IT資格道場

LinuC レベル4 4SA01試験 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)

LPI-Japan公式 LinuC レベル4 システムアーキテクト 4SA01試験(LinuC-4SA Version 1.0) 出題範囲 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）。暗記用は本線に入れず、空き時間や用語の再確認に並走させてください。

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。

試験の基本情報

- 対象試験: LinuC レベル4 システムアーキテクト **4SA01試験**（LPI-Japan・LinuC-4SA Version 1.0）
- 出題範囲: 主題 4SA.01「システムアーキテクチャ」／4SA.02「ネットワークとストレージの選定」／4SA.03「可用性の設計」／4SA.04「性能・拡張性の設計」の **4主題・12副主題**
- 出題数: 公式の試験概要に **約40問／1試験** と示されています

- 出題形式: CBT。公式は「マウスによる選択方式がほとんど。キーボード入力問題も多少出題される場合がある。実技や面接はありません」としています。団体受験用にペーパーテスト（PBT）もあります
- 試験時間: **90分**。ただし試験後のアンケートに5分を要するため、公式は「試験問題を解く時間は実質 **85分**」としています
- 前提条件: 受験自体に実務経験や前提資格の条件はありません。ただし **LinuC レベル4 システムアーキテクトとして認定されるには、「有意なLinuCレベル2の認定」を持ち、かつ 4SA01・4SA02 の両試験に5年以内で合格する**必要があります（受験順は問いません）。レベル2認定を持たずに2試験へ合格した場合は、レベル2認定を取得した時点で認定されます
- 受験方法: ピアソンVUE経由のCBT（テストセンター受験、または自宅・職場からの OnVUE オンライン受験）
- 試験実施言語: 日本語・英語
- 合格ライン: **公表されていません**。本書では点数・正答率の目安を一切示していません
- 出題比重: 公式は副主題ごとの **「重要度」**（4SA01は合計40）だけを公表しており、「配点◯%」という比率は非公表です。公式は「重要度が高いトピックスほど、試験において多くの問題が出題されます」としています

重要度の合計40と、出題数「約40問」が一致することを覚えておくと便利です。**重要度4の副主題は「40問中およそ4問」**と読めるので、どこに時間を使うべきかがそのまま分かります。

主題	副主題	重要度	本書の章
4SA.01 システムアーキテクチャ	4SA.01.1 アーキテクチャの設計原則と主要パターン	4	第1章

主題	副主題	重要度	本書の章
	4SA.01.2 柔軟性を高めるアーキテクチャパターン	5	第2章
4SA.02 ネットワークとストレージの選定	4SA.02.1 拠点内のネットワーキング	3	第3章
	4SA.02.2 拠点間のネットワーキング	3	第4章
	4SA.02.3 ストレージとアクセスプロトコル	2	第5章
	4SA.02.4 集中型および分散型ストレージ	4	第6章
4SA.03 可用性の設計	4SA.03.1 フェイルオーバークラスタ	3	第7章
	4SA.03.2 負荷分散と障害の局所化	4	第8章
	4SA.03.3 データレプリケーションと災害対策	3	第9章
4SA.04 性能・拡張性の設計	4SA.04.1 性能見積もりと評価	2	第10章
	4SA.04.2 性能の改善	3	第11章
	4SA.04.3 性能の拡張	4	第12章

章立ては副主題そのままの全12章です。**章の厚みは重要度に比例させてあります**（重要度5の第2章がいちばん厚く、重要度2の第5章・第10章がいちばん薄い）。時間が足りないときは重要度の大きい章 —— 第2章（5）、第1章・第6章・第8章・第12章（4）—— から手を付けてください。

4SA01 の設問はどう問われるか

レベル4の 4SA01 は、**手を動かす試験ではなく「選ぶ試験」**です。副主題の到達目標に並ぶ動詞を見ると性格がはっきりします ——「比較できる」「選定できる」「導入可否判断ができる」「トレードオフに基づいて比較できる」。**手順の暗記ではなく、要件を読んで手段を決める力**が問われます。

特徴	内容
設問の立て方	要件（利用者数・許容停止時間・許容データ損失・帯域・拠点数・予算）を先に示し、**「最も適切な構成はどれか」**を選ばせる形が中心
誤り肢の作り	「その技術では実現できない」ではなく、**「別の技術ならできるが、これではできない」**。近接する技術の性質を持ち込む形が最頻
用語の扱い	単独の用語定義を聞く設問は少なく、**2つの技術の対比**（同期／非同期、オーケストレーション／コレオグラフィ、TCC／Saga など）として問われる
コマンド入力	出題範囲が具体的な OSS・設定を明示している副主題（拠点内ネットワーク・集中型／分散型ストレージ・フェイルオーバークラスタ・負荷分散・性能）で、名称やコマンド名を打たせる形が入る

本書はこの性格に合わせ、**比較表・使い分けの判断軸・典型的なひっかけ**を厚く取っています。コマンドや設定例は、出題範囲が名前を挙げているものに限って、理解を助ける範囲で載せています。

壊され方（ひっかけ）は7つに整理できる

各章末の「章の試験ポイント」では、この番号で型を示します。

型	仕掛け
① 近接概念ペアの入れ替え	対になる2つの中身を丸ごと入れ替える（オーケストレーション／コレオグラフィ、TCC／Saga、シック／シン、pull／push、垂直／水平シャーディングなど。**最頻**）
② 層の取り違え	別の層（L2／L3／L4／L7、ブロック／ファイル／オブジェクト）の仕組みを持ち込んで「これで解決する」と書く
③「量」と「配置」のすり替え	資源の**量**を増やす手段と、同じ量を**どこに置くか**を変える手段を入れ替える（サーバー増強 ⇔ CDN・キャッシュ・シャーディング）
④ 前提条件の付け替え	成立条件を別の技術の条件に差し替える（DSR の前提に NAT の条件を持ち込む、cLVM の前提に共有ストレージ以外を持ち込む）
⑤ 目標値の取り違え	RPO と RTO、レイテンシとスループット、可用性と整合性のように、**測っている軸が違うもの**を入れ替える
⑥ 効かない対処	症状は消えるが原因は残る対処を「解決」として書く（タイムアウトを延ばす、リトライ回数を増やす、キャッシュを増やす）

型	仕掛け
⑦ 存在しない機能・オプションの捏造	もっともらしいが実在しない指定・組み合わせを書く

⑦は特に注意してください。この試験の誤り肢は「日本語として自然で、やりたいことにも合っている」ように書かれます。**その組み合わせが実在するか**を思い出せるかが分かれ目になります。

設計の試験を解く「3つの問い」

要件を読んだら、次の3つを順に自問してください。本書の各章はこの3問に答えられる形で書いてあります。

1. **何を守りたいのか**（可用性か、整合性か、レイテンシか、スループットか、コストか）
2. **何を捨てられるのか**（データ損失を秒単位で許すのか、切替に人手を挟むのか、性能を落として待つのか）
3. **その手段は、捨てたものを本当に捨てているか**（同期レプリケーションを選びながら「遠隔地でも性能は落ちない」とする肢は、捨てたはずのものを捨てていない）

4SA01 でいちばん多い誤り肢は、「全部を同時に手に入れる」と書いてある肢です。分散システムでは整合性・可用性・分断耐性が同時には成立せず（第6章）、遠距離の同期レプリケーションは必ず待ち時間を持ち込み（第9章）、キャッシュは必ず古いデータを許容します（第12章）。**トレードオフが書かれていない肢を疑う** —— これが本書を通じた読み方です。

想定読者

LinuC レベル2 相当の Linux 運用実務（パッケージ管理、ネットワーク設定、ストレージとファイルシステム、サービス管理、ログ調査）を前提にしています。個々のコマンドの使い方はいちいち説明せず、**「複数台を組み合わせて1つのシステムにするとき、何をどう選ぶか」**に紙面を割いています。

各章は次の3点セットで構成しています。

1. **本文** …… 「そう決まっている」ではなく「なぜその設計になるのか」まで書いています
2. **混同しやすい対の概念** …… 入れ替えられやすい組を表にしてあります
3. **章の試験ポイント** …… 壊されやすい箇所と、その見分け方を並べています

それでは第1章から始めます。

第1章 アーキテクチャの設計原則と主要パターン（4SA.01.1）[1.1]

この章の到達目標は2つです。**複数ノード間の業務機能の分割・制御・連携集約の方式に着目してアーキテクチャパターンを比較できること**、そして**主要なアーキテクチャパターンの特徴・制約・適するユースケースを網羅的に理解し、設計時の判断材料にできること**です。重要度は4で、40問中およそ4問に相当します。

この章の設問は「用語の意味を答えさせる」形では出ません。**要件を読ませて、どのパターンを採るかを選ばせる**形です。だからこの章は、パターンを並べるだけでなく「**どの軸で分かれているのか**」を先に押さえます。

1-1 アーキテクチャを比較する4つの軸 [1.1]

複数ノードに分かれたシステムの設計は、次の4つの観点に分解できます。この4軸がそのまま本章の構成です。

軸	問い	選択肢
配置	処理をどこに置くか	集約型 ／ 分散型
分割	機能をどう切るか	layer 構造（役割で横に切る） ／ tier 構造（配置で縦に切る）
制御	誰が誰を呼ぶか	中央が指揮する ／ 各自が反応する ／ 仲介者を挟む
タイミング	呼び出しは待つか	同期 ／ 非同期

この4軸は独立しています。「分散型だから非同期」「マイクロサービスだからコレオグラフィ」ではありません。分散配置しながら同期呼び出しをすることも、集約型の中で非同期処理を使うこともできます。設問では**片方から他方を勝手に導く肢**がよく置かれます ―― 「分散構成なので結果整合になる」「イベント駆動なので中央の制御は存在しない」といった肢は、軸の混同です。

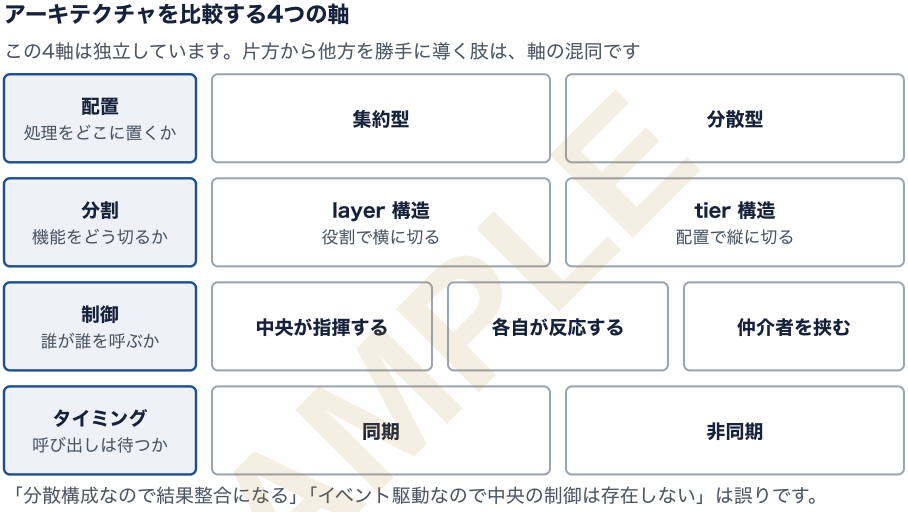

図 1-1　アーキテクチャを比較する4つの軸 ―― 4軸は独立している

設計原則としての「関心の分離」と「疎結合」[1.1]

どのパターンを選ぶ場合でも、判断のよりどころになるのは次の性質です。設問の選択肢はこの語彙で書かれます。

- **凝集度（cohesion）** …… 1つのコンポーネントの中身が、どれだけ1つの目的に集まっているか。**高いほどよい**
- **結合度（coupling）** …… コンポーネント同士がどれだけ相手の事情に依存しているか。**低い（疎結合）ほどよい**

- **関心の分離** …… 変更理由が違うものを同じ場所に置かない。「業務ルールが変わったとき」と「DB 製品を変えたとき」で直す場所が別になっているか
- **単一責任** …… 1つのコンポーネントが変更される理由は1つに保つ

疎結合が効くのは、**変更・障害・スケールの3つが伝播しないこと**です。逆に言えば、疎結合にすると**呼び出し1回ごとの遅延と、全体を見通す難しさ**が増えます。ここに必ずトレードオフがあります。「疎結合にすればレスポンスも速くなる」と書いてある肢は誤りです —— 疎結合化はネットワーク越しの往復を増やす方向に働きます。

1-2 配置 —— 集約型と分散型 [1.1]

集約型は、業務処理を1つのノード（あるいは1つの強く結合したノード群）に集めます。**分散型**は、複数ノードへ分けて配置します。

	集約型	分散型
処理の置き場	単一ノードに集約	複数ノードへ分散
性能の伸ばし方	**スケールアップ**（1台を強くする）が主	**スケールアウト**（台数を足す）が可能
上限	1台の CPU・メモリ・I/O が上限になる	台数で伸ばせるが、分割できない処理が上限になる
データ整合	同一プロセス・同一 DB 内で完結し、**トランザクションで一貫性を取りやすい**	ノードをまたぐため、分散トランザクションか結果整合の設計が要る
障害の影響	単一障害点になりやすい	一部の停止で全体が止まらない設計にできる

	集約型	分散型
運用	監視・デプロイ・障害調査の対象が少なく単純	対象が増え、分散トレーシングなどの仕組みが要る
遅延	プロセス内・ノード内で完結し**小さい**	ネットワーク越しの往復が積み上がる

集約型は「悪い設計」ではありません。 4SA01 の設問では、要件が「利用者数が限定的で増加傾向が乏しい」「強い整合性が業務上必須」「運用要員が少ない」と書かれているときに、**分散型を選ぶと過剰**になります。分散化は、①1台では処理しきれない、②部分的に落ちても止められない、③部分ごとに独立して変更・拡張したい、のいずれかが要件にあるときに初めて元が取れます。

ひっかけ: 「可用性を上げたいので分散型にする」だけでは足りません。分散配置しても、**全ノードが同じ単一のデータベースを見ている**なら、そこが単一障害点として残ります。分散化の効果は「どこまで分散したか」で決まります。

1-3 分割 —— layer 構造と tier 構造 [1.1]

layer（レイヤ）構造は論理的な分割、tier（ティア）構造は物理的な分割です。ここは毎回どこかで問われます。

	layer 構造	tier 構造
何の分割か	**役割・関心事による論理的な分割**	**配置（実行環境・ノード）による物理的な分割**
例	プレゼンテーション層／業務ロジック層／データアクセス層	Web サーバー／アプリケーションサーバー／データベースサーバー
分けても	同一プロセス・同一ノードに同居してよい	別ノードに置かれる（ネットワークを挟む）

	layer 構造	tier 構造
主な効果	保守性・変更容易性・再利用性	個別のスケール、境界での防御、障害の局所化
主なコスト	呼び出しの間接化（実行コストは小さい）	**ネットワーク遅延・障害点の増加・運用対象の増加**

1つの layer 構造を、そのまま tier に割り付けるとは限りません。3層（layer）で設計したアプリケーションを1台のノードに置けば 1-tier ですし、3台に分ければ 3-tier です。設問では「3層アーキテクチャなので3台のサーバーが必要」といった**論理と物理を同一視する肢**が置かれます。誤りです。

layer 構造の重要な規約が**「上位層は下位層を呼ぶが、下位層は上位層を呼ばない」**という一方向の依存です。これが崩れると、層を分けた意味（下位を差し替えても上位が影響を受けない）が失われます。データアクセス層から業務ロジック層の関数を呼び出す設計は、この原則違反として問われます。

tier を増やす判断は、次のどれかが要件にあるときです。

- **層ごとに負荷特性が違い、別々にスケールしたい**（Web は接続数、DB は I/O）
- **セキュリティ境界を引きたい**（DMZ に Web、内部ネットワークに DB を置き、DB を外部から到達不能にする）
- **可用性の単位を分けたい**（アプリを入れ替えても DB は動かし続ける）

逆に、上のどれも要件になければ、tier を増やすほど**遅延と運用負荷だけが増えます。**

図 1-2　layer 構造と tier 構造 —— 論理的な分割と、物理的な分割

1-4 業務とデータの関係 —— 共有か、非共有か [1.1]

ノードを分けたとき、**データをどう扱うか**が設計の本体になります。「業務機能を分けたのに、データは1つの共有 DB に置いたまま」という構成は非常によく見ますが、これは**分割が半分しか終わっていない状態**です。

方式	内容	得るもの	失うもの
データ共有型	複数ノードが**同一のデータストアを共有**する	整合性を DB のトランザクションに任せられる。実装が単純	データストアが**単一障害点・性能上限・変更の足かせ**になる。スキーマ変更が全ノードに波及する

方式	内容	得るもの	失うもの
データ非共有型（shared nothing）	ノードごとに**自分のデータを占有**し、他ノードのデータへは**インターフェース経由でしか触らない**	ノード単位で独立して変更・拡張・障害隔離ができる	ノードをまたぐ整合性を**自分で設計**しなければならない（TCC・Saga・結果整合）

「業務とデータの関係」を見る目線は、次の問いに集約されます —— **その業務機能を変更したとき、他のノードのデータ定義まで直す必要があるか**。必要があるなら、それは分割できていません。逆に、非共有にすると「他ノードのデータが欲しいときは相手に聞く」しかなくなり、**参照のたびに通信が発生**します。参照が多い業務では、後述のレプリケーション（非同期データ複製）で手元にコピーを持つ判断が出てきます。

ひっかけ: 「共有 DB をやめれば整合性の問題がなくなる」は逆です。共有をやめた瞬間に、**整合性は DB の仕事から設計者の仕事になります**。

1-5 タイミング —— 同期と非同期 [1.1]

呼び出し側が**相手の処理完了を待つかどうか**の違いです。ここは制御パターンの前提になります。

	同期	非同期
呼び出し側	応答が返るまで**待つ**（ブロックする）	依頼を渡したら**先へ進む**
結果の受け取り	戻り値としてその場で	コールバック・イベント・ポーリングで後から

	同期	非同期
障害の伝播	相手が遅い／落ちていると**自分も遅くなる／落ちる**	相手が落ちていても依頼を溜めておける（キューがあれば）
実装	単純。処理の順序が読みやすい	複雑。順序保証・重複・失敗の扱いを自分で設計する
向く場面	結果が無いと次へ進めない（残高照会、認証）	結果を待たなくてよい（通知送信、集計、変換、印刷）
応答時間	全区間の合計が利用者の待ち時間になる	受け付けまでが利用者の待ち時間になり、**体感を短くできる**

非同期化は「速くする」技術ではありません。全体の処理量が減るわけではなく、**利用者を待たせる区間を短くし、負荷の山を時間方向にならす**技術です。設問では「非同期にしたのでスループットが必ず向上する」といった肢が置かれますが、下流の処理能力が同じなら、溜まるだけです（キューが伸びる）。

同期の連鎖が危険な理由も押さえてください。A→B→C と同期で呼ぶと、**利用者の待ち時間は3区間の合計**になり、**可用性は3つの積**になります（どれか1つが落ちれば全体が落ちる）。同期呼び出しを深く重ねる構成は、第8章のタイムアウト・サーキットブレーカーとセットで設計します。

1-6 制御と連携集約のパターン —— 主従関係で整理する [1.1]

ここからが本章の中心です。**「誰が全体の流れを知っているか」**で整理すると、丸暗記せずに済みます。

パターン	全体の流れを知っているのは	主従関係	呼び方
パイプライン（パイプ／フィルター）	**誰も持たない**（つなぎ方が流れそのもの）	前段→後段の一方向	データを流す
コーディネーション	**参加者どうしが合意して進める**	対等に近い	互いに調整する
オーケストレーション	**指揮者（オーケストレーター）が持つ**	明確な主従（中央が従を呼ぶ）	中央が順に呼ぶ
コレオグラフィ	**誰も全体を持たない**（各自が自分の担当だけ知る）	主従なし	各自がイベントに反応する
メディエーター／ブローカー	**仲介者が経路を持つ**（業務の流れは持たないこともある）	送信側と受信側が互いを知らない	仲介者を経由する

パイプライン（パイプ／フィルター） [1.1]

フィルター（処理の単位）を**パイプ**（データの通り道）でつなぎ、**データを一方向に流していく**構造です。

- 各フィルターは**入力を受け取り、変換し、出力する**ことだけを知っています。前後に誰がいるかは知りません
- そのため**フィルターの差し替え・追加・並べ替えが容易**です
- 段ごとに独立して並列化・多重化できます

向くユースケース: ログの収集と加工、ETL（抽出・変換・格納）、画像や動画の変換、ビルドパイプライン、テキストの逐次処理。**共通点は「データの流れが一方向で、分岐や巻き戻しが少ない」**ことです。

制約: 双方向のやり取りや、途中で全体を止めて分岐するような複雑な業務フローには向きません。また、**段数が増えるほど遅延が積み上がり**、途中でエラーが起きたときに「どこまで進んだか」を追いにくくなります。段ごとにバッファ（キュー）を置くと、速度差のある段をつなげられます。

コーディネーションとオーケストレーション [1.1]

オーケストレーションは、**中央のオーケストレーターが業務フロー全体を保持し、各サービスを順に呼び出す**方式です。指揮者とオーケストラの関係で、**主従がはっきりしています**。

- **利点:** 業務フローが**1か所に書かれている**ので、全体の流れを読める・変更できる・進捗と失敗の状態を追える。補償処理（後述の Saga）の実装も中央に集約できる
- **欠点:** オーケストレーターが**単一障害点**かつ**変更のボトルネック**になる。サービスを1つ足すたびに中央を直す必要があり、結合が中央に集まる

コーディネーションは、**参加者どうしが調整して合意に至る**方式です。中央の指揮者を置かず、参加者が互いに状態を確認しながら足並みをそろえます。代表例は分散トランザクションの合意（コミットするかどうかを参加者間で決める）や、クラスタでのリーダー選出（第7章の Quorum）です。

設問での見分け方:「**中央に流れが書かれているか**」を見ます。「注文サービスが在庫→決済→配送の順に呼び出す」ならオーケストレーション、「参加者が互いの可否を確認して足並みをそろえる」ならコーディネーションです。

イベント駆動とコレオグラフィ [1.1]

イベント駆動は、「**起きた事実（イベント）を発行し、関心のあるコンポーネントがそれに反応する**」方式です。**発行側は誰が受け取るかを知りません。**

コレオグラフィは、イベント駆動を業務フローに適用した形で、**全体の流れを保持する中央が存在せず、各サービスが「このイベントが来たら自分は何をして、次に何を発行するか」だけを知っている**方式です。振り付け（choreography）どおりに各自が踊れば、全体として1つの流れになります。

	オーケストレーション	コレオグラフィ
全体フローの所在	**中央のオーケストレーター**	**どこにも無い**（各サービスに分散）
サービス間の結合	中央 ⇔ 各サービス（中央に集中）	イベントの型のみ（**疎結合**）
サービス追加	**中央の変更が必要**	**既存を変えずに、購読側を足せる**
全体の把握	容易（1か所を読む）	**困難**（分散トレーシングが要る）
障害時の補償	中央に集約できる	各サービスに分散し、設計が難しい
単一障害点	オーケストレーターがなり得る	中央は無いが、**イベント基盤**がなり得る

この2つの入れ替えが、この章で最も頻出のひっかけです。「中央が各サービスを順に呼ぶのがコレオグラフィ」と書いてある肢は誤りです。**指揮者がいるのがオーケストレーション、振り付けだけ決めて各自が踊るのがコレオグラフィ**と覚えてください。

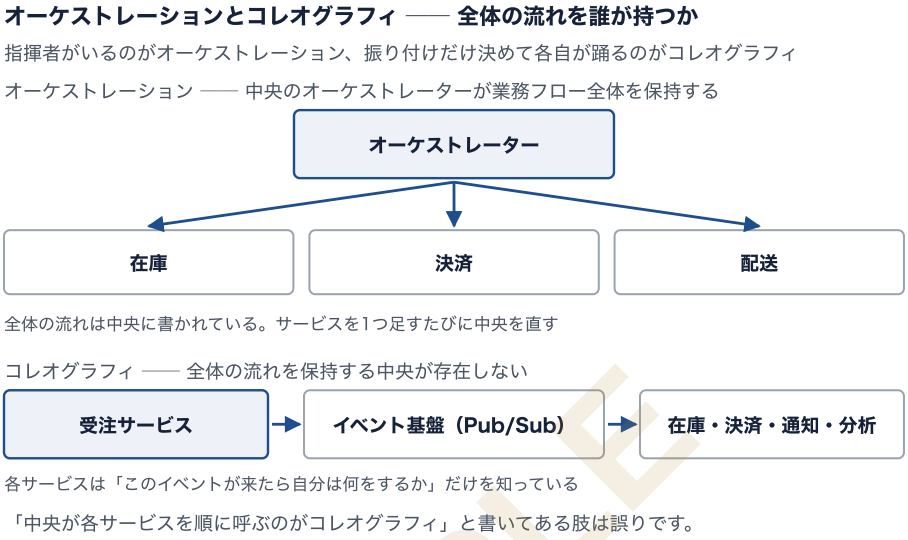

図 1-3　オーケストレーションとコレオグラフィ —— 全体の流れを誰が持つか

選び分けの判断軸:

- 業務フローが**複雑で、順序と補償を厳密に管理したい**、進捗を可視化したい → **オーケストレーション**
- 参加者が**今後増える見込みで、既存を触らずに拡張したい**、各チームが独立して動きたい → **コレオグラフィ**

メディエーターとブローカーによる仲介 [1.1]

送信側と受信側の間に**仲介者**を挟み、**互いを直接知らなくてよい**状態を作る方式です。

- **メディエーター** …… 仲介者が**やり取りのルール（経路・変換・順序）を持つ**。参加者は仲介者だけを知る

- **ブローカー** …… 仲介者は**受け渡しに徹し**、経路の決定は宛先（キュー名・トピック名）に委ねる

仲介者を挟むと、**送信側と受信側は「時間的にも」切り離されます。**受信側が停止していても送信側は送れる（溜まる）、というのが最大の効果です。

メッセージングキュー [1.1]

1つのメッセージを、1つの受信者が取り出して処理するモデルです（point-to-point）。

- 受信者を増やせば**処理を分担**でき、そのまま**負荷分散**になります（**競合コンシューマ**）
- 送信側の速度と受信側の速度が違っても、キューが**バッファ**として吸収します（**負荷平準化**）
- 受信側が落ちていても、メッセージはキューに残ります（**耐障害**）

Pub/Sub パターン [1.1]

1つのメッセージを、購読している「すべての」受信者が受け取るモデルです（publish/subscribe）。発行者（publisher）はトピックへ発行し、購読者（subscriber）がトピックを購読します。

キューと Pub/Sub の違いは「1対1か、1対多か」です。ここは入れ替え型のひっかけの定番です。

	メッセージングキュー	Pub/Sub
配送	**1メッセージ＝1受信者**が処理	**1メッセージ＝購読者全員**が受け取る

	メッセージングキュー	Pub/Sub
受信者を増やすと	**処理が分担される**（速くなる）	**同じ処理が受信者の数だけ実行される**
主な用途	非同期のタスク処理、負荷分散、平準化	状態変化の通知、複数システムへの一斉連携
典型的な誤り肢	「受信者を増やすと全員が同じメッセージを処理する」	「購読者を増やすと処理が分担される」

コレオグラフィの実装基盤は、多くの場合 Pub/Sub です。「注文が確定した」というイベントを1つ発行すると、在庫・決済・通知・分析がそれぞれ反応します。イベントを増やしても、発行側は変わりません。

1-7 データ連携と整合のパターン [1.1]

ノードをまたいで**1つの業務を成立させる**ときの整合性の取り方です。データ非共有型を選んだ瞬間に、この設計が必要になります。

分散トランザクションが難しい理由 [1.1]

単一 DB のトランザクションは、DB が「全部やるか、全部やめるか」を保証してくれます。ノードをまたぐと、**各ノードのトランザクションは別々**になるため、途中で失敗したときに**すでにコミットしてしまった処理を戻す**必要が出てきます。さらに、全ノードをまたいでロックを保持し続けると、**その間ずっと他の処理が待たされ**、可用性とスループットを大きく落とします。

そこで使われるのが **TCC** と **Saga** です。**どちらも「長時間ロックを持たない」ための工夫**という点は共通で、**違いは「仮押さえをするかどうか」**です。

TCC（Try-Confirm/Cancel） [1.1]

処理を **Try → Confirm または Cancel** の2段階に分けます。

1. **Try** …… 各サービスが**必要な資源を仮押さえ（予約）し、実行可能かを確認**する。まだ確定はしない
2. **Confirm** …… 全サービスの Try が成功したら、**仮押さえを確定**する
3. **Cancel** …… どれか1つでも Try が失敗したら、**仮押さえを解放**する

- **仮押さえの状態が存在する**のが特徴です（在庫を「引当済み」にする、与信枠を「確保」する）
- **業務的に見える中間状態が短い**ため、確定の直前まで「まだ他人に取られない」ことを保証できます
- 代償として、**各サービスに Try／Confirm／Cancel の3つの API を実装する必要**があり、実装コストが高くなります
- 仮押さえを解放しないまま呼び出し側が落ちると資源が残るため、**タイムアウトによる自動解放**を設計に含めます

Saga [1.1]

業務を**ローカルトランザクションの連なり**として実行し、途中で失敗したら**それまでに完了した処理を打ち消す「補償トランザクション」を逆順に実行**します。

1. 各サービスが**自分のトランザクションを即座にコミット**する（仮押さえをしない）
2. 後続が失敗したら、**すでにコミットした処理に対する補償処理**（返金する、引当を戻す、キャンセル通知を出す）を実行する

- **仮押さえが無いぶん実装が軽く、ロックを持たない**のが利点です

- 代償として、**中間状態が外から見えてしまいます**（一度確定した注文が、あとでキャンセルされる）。**業務としてその見え方が許されるか**が採用条件です
- **補償できない処理には使えません。**メールを送ってしまった、現金を払い出してしまった、といった「取り消せない副作用」がある処理は、Saga の対象にできない（あるいは最後に回す）
- Saga の実行方式は、**オーケストレーション型**（中央が順に呼び、失敗時に補償を呼ぶ）と**コレオグラフィ型**（各サービスがイベントに反応し、失敗イベントで補償する）の2通りがあります

	TCC	Saga
仮押さえ	**する**（Try で予約）	**しない**（都度コミット）
打ち消し方	Cancel で**仮押さえを解放**	補償トランザクションで**確定済みの結果を打ち消す**
中間状態の見え方	「予約中」として制御できる	**確定して見えたものが後で取り消される**
実装コスト	高い（3つの操作を全サービスに）	低め（補償処理を用意すればよい）
適する場面	在庫や座席の**二重取りを絶対に避けたい**	中間状態が業務上許容でき、**処理を止めたくない**

ひっかけ:「Saga は資源を仮押さえして確定・取消を行う」「TCC は補償トランザクションを逆順に実行する」—— **中身が入れ替わった肢**が最頻出です。**T が Try＝仮押さえ**と結びつけてください。また「Saga は ACID を保証する」も誤りです。Saga が保証するのは**最終的に業務として辻褄が合うこと**であり、途中経過の分離性（Isolation）は保証されません。

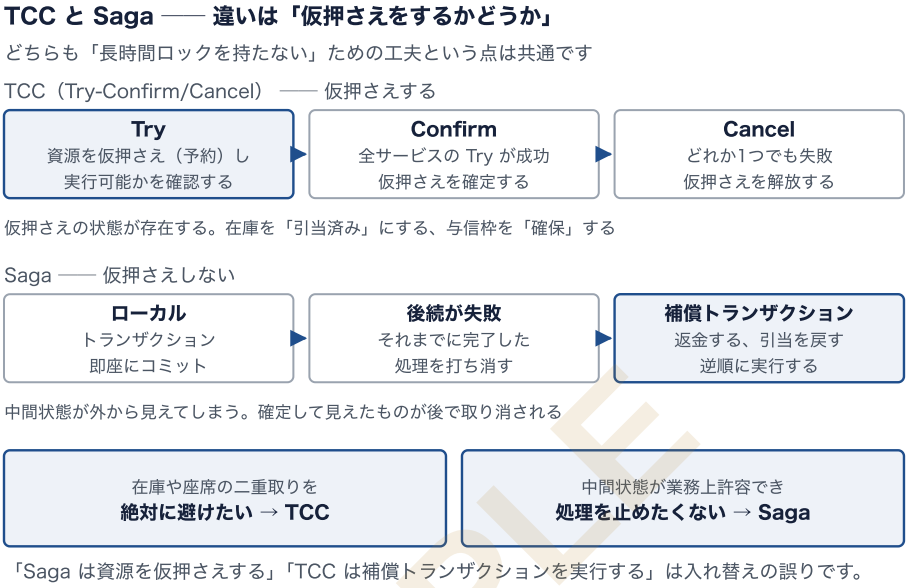

図 1-4　TCC と Saga —— 違いは「仮押さえをするかどうか」

レプリケーション（非同期データ複製） [1.1]

同じデータの写しを別ノードに持たせる方式です。ここでは連携パターンとしての側面（詳細な実装は第9章）を押さえます。

- **非同期複製**は、更新をコミットしてから**あとで写しへ反映**します。更新側は複製の完了を待たないため、**書き込みの応答時間が伸びません**
- 代償は**遅延（レプリケーションラグ）**です。写しを読むと**少し古いデータが返る**ことがあります（結果整合）
- 連携パターンとしての用途は、**参照専用の写しを相手側に置いて、問い合わせのたびに他ノードへ通信するのをやめる**ことです。第12章のリードレプリカ、第9章の災害対策と同じ道具を、別の目的で使っています

判断軸:「古いデータを読んでも業務が壊れないか」。壊れるなら（残高照会からの引き落とし、在庫の引当）、非同期複製の写しを判断根拠に使ってはいけません。壊れないなら（商品一覧、レコメンド、分析）、非同期複製は最も費用対効果の高い手段です。

1-8 要件からパターンを選ぶ —— 判断の手順 [1.1]

設問はほぼ必ず「状況＋要件」から始まります。次の順で読むと、選択肢を機械的に絞れます。

手順1: データの持ち方を先に決める。「機能を分けたいが、同じデータを全員が更新する」なら、分けても効果は限定的です。**分割の境界は、業務の境界ではなくデータの境界**で決めます。

手順2: 待てるかどうかを決める。利用者が結果を見るまで待つ必要があるなら同期、通知や集計のように後回しでよいなら非同期です。**「待てない処理を非同期にする」「待てる処理を同期の連鎖にする」がどちらも典型的な誤設計**です。

手順3: 全体の流れを誰が持つかを決める。業務フローが複雑で監査・進捗管理が必要ならオーケストレーション、参加者が増える前提で各チームが独立して動きたいならコレオグラフィです。

手順4: 失敗したときの戻し方を決める。仮押さえが要るなら TCC、都度確定して打ち消してよいなら Saga、そもそも打ち消せない処理は**フローの最後に置く**か、人手の運用に逃がします。

ケース1: 受注処理 —— 在庫の二重引当を避けたい [1.1]

要件が「同じ商品を複数の注文が同時に取り合う」「在庫を超えて売ってはならない」であれば、**確定前に押さえておく必要**があります。都度コミットして後

から補償する Saga では、**補償が走るまでの間に他の注文へ売れてしまう**可能性が残ります。よって **TCC** が第一候補です。

- Try で在庫を引当（仮押さえ）し、決済の Try で与信枠を確保する
- 両方成功したら Confirm で確定、どちらかが失敗したら Cancel で解放する
- 呼び出し側が落ちた場合に備え、**仮押さえにタイムアウトを持たせて自動解放**する

誤り肢の作られ方:「共有 DB のトランザクションで囲めばよい」―― データ非共有型を選んだ前提と矛盾します。「全体を同期呼び出しで直列につなげばよい」―― 二重引当は防げても、決済サービスの遅延がそのまま受注全体の遅延と停止になります。

ケース2: 注文確定後の後続処理 ―― 通知・ポイント・分析 [1.1]

「注文が確定したら、メール通知、ポイント付与、分析基盤への連携を行う。今後、連携先が増える見込みがある」―― この要件はほぼ一択で、**イベント駆動＋Pub/Sub によるコレオグラフィ**です。

- 受注サービスは「注文が確定した」というイベントを**1回だけ発行**します。誰が受け取るかは知りません
- 連携先が増えても、**購読側を追加するだけ**で受注サービスは変わりません
- 通知が遅れても受注は成立します（**待たなくてよい処理**だから非同期でよい）

ここで**メッセージングキュー（1受信者）を選ぶと、連携先が増えるたびにキューを増やして送信側を直す**ことになり、疎結合の利点を失います。**「複数の相**

手が同じ事実を必要とする」なら Pub/Sub です。

ケース3: 大量データの加工 —— 段ごとに速度差がある [1.1]

「収集 → 変換 → 集計 → 格納」のように**一方向に流れ、段ごとに処理速度が違う**なら、**パイプライン（パイプ／フィルター）** です。段と段の間に**キューを挟む**と、速い段が遅い段に足を引っ張られず、遅い段だけ受信者を増やして（競合コンシューマ）追いつかせられます。

ここで問われるのは「どこを増やすか」です。 全段を一律に増やしても、**いちばん遅い段（ボトルネック）が変わらなければ全体のスループットは変わりません。**増やすべきは詰まっている段だけです。この考え方は第10章の性能評価、第12章のスケールアウトと同じものです。

ケース4: 参照が多く、他ノードへの問い合わせが負荷になっている [1.1]

「商品サービスが持つマスタを、注文・検索・レコメンドの各サービスが毎回参照していて、商品サービスが詰まっている」—— ここで採るのは **レプリケーション（非同期データ複製）** です。参照側が**自分の手元に写しを持ち**、更新はイベントで伝えます。

採用条件は**「少し古くても業務が壊れないこと」**です。商品名や説明文なら許容できますが、**在庫数や価格の最終確定**を写しで判断すると事故になります。**同じデータでも、用途によって写しでよいかどうかが変わる** —— ここが設問の分かれ目です。

1-9 混同しやすい対の概念 [1.1]

組	違い
layer ／ tier	**論理的な役割分割** ／ **物理的な配置分割**
集約型 ／ 分散型	スケールアップ・強い整合性が取りやすい ／ スケールアウト・整合性は自分で設計
同期 ／ 非同期	完了を待つ・可用性は積になる ／ 待たない・順序と重複を自分で設計
オーケストレーション ／ コレオグラフィ	**中央が全体フローを持つ** ／ **各自が自分の反応だけ知る**
コーディネーション ／ オーケストレーション	参加者が対等に合意 ／ 明確な主従で中央が呼ぶ
メッセージングキュー ／ Pub/Sub	1メッセージを**1受信者**が処理（分担） ／ **購読者全員**が受信（一斉）
メディエーター ／ ブローカー	仲介者が**経路と変換のルールを持つ** ／ 受け渡しに徹する
TCC ／ Saga	**仮押さえ**して Confirm／Cancel ／ 都度コミットし**補償**で打ち消す
データ共有 ／ 非共有	DB が整合性を担保・DB が制約になる ／ 独立して動ける・整合性は設計事項
パイプライン ／ イベント駆動	データが**決まった順路**を流れる ／ 誰が反応するかは**購読者側**が決める

1-10 章の試験ポイント [1.1]

- **①（入れ替え）オーケストレーション／コレオグラフィ。**「中央が順に呼ぶ」がオーケストレーション。**この章の最頻出**

- **①（入れ替え）TCC／Saga。**「Try＝仮押さえ」があるのが TCC、「補償で打ち消す」のが Saga
- **①（入れ替え）キュー／Pub/Sub。**受信者を増やしたときに**分担されるのがキュー、全員が受け取るのが Pub/Sub**
- **②（層の取り違え）layer と tier。**「3層アーキテクチャだから3台必要」は論理と物理の同一視
- **⑤（目標値の取り違え）非同期化の効果。**非同期は**体感の待ち時間と負荷の山**を改善する。**総処理量やスループットが自動的に増えるわけではない**
- **⑥（効かない対処）分散化そのもの。**分散配置しても**共有 DB が残っていれば単一障害点は消えない**
- **④（前提条件の付け替え）Saga の適用条件。**補償できない副作用（送信済みのメール、払い出した現金）がある処理には使えない
- **トレードオフが書かれていない肢を疑う。**「疎結合にすればレスポンスも速くなる」「非同期にすれば整合性も保たれる」型の肢は、捨てたはずのものを捨てていない

サンプルはここまでです

続き(第2章〜巻末)は、LinuC 4SA01 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。

LinuC レベル4 4SA01試験 学習用テキスト(無料サンプル)

版

2026年7月版(LPI-Japan公式 LinuC レベル4 システムアーキテクト 4SA01試験(LinuC-4SA Version 1.0) 出題範囲 準拠)

発行

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

お問い合わせ

info@shikaku-doujou.com