です。

資格や学習ロードマップも、そのための入口のひとつに過ぎません。

KCNA、CKA、CKAD、CKS といった資格体系も、本来は知識を増やすためだけではなく、
「どの視点でシステムを見るのか」を整理するためのものともいえます。

学習の進め方に、絶対的な正解はありません。

運用を深める人もいれば、設計を深める人もいる。
セキュリティや可観測性へ進む人もいるでしょう。

重要なのは、自分がどのような視点でシステムを見たいのか、
どのような責任を担いたいのかを考え続けることです。

本書が、クラウドネイティブという変化し続ける世界を、
構造として捉えるための入口となれば幸いです。

クラウドネイティブ教科書

1.0.0

発行元 LPI-Japan