Principles of Transaction Processing, 2nd ed.

久野　靖*

2011.9.1

Philip A. Bernstein, Eric Newcomer, Principles of Transaction Processing, 2nd ed., Morgan Kaufmann, 2009.

0 Preface

0.1 WHY READ THIS BOOK?

□ トランザクションプロセシング (TP) は 40 年に渡って重要なソフトウェア技術であり続けている

- 交通、金融、小売、通信、製造、行政、軍事
- 予約、入出金、株取引、注文処理、音楽/ビデオサービス、出荷追跡、回線交換、在庫管理、指令/制御
- IBM、HP、Oracle、MS、Dell、Red Hat、EMC
- 数$10,000,000,000/year 規模、我々が日常的に使用

□ TP システムの動作

- かっては商業データ処理のプロだけが関心
- 今では広く使われていることからより広い範囲の人が知るべき
- しかし分かりやすい解説はあまりなかった→本書の目的

□ 多くの TP 環境は TP ミドルウェアに基づいている

- TP ミドルウェア --- 多くのコンポーネントを組み合わせる
- コンポーネント:　フロントエンド (Web サーバ)、中間層 (入力をサービスに接続)、サーバ (ビジネスロジック)
- TP ミドルウェアの例:　IBM CICS、.NET Enterprise Service、Java EE 製品 (IBM WebSphere Server、Oracle WebLogic Server、Red Hat JBoss Applicaiton Server)
- 本書の前半は TP ミドルウェアに焦点

□ 多くのソフト技術者にとって TP ミドルウェアはよく分からない世界

- OS の上に載ったヘンな glue、データベース、通信システム、アプリケーションプログラミング言語
- 本書はこれらの謎を解明←それぞれが TP システムの性能、セキュリティ、スケーラビリティ、可用性、保守性、使いやすさにどう影響しているかを説明
- 前半では TP ミドルウェアを外→中の順で説明:　AP プログラマにどのような機能をどういう風にして提供しているか

□ トランザクションという抽象化は DB システムが提供

- トランザクションは全体として実行され、他のトランザクションと干渉せず、ハード/ソフトの障害があっても失われない → 本書の後半のテーマ
- 対象読者:　AP プログラマ、DB 管理者、AP アナリスト、関連領域の開発者 (DB、OS、通信)、マーケティング技術者、学部生/院生

□ 焦点は「原理」であり、TP システムの作り方ではない

- 各種製品に基づく具体例多数←原理がどのように実際に適用されているかが分かるから。特定製品に深入りはしない
- 実際に使われている技術に重点を置き、そうでないものはそこそこに

□ 前提:　システム的に考えられること。SQL と DB は分かっているものと期待するが、そうでなくても大丈夫

□ 結果:　TP ミドルウェアの働きや、信頼できる分散 TP システムのためにいつそれを使うべきかが分かる。実際の TP ミドルウェアや DB 製品を速やかに学んでそれを活用して TP アプリケーションを作れるようになる

0.2 WHAT'S NEW IN THIS SECOND EDITION

□ 初版は 12 年前→そこから大きく変更した。その要因はおもに 2 つ

□ その 1:　状況の大きな変化

*筑波大学大学院経営システム科学専攻

- インターネット上の E コマースによる B2B、B2C の普及
- サーバや OS のコモディティ化による TP 製品の変化
- ブラウザがフロントエンドになり、TP ミドルウェアも TP モニタからインターネット向けのさまざまな製品として発展 (AP サーバ、ORB、メッセージミドルウェア、ワークフロー製品)
- OOP や SOA が主流になり、DB システムが TP 機能をきちんと搭載

□ その 2: 内容の充実: 伝統的 TP システムについてより広く/深く

- ワシントン大で教科書として使用してきた。技術の発展に追従
- TP ミドルウェアについての 3 章 (原理 2 章、実例 1 章) は全面書き換え。例に挙げる製品の充実 (J2EE、.NET)
- ビジネスプロセス管理に関する章を新設
- ロックに関する章: 楽観的並行制御、B 木ロック、多粒度ロック、入れ子トランザクションを追加
- 新しい章: TPC-E、状態管理、スケーラビリティ、シャドウページ、データ共有システム、合意アルゴリズム、ログベース重複化、複数マスター重複化。
- 全体的に: SOA、REST、Web サービスについて言及

□ サポートページもあります

0.3 SUMMARY OF TOPICS

□ 多様なニーズ

- スポンサー: 速く、安く、スケーラブルで、機能追加しやすい
- AP プログラマ: TP のさまざまな複雑さからは隔離されたい (トランザクションプロトコル、メッセージプロトコル、トランザクション RPC、永続キュー、マルチスレッド、リソースプール、セッション管理、重複化プロトコル)
- AP プログラマの責務: ビジネスが必要とするものを理解し、トランザクションを用いてそれを実現する
- システムソフトウェアの仕事: 速く、効率的で、スケーラブルで、信頼性のあるシステム → TP ミドルウェアの役割 → 1〜5 章、＋ 10 章 (現況)

□ 理想と現実と実際

- TP システムのユーザが考えたいもの: トランザクションを順次的に処理し、無限に信頼性が高く、自分にだけサービスをしてくれ、トランザクション全体を完遂してくれ、結果は永遠に残してくれる
- 現実には: 多くのトランザクションを並行処理し、しょっちゅうソフトやハードのエラーが起き (しかも最悪のタイミングで --- あなたのトランザクションの実行中に)、記憶容量も有限
- にもかかわらず、ソフトウェア的手法を組み合わせることで、その現実の上で理想とするものを実現する → 6〜9 章
- 技術の進歩にともなう発展→ 11 章 (クラウド、スケーラブルな分散コンピューティング、SDD、ストリーム/イベント処理)

□ 各章のサマリー:

- 1 章 はじめに --- TP アプリケーション/システムの全体像。SOA、ACID、2PC、TPC ベンチマーク、可用性、TP 対バッチ/リアルタイム
- 2 章 TP の抽象化 --- TP システムがどのような抽象化を提供するか。トランザクション、プロセス/スレッド、RPC、共有状態 (コンテキスト、セッション、クッキー)、スケーラビリティ技術 (キャッシング、資源プーリング、パーティショニング、重複化)
- 3 章 TP アプリケーションアーキテクチャ --- 多層アーキテクチャの価値と各層の役割。フロントエンド (フォーム、Web サーバ)、リクエストコントローラ (トランザクションで囲む)、トランザクションサーバ (トランザクション実行)。これらと TP ミドルウェア、DB サーバとの対応
- 4 章 キューによる TP --- 永続メッセージキューによる信頼性。リカバリシナリオを詳細に説明し、キューによる公開/購読型、ブローカ型、バス型ミドルウェアの構成を説明。キューマネージャの内部も IBM Websphere MQ、Oracle Stream AQ を例に説明
- 5 章 ビジネスプロセス管理 --- 複数の関連するトランザクションとして実行されるビジネスプロセスの生成/管理/モニタリングをサポートするメカニズムについて。複数トランザクションリクエストにおいて適切な原子性、隔離性、耐久性を得ること。標準としての BPEL(ビジネスプロセス実行言語) と実装例としての MS SQL Service Broker。
- 6 章 ロッキング --- 2PC がどのように動き、AP プログラマがその正しさや性能にどう影響し得るか。ロックマネージャの実装、デッドロックの対処、ロック粒度による性能の調整、楽観的手法、バッチ化、ホットスポット除去、隔離度の低い手法、多重バージョン手法、B 木ロッキング、多粒度ロック、入れ子トランザクション。

- 7 章 システムリカバリ --- 障害の原因とトランザクションでいかに障害をマスクできるか。チェックポイント、ステートレス、ウォーム/ホットスタンバイ、DB におけるロギング (トランザクション/システム/メディアの障害に対処)、undo/redo パラダイム、ログチェックポイント、リカバリアルゴリズム、シャドウページ、ロギング最適化 (ARIES アルゴリズムなど)、アーカイブのリカバリ。
- 8 章 2 相コミット (2PC) --- 2PC プロトコルの詳細、リカバリの状況分析、ユーザの関与場面。よく使われる最適化 (アボート予約、フェーズ 0、コーディネーションの転送)。X/Open トランザクション管理アーキテクチャを例に DB と TP マネージャのやりとりを説明。
- 9 章 重複化 --- サーバ重複化と資源重複化のトレードオフ、正当性基準、1 コピーによる直列化可能。主要な重複化手法 (主要コピー方式、複数マスター方式)。重複キャッシュの同期、合意アルゴリズム。
- 10 章 トランザクションミドルウェアの製品と標準 --- J2EE、.NET Enterprise Service、旧来製品 (CICS、IMS、Tuxedo、ACMS、Pathway)。各種コンポーネント (Windows Communication Foundation、EJB、JDBC、Java Transaction API、Spring。標準 (OMB、X/Open、OASIS)。
- 11 章 将来動向 --- TP 技術の主要な方向性。クラウド、SDD、データ/イベントストリーム、など。

0.4 GUIDANCE FOR INSTRUCTORS

□ ワシントン大での授業経験

- 宿題、プロジェクト、講義ビデオなど公開 http://www.cs.washingotn.edu/education/course/csep545/
- いろいろやったが、まず 1 章、つぎに 6 章、7 章をやり、宿題と試験でしごいてからコースプロジェクトに進むのがよかった
- プロジェクトは 3 種類: AP のケーススタディ、商用製品 (J2EE か.NET) で AP を作ってみる、分散トランザクションを実行する TP ミドルウェアを作ってみる → 最後のが最も効果的だったのでこれを必須とした
- 旅行予約システム (飛行機、ホテル、レンタカー) のひな型を作る→ ロック、リカバリ、2PC を実装したリソースマネージャとリクエストを転送できるトランザクションミドルウェアの実装が必要 (10 週ではプロの院生でもちょっと大変すぎ)。なので、一部はもうできているものを与えている (これもダウンロード可能)

0.5 ACKNOWLEGMENTS

1 Introduction

1.1 THE BASICS

1.1.1 The Problem

□ ビジネストランザクション

- 現実世界における (企業と企業/個人との) やり取り (たいていはお金、品物、情報、サービスのやり取りを伴う)
- 何があったかの記帳が必要→コンピュータで (規模、信頼性、コストのため)
- このための通信もコンピュータ/ネットで→ TP(トランザクション処理)

□ TP における代表的な要求

- 複数の操作が関与 (支払い+出荷など) →ナイーブな方法なら簡単だが、規模/信頼性/コストがからむと複雑
- 量とデータベース規模→複雑、効率が必須。店員が端末の応答待ちになって待たされたり、Web ページの取り寄せに待たされたりする。企業は「迅速で、低コスト」を求める
- 規模+性能のために並行処理が必須だが、制御がないと間違った処理。公平性も必要 (Amazon は最初の Xbox 千台の発売時に各顧客のチャンスが公平であるようにかなり労力を払った)
- トランザクションは全体として実行される必要。支払って品物を受け取るか、どちらも起きないか。エラー (避けられない) が起きた場合でも部分的に実行 (支払ったが品物は届かない) が起きると問題
- トランザクションは「OK」「NO」を正しく返すべき。これが無いと顧客はもう 1 度実行した方がいいのかどうか判断できない
- 漸進的規模拡大。機器を少しずつ買い増すことでスケールしたい。全システム入れ替えや、ましてアプリケーション作り直しは避けたい
- オンラインショップが止まってしまったらその小売店は商売できない。ビジネス活動にとってミッションクリティカル→ダウンは避けたい

- 完了した取引記録は永続的かつ正当である必要 (金融では法的要求)。トランザクションが失われてはいけない
- 地理的に分散していてもうまく稼働する必要→システム自体も分散システムであることが多い。法的要求のことも (システムはビジネスの起きる地域になければいけない)。また技術的要求 (効率、漸進的規模拡大、障害対策) のことも。
- パーソナライゼーションも必要。小売なら各顧客に向けた割引やお勧めなど
- インターネットからの数百万の顧客の可能性に対して、予測可能な形で、かつ低コストで規模拡大可能であること。ネットからの顧客数は制御できない
- 運用が容易であること。さもないと運用スタッフのコストがかさむし、複雑なシステムだと間違いやダウンの可能性が増し、それによりストレスや残業が生じてされにコスト増

□ まとめると: TP システムは大変。それをどうやって実現するかが本書のテーマ

- トランザクション自動処理の原理 (従来型、ネット)
- そのための基本技術の複雑なところ (ログ、ロック)
- 今日の商用トランザクションミドルウェア (上記機能を提供)

1.1.2 What Is a Transaction?

□ TP アプリケーション → トランザクションプログラムの集まりで、あるビジネス活動を自動化すべく設計されたもの

□ 最初の OLTP アプリ → SABRE (1960's、IBM+AA)

- 当時最大のコンピュータシステム、今日でも大規模なものの 1 つ
- 今日は Sabre ホールディングスが所有し、200 の航空会社、何千もの旅行代理店をサポート
- 航空券、特殊ミール、マイレージ、機体保守スケジュールなど管理。最大処理能力 20000 メッセージ/秒

□ 今日の TP の用途→図 1.1

- 初期の TP →大企業による大規模なもの→この部分は今日でも (ネットの発達により) より重要に
- さらに今日では (機器や DB、TP ソフトのコスト低下により)、数端末+サーバ程度の小規模な TP も増加 (小さい通販店、学校のコース登録、歯科医院の予約など)
- いずれも技術的には同じシステム構造、同じ技術

□ TP サービスを他社に提供する事業者も

- Amazon のサービス、航空予約機能を他者に提供する航空会社
- SaaS などによるアウトソース (TP の設置/管理はスキルが必要)

1.1.3 A Transaction Program's Main Function

□ TP プログラムのやることは次の 3 つ

- (1) ブラウザや他の端末 (POS、センサなど) から入力を受け取る
- (2) 実際の処理を行う
- (3) 応答を作り出し、(おそらく) 返送

□ 以上の処理を入力ごとに独立して「1 度だけ」行い結果を「残す」

□ 多くの TP システムではトランザクションとして実行しないコードも

- 情報収集、メンテナンス、データの掃除、再構成、エラーチェック…
- 本書ではこれらも TP アプリケーションの一部として扱う

1.2 TP SYSTEM ARCHITECTURE

□ TP システムは「コンピュータシステムで、トランザクションプログラムを稼働させているもの」→そのソフトウェア部分の構造:図 1.2

- エンドユーザ機器: エンドユーザ (トランザクションの実行を要求する人 --- 銀行の顧客、ネット上の購買者など) が操作。機器は物理デバイス (レジスター、ガソリンポンプなど) のことも、PC 上のブラウザのことも、ダム端末のこともある。スマート端末であればアプリのコードを端末上でも実行
- フロントエンドプログラム: エンドユーザ機器とやりとりするアプリケーションコード。メニューやフォームの表示、ユーザ入力の収集をおこなう。ブラウザ上で動くコードをサーバから提供することも。収集した入力はチェックしてから本体部分に送る
- リクエストコントローラ: フロントエンドからの入力を受け取り、適切な (もしかしたら複数の) トランザクションプログラム呼び出しに変換。分散 TP ならメッセージを送る必要。複数プログラムが関与するならその状態を管理

- トランザクションサーバ： ユーザの要求した仕事を実行する部分。DB の読み書き、他のプログラムの呼び出し、結果の作成など
- DB システム： アプリケーションが必要とする共有データを管理

□ 例： インターネット上の注文処理

- フロントエンドは Web サーバ上で走り、フォームを読み書きし、ショッピングカードを管理 (本当か?)
- リクエストコントローラは Web サーバからトランザクションサーバを呼ぶ
- トランザクションサーバは注文を処理 (注文 DB の読み書き、カタログ情報や在庫情報にアクセス、クレジットカード処理のための別トランザクションを起動など)

□ サーバ上で動くトランザクションプログラムの種類はある程度限定 --- 数十から多くて数百程度

- それぞれビジネス上の処理に対応 (注文に対応する出荷、資金の移動など)
- アプリケーションが大きくなった場合、複数のアプリケーションに分割
- それぞれのトランザクションは小規模。典型的ディスクアクセス数 0〜30、実行命令数数千〜数百万、送受メッセージ数 2〜(場合によるが通常はずっと多い)
- 分散されていることが多い (複数のサーバの機能が必要だったり、負荷対応のため同機能のサーバが複数動いていたりするため)
- 動作時間は 1〜2 秒 (ユーザを待たせない、ロック競合を減らす)

□ DB システムの役割は重要 (おそらくアプリケーションコード自体よりも)

- メモリに入り切るサイズのこともあるが、もっと大きいことが多い
- 非揮発性記憶 (磁気ディスク、SDD) を多数搭載することが多い
- ディスク技術、DB 技術を最大限活用。重複化や分散化も

□ もう 1 つの重要なカテゴリ： トランザクションミドルウェア

- TP アプリケーションと下位層 (DB、OS、管理ツール) の中間に位置
- DB 接続、OS プロセス、通信機能などを効率よく活用させてくれ、スケールアップに対応させてくれる
- メッセージを適切なサーバに配送したりトランザクション機能をアプリケーションと統合したり分散トランザクションを実現
- 管理ツールと統合されて負荷分散に対応したり使いやすい API を提供

□ TP ミドルウェアはここ 15 年間で急速に発達してきた

- WWW の普及以前は TP モニタ、OLTP モニタなどと呼ばれて来た
- 1990 年代半ば以降、Web サーバ上のトランザクションのためにアプリケーションサーバが発展
- 最初は TP モニタとネットの橋渡し機能→やがてアプリケーションサーバと TP モニタが統合されるように
- 並行して、メッセージ型トランザクションミドルウェアや ORB が普及→ エンタプライズアプリケーション統合システム
- インターネットの標準プロトコルへの対応→ Web サービス→エンタプライズサービスバス
- ワークフロー製品→長時間実行されるビジネスプロセスに対応
- ミドルウェア製品は環境一式として導入されることが多いが、顧客によっては複数製品の機能を集めて来て独自環境を構築することも

1.2.1 Service Oriented Computing

□ SOA(Service Oriented Architecture): アプリケーションないしその再利用可能な一部を構成するスタイルの 1 つ

- 機能が中心となるという点で、OO(「もの」が中心) と対照的
- ビジネスサービス (ビジネスが顧客や取引企業に提供するサービス) の概念をモデル化するという点では自然
- サービスはオブジェクトにより実現されてもよいが、手続き、メッセージキュー、スクリプトなどで実現されてもよい
- サービスはやりとりするメッセージとインタフェースに関する契約 (contracts) により特徴づけられる (実装プログラムではなくて)

□ サービス指向は概念としては長くあるが…

- 主流になったのは大規模な検索、SNS、e コマースなどのサイトがサービス指向のアクセスを提供するようになったため

- その背景は標準的な Web サービスプロトコルの普及 ← 独立したプログラムがそれぞれネット経由で安全にかつ信頼できる形で他のプログラムを起動できるという実装技術
- 多くのベンダが Web サービスプロトコルを提供→マルチベンダ SOA が可能に

□ SOA を一部または全体として取り入れた TP システム→1 つのトランザクションや複数の分散したサービスを用いて自サービスを提供

- 環境や必要とする機能に応じて同期/非同期通信機能を提供
- SOA ベースの TP システムは複数のアプリケーション、OS、ミドルウェア、プラットフォーム、プログラミング言語から組み立て可能

□ 図 1.3 → SOA のコンポーネント

- サービスプロバイダ: サービスを提供、サービスリクエスタ: サービスを起動、レジストリ (リポジトリ): サービスの記述を公開
- サービス記述: インタフェース、交換するデータの名前やフォーマット、通信プロトコル、品質 (セキュリティ、信頼性、トランザクションのふるまいなど) を含む

□ 呼び出し方: メッセージ送信 (メッセージ交換パターンを参考に)

- 基本パターン: 1 方向の非同期メッセージ (送ったらそれきり、受け取ったら実行するだけ)
- 別のパターン: 要求=応答型、公開=購読型

□ レジストリ: 無い場合もある

- 呼び出し側がサービス記述を別の方法で取得する --- Web から取れるようになっている、提供側に情報を予めもらうなど

□ Web サービス: SOA の実装メカニズムの 1 つ

- SOAP を使って呼び出す (SOAP は前は Simple Object Access Protocol だったが SOAP 1.2 では何かの略号ではないことになった)
- サービス記述は WSDL(Web Service Descriptio Language) による
- この記述をレジストリに登録し、レジストリは UDDI(Universal Description, Discovery, and Integration) プロトコルで検索可能
- 提供側と利用側は別プラットフォーム (例: JavaEE vs .NET) でも可

□ サービス化により既存サービスを組み合わせて新しいサービスを構築することが容易に

- そのような組み合わせを容易にするツール/技法も出現: Service Component Architecture (Java), Windows Communicaiton Foundation (Windows)

□ TP アプリも再利用可能なサービスの集まりとして存在し得る

- その場合でもフロントエンド、リクエストコントローラ、トランザクションサーバの役割は不変
- ただし機能のモデル化や設計は変わる必要があるかも
- たとえば図 1.2 でリクエストコントローラを Web サービスとして実現するなら、実装技術は WDSL+SOAP などに変更
- エンドユーザのインタフェースもブラウザから呼ぶように変更するかも (より詳しくは 3 章)

□ REST(REpresentational State Transfer): SOA への代替アプローチ (Web サービスとはかなり違っている)

- REST という用語は 2 つの違う意味でおもに使われる
- (1) WWW のためのプロトコルインフラストラクチャ (=HTTP)
- (2) WWW プロトコルによって実現可能なソフトウェアアーキテクチャ
- ここでは (1) の意味 --- REST/HTTP (より詳しくは 3.3 節)

□ REST/HTTP ではリソースを少数の汎用的な HTTP 操作 (GET、PUT、POST、DELETE) を使うことで再利用可能にする

- cf. Web サービスでは各アプリケーション固有の呼び出し
- 各 HTTP 操作はリソースを表す URI に対して適用
- レジストリ機能により URI を対応するネットワークアドレスに変換 (DNS によりドメイン名を IP アドレスに変換することに相当)

□ 汎用の HTTP 操作がアプリケーション毎固有の機能に対応づけられる

- 例: Web サービスで AddCustomer を呼ぶ → REST では URI に対する POST
- アプリケーション固有の機能やパラメタは URI に埋め込まれる → REST という名前の由来
- 転送に使われる表現形式としては JSON などの標準的形式が使われる

□ どの表現形式を使うかは HTTP の content-type ヘッダ、accept ヘッダなどで指定

- これにより、Web サービスのようにインタフェース定義で表現形式が決められているのと違い、呼び側に自由度 → 多様な呼び手を許容

□ REST/HTTP の人気←速さ、簡潔さ

- Web サービスの SOAP では XML で表現するためパースが重い
- XML の汎用性やインタオペラビリティは常に重要というわけではない
- 単純なサービスでは JavaScript のような特定言語での操作に特化して速いのがよいかも

1.2.2 Hardware Architecture

□ TP を動かすシステムの能力はさまざま

- ディスプレイも文字端末、モバイル機器、廉価な PC、ワークステーションなどさまざま
- フロントエンド、リクエストコントローラ、トランザクションサーバ、DB はどのようなサーバマシンでも動かせる (小型のサーバから大型メインフレームまで)
- 分散システムの場合、複数のマシン (1 つの部屋にある場合からキャンパス内、地域間、世界に分散まで)

□ 小規模な場合: 数台のディスプレイと小型 PC の LAN 接続

□ 大規模な場合 (航空予約など): 全企業、インターネット対象

- ディスプレイ台数で 10 万 (端末、チケットプリンタ、搭乗券プリンタなど)
- ディスク台数で数千、毎秒数千トランザクション
- 最大規模のネット上のシステムではユーザ数1 億、同時使用数 1 千万など

□ このように規模に大小があるので用語の区分が必要。基本的に一般用語だが、一部はこの本固有の意味として狭く定義

- マシン: 単一 OS イメージを実行しているコンピュータ。1 コア、マルチコア、共有メモリ MP など。VM でもよい (他の VM とハードを共有)
- サーバマシン: クライアントや他のコンピュータのためのプログラムを実行するマシン
- システム: 1 つ以上のマシンの集まりで協調して 1 つの機能を実現
- TP システム: 1 つ以上の TP アプリケーションをサポートするシステム
- ノード: ネットワーク上で他のマシンから 1 つのマシンとしてアクセスされるようなシステム (複数のマシンから成っていてもよい)
- サーバプロセス: OS のプロセス P で、自マシンまたは他マシンで動いているプロセスの下請けとして動くようなもの
- 意味がはっきりしているときは「サーバマシン」「サーバプロセス」と言うかわりに単に「サーバ」と言う

1.3 ATOMICITY, CONSISTENCY, ISOLATOIN, AND DURABILITY

□ トランザクションの重要な 4 つの特性

- 原子性 (atomicity) --- トランザクションは全体として実行されるかされないかのいずれか
- 一貫性 (consistency) --- トランザクションはデータベースの内部整合性を壊さない
- 隔離性 (isolation) --- トランザクションはそれ単独で (他のトランザクションが存在しないかのように) 実行
- 耐久性 (durability) --- トランザクションの実行結果は障害が発生しても失われない

□ これらをまとめて ACID と呼ぶ。「ACID トランザクション」「TP システムは ACID テストを通る」。以下で順に見ていく

1.3.1 Atomicity

□ トランザクションは atomic (全てかゼロか): 一部分だけが実行されるようなことがあってはいけない

- 例: $100 を口座 A から B に移す --- 「A から$100 引き」「B に$100 追加」この両方が起きるか、両方とも起きないか。片方だけ起きてはならない

□ TP システムではこれをデータベース操作を通じて実現

- トランザクションが最後までうまく実行したとき (commit) はじめて効果は恒久的なものとなる
- 途中で失敗したら (abort) すべての変更は元に戻される
- システム障害が起きた場合も同様 → 整合性のある状態に戻る

□ 各ビジネス操作 (口座間の転送、座席予約、株売買): 複数のデータの更新を必要とする

- これらをトランザクションとして原子的に実行することですべての更新がなされるか何もなされないかいずれかであることを保証

1.3.2 Handling Real-World Operations

□ トランザクションは実行の過程でユーザに返す結果を生成するかも

- トランザクションは全てかゼロか、なので commit するまでユーザに提示したものは本当ではない (アボートしたらすべてチャラ)
- トランザクションがユーザに表示したものは commit した後でなければ使用可能でない → ユーザも注意する必要 (図 1.4)
- トランザクションが commit 前にデータの一部を表示し、それをユーザが見て何かに入力してしまったら問題

□ システムによっては commit まで結果を見せないことで解決しようとしている (図 1.5)

- でも、commit したけれどその直後 (結果を表示する前) にクラッシュしたらどうなのか? 更新は実行したが結果は表示されていない
- 具体例 (図 1.6): ATM で$100 引き出す
- $100 を出してから commit だと、出した直後に abort したら銀行はお金を損する
- commit してから$100 出すのだと、commit 直後にクラッシしたらユーザはお金を損する

□ いずれも「全てかゼロか」ではない…

□ 関連した問題: 各トランザクションが「1 回だけ」実行されること

- そのためには呼び手に「OK」を返さないといけない。返らないと呼び手はアボートしたのかと思って再実行してしまう
- しかし OK だけだと「OK が無い」ことをどう解釈すべきか困る (commit したけれど OK を返す前にクラッシュ?)
- 解決不能に思えるかも知れないが、永続キュー (4 章) で解決可能

1.3.3 Compensating Transactions

□ commit は取消不能 (commit したら abort はできない)

- でも人は間違える→ commit したトランザクションの効果を元に戻す別のトランザクション (補償トランザクション) を実行
- 間違ってお金を転送したら、逆に同額を転送するなど

□ 完全な補償は難しい場合もある

- たとえば、スプレーガンで間違った色を塗ったら (?!)、そしてその塗った位置がスプレーガンの届かないところに行ってしまっていたら (?!)
- 補償トランザクションは、エラーを DB に記録し、誰かに何とかしてというメールを送ること (?!)
- 実質、どのようなトランザクションも間違って実行されることがある

□ だからよくできた TP システムはすべてのトランザクションに対してそれを補償するトランザクションを持つべき

1.3.4 Multistep Business Process

□ ビジネス活動によっては 1 つのトランザクションとして実行しないものも

- 注文受け付けトランザクションはその注文を処理するトランザクションとは別のものであるのが普通
- 注文の受け付け・記録は単純→すぐ OK を返せる
- 注文の処理→複数のトランザクション (与信チェック、在庫確認、集荷、配送) を含み複雑・時間が必要

□ ビジネスプロセスが複数トランザクションであってもユーザは原子性を欲する→補償トランザクションが必要

- 例: 注文が受け付けられても、後のトランザクションが実行不能であれば、注文受け付けの補償トランザクションを実行
- この場合、顧客の不満をなだめる一般的な補償トランザクション→ お詫び+無償ギフトカード (+キャンセルまたは商品到着後通知の選択)

□ トランザクションミドルウェア → 複数ステップから成るビジネスプロセスの遂行をサポート

- 例: 複数ステップの遂行状況を追跡し、途中で遂行不能になったら補償トランザクションを動かすなど
- より詳しくは 5 章

1.3.5 Consistency

□ consistency(一貫性) --- トランザクションは DB の一貫性を損なわないこと

- つまりDBが一貫性のある状態でトランザクションを開始したら、終了した時も一貫性がある
- これは「内部一貫性」つまりDB自体が持つ性質

□ DBの整合性制約として典型的には次のものがある

- すべての主キーが一意的
- 参照整合性: あるフィールドに他のオブジェクトのキーを保持しているなら、そのオブジェクトもきちんと存在している
- 値の範囲の制約: 年齢が120未満、SSNがNULLでない、など

□ DB自体ではふつう維持できないような制約もある

- 各部門の支出がその部門の予算以下であること
- 社員の給与が職位によって決まった範囲内であること
- 社員の給与は降格されない限り減少しないこと

□ AIDと違い、Cはトランザクションプログラムと DBとの共同責任

- TPシステムはトランザクションのA、I、Dを保証し、これを用いてトランザクションプログラムがCを維持する(アプリの責任)
- だからACIDテストというのは厳密には言いすぎ

□ TPシステムから外れて実世界に関わるCもある

- 例: DB中の商品数が実際に棚にある商品数と一致 ← 物理世界での動きに依存(正しい入庫、出庫、記録)

1.3.6 Isolation

□ isolation(隔離) --- トランザクションを実行した効果が、そのトランザクションだけを単独で実行した場合と一致すること

- これを技術的用語で言えば「直列可能(serializable)」つまり複数のトランザクションが「順番に1つずつ」実行されるのと同等
- 古典的な例: $100しかない口座から2つのトランザクションが同時に$100引き出してしまう
- その場合の結果は順番にやったのと異る(直列可能でない)
- 隔離は原子性とは異る: 原子性は「途中でやめることはない」→上記の例も途中でやめていないが、結果は正しくない

□ 実行が直列可能 → トランザクションに相当するリクエストを発行するユーザはシステムが自分専用に動いているように思える

- もちろん、2つのトランザクションを順番に実行したら、その間には他人のがはさまることはあるが…
- 1つのトランザクションについては「自分が占有」に思える
- 実際にはもちろん「思える」だけで実は沢山並列に動いている(本当に順番だと効率が悪すぎる)

□ トランザクション群が直列可能であり、かつそれぞれがCを維持 →全体としてのトランザクション群もCを維持

- 各実行の結果は(直列可能により)順番に実行しているのと同等だから

□ 実際には内部でロックを使って隔離を実現 → これにより多数が並行に動いても順番であるかのように見える(6章で詳しく)

- DBに整合性機能があるから直列可能は重要でない、という誤解があるがそれは違う
- 先に述べたように、Cはアプリ側がかなり責任を持つ。それをトランザクションで囲んで直列可能にすることで全体としてCを維持

1.3.7 Durability

□ durability(耐久性) --- トランザクションが完了したら、その結果は安定記憶に格納されていて消えない(停電やシステムダウンがあっても残る)

- 安定記憶(stable storage)、ないしnonvolatile storagi(不揮発性記憶)、ないし persistent storage(恒久記憶) → 典型的には磁気ディスクだが、最近はSSDも
- 一旦commitしたらその後プログラムエラーやシステムダウンがあっても結果は保存されていて次に立ち上がった時にそのように修復

□ 耐久性が重要な理由: トランザクションが提供するサービスはユーザと企業との契約

- たとえば口座間である金額を移すトランザクションが「完了」といったらその後で何が起ころうと金額は移動した状態(法的合意)
- このことを安定記憶に記録することで、エラーがあってもそこを参照して正しくする

- さらに「長期間残る」という意味も。当座預金に N 円入っていてそれをほっておいたとして、数年後に見てもその N 円が入っているはず

□ 耐久性の実現方法: トランザクション実行時に、すべての操作をログファイルに追記していく

- commit のときにはそれまでのすべての記録がログに書かれたことを確認したのちに OK の返答を返す
- その後で (直後でもすこしゆっくりでも)DB 本体に書く
- その途上でエラーが起きた場合も、ログを順に見てコミットされた内容が DB に既に書かれているかチェックし、そうなっていなければ改めて書けばよい
- 回復処理が終った後は最新の commit 後の状態になっている