IT資格道場

KCNA-JP 学習用テキスト

予備知識ゼロから合格ラインへ ―― 読んで理解する1冊目

無料サンプル(第1章まで)

CNCF 公式 KCNA Exam Curriculum(2025-11-24 版・2026-09-07 取得)準拠
2026年7月版

本書のご利用にあたって

- 本書は「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルであっても著作物です。本書の転載・改変しての再配布・二次販売を禁じます。
- 本書を AI モデルの学習データとして利用すること(学習目的の入力・データセット化を含む)を禁じます。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

不正な配布を見つけた場合は info@shikaku-doujou.com までご連絡ください。

目次

この教材群の使い方(3点セット)

種類	役割
① 学習用テキスト(本書)	これだけで問題が解けるようになる本編
② 頻出度別ノート	テキストを読んだら次はこれ。頻出順に絞った問題と解説で「解ける」に橋渡し。試験直前の最終チェックにも
③ 暗記用ノート	覚えるべき要点だけを一問一答に凝縮。空き時間の反復と用語の再確認用

使い方: ① 学習用を読む → ② 頻出度別で解き方を掴む → ③ 問題演習で腕試し → ④ 頻出度別に戻って再確認(試験直前もここ)。暗記用は空き時間や用語の再確認に。

このテキストの位置づけ: 本書は①の学習用テキストです。まずはここを通読し、これだけで問題が解けるようになることを目標にしてください。

はじめに ―― この教材の使い方

このテキストは、Linux Foundation と CNCF が実施する **Kubernetes and Cloud Native Associate（KCNA-JP・日本語版）** の合格を目的に作られています。

この試験が問うのは、クラスタを手で運用する腕前ではありません。問われるのは、**Kubernetes の構成要素とオブジェクトの役割、コンテナ・ネットワーク・ストレージ・セキュリティの基本、アプリケーション配信（GitOps・CI/CD）、可観測性、そして CNCF エコシステムの全体像を、用語のレベルで**

正しく理解していることです。CNCF のカリキュラムは、この試験を Kubernetes の入口（CKA・CKAD の前段）と位置づけています。

本書は Linux Foundation / CNCF の公式教材ではなく、IT資格道場が独自に書き下ろしたオリジナルの教材です。実際に出題された問題を再現したものではありません。

試験の基本情報

項目	内容
試験名	Kubernetes and Cloud Native Associate（KCNA-JP）
実施	Linux Foundation（オンライン監督試験）
問題数	非公表
試験時間	**90分**
出題形式	**択一（multiple choice）**
合格ライン	非公表
受験言語	日本語（KCNA-JP）・英語（KCNA）

受験料・試験時間・出題数・試験ガイドは改定されることがあります。お申し込みの前に、Linux Foundation 公式サイトで最新の情報をご確認ください。

想定される受験者

公式カリキュラムが想定するのは、**これから Kubernetes とクラウドネイティブ技術を学び始める人**（開発者・運用者・学生）です。本書は、**コンテナの基礎から Kubernetes の構成要素、エコシステムまでをゼロから順に積み上げる構成**にしてあります。

出題範囲の全体像 —— 4つのドメインと本書の章

ドメイン	分野	採点対象に占める割合（公式）	本書
1	Kubernetesの基礎	**44%**	第1章
2	コンテナオーケストレーション	**28%**	第2章
3	クラウドネイティブアプリケーション配信	**16%**	第3章
4	クラウドネイティブアーキテクチャ	**12%**	第4章

本書の章の並びは、CNCF カリキュラムの並びとまったく同じです。節見出しの末尾にある `[1.1]` のような番号は、本書がカリキュラムのドメインに付けた番号です。CNCF はドメインごとの割合だけを公表しており、細目単位の出題数は公表していません。本書もそれ以上の数字は書きません。

サービス名・機能名は英語のまま覚える

本文では、**製品名・機能名・用語を英語表記のまま**書いています（Pod、Deployment 、Service 、kubelet 、etcd 、CRI/CNI/CSI 、Helm 、GitOps 、Prometheus、OpenTelemetry …）。本番の設問文と選択肢に出るのはこの表記であり、日本語訳だけを覚えても画面では出会いません。英語の名前を見た瞬間に「どのドメインの、どの場面の機能か」が反射で出る状態を目指してください。

全設問に効く判断軸 ——「3つの問い」

問い	何を切り分けるか	分かれ方
問い1: どの層の話か	コントロールプレーン／ノード／ワークロード／ネットワーク・ストレージ／エコシステム	同じ「設定を渡す」でも、ConfigMap・Secret・環境変数・ボリュームは層が違い、要件がどの層を指しているかで答えが決まります
問い2: 誰がその役割を担うか	API server・scheduler・controller-manager・etcd・kubelet・kube-proxy	「Pod をノードに割り当てる」「望ましい状態へ収束させる」は、この軸だけで一意に決まります
問い3: クラウドネイティブの原則に沿っているか	宣言的・自動化・疎結合・観測可能・回復性	要件を満たす案が複数あるとき、正解はより宣言的で、より自動化されたものです

4ステップ学習法

購入者限定テキストは3冊で1セットです。

- **STEP 1（理解する）**: 本書「学習用テキスト」を通読し、4つのドメインの地図と3つの問いを頭に入れる
- **STEP 2（解き方を掴む）**: 「頻出度別ノート」で頻出度の高い論点を解いて、解き方を掴む
- **STEP 3（腕試し）**: 本サイトの問題演習で、本番と同じ択一形式に慣れる
- **STEP 4（再確認）**: 「頻出度別ノート」に戻って総ざらい。試験直前もここへ戻る

「暗記用テキスト」は本線には入れません。通勤時間や休憩時間など、**空き時間の反復と用語の再確認**に並走させてください。

それでは、第1章から始めます。

第1章 Kubernetes Fundamentals（Kubernetesの基礎）（ドメイン1・採点対象の44%）

このドメインは KCNA-JP の全4ドメインのうち最大の配点を占めます。試験範囲の小項目は Kubernetes Core Concepts、Administration、Scheduling、Containerization の4つです。ここで扱う語彙と対応関係が、以降の第2章から第4章までの土台になります。

読み方の指針を3つだけ述べます。第一に、Kubernetes の用語は英語表記のまま覚えてください。試験は日本語ですが、オブジェクト名（Pod、Deployment、Service など）は英語のまま出題されます。第二に、「何をするものか」より「誰の責任か」を覚えてください。KCNA は Associate レベルの試験なので、コマンドの細かいオプションではなく、コンポーネントと役割の対応関係が問われます。第三に、似た用語の対比を必ずセットで押さえてください。Deployment と StatefulSet、requests と limits、liveness probe と readiness probe のように、対で覚えないと誤答肢に引っかかります。

1-1 Kubernetes Core Concepts（Kubernetesの中核概念）[1.1]

Kubernetes Core Concepts は、Kubernetes がどんな部品でできていて、どんな考え方で動いているかを扱う小項目です。ドメイン1のなかでも中心にあたる部分で、ここが曖昧だと Scheduling も Administration も理解できません。

1-1-1 Kubernetesとは何か —— コンテナオーケストレータの役割 [1.1]

Kubernetes は、コンテナ化されたアプリケーションのデプロイ、スケーリング、管理を自動化するためのオープンソースのプラットフォームです。名前はギリシャ語で「操舵手」「水先案内人」を意味し、K と s の間に8文字あることから K8s と略記されます。もともと Google の社内システムの設計思想を引き継いで開発され、2015年に最初のバージョンが公開されると同時に Cloud Native Computing Foundation（CNCF）へ寄贈されました。CNCF が最初にホストしたプロジェクトであり、現在は Graduated（卒業）ステータスにあります。

コンテナオーケストレータが解く問題を、場面で考えてみます。ある物流会社の配送管理チームが、注文受付、在庫照会、配送ルート計算の3つのサービスをコンテナとして動かしているとします。サーバが1台で、コンテナが3個なら、手作業でも回ります。しかしサーバが50台、コンテナが1,000個になった瞬間に、次の問いが人間の手に負えなくなります。

問い	Kubernetes がやること
このコンテナをどのサーバで動かすか	kube-scheduler が空き容量と制約を見て配置先を決める
コンテナが落ちたらどうするか	コントローラが検知して自動的に再作成する
サーバごと落ちたらどうするか	そのノード上の Pod を別ノードへ再配置する
アクセスが増えたらどうするか	HorizontalPodAutoscaler が replica 数を増やす
新バージョンをどう出すか	Deployment がローリングアップデートを段階的に実行する

問い	Kubernetes がやること
どうやってコンテナ同士を見つけるか	Service と DNS が安定した名前とIPを提供する
設定や機密情報をどう渡すか	ConfigMap と Secret がイメージの外から注入する

Kubernetes の中核にある考え方は「宣言的（declarative）」です。利用者は「こうなっていてほしい」という望ましい状態（desired state）を宣言し、Kubernetes がその状態に近づける努力を続けます。「この手順を実行せよ」という命令的（imperative）な指示とは対照的です。この違いは KCNA で頻出します。

一方で、Kubernetes が「やらないこと」も試験に出ます。公式ドキュメントは Kubernetes を PaaS ではないと明記しています。具体的には、次のことは Kubernetes の責任範囲外です。

- ソースコードのビルドやCI/CDパイプラインそのもの（Argo CD や Tekton などの別プロジェクトが担う）
- アプリケーションレベルのサービス（メッセージバス、データベース、キャッシュを標準機能として提供はしない）
- ログ・監視・アラートの仕組みそのもの（収集の口は用意するが、Prometheus や Fluentd などのエコシステムに委ねる）
- ミドルウェアや設定言語の強制（どの言語・どのフレームワークで書くかは問わない）

よくある取り違えを表にします。

誤解	正しい理解
Kubernetes はコンテナランタイムである	ランタイムではない。containerd などのランタイムを CRI 経由で使う側
Kubernetes を入れればCI/CDが手に入る	CDの対象になるだけで、パイプラインは別途用意する
Kubernetes はDockerの代わり	Docker はイメージのビルドと実行の道具、Kubernetes は多数のノードにまたがる調整役
Kubernetes は自動でアプリを速くする	配置と冗長化を自動化するだけで、アプリの性能は変わらない

誤答肢の切り方としては、「Kubernetes が○○を提供する」という選択肢のうち、CI/CDパイプライン、ソースコード管理、アプリケーションのビルド、モニタリングのダッシュボードを挙げているものは、まず疑ってください。これらはエコシステム側の役割です。

1-1-2 クラスタアーキテクチャ —— コントロールプレーンとノード [1.1]

Kubernetes クラスタは、コントロールプレーン（control plane）と、ワーカーノード（node）の集合でできています。コントロールプレーンが「頭脳」、ノードが「手足」です。本番環境では可用性のためにコントロールプレーンを複数台構成にしますが、学習環境では1台に同居させることもあります。

コントロールプレーンのコンポーネントは次の5つです。この対応関係は必ず暗記してください。

コンポーネント	役割	覚え方
kube-apiserver	Kubernetes HTTP API を公開する中核サーバ。すべての操作の唯一の入口	玄関
etcd	クラスタの全データを保持する一貫性と高可用性を備えたキーバリューストア	台帳
kube-scheduler	まだノードに割り当てられていない Pod を探し、適切なノードへ割り当てる	配車係
kube-controller-manager	各種コントローラを動かし、API のふるまいを実現する	現場監督
cloud-controller-manager	クラウドプロバイダ固有の処理（ロードバランサ、ノード、ルート）を担う。任意	外部との窓口

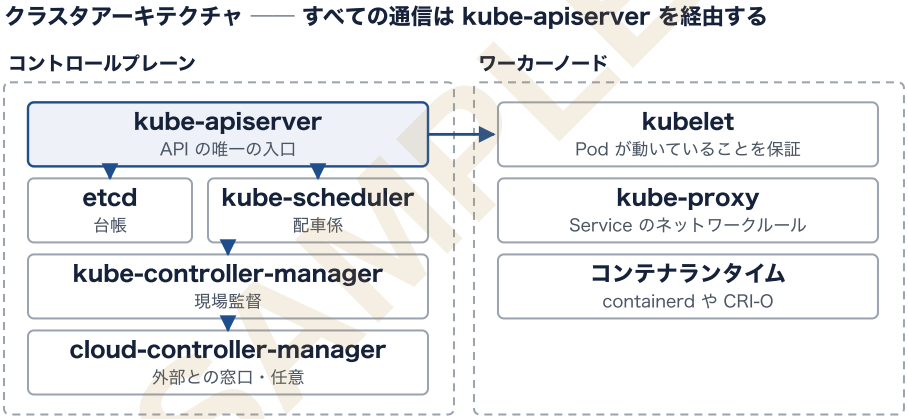
ノード側のコンポーネントは次の3つです。

コンポーネント	役割
kubelet	ノード上で Pod とそのコンテナが動いていることを保証するエージェント
kube-proxy	Service を実現するためのネットワークルールをノード上で維持する。任意（CNIが代替する場合がある）
コンテナランタイム	実際にコンテナを動かすソフトウェア。containerd や CRI-O

ここで注意すべき点が3つあります。第一に、kube-apiserver 以外のコンポーネントは、互いに直接通信しません。すべて API server を経由します。「kube-

scheduler が kubelet に直接 Pod の起動を指示する」という選択肢は誤りです。正しくは、kube-scheduler が API server 経由で Pod にノード名を書き込み、そのノードの kubelet が API server を監視していて自分宛ての Pod を見つけて起動します。

第二に、etcd に直接アクセスするのは API server だけです。コントローラも kubectl も etcd を直接触りません。第三に、kubelet はコントロールプレーンではなくノード側のコンポーネントです。コントロールプレーンのノードにも kubelet は動いていますが、分類上はノードコンポーネントです。



kube-apiserver 以外のコンポーネントは互いに直接通信せず、すべて API server を経由します。
etcd に直接アクセスするのは API server だけです。kubelet はノード側のコンポーネントです。

図 1-1　クラスタアーキテクチャ —— すべての通信は kube-apiserver を経由する

アドオン（Addons）は、クラスタの機能を拡張するもので、多くは Namespace の kube-system に配置されます。公式ドキュメントが挙げるものは次のとおりです。

- DNS —— クラスタ全体の名前解決。実質必須で、通常は CoreDNS が使われます
- Web UI（Dashboard）—— ブラウザからクラスタを管理する画面
- Container Resource Monitoring —— コンテナのメトリクスを収集して保存する
- Cluster-level Logging —— コンテナのログを中央のログストアに保存する

このほか、ネットワークプラグイン（CNI プラグイン）も実質的なアドオンで、これが無いと Pod 同士が通信できません。

コントロールプレーンが停止した場合に何が起きるかも、よく問われます。既に動いている Pod はノード上で動き続けます。kubelet はコンテナを再起動させることもできます。しかし、新しい Pod の作成、スケール、自動復旧、kubectl による操作はすべてできなくなります。「コントロールプレーンが落ちるとすべてのアプリが即停止する」という選択肢は誤りです。

1-1-3 Kubernetes API とオブジェクトモデル [1.1]

Kubernetes を操作するとは、Kubernetes API に対してオブジェクトを作成・更新・削除することです。kubectl も、ダッシュボードも、CI/CDツールも、内部では同じ REST API を叩いています。

Kubernetes オブジェクトは、ほぼすべて次の4つのトップレベルフィールドを持ちます。

フィールド	内容
apiVersion	どのAPIグループ・バージョンのオブジェクトか（例: v1、apps/v1、batch/v1）

フィールド	内容
kind	オブジェクトの種類（例: Pod、Deployment、Service）
metadata	名前、Namespace、labels、annotations、UID など
spec	望ましい状態（desired state）。利用者が書く
status	実際の状態（current state）。Kubernetes が書く

spec と status の分離が、宣言的モデルの核心です。利用者は spec に「replicas: 3」と書きます。Kubernetes は現実を観測して status に「readyReplicas: 2」と書きます。この差分を埋めようとする動きが、次項の調整ループです。「status は利用者が書くフィールドである」という選択肢は誤りです。

API はグループ（API group）に分かれています。中核オブジェクト（Pod、Service、ConfigMap、Secret、Namespace、Node、PersistentVolume など）は「core」グループに属し、apiVersion は単に v1 と書きます。ワークロードは apps/v1（Deployment、ReplicaSet、StatefulSet、DaemonSet）、バッチは batch/v1（Job、CronJob）、ネットワークは networking.k8s.io/v1（Ingress、NetworkPolicy ）、認可は rbac.authorization.k8s.io/v1（Role 、RoleBinding、ClusterRole、ClusterRoleBinding）です。

APIバージョンには安定度の段階があります。

段階	表記例	意味
Alpha	v1alpha1	既定で無効。バグを含む可能性があり、予告なく削除される。本番非推奨
Beta	v1beta1	既定で有効なことが多い。詳細は変わりうるが機能は残る
Stable（GA）	v1	以後のリリースでも提供され続ける

API server へのリクエストは、必ず次の3段階を通ります。これはセキュリティの章でも再登場します。

1. Authentication（認証）—— 誰からのリクエストかを確認する
2. Authorization（認可）—— その人にその操作が許されているかを判定する。RBAC がここ
3. Admission Control（アドミッション制御）—— リクエストの内容を検証したり書き換えたりする

認証が通っただけでは操作できません。「認証に成功すればすべてのリソースを操作できる」は誤りです。

1-1-4 Pod —— 最小のデプロイ単位 [1.1]

Pod は、Kubernetes で作成・管理できる最小のデプロイ単位です。ここで重要なのは、最小単位がコンテナではなく Pod だということです。Pod は1つ以上のコンテナをまとめたもので、次のものを共有します。

- ネットワーク名前空間 —— 同じ Pod 内のコンテナは同一のIPアドレスとポート空間を共有し、localhost で互いに通信できます

- ストレージボリューム —— Pod に定義したボリュームを複数コンテナがマウントできます
- ライフサイクル —— Pod として一緒にスケジュールされ、一緒に終了します

Pod は基本的に使い捨て（ephemeral）です。停止した Pod が自動的に復活することはなく、代わりに新しい Pod が作られます。新しい Pod は新しい名前と新しいIPアドレスを持ちます。だからこそ、後述する Service という安定した宛先が必要になります。「Pod は再起動しても同じIPアドレスを保持する」は誤りです。

Pod の phase（フェーズ）は5種類です。これは Troubleshooting と Debugging でも使うので、ここで押さえます。

phase	意味
Pending	API に受理されたが、まだ全コンテナが起動していない。スケジュール待ち、イメージ取得中を含む
Running	ノードに割り当てられ、少なくとも1つのコンテナが動作中
Succeeded	全コンテナが正常終了（exit code 0）し、再起動されない。Job で見る
Failed	全コンテナが終了し、少なくとも1つが異常終了した
Unknown	Pod の状態を取得できない。多くはノードとの通信断

Pod 内の補助コンテナには、代表的な2つのパターンがあります。

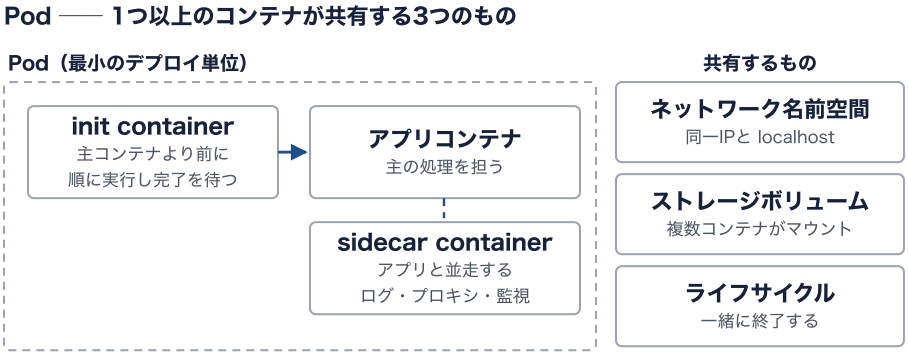
init container は、アプリコンテナが起動する前に順番に実行され、完了まで待たれるコンテナです。設定ファイルの取得、依存サービスの起動待ち、マイグレーションの実行に使います。init container が失敗すると、Pod は再試行を繰り返し、アプリコンテナは起動しません。

sidecar container は、アプリコンテナと並走して補助機能を提供するコンテナです。ログ収集エージェント、プロキシ、証明書の更新などが典型です。サービスメッシュの Envoy プロキシも sidecar として注入されます。

項目	init container	sidecar container
起動タイミング	主コンテナより前	主コンテナと並走
終了条件	完了して終了する	主コンテナが動く間ずっと動く
用途	前準備、依存待ち	ログ、プロキシ、監視

Pod のライフサイクル管理には restartPolicy があります。Always（既定。Deployment 系で使う）、OnFailure（失敗時のみ再起動。Job で使う）、Never（再起動しない）の3つです。Deployment が管理する Pod で restartPolicy に Never を指定することはできません。

Pod を直接 YAML で作ることは可能ですが、実務では推奨されません。裸の Pod（naked Pod）はノード障害時に再作成されないためです。実務では、次項のワークロードリソースを通じて Pod を作ります。



Pod は使い捨てで、停止しても復活せず、新しい名前と新しいIPアドレスの Pod が作られます。restartPolicy は Always（既定）、OnFailure、Never の3つです。

図 1-2　Pod —— 1つ以上のコンテナが共有する3つのもの

1-1-5 ワークロードリソース —— Pod を管理する仕組み [1.1]

ワークロードリソース（workload resources）は、Pod を直接管理する代わりに「こういう Pod がこれだけ欲しい」を宣言するオブジェクトです。KCNA で最も出題されやすい対応関係なので、用途と特徴を対比で覚えてください。

リソース	apiVersion	用途	特徴
Deployment	apps/v1	ステートレスなアプリを動かす最も一般的な方法	ローリングアップデートとロールバックができる。ReplicaSet を介して Pod を管理する
ReplicaSet	apps/v1	指定数の Pod レプリカを維持する	通常は Deployment 経由で間接的に使う。直接作ることは少ない

リソース	apiVersion	用途	特徴
StatefulSet	apps/v1	固有のアイデンティティが必要な Pod を管理する	安定したホスト名、順序付きの起動と停止、Pod ごとの PersistentVolume
DaemonSet	apps/v1	すべてのノード（または一部のノード）に1つずつ Pod を動かす	監視エージェント、ログ収集、CNIプラグイン、GPUドライバ
Job	batch/v1	完了までを1回実行するタスク	バッチ処理、データ分析。完了したら終わる
CronJob	batch/v1	スケジュールに従って Job を繰り返す	Unix の cron 形式。バックアップ、定期レポート
ReplicationController	v1	レガシー。Deployment と ReplicaSet に置き換えられた	新規に使わない

Deployment の階層関係は必ず問われます。Deployment が ReplicaSet を作り、ReplicaSet が Pod を作ります。ローリングアップデートのとき、Deployment は新しい ReplicaSet を作って Pod を増やしつつ、古い ReplicaSet の Pod を減らします。古い ReplicaSet は replicas を 0 にした状態で残るので、ロールバックが可能になります。「Deployment が直接 Pod を作る」は誤りです。

Deployment の更新戦略は2つです。

戦略	動作	ダウンタイム
RollingUpdate（既定）	新旧の Pod を入れ替えながら段階的に更新。maxSurge と maxUnavailable で速度を制御	無し
Recreate	すべての旧 Pod を削除してから新 Pod を作る	発生する

StatefulSet と Deployment の違いは、KCNA の定番の対比です。

観点	Deployment	StatefulSet
Pod の名前	ランダムな接尾辞（web-6d4cf56db6-x8k2p）	序数付きの安定名（web-0、web-1、web-2）
Pod の入れ替え	交換可能。どれでも同じ	固有。同じ名前と同じストレージで再作成される
起動・停止の順序	並行	順序どおり（0→1→2、削除は逆順）
ストレージ	共有または無し	volumeClaimTemplates により Pod ごとに専用の PersistentVolume
ネットワーク	通常の Service	Headless Service と組み合わせて Pod ごとのDNS名を得る
典型用途	Webサーバ、APIサーバ	データベース、分散キーバリューストア、メッセージブローカ

DaemonSet は replicas を指定しません。ノード数に追随して自動的に1ノード1 Pod になります。ノードを追加すれば自動的に Pod が増え、ノードを削除すれば消えます。「DaemonSet の replicas を3に設定する」という選択肢は誤りです。

Job と CronJob の関係も対で覚えます。Job は completions（何回成功したら完了か）と parallelism（同時に何個動かすか）で制御します。backoffLimit は失敗時の再試行回数の上限です。CronJob は schedule フィールドに cron 式を書き、時刻が来るたびに Job オブジェクトを新しく作ります。concurrencyPolicy で、前回の Job がまだ動いているときの挙動（Allow / Forbid / Replace）を決めます。

場面で整理します。ある小売企業の在庫システムで、次の割り当てが妥当です。

- 商品検索API（ステートレス、負荷に応じて増減したい）→ Deployment
- 在庫データベース（各インスタンスが自分のディスクを持ち、順序よく起動したい）→ StatefulSet
- 全ノードのログを集める収集エージェント → DaemonSet
- 毎晩0時に売上を集計する処理 → CronJob
- 一度きりのデータ移行スクリプト → Job

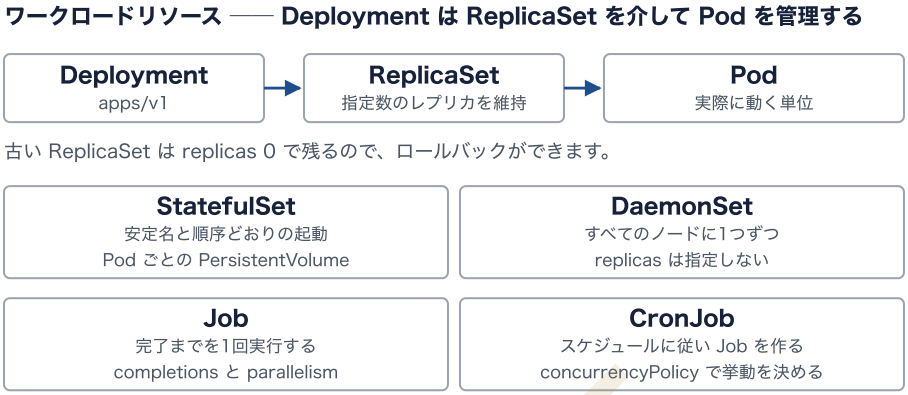


図 1-3　ワークロードリソース —— Deployment は ReplicaSet を介して Pod を管理する

1-1-6 コントローラと調整ループ（reconciliation loop）[1.1]

Kubernetes の自己修復（self-healing）は、コントローラ（controller）という仕組みで実現されています。コントローラは、次の3ステップを延々と繰り返すループです。

1. Observe（観測）—— API server から、担当するオブジェクトの現在の状態を読む
2. Diff（差分検出）—— spec に書かれた望ましい状態と、status の実際の状態を比較する
3. Act（実行）—— 差分を埋めるための操作を API server に対して行う

このループを調整ループ（reconciliation loop）または制御ループ（control loop）と呼びます。家庭のサーモスタットに例えられます。設定温度が spec、室温が status、エアコンを動かすのが act です。

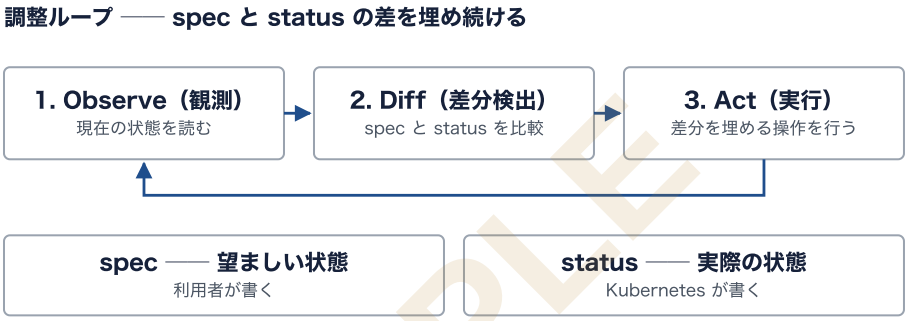
具体例で追います。ReplicaSet の spec に replicas: 3 と書いてあり、実際には Pod が2個しかないとします。ReplicaSet コントローラはこの差分を検出し、Pod を1個追加で作成します。ノード障害で Pod が消えても、次のループで再び差分が検出され、Pod が作られます。人間が「再起動せよ」と命令したわけではありません。これが宣言的モデルの利点です。

kube-controller-manager は、多数のコントローラを1つのプロセスにまとめて動かしています。代表的なものを挙げます。

コントローラ	担当
Node controller	ノードのダウンを検知して通知・対処する
ReplicaSet controller	指定数の Pod レプリカを維持する
Deployment controller	ReplicaSet を作り、ローリングアップデートを進める
Job controller	Job に対応する Pod を作り、完了を追跡する
EndpointSlice controller	Service に対応する EndpointSlice を維持する
ServiceAccount controller	新しい Namespace に既定の ServiceAccount を作る
Namespace controller	削除された Namespace の中身を掃除する

この考え方の応用が Operator パターンです。Operator は、独自のオブジェクト（CustomResource）に対する独自のコントローラを書くことで、データベースのバックアップやフェイルオーバーといった運用知識をソフトウェアに埋め込む手法です。Operator は Kubernetes 本体の機能ではなく、Kubernetes の拡張の仕方の名前です。

よくある取り違えとして、「コントローラは障害を検知したときだけ動く」というものがあります。実際には、障害の有無にかかわらず常時ループしています。また「調整ループは kubelet が回している」も誤りです。kubelet はノード上の Pod について自分の責務を果たしますが、ReplicaSet の replicas を保つのはコントロールプレーン側の仕事です。



コントローラは障害の有無にかかわらず常時ループしています。
ReplicaSet の replicas を保つのはコントロールプレーン側の仕事で、kubelet ではありません。

図 1-4　調整ループ —— spec と status の差を埋め続ける

1-1-7 Namespace・label・selector・annotation [1.1]

Namespace は、1つのクラスタの中を論理的に分割する仕組みです。同じ Namespace の中ではオブジェクト名が一意である必要がありますが、Namespace が違えば同じ名前を使えます。開発チームごと、環境ごと（dev / staging / prod）、テナントごとの区切りに使います。

クラスタ作成時に既定で存在する Namespace は4つです。

Namespace	用途
default	Namespace を指定しなかったときの既定の置き場

Namespace	用途
kube-system	Kubernetes 本体が作るオブジェクト（CoreDNS、kube-proxy など）
kube-public	全ユーザから読める。クラスタ情報の公開に使う
kube-node-lease	ノードのハートビート（Lease オブジェクト）用

重要なのは、すべてのオブジェクトが Namespace に属するわけではないことです。Namespace に属さない（クラスタスコープの）オブジェクトの代表は次のとおりです。

- Node
- PersistentVolume
- StorageClass
- ClusterRole、ClusterRoleBinding
- Namespace 自身
- CustomResourceDefinition
- IngressClass

一方、Pod、Service、Deployment、ConfigMap、Secret、PersistentVolumeClaim、Role、RoleBinding、ServiceAccount、NetworkPolicy は Namespace に属します。「PersistentVolumeClaim はクラスタスコープである」は誤りです。PV はクラスタスコープ、PVC は Namespace スコープ、という対比で覚えてください。

なお Namespace は論理的な区切りであって、既定ではネットワークを遮断しません。Namespace が違っても Pod 同士は通信できます。遮断したいなら

NetworkPolicy が必要です。この点は KCNA でよく狙われます。

label（ラベル）は、オブジェクトに付けるキーと値の組で、識別と選択に使います。annotation（アノテーション）も同じくキーと値の組ですが、用途が異なります。

観点	label	annotation
目的	選択（selector で絞り込む）	付随情報の記録
セレクタから使えるか	使える	使えない
典型例	app=frontend、env=prod、tier=backend	変更履歴、ビルドID、ツール向けの設定、連絡先
値の制約	短い識別子向け（63文字以内など）	構造化された長い文字列も可

selector（セレクタ）は、label を条件にオブジェクトを絞り込む仕組みです。Service がどの Pod へトラフィックを流すか、ReplicaSet がどの Pod を自分の管理下と見なすかは、すべて selector で決まります。等価ベース（app=frontend）と集合ベース（environment in (prod, staging)）の2種類があります。

ここが実務でも試験でも落とし穴です。Service の selector と Pod の label が一致していないと、Service は宛先を1つも見つけられず、接続が失敗します。第2章の Troubleshooting でも再登場する典型的な原因です。

1-1-8 拡張性 —— CRD、Operator、アドオン [1.1]

Kubernetes は、本体を書き換えずに機能を足せるように設計されています。KCNA では、どの拡張点が何を担うかが問われます。

CustomResourceDefinition（CRD）は、Kubernetes API に独自のオブジェクト種別を追加する仕組みです。CRD を登録すると、たとえば kind: Database というオブジェクトを kubectl で作れるようになります。ただし CRD はデータの置き場を増やすだけで、それだけでは何も起きません。そのオブジェクトを見て動くカスタムコントローラを併せて用意して、はじめて意味を持ちます。この「CRD＋カスタムコントローラ」の組み合わせが Operator です。

そのほかの拡張点を整理します。

拡張点	何を差し替えるか	代表例
CRI（Container Runtime Interface）	コンテナランタイム	containerd、CRI-O
CNI（Container Network Interface）	Pod ネットワーク	Calico、Cilium、Flannel
CSI（Container Storage Interface）	ストレージドライバ	各クラウドのブロックストレージ、Ceph
Admission Webhook	API リクエストの検証・書き換え	OPA/Gatekeeper、Kyverno
CRD + コントローラ	API そのものの拡張	Prometheus Operator、cert-manager
Scheduling Framework	スケジューラのプラグイン	独自の配置ポリシー
Device Plugin	GPU などの特殊デバイス	NVIDIA device plugin

3つのインタフェース CRI・CNI・CSI の対応は、そのまま出題されます。C の後ろの文字で覚えます。R は Runtime（実行）、N は Network（通信）、S は Storage（保存）です。

Admission Webhook には2種類あります。Mutating（変更を加える。sidecar の自動注入、既定値の付与）と Validating（検証だけ行い、条件を満たさなければ拒否する。必須ラベルの強制、特権コンテナの禁止）です。順番は Mutating が先、Validating が後です。

1-2 Administration（クラスタとワークロードの運用管理）[1.1]

Administration は、クラスタを構築し、日々運用し、資源を割り当て、拡大縮小する作業を扱う小項目です。KCNA は Associate レベルなので、実際の運用手順そのものより「どの道具で何ができるか」「誰の権限で何が起きるか」の理解が問われます。

1-2-1 kubectl の基本 —— 宣言的管理と命令的管理 [1.1]

kubectl は、Kubernetes API を叩くためのコマンドラインツールです。接続先クラスタ、認証情報、既定の Namespace は kubeconfig ファイル（既定で ~/.kube/config）に記録されます。kubeconfig は cluster（接続先）、user（認証情報）、context（cluster と user と namespace の組）の3要素で構成されます。

管理のスタイルは3つに分類されます。

スタイル	コマンド例	特徴
命令的コマンド	kubectl create deployment web --image=nginx	手早い。履歴が残らない。学習や検証向け
命令的オブジェクト設定	kubectl create -f pod.yaml、kubectl replace -f pod.yaml	ファイルは残るが、差分の扱いが荒い

スタイル	コマンド例	特徴
宣言的オブジェクト設定	kubectl apply -f ./configs/	望ましい状態を宣言する。本番と GitOps の標準

kubectl apply が本番で推奨されるのは、ファイルに書かれた望ましい状態と現在の状態の差分を Kubernetes に計算させる方式だからです。同じファイルを何度適用しても結果が同じ（冪等）になります。create は既にオブジェクトが存在するとエラーになりますが、apply は更新として扱われます。

KCNA で名前を覚えておきたい主要サブコマンドを挙げます。

コマンド	用途
kubectl get	オブジェクトの一覧。-o wide、-o yaml、-o json で出力形式を変える
kubectl describe	1オブジェクトの詳細とイベント。トラブルシュートの起点
kubectl logs	コンテナの標準出力。--previous で直前のクラッシュ時のログ
kubectl exec	動いているコンテナ内でコマンドを実行する
kubectl apply	宣言的に作成・更新する
kubectl delete	削除する
kubectl scale	replica 数を変更する
kubectl rollout	ローリングアップデートの状態確認、履歴、ロールバック
kubectl port-forward	ローカルのポートを Pod へ転送する。デバッグ用

コマンド	用途
kubectl explain	オブジェクトのフィールド定義を調べる
kubectl api-resources	利用可能なリソース種別と、Namespace スコープかどうかを一覧する
kubectl top	ノードや Pod のリソース使用量。Metrics Server が必要
kubectl cordon / drain / uncordon	ノードのメンテナンス操作
kubectl debug	エフェメラルコンテナを使ったデバッグ

注意点として、kubectl top は Metrics Server というアドオンが導入されていないと動きません。「kubectl top は標準で常に使える」は誤りです。

1-2-2 クラスタの構築形態とディストリビューション [1.1]

クラスタをどう用意するかは、責任分界点の違いとして理解します。

形態	例	コントロールプレーンの管理	特徴
マネージドサービス	クラウド事業者のマネージド Kubernetes	事業者	運用負荷が小さい。バージョンや設定の自由度は制限される
セルフマネージド（自前構築）	kubeadm で構築	利用者	自由度が高い。アップグレードや etcd のバックアップも自分の責任
ローカル学習環境	minikube、kind、k3d	利用者（単一ホスト）	開発と学習用。本番には使わない

形態	例	コントロールプレーンの管理	特徴
軽量ディストリビューション	K3s、MicroK8s	利用者	エッジやIoT、リソースの限られた環境向け

kubeadm は、準拠したクラスタを立ち上げるための公式ツールです。kubeadm init でコントロールプレーンを初期化し、kubeadm join でノードを参加させます。ただし kubeadm は CNI プラグインを入れてくれません。CNI を導入するまで、ノードは NotReady のままで、CoreDNS の Pod も Pending のままです。これは頻出の落とし穴です。

kind（Kubernetes IN Docker）は、Docker コンテナをノードに見立ててクラスタを作るツールで、CI での検証によく使われます。minikube は仮想マシンやコンテナ上に単一ノードのクラスタを作ります。

CNCF は Certified Kubernetes Conformance Program を運営しており、適合テストに合格したディストリビューションだけが「Certified Kubernetes」を名乗れます。これは、どの事業者の Kubernetes でも同じ API が使えることを保証する仕組みで、移植性（portability）を支えています。KCNA では、この適合プログラムがベンダーロックインを避けるための仕組みだという趣旨で出題されます。

1-2-3 ノードのライフサイクルとメンテナンス [1.1]

ノードは Node オブジェクトとして API に登録されます。ノードの状態は conditions で表現され、Ready が True なら Pod を受け入れられます。ほかに MemoryPressure、DiskPressure、PIDPressure があります。

ノードをメンテナンスするときの手順は3段階で、この語彙は必ず覚えます。

コマンド	動作
kubectl cordon	ノードをスケジュール不可（SchedulingDisabled）にする。既存の Pod はそのまま
kubectl drain	ノード上の Pod を退去させる。内部で cordon も行う。DaemonSet の Pod は既定で対象外
kubectl uncordon	スケジュール可能に戻す

drain は API-initiated eviction（API 起点の退去）を使うので、PodDisruptionBudget が尊重されます。PodDisruptionBudget（PDB）は「同時に何個まで落としてよいか」を宣言するオブジェクトで、minAvailable または maxUnavailable で指定します。PDB があると、drain は可用性を壊さない範囲で少しずつ Pod を退去させます。

障害の種類は2つに分けて覚えます。

種類	内容	例
Voluntary disruption（自発的中断）	人やツールが意図して起こす	ノードのアップグレード、drain、Deployment の更新
Involuntary disruption（非自発的中断）	避けられない事由	ハードウェア障害、カーネルパニック、ノードのリソース枯渇

PodDisruptionBudget が守るのは自発的中断だけです。「PDB があればハードウェア障害でも Pod は減らない」は誤りです。

クラスタのアップグレードは、コントロールプレーンを先に、ノードを後に行うのが原則です。kubelet のバージョンは kube-apiserver より新しくてはいけ

ません。バージョンスキューのポリシーとして、kubelet は API server より最大3マイナーバージョンまで古くてよい、と定められています。

1-2-4 リソース管理 —— requests、limits、ResourceQuota、LimitRange [1.1]

コンテナごとに、CPU とメモリの requests と limits を指定できます。この2つの違いは KCNA の頻出項目です。

項目	requests	limits
意味	このコンテナが最低限必要とする量	このコンテナが超えてはいけない上限
誰が使うか	kube-scheduler が配置先を決める計算に使う	kubelet とコンテナランタイムが実行時に強制する
超えたら	概念上「超える」という状態が無い	CPU はスロットリング、メモリは OOMKilled

CPU の単位は「1」が1コア（1 vCPU）相当で、500m は 0.5 コアを意味します。メモリの単位はバイトで、Mi（メビバイト）や Gi（ギビバイト）を使います。ここで、CPU は圧縮可能（compressible）な資源、メモリは圧縮不可能（incompressible）な資源、という区別が重要です。CPU が上限に達すると処理が遅くなるだけですが、メモリが上限に達するとコンテナは OOMKilled で強制終了されます。

requests と limits の組み合わせで、Pod に QoS クラス（Quality of Service class）が自動的に割り当てられます。

QoS クラス	条件	退去されやすさ
Guaranteed	すべてのコンテナで requests と limits が指定され、かつ両者が等しい	最も退去されにくい
Burstable	requests は指定されているが、limits と一致しない、または一部だけ指定	中間
BestEffort	requests も limits も一切指定されていない	最初に退去される

ノードのリソースが逼迫して node-pressure eviction が起きるとき、BestEffort から順に退去されます。「Guaranteed の Pod は絶対に退去されない」は言い過ぎで、誤りです。

Namespace 単位で資源を制御する仕組みが2つあります。

オブジェクト	効果
ResourceQuota	Namespace 全体の合計に上限を設ける。CPU、メモリの合計に加え、Pod 数、Service 数、PVC 数などのオブジェクト数も制限できる
LimitRange	個々のコンテナや Pod に対する既定値と最小値・最大値を設ける。requests を書き忘れた場合に既定値を注入することもできる

ここも取り違えが起きます。ResourceQuota は Namespace の合計、LimitRange は1つあたり、という対比で覚えてください。なお、ResourceQuota で CPU やメモリの制限を設定した Namespace では、requests や limits を書いていない Pod の作成が拒否されることがあります。

1-2-5 設定の外部化 —— ConfigMap と Secret [1.1]

アプリケーションの設定をコンテナイメージに焼き込むと、環境ごとにイメージを作り直すことになります。Kubernetes は設定をイメージの外に置く仕組みとして ConfigMap と Secret を用意しています。

観点	ConfigMap	Secret
用途	機密でない設定値	パスワード、APIキー、トークン、TLS証明書
保存形式	プレーンテキスト	base64 エンコードされた値（暗号化ではない）
既定のサイズ上限	1 MiB	1 MiB
追加でできる保護	特に無し	etcd の保存時暗号化、RBAC による参照制限

ここが最頻出の取り違えです。base64 は符号化であって暗号化ではありません。Secret の値は誰でもデコードできます。「Secret は暗号化されているので安全だ」は誤りです。実際に保護するには、etcd の encryption at rest を有効にし、RBAC で Secret の get 権限を絞り、外部のシークレット管理サービスと連携させます。

Pod への渡し方は3つあります。

方法	特徴
環境変数（env、envFrom）	簡単。ただし Pod 起動後に値を変えても反映されない
ボリュームとしてマウント	ファイルとして見える。ConfigMap の更新が（やがて）ファイルに反映される
コマンドライン引数	環境変数を経由して args に埋め込む

Secret のタイプにはいくつか決まったものがあります。Opaque（既定の任意データ）、kubernetes.io/dockerconfigjson（プライベートレジストリの認証情報。Pod の imagePullSecrets で参照する）、kubernetes.io/tls（TLS証明書と鍵。Ingress で使う）、kubernetes.io/service-account-token（ServiceAccount のトークン）などです。

Secret も ConfigMap も Namespace スコープです。別 Namespace の Secret を Pod から参照することはできません。

1-2-6 高可用性、バックアップ、etcd [1.1]

本番クラスタでは、コントロールプレーンを3台以上の奇数台で構成します。etcd は Raft という合意アルゴリズムを使うため、過半数（quorum）が生きている必要があるからです。3台なら1台の障害に耐え、5台なら2台の障害に耐えます。2台構成は、1台落ちると過半数を失うので、1台構成より脆くなります。

etcd の位置づけを整理します。

- クラスタのすべての API オブジェクトの唯一の保存先です
- アクセスするのは kube-apiserver だけです
- etcd を失うとクラスタの状態をすべて失います。したがってバックアップの対象は第一に etcd です
- etcd は CNCF の Graduated プロジェクトで、Kubernetes 専用ではなく汎用の分散キーバリューストアです

バックアップの考え方は2通りあります。etcd のスナップショットを取る方法と、すべてのマニフェストを Git に置いて再適用できるようにする方法です。後者は第3章の GitOps につながります。宣言的マニフェストが Git にあれば、クラスタを作り直しても同じ状態を再現できます。これがクラウドネイティブでの「再構築可能性」の考え方です。

なお、PersistentVolume に入っている実データは etcd のバックアップには含まれません。etcd に入るのは PV や PVC というオブジェクトの定義だけで、中身のデータはストレージ側にあります。Velero のようなツールが、オブジェクトとボリュームスナップショットを併せてバックアップする役割を担います。

1-2-7 オートスケーリング（HPA、VPA、Cluster Autoscaler、KEDA）[1.1]

autoscaling（オートスケーリング）は、負荷に応じて自動的に規模を変える仕組みです。次元が3つあることを、まず押さえてください。

次元	何を増減するか	主な仕組み
水平（horizontal）	Pod のレプリカ数	HorizontalPodAutoscaler（HPA）、KEDA
垂直（vertical）	1 Pod あたりの CPU とメモリ	VerticalPodAutoscaler（VPA）
クラスタ	ノードの台数	Cluster Autoscaler / Node Autoscaler

HorizontalPodAutoscaler は Kubernetes に標準で組み込まれた API リソースとコントローラです。CPU 使用率、メモリ使用率、あるいはカスタムメトリクスを見て、Deployment や StatefulSet の replicas を増減させます。動作には Metrics Server が必要です。

VerticalPodAutoscaler は Kubernetes 本体には含まれず、アドオンとして別途導入します。Pod の requests と limits を実際の使用量に合わせて調整します。HPA と VPA を同じメトリクス（たとえば CPU）に対して同時に使うのは推奨されません。互いの判断が衝突するためです。

Cluster Autoscaler は、Pod が資源不足で Pending のままになったときにノードを追加し、使われていないノードを削除します。ノードを増やすにはクラウドプロバイダ側のAPIが必要なので、環境に依存します。

KEDA（Kubernetes Event-driven Autoscaling）は CNCF の Graduated プロジェクトで、イベント駆動のスケーリングを提供します。メッセージキューの滞留数、データベースの行数、Kafka のラグ、cron のスケジュールなど、多数のスケーラ（scaler）に対応します。CPU 使用率では捉えられない負荷に反応でき、レプリカ数を 0 まで落とす scale to zero もできます。これは HPA 単体ではできないことで、KCNA で問われる差分です。

場面で整理します。ある動画配信サービスで、次の割り当てが妥当です。

- 昼のアクセス増でWebフロントの Pod を増やしたい → HorizontalPodAutoscaler
- 変換処理のジョブがメモリ不足で頻繁に OOMKilled になる。適切な requests が分からない → VerticalPodAutoscaler
- Pod は増えたがノードの空きが無く Pending が続く → Cluster Autoscaler
- 変換キューに溜まったメッセージ数に応じてワーカーを増やし、空なら 0 にしたい → KEDA

1-3 Scheduling（スケジューリング）[1.1]

Scheduling は、Pod をどのノードに置くかを決める仕組みを扱う小項目です。決めるのは kube-scheduler ですが、その判断に影響を与える手段が複数あり、それぞれの守備範囲が問われます。

1-3-1 kube-scheduler の動作 —— フィルタリングとスコアリング [1.1]

kube-scheduler は、まだノードに割り当てられていない（spec.nodeName が空の）Pod を監視し、割り当て先を決めます。判断は2段階です。

1. Filtering（フィルタリング / 述語）—— その Pod を置ける候補ノードだけを残します。資源が足りるか、nodeSelector に合うか、taint を許容できるか、必要なボリュームを接続できるか、ポートが空いているか、などを見ます
2. Scoring（スコアリング / 優先度）—— 残った候補に点数を付け、最も高い1台を選びます。資源の空き具合、イメージが既にあるか、topology spread の均等さ、affinity の優先条件などが加点対象です

選ばれたら、scheduler は API server に対して binding を作り、Pod の spec.nodeName が設定されます。あとはそのノードの kubelet が Pod を起動します。この流れの中で、scheduler が直接コンテナを起動することは一度もありません。

候補ノードが1つも残らなかった場合、Pod は Pending のままになり、イベントに「0/5 nodes are available」のような理由が記録されます。KCNA では、Pending の代表的な原因を答えさせる形で出ます。

Pending の原因	確認の手がかり
CPU やメモリの requests に見合う空きが無い	Insufficient cpu / Insufficient memory
taint を許容する toleration が無い	node(s) had untolerated taint
nodeSelector / node affinity に合うノードが無い	didn't match Pod's node affinity/selector

Pending の原因	確認の手がかり
PersistentVolumeClaim が Bound されていない	pod has unbound immediate PersistentVolumeClaims
topology spread 制約を満たせない	didn't match pod topology spread constraints

なお、スケジューラは Pod 単位で判断します。1つの Pod のコンテナ群が別々のノードに散ることはありません。「Pod 内のコンテナは負荷に応じて別ノードへ分散される」は誤りです。

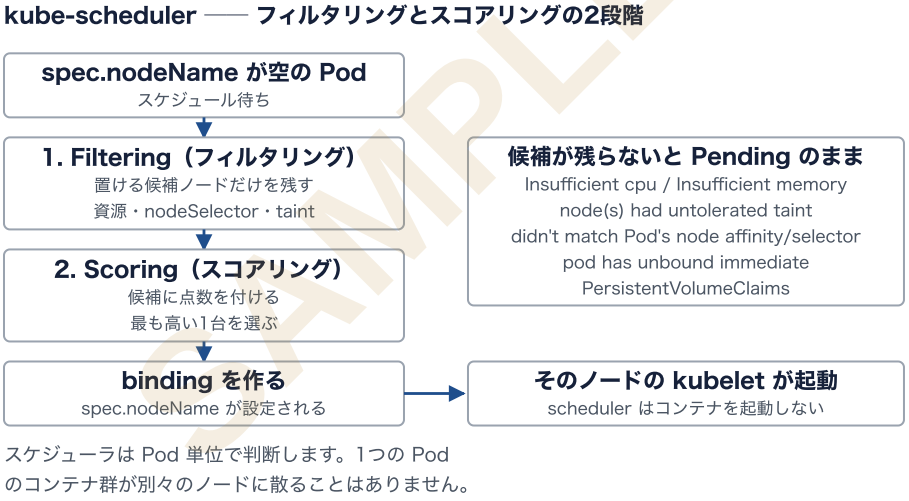


図 1-5　kube-scheduler —— フィルタリングとスコアリングの2段階

1-3-2 nodeSelector と Node Affinity [1.1]

配置先を指定する最も単純な方法が nodeSelector です。Pod の spec にキーと値を書き、そのラベルを持つノードにだけ置きます。たとえば disktype: ssd というラベルの付いたノードにだけ置く、という指定です。条件は完全一致のみで、否定や範囲は書けません。

Node Affinity は nodeSelector の表現力を拡張したもので、2つの強さがあります。

種別	意味
requiredDuringSchedulingIgnoredDuringExecution	必須条件。満たすノードが無ければ Pod は Pending のまま
preferredDuringSchedulingIgnoredDuringExecution	優先条件。満たすノードが無くても、別のノードに置かれる

名前の後半 IgnoredDuringExecution は「一度スケジュールされた後にノードのラベルが変わっても、既存の Pod は追い出さない」という意味です。ここは名前から意味を答えさせる問題になりやすい箇所です。

Node Affinity では In、NotIn、Exists、DoesNotExist、Gt、Lt といった演算子が使え、「zone が a か b のどちらか」「GPU ラベルを持たないノード」といった条件が書けます。

Pod Affinity と Pod Anti-Affinity は、ノードのラベルではなく「既にそのノード（あるいはゾーン）に居る Pod」を条件にします。

- Pod Affinity —— 特定のラベルを持つ Pod と同じ場所に置きたい。例: キャッシュと同じノードに置いてレイテンシを下げる
- Pod Anti-Affinity —— 特定のラベルを持つ Pod と別の場所に置きたい。例: 同じアプリのレプリカを同一ノードに固めない

topologyKey で「同じ場所」の粒度を決めます。kubernetes.io/hostname ならノード単位、topology.kubernetes.io/zone ならゾーン単位です。

1-3-3 Taints と Tolerations [1.1]

Taints（テイント）と Tolerations（トレランス）は、Node Affinity と逆向きの仕組みです。Node Affinity が「Pod がノードを選ぶ」のに対し、taint は「ノードが Pod を拒む」仕組みです。この方向の違いが最頻出の対比です。

taint はノードに付けます。key、value、effect の3要素で、effect は3種類です。

effect	動作
NoSchedule	toleration を持たない Pod は新たにスケジュールされない。既存の Pod はそのまま
PreferNoSchedule	できれば置かない。ほかに置けなければ置く（ソフト版）
NoExecute	toleration を持たない Pod は既存のものも退去させられる

toleration は Pod に付けます。ノードの taint に対応する toleration を持つ Pod だけが、そのノードに置かれます。

重要なのは、toleration は「置ける」ようにするだけで、「必ずそこに置く」わけではないという点です。GPU ノードに taint を付け、GPU を使う Pod に toleration を付けても、その Pod が普通のノードに置かれる可能性は残ります。確実に GPU ノードへ寄せたいなら、toleration に加えて nodeSelector や Node Affinity を併用します。この組み合わせが KCNA の応用問題になります。

実際のクラスタでは、コントロールプレーンのノードに node-role.kubernetes.io/control-plane:NoSchedule という taint が付いています。だから通常のワークロードはコントロールプレーンに置かれません。一方、

DaemonSet が配る監視エージェントは、この taint を許容する toleration を持つことでコントロールプレーンにも配置されます。

また、ノードに問題が起きると Kubernetes が自動的に taint を付けます。node.kubernetes.io/not-ready 、 node.kubernetes.io/unreachable 、 node.kubernetes.io/memory-pressure 、 node.kubernetes.io/disk-pressure などです。ノードが到達不能になったとき Pod が別ノードへ移るのは、この NoExecute taint の働きです。

1-3-4 Pod Topology Spread Constraints [1.1]

Pod Topology Spread Constraints（Pod トポロジー分散制約）は、Pod を障害ドメインにまたがって均等に散らすための仕組みです。ゾーンやノードといった topologyKey ごとに、Pod 数の偏りを maxSkew 以下に抑えます。

主なフィールドは4つです。

フィールド	意味
maxSkew	許容する偏りの最大値。1なら「最も多い場所と最も少ない場所の差が1まで」
topologyKey	分散の単位となるノードラベル（ゾーン、ノード、リージョン）
whenUnsatisfiable	制約を満たせないときの動作。DoNotSchedule（置かない）か ScheduleAnyway（置く）
labelSelector	どの Pod 群を数えるか

Pod Anti-Affinity との違いを対比します。Anti-Affinity は「同じ場所に置くな」という二値の指定で、レプリカ数がノード数を超えると Pending になりが

ちです。Topology Spread は「差を maxSkew 以下に保て」という度合いの指定なので、より柔軟に均し込めます。

1-3-5 PriorityClass と Preemption [1.1]

PriorityClass はクラスタスコープのオブジェクトで、value という整数の優先度を定義します。Pod は priorityClassName でこれを参照します。値が大きいほど優先度が高いという規則です。

preemption（プリエンプション）は、優先度の高い Pod が Pending になったとき、スケジューラが優先度の低い Pod を退去させて場所を空ける動作です。退去させられた Pod は、条件が整えば別のノードで再スケジュールされます。

- globalDefault: true を持つ PriorityClass は、priorityClassName を書かなかった Pod の既定値になります。クラスタに1つだけ設定できます
- preemptionPolicy: Never を指定すると、その Pod は優先的にキューへ並びますが、他の Pod を追い出しません
- system-cluster-critical と system-node-critical という組み込みの PriorityClass があり、クラスタ運用に不可欠なコンポーネントに使われます

QoS クラスとの違いを混同しないでください。PriorityClass はスケジューリング時の順番と preemption に効きます。QoS クラスはノードの資源が逼迫したときの node-pressure eviction の順序に効きます。効く場面が別です。

1-3-6 Eviction —— node-pressure と API-initiated [1.1]

eviction（退去）には2系統あります。

種別	起こす主体	きっかけ	特徴
node-pressure eviction	kubelet	ノードのメモリ、ディスク、inode、PID の逼迫	PodDisruptionBudget を尊重しない。QoS クラスと優先度の順で選ばれる
API-initiated eviction	利用者やツール（kubectl drain など）	メンテナンス	Eviction API を使う。PodDisruptionBudget を尊重する

node-pressure eviction では、kubelet が閾値（eviction threshold）を監視します。soft threshold は猶予期間を伴い、hard threshold は即座に退去させます。退去対象は、まず BestEffort、次に Burstable のうち requests を超えて使っている Pod、最後に Guaranteed の順で選ばれます。

なお、メモリ不足によるコンテナ単位の OOMKilled と、ノード全体の逼迫による Pod の eviction は別の出来事です。前者は limits を超えた1コンテナがカーネルに殺されるもので、後者はノードを守るために kubelet が Pod を退去させるものです。

1-3-7 手動配置とスケジューラの拡張 [1.1]

スケジューラを介さない配置方法もあります。

- spec.nodeName を直接指定すると、スケジューラを飛ばして特定ノードに固定されます。ノードが存在しなかったり資源が足りなかったりしても、そのまま失敗します。実務ではほぼ使いません
- static Pod は、ノードの kubelet が特定ディレクトリのマニフェストを直接読んで起動する Pod です。API server が動いていなくても起動できるた

め、kubeadm はコントロールプレーンのコンポーネント自身を static Pod として動かします

- DaemonSet の Pod は、DaemonSet コントローラがノードごとに Pod を作り、既定の scheduler が配置します

スケジューラ自体の拡張には Scheduling Framework があります。プラグイン方式で、フィルタやスコアの各拡張点に独自の処理を差し込めます。また、複数のスケジューラを同時に動かし、Pod の spec.schedulerName でどれを使うか指定することもできます。

さらに、配置後の偏りを是正する Descheduler というプロジェクトがあります。時間が経つと配置が偏るので、条件に合う Pod を退去させて再スケジュールを促します。Descheduler は Kubernetes 本体ではなく別プロジェクトです。

1-4 Containerization（コンテナ化）[1.1]

Containerization は、Kubernetes が動かす対象そのものを扱う小項目です。コンテナとは何か、イメージはどう作られるか、どの標準に従っているか、Kubernetes とコンテナランタイムはどう繋がっているかを扱います。

1-4-1 コンテナとは何か —— 仮想マシンとの違い [1.1]

コンテナは、アプリケーションとその実行に必要な依存関係（ライブラリ、設定ファイル、ランタイム）を1つのパッケージにまとめ、ホストOSのカーネルを共有しながら隔離された環境で実行する技術です。

仮想マシンとの違いを表で対比します。KCNA では必ず問われます。

観点	仮想マシン（VM）	コンテナ
隔離の単位	ハイパーバイザによる仮想ハードウェア	カーネル機能による名前空間の分離
ゲストOS	各VMに完全なOSがある	無い。ホストのカーネルを共有する
起動時間	数十秒から数分	数百ミリ秒から数秒
サイズ	ギガバイト単位	メガバイト単位
隔離の強さ	強い（カーネルも分離）	相対的に弱い（カーネルを共有）
密度	1ホストあたり少数	1ホストあたり多数

「コンテナは軽量な仮想マシンである」という言い方は入門的な比喩としては通じますが、試験の選択肢としては誤りです。コンテナはゲストOSを持たないという点が本質だからです。逆に「コンテナは仮想マシンより隔離が強い」も誤りです。カーネルを共有する分、隔離は弱くなります。強い隔離が必要な場面では、gVisor や Kata Containers のようなサンドボックス型ランタイムを使います。

Linux でこの隔離を実現している基盤技術は2つです。

技術	役割
namespaces（名前空間）	見えるものを分ける。PID、network、mount、UTS、IPC、user、cgroup の各名前空間
cgroups（コントロールグループ）	使える量を制限する。CPU、メモリ、ディスクI/O、PID数

Kubernetes の limits は、最終的にこの cgroups の設定として反映されます。namespaces と cgroups の役割分担は「見え方を分けるのが namespaces、量を絞るのが cgroups」と覚えます。なお、この Linux の namespaces は、Kubernetes の Namespace オブジェクトとは全く別物です。名前が同じなので、選択肢で引っかけに使われます。

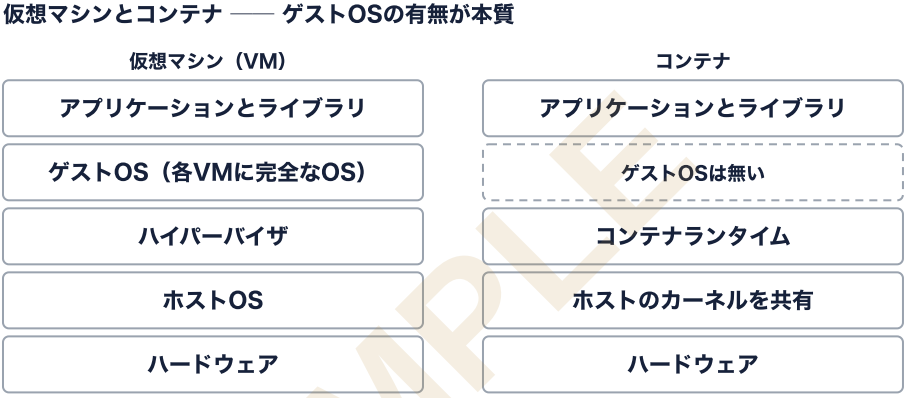


図 1-6　仮想マシンとコンテナ —— ゲストOSの有無が本質

コンテナの利点を、場面で言い直します。ある金融系の開発チームが「開発者のノートPCでは動くのに本番で動かない」問題を抱えているとします。コンテナは、アプリと依存関係を一緒に固めて、どこでも同じイメージを動かせるようにします。これが移植性（portability）です。加えて、イメージは変更されない（immutable）ので、同じイメージから何度でも同じ環境を再現できます。このimmutable infrastructure（不変のインフラ）という考え方は、第4章のクラウドネイティブ原則にもつながります。

1-4-2 コンテナイメージとレジストリ [1.1]

コンテナイメージは、読み取り専用のレイヤ（layer）を積み重ねた構造をしています。ベースイメージの上に、パッケージのインストール、アプリのコピーといった各手順が1つのレイヤになります。コンテナを起動すると、その上に書き込み可能なレイヤが1枚被さります。

この構造には次の効果があります。

- レイヤは複数のイメージで共有されるので、ディスクと転送量が節約されます
- 変更のあったレイヤだけを転送すればよいので、更新が速くなります
- 書き込み可能レイヤはコンテナの終了とともに消えます。永続化したいデータはボリュームに置く必要があります

イメージの指定は、レジストリ、リポジトリ、タグ（またはダイジェスト）で行います。たとえば registry.example.com/team/app:1.4.2 という形です。タグを省略すると latest が使われます。

指定方法	例	性質
タグ	app:1.4.2	人が読める。ただし同じタグで中身が差し替わりうる
ダイジェスト	app@sha256:abc…	内容のハッシュ。中身が一意に定まる（immutable）

本番で latest タグを使うのは避けるべきだ、というのが定番の出題ポイントです。理由は2つあります。第一に、いつ何が入ったか分からず再現性が無いこと。第二に、imagePullPolicy が絡むことです。

imagePullPolicy は3値です。

値	動作
Always	毎回レジストリから取得する
IfNotPresent	ノードに無ければ取得する
Never	取得しない。ノードに既にある前提

既定値は、タグが latest の場合または省略された場合は Always、それ以外のタグを明示した場合は IfNotPresent です。

レジストリ（registry）はイメージの保管庫です。公開レジストリと、企業内のプライベートレジストリがあります。プライベートレジストリから取得するには、認証情報を kubernetes.io/dockerconfigjson 型の Secret に入れ、Pod の imagePullSecrets で参照します。認証情報が無いと ImagePullBackOff で失敗します。CNCF プロジェクトの Harbor は、脆弱性スキャンや署名の機能を備えたプライベートレジストリの実装です。

1-4-3 OCI 標準 [1.1]

OCI（Open Container Initiative）は、コンテナの形式と実行方法を標準化するために設立された団体で、Linux Foundation の下にあります。仕様は3つです。

仕様	定めるもの
Image Specification（image-spec）	コンテナイメージの形式。レイヤ、マニフェスト、設定の構造
Runtime Specification（runtime-spec）	展開されたファイルシステムバンドルをどう実行するか
Distribution Specification（distribution-spec）	レジストリとの間でイメージをやり取りする API

OCI 標準があるおかげで、あるツールで作ったイメージを別のツールで実行でき、どのレジストリにも置けます。「Docker で作ったイメージは Docker でしか動かせない」は誤りです。Docker が作るイメージは OCI 準拠なので、containerd でも CRI-O でも動きます。

ここは KCNA が好むテーマである相互運用性（interoperability）とベンダーロックインの回避に直結します。CNCF の Certified Kubernetes 適合プログラムと同じ趣旨で、標準が移植性を担保しているという文脈で出題されます。

1-4-4 CRI とコンテナランタイム [1.1]

CRI（Container Runtime Interface）は、kubelet とコンテナランタイムの間の gRPC ベースのインタフェースです。CRI があることで、kubelet はランタイムの実装を知らずにコンテナを起動できます。

用語が階層になっているので、整理します。

層	名称	役割	例
上位	高レベルランタイム（CRI ランタイム）	kubelet からの CRI 呼び出しを受け、イメージの取得と管理、コンテナのライフサイクル管理を行う	containerd、CRI-O
下位	低レベルランタイム（OCI ランタイム）	runtime-spec に従って実際にプロセスを起動し、namespaces と cgroups を設定する	runc、crun、gVisor（runsc）、Kata Containers

呼び出しの流れは、kubelet → CRI → containerd → OCI ランタイム（runc）→ Linux カーネル、となります。

Docker との関係も試験に出ます。Kubernetes はかつて dockershim という仲介層で Docker Engine を直接使えるようにしていましたが、これは Kubernetes v1.24 で削除されました。これを Docker サポートの終了と呼ぶことがありますが、意味を正確に押さえてください。

- Docker で作ったイメージは、これまでどおり Kubernetes で動きます。OCI 準拠だからです
- 変わったのは、kubelet が Docker Engine を直接呼ぶ経路が無くなったことだけです
- Docker Engine 自体、内部では containerd を使っています

「dockershim の削除により Docker で作ったイメージが使えなくなった」は誤りです。この誤解を選ばせる問題は定番です。

サンドボックス型ランタイムも押さえます。gVisor はユーザ空間でシステムコールを代行することで、ホストカーネルへの接触面を減らします。Kata Containers は各コンテナを軽量な仮想マシンで包み、VM 並みの隔離を得ます。どちらもマルチテナント環境や信頼できないコードの実行に使われ、Kubernetes では RuntimeClass オブジェクトで Pod ごとに選択します。

1-4-5 イメージのビルドとベストプラクティス [1.1]

イメージのビルドは Dockerfile（Containerfile）に手順を書いて行うのが一般的です。ビルド用のツールには Docker のほか、BuildKit、Buildah、Kaniko、CNCF の Buildpacks などがあります。Kaniko や Buildah は特権を必要とせずに Kubernetes クラスタ内でイメージを作れるため、CI パイプラインで使われます。

KCNA で問われるベストプラクティスを挙げます。

手法	目的
マルチステージビルド	ビルド用の重い環境（コンパイラ、SDK）を最終イメージに含めない。サイズと攻撃面の削減
最小ベースイメージ（distroless、Alpine、scratch）	シェルやパッケージマネージャを含めず、脆弱性の対象を減らす
非rootユーザで実行	侵害されたときの影響を抑える。USER 命令、または securityContext の runAsNonRoot
レイヤ数を減らす、順序を工夫する	キャッシュを効かせてビルドを速くする
.dockerignore	不要なファイルをビルドコンテキストに含めない
機密情報を焼き込まない	イメージのレイヤに残る。ConfigMap と Secret で外から渡す
タグをバージョンで固定する	再現性の確保。latest を避ける
1コンテナ1プロセス	責務を分け、ログとライフサイクルを単純にする

イメージの中に認証情報を入れてはいけない理由は、レイヤに履歴として残るからです。後のレイヤで削除しても、前のレイヤから取り出せます。これは頻出の引っかけです。

イメージの安全性を確かめる手段も併せて覚えます。脆弱性スキャン（Trivy、Clair など）、イメージの署名と検証（Notary、Sigstore の cosign）、SBOM（Software Bill of Materials、ソフトウェア部品表）の生成です。CNCF のプロ

ジェクトでは、Harbor がスキャンと署名の統合を、in-toto と The Update Framework（TUF）がサプライチェーンの完全性を担います。

1-4-6 Pod 内のコンテナ設計パターン [1.1]

1つの Pod に複数のコンテナを入れるのは、切り離せない補助的な役割がある場合だけです。代表的なパターンは3つです。

パターン	内容	例
sidecar	主コンテナの機能を補強する	ログ収集、プロキシ、証明書更新、サービスメッシュのデータプレーン
ambassador	外部への接続を代理する	データベース接続のプロキシ、シャーディングの隠蔽
adapter	主コンテナの出力を標準形式に変換する	独自形式のメトリクスを Prometheus 形式に直す

これに init container を加えた4つが、KCNA で名前を問われる範囲です。

同一 Pod のコンテナは、ネットワーク名前空間を共有するため localhost で通信できます。ポート番号は Pod 内で重複できません。ボリュームを共有すれば、ファイル経由でのやりとりもできます。一方、別 Pod のコンテナとは Service や Pod IP を通じて通信します。

判断基準は「一緒にスケールするか」です。Web サーバとログ収集エージェントは常に1対1なので同じ Pod に入れます。Web サーバとデータベースは独立にスケールするので、別々の Pod（別々のワークロードリソース）にします。「フロントエンドとバックエンドを1つの Pod にまとめる」は誤った設計として選択肢に出ます。

1-4-7 コンテナのライフサイクルとプローブ [1.1]

kubelet は、コンテナの健全性を3種類のプローブ（probe）で確認します。この3つの区別は KCNA で頻出です。

プローブ	問い	失敗したときの動作
liveness probe	このコンテナは生きているか	コンテナを再起動する
readiness probe	このコンテナはトラフィックを受けられるか	Service の宛先（EndpointSlice）から外す。再起動はしない
startup probe	起動処理は完了したか	完了するまで liveness と readiness の判定を待たせる。失敗し続ければ再起動

readiness probe が失敗しても、コンテナは再起動されません。ここが最大の対比点です。起動に時間のかかるレガシーアプリで liveness probe が早すぎて再起動ループに陥る問題は、startup probe で解決します。

プローブの実装方法は4つです。httpGet（HTTPステータス200〜399なら成功）、tcpSocket（TCP接続できれば成功）、exec（コンテナ内でコマンドを実行し、終了コード0なら成功）、grpc（gRPC のヘルスチェック）です。

主なパラメータは、initialDelaySeconds（最初の実行までの待ち）、periodSeconds（実行間隔）、timeoutSeconds（タイムアウト）、failureThreshold（何回失敗したら失敗と見なすか）、successThreshold（何回成功したら成功と見なすか）です。

ライフサイクルフックとして、postStart（コンテナ起動直後）と preStop（終了直前）があります。Pod の終了は次の順で進みます。

1. Pod が Terminating になり、EndpointSlice から外される
2. preStop フックが実行される
3. コンテナに SIGTERM が送られる
4. terminationGracePeriodSeconds（既定30秒）待つ
5. まだ生きていれば SIGKILL で強制終了する

この猶予期間があるので、処理中のリクエストを捌き切ってから終了する（graceful shutdown、優雅な終了）ことができます。

図 1-7　3つのプローブと Pod 終了の順序

1-4-8 コンテナの状態と終了理由 [1.1]

コンテナの state は3種類で、Pod の phase とは別の概念です。

state	意味
Waiting	まだ動いていない。reason に ImagePullBackOff、CrashLoopBackOff、CreateContainerConfigError などが入る

state	意味
Running	実行中
Terminated	終了済み。exitCode と reason（Completed、Error、OOMKilled）が入る

KCNA で頻出の reason を、原因とともに整理します。

reason	何が起きているか	主な原因
ImagePullBackOff / ErrImagePull	イメージを取得できない	タグの綴り違い、レジストリの認証情報不足、レジストリに到達できない
CrashLoopBackOff	コンテナが起動しては落ちるのを繰り返している	アプリの設定不備、依存先に繋がらない、liveness probe が厳しすぎる
OOMKilled	メモリの limits を超えて強制終了された	limits が小さすぎる、アプリのメモリリーク
CreateContainerConfigError	起動前の設定解決に失敗	参照している ConfigMap や Secret が存在しない
Completed	正常終了した	Job では正常。Deployment ではアプリがすぐ終了している設計ミス

BackOff という語は、再試行の間隔を指数的に延ばす仕組みを指します。CrashLoopBackOff は「落ちている」状態そのものではなく「再起動を待っている」状態です。ここも選択肢の切り分けに使われます。

1-5 ドメイン1の総まとめ —— 対応表と誤答肢の切り方 [1.1]

このセクションは、Kubernetes Core Concepts、Administration、Scheduling、Containerization の4小項目を横断して、試験直前に見返すための整理です。新しい事実は増やさず、対応関係と切り分けだけを並べます。

1-5-1 コンポーネントと役割の対応表 [1.1]

問われ方	答え
クラスタの全データを保存するのは	etcd
すべての操作の唯一の入口は	kube-apiserver
Pod をノードに割り当てるのは	kube-scheduler
調整ループを回すのは	kube-controller-manager
クラウドのロードバランサを作るのは	cloud-controller-manager
ノード上で Pod を動かし続けるのは	kubelet
Service のネットワークルールを維持するのは	kube-proxy
コンテナを実際に起動するのは	コンテナランタイム（containerd など）
クラスタ内の名前解決をするのは	DNS アドオン（CoreDNS）
Pod 間の通信を成立させるのは	CNI プラグイン

覚え方として、コントロールプレーンは「決める側」、ノードコンポーネントは「動かす側」と分けます。決める側にあるのに動かす側と誤答させる、あるいはその逆の選択肢が典型です。

1-5-2 オブジェクトと用途の対応表 [1.1]

場面	使うオブジェクト
ステートレスなWebアプリを3台動かしたい	Deployment
データベースを固有名と専用ストレージで動かしたい	StatefulSet
全ノードにログ収集エージェントを1つずつ置きたい	DaemonSet
一度きりのバッチ処理を走らせたい	Job
毎晩定時に処理を走らせたい	CronJob
設定ファイルをイメージの外から渡したい	ConfigMap
パスワードを渡したい	Secret
クラスタ内を論理的に区切りたい	Namespace
Namespace の合計使用量を制限したい	ResourceQuota
コンテナ1つあたりの既定値と上限を決めたい	LimitRange
メンテナンス時に同時停止数を抑えたい	PodDisruptionBudget
負荷に応じて Pod 数を増減したい	HorizontalPodAutoscaler
Pod の優先度を決めたい	PriorityClass
独自のオブジェクト種別を足したい	CustomResourceDefinition

1-5-3 まぎらわしい対の一覧 [1.1]

対	決め手
spec と status	spec は利用者が書く望ましい状態、status は Kubernetes が書く実際の状態
宣言的と命令的	apply が宣言的、create や scale が命令的
Deployment と StatefulSet	Pod が交換可能か、固有かで分ける
ReplicaSet と Deployment	ReplicaSet は数を保つだけ。更新とロールバックは Deployment
requests と limits	requests はスケジューリングの計算、limits は実行時の強制
ResourceQuota と LimitRange	Namespace の合計か、1つあたりか
QoS クラスと PriorityClass	退去の順序か、スケジューリングの順序と preemption か
Node Affinity と Taints	Pod がノードを選ぶか、ノードが Pod を拒むか
nodeSelector と Node Affinity	完全一致だけか、演算子と優先条件が使えるか
Pod Anti-Affinity と Topology Spread	同居禁止の二値か、偏りの度合いの制御か
liveness probe と readiness probe	再起動するか、宛先から外すか
init container と sidecar	先に完了して終わるか、並走し続けるか
ConfigMap と Secret	機密でないか、機密か（ただし Secret も既定では暗号化されない）
CRI と CNI と CSI	実行と通信と保存
namespaces（Linux）と Namespace（Kubernetes）	見え方の分離か、API オブジェクトの論理区分か

対	決め手
node-pressure eviction と API-initiated eviction	kubelet が起こし PDB を無視するか、利用者が起こし PDB を守るか
Namespace スコープと クラスタスコープ	PVC・Role は Namespace、PV・StorageClass・ClusterRole はクラスタ

1-5-4 誤答肢に出やすい断定 [1.1]

次の記述は、いずれも誤りです。文の形ごと覚えておくと、選択肢を素早く落とせます。

- Kubernetes はコンテナイメージをビルドし、CI/CD パイプラインを提供する
- Kubernetes の最小のデプロイ単位はコンテナである
- Pod は再作成されても同じ名前と同じIPアドレスを保つ
- status フィールドは利用者がマニフェストに書く
- kube-scheduler がノード上のコンテナを直接起動する
- コントローラは障害が起きたときにだけ動く
- Namespace が違えば Pod 同士は通信できない
- PersistentVolumeClaim はクラスタスコープのオブジェクトである
- Secret は暗号化されているのでマニフェストに直接書いても安全である
- DaemonSet では replicas を指定して台数を決める
- Deployment は Pod を直接作成する
- toleration を付けた Pod は必ずその taint を持つノードに配置される
- PodDisruptionBudget があればハードウェア障害でも Pod 数は減らない

- Guaranteed の QoS クラスであれば決して退去されない
- readiness probe が失敗するとコンテナが再起動される
- コンテナは仮想マシンよりも強い隔離を提供する
- dockershim の削除により Docker で作成したイメージは動かなくなった
- kubectl top は追加のコンポーネントなしで常に利用できる
- HorizontalPodAutoscaler と VerticalPodAutoscaler は同じメトリクスで併用するのが推奨される
- コントロールプレーンが停止すると、稼働中の Pod は直ちにすべて停止する

1-5-5 ドメイン1で押さえる数字と既定値 [1.1]

数字は、根拠のあるものだけを覚えます。

項目	値
ドメイン1の配点	44%（KCNA の4ドメイン中で最大）
ConfigMap と Secret のサイズ上限（既定）	1 MiB
terminationGracePeriodSeconds の既定値	30秒
imagePullPolicy の既定値	タグが latest または省略なら Always、それ以外は IfNotPresent
restartPolicy の既定値	Always
コントロールプレーンの推奨構成	3台以上の奇数（etcd の過半数確保のため）
kubelet のバージョンスキュー	kube-apiserver より最大3マイナーバージョンまで古くてよい

項目	値
既定で存在する Namespace	default、kube-system、kube-public、kube-node-lease の4つ
CPU の 1 単位	1コア相当。500m は 0.5 コア

試験時間は90分、出題形式は多肢選択式（multiple choice）で、オンライン監督下で実施されます。認定の有効期間は2年です。ドメインごとの配点は、Kubernetes Fundamentals が44%、Container Orchestration が28%、Cloud Native Application Delivery が16%、Cloud Native Architecture が12%です。

サンプルはここまでです

続き(第2章〜巻末)は、KCNA-JP 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

KCNA-JP 学習用テキスト(無料サンプル)

版

2026年7月版(CNCF 公式 KCNA Exam Curriculum（2025-11-24 版・2026-09-07 取得）準拠)

発行

IT資格道場(https://www.shikaku-doujou.com)

お問い合わせ

info@shikaku-doujou.com