# データ管理
# Magic xpa


OUTPERFORM THE FUTURE™

本マニュアルに記載の内容は、将来予告なしに変更することがあります。これらの情報について MSE（Magic Software Enterprises Ltd.）および MSJ（Magic Software Japan K.K.）は、いかなる責任も負いません。

本マニュアルの内容につきましては、万全を期して作成していますが、万一誤りや不正確な記述があったとしても、MSE および MSJ はいかなる責任、債務も負いません。

MSE および MSJ は、この製品の商業価値や特定の用途に対する適合性の保証を含め、この製品に関する明示的、あるいは黙示的な保証は一切していません。

本マニュアルに記載のソフトウェアは、製品の使用許諾契約書に記載の条件に同意をされたライセンス所有者に対してのみ供給されるものです。同ライセンスの許可する条件のもとでのみ、使用または複製することが許されます。

当該ライセンスが特に許可している場合を除いては、いかなる媒体へも複製することはできません。ライセンス所有者自身の個人使用目的で行う場合を除き、MSE または MSJ の書面による事前の許可なしでは、いかなる条件下でも、本マニュアルのいかなる部分も、電子的、機械的、撮影、録音、その他のいかなる手段によっても、コピー、検索システムへの記憶、電送を行うことはできません。

サードパーティ各社商標の引用は、MSE および MSJ の製品に対するコンパチビリティに関しての情報提供のみを目的としてなされるものです。

本マニュアルにおいて、説明のためにサンプルとして引用されている会社名、製品名、住所、人物は、特に断り書きのないかぎり、すべて架空のものであり、実在のものについて言及するものではありません。

Magic は Magic Software Japan K.K. の登録商標です。
Magic xpa は Magic Software Enterprises Ltd. のイスラエルその他の国での商標または登録商標です。
Magic xpa Enterprise Studio、Magic xpa Enterprise Client、Magic xpa Enterprise Server および Magic xpa RIA Server は Magic Software Japan K.K. の商標です。
Pervasive.SQL® は Pervasive Software, Inc. の商標です。
IBM®, iSeries™, xSeries®, DB2® および WebSphere® は、IBM Corporation の商標または登録商標です。
Microsoft® および FrontPage® は、Microsoft Corporation の登録商標です。また、Windows™, WindowsNT™ および ActiveX ™ は Microsoft Corporation の商標です。
Oracle® は Oracle Corporation の登録商標です。
Linux® は Linus Torvalds の登録商標です。
GLOBEtrotter® と FLEXlm® は、Macrovision Corporation の登録商標です。

一般に、会社名、製品名は各社の商標または登録商標です。
MSE および MSJ は、本製品の使用またはその使用によってもたらされる結果に関する保証や告知は一切していません。この製品のもたらす結果およびパフォーマンスに関する危険性は、すべてユーザが責任を負うものとします。

この製品を使用した結果、または使用不可能な結果生じた間接的、偶発的、副次的な損害（営利損失、業務中断、業務情報の損失などの損害も含む）に関し、事前に損害の可能性が勧告されていた場合であっても、MSE および MSJ、その管理者、役員、従業員、代理人は、いかなる場合にも一切責任を負いません。



2016 年 9 月 7 日

OUTPERFORM THE FUTURE™

# 第 1 章 トランザクションとは？

*トランザクションとは何か ?なぜトランザクションを使用しなければならないのか ？*

トランザクションはデータバウンド・アプリケーションの開発には不可欠なものです。データベース環境でアプリケーション開発を行う際に重要なことは、データの一貫性を確保するためにトランザクションを最適に使用できることです。

「トランザクション」という言葉は、SQL アプリケーションについて論じる際に非常によく使われます。トランザクションは、アプリケーション全体を組み立てる際に役立つ不可欠な処理であり、「全体としてコミットされるかアボートされなければならない、一連のデータ修正処理から成る小さな作業単位」と定義されます。

つまり、処理全体は成功するか失敗するかのどちらかとなります。中途半端はありません。UPDATE、DELETE、そして INSERT といったいくつかの操作が一つの単位を作ります。すべての操作が成功した場合にのみこの論理単位が成功したことになります。

トランザクション処理は、ビジネスロジックの処理全体である場合と、ビジネスロジックの一部のより小さな単位である場合があります。

トランザクションは、読込／書込処理とは対照的に、セキュアな読込処理で使用することができます。読込トランザクションは、トランザクション内で読み込まれるデータが他のユーザによって修正されないことを保証します。

トランザクション処理の技術は、一時的なトランザクションファイルに自動的にトランザクションの更新の全てを記録します。トランザクションが完了した時だけ、このファイル内の更新は消去されます。つまり、すべての通常データベーステーブルに対する更新が正常に終了します。トランザクションに影響するテーブルで問題が発覚した場合、トランザクション全体はキャンセルされます。そして、トランザクションが発生する前までデータベースはロールバックされ、元の状態に戻ります。ロールバックは、プロジェクトを復旧させるためにトランザクションログファイルで保存される情報を使用します。

## なぜトランザクションが必要なのか

すべてのアプリケーション開発者は、データベースの一貫性を確保するという同じ目標を持っています。データベースの一貫性を確保するためにトランザクションを使用しなければ問題が起こるでしょう。これは次のような例で説明できます。

フレッドというユーザが、普通預金口座から当座預金口座へ 50 ドル振替したいとします。この場合計算は単純です。

- 普通預金 = 普通預金 -50
- 当座預金 = 当座預金 + 50

二つの操作の間で何らかの失敗があったらどうなるでしょう？預金口座からは 50 ドル減りますが、当座預金は更新されなかったとします。フレッドは 50 ドル足りなくなってしまい、銀行の資金もなくなってしまっています。この例ではデータの一貫性がひどく損なわれてしまっているといえます。

トランザクションを使用して二つの操作が必ず一つの論理単位として実行されるようになっていたら、この単純な処理の間に失敗があったとしても処理全体がアボート（SQL 言語でいうロールバック）され、一貫性は保持されていたでしょう。そうすれば、口座のお金もトランザクションを開始する前の金額に戻っていたでしょう。

ここでもう一つ明白なことを追加すべきでしょう。つまり、上の例では二つの操作しかないように見えますが、実際にはあと二つの操作があったのです。

- 普通預金の読み込み
- 当座預金の読み込み

これはデータベースに対する実際の変更を考慮していません。表面上単純に見える処理も見た目ほど単純ではないのです。

このように、トランザクションを使用して開発することはデータバウンド・アプリケーションで必要なことなのです。

## マルチユーザ環境ではどうでしょうか？



二人のユーザが同時に同じデータにアクセスすることが可能なマルチユーザのアプリケーションを開発する場合、トランザクションを使用して開発する必要性が更に増します。

**Magic xpa のトランザクション：**

Magic xpa では、［タスク特性］の［データ］タブの［トランザクションモード］特性と［トランザクション開始］特性を設定することによって、任意のタスク実行レベルでトランザクション処理を実行させることができます。

同じシナリオでは、フレッドは残高の 50 ドルすべてを普通預金口座から当座預金口座に振替しようとしています。しかし、彼の妻のウィルマは、今週分の食料雑貨を購入するために銀行の別の支店で当座預金口座から 50 ドルを引き出そうとしています。

このとき何が起きるでしょうか？シナリオは以下のように様には単純でありません。

| フレッド | ウィルマ |
|---|---|
| 普通預金口座を確認 | |
| 普通預金 = 普通預金 -50 | |
| 当座預金口座を確認 | |
| 当座預金 = 当座預金 +50 | 当座預金口座を確認 |

フレッドが普通預金から当座預金へお金を振替する途中で、ウィルマが現金を引き出すために当座預金の残高を確認したとします。この場合、フレッド側では、普通預金の処理のみが完了し当座預金への入金が完了していないので、当座預金の残金はゼロです。そこで、ウィルマが普通預金の残高を照会すると、そこにもお金は残っていません。ウィルマから見ると、データの整合性が失われてしまっていることになります。

トランザクションのある世界ではこの例は少し違ったものになります。

処理が一つの論理単位として実行される場合、フレッドが行ったデータ更新は、トランザクションがコミットされるまで、ウィルマからは見えません。また、更新されたすべてのデータはトランザクションが完了するまでロックされます。

この場合、ウィルマが口座の残金を見ると、当座預金はゼロですが、普通預金の残高を調べるとそこにお金が残っているので、データの整合性は保たれています。

トランザクション内ではロックが積み重なってしまうことに注意してください。例えば、あるレコード（結果としてロックされる）が更新された後、次のレコードを更新したとします。これらのレコードは両方ともトランザクションの一部であり、トランザクションがコミットされるかロールバックされるまでロックされています。

**Rollback 関数：**

Rollback 関数は、指定されたネストレベルにトランザクションをロールバックするか、完全にトランザクションを破棄する場合に使用することができます。Rollback 関数は、ロールバック地点に対するネストレベルの値をパラメータとして指定することができます。パラメータの値は、オープンしたネストトランザクションの値でなければなりません。パラメータとして「0」を指定した場合、全てのトランザクションを最初の位置にロールバックします。ロールバックを強制した後、Magic xpa は［エラー動作］特性で定義されたロールバック・レベルにもとづいて処理を再開します。

実際のロールバック処理が開始される前に、Rollback 関数によって［確認］ダイアログを表示させることができます。

ウィルマがフレッドに電話する代わりに、彼女がお金を振り替えようと決心したと仮定します。しかし、このようなことが起こる頃には、フレッドのトランザクションは終了（コミット）されます。次に、この場合、彼女が預金口座の現在の残高をチェックしようとするとき、金額はすでに更新されます。彼女は、当座預金からお金を運良く引き出すことができます。

このような特殊な状況で実際に起こることは、使用している DBMS によって変わってきますが、ここでの目的はトランザクションの簡単な概要を示すことです。

## いいことばかりではありません。



データバウンド・アプリケーションを開発するには、トランザクションを使用して作業するしかありません。しかし、そのためには、開発者はトランザクションを使用する際の落とし穴について知らなければなりません。

フレッドとウィルマに関する問題に戻りましょう。ウィルマは、間違った情報を受けました。トランザクションの場合、ウィルマはトランザクションがまったく実行されなかったという情報を受けました。実際のところ RDBMS では、トランザクションが終了するまで待たなければなりません。これは、デッドロックとして知られていることです。マルチユーザ環境では、デッドロックはトランザクションによる障害の一般的な原因になります。

デッドロックが発生した場合、データベース管理システム（DBMS）によって動作が異なります。多くの DBMS システムは、デッドロックを見つけて、警告を表示させることができます。この場合、Magic xpa はトランザクションをロールバックして、トランザクションに対するエラー動作の設定にもとづいて実行を継続します。

ISAM 系のデータベースはデッドロック状態でハングアップするかもしれないため、Magic xpa はデッドロック防止機能を ISAM と SQL データベースに提供しています。デッドロック防止機能を作動させるには、［環境設定］ダイアログの［デッドロック防止］設定を「Yes」に設定してください。デッドロック防止機能は、トランザクション中、書込アクセスでオープンしている全てのテーブルを定義順に排他的にロックします。一時テーブルは、ロックされません。

トランザクションとデータベースの詳細については、第 7 章「付録」を参照してください。

これまでの場合、簡単な状況でした。この状況を少し変えてみましょう。トランザクションの副産物のひとつとして、修正されたデータがロックされるということについて検討しました。そこで、フレッドがトランザクションを行おうとしているときのシナリオを考えてみましょう。

銀行員は普通預金口座の詳細を画面に出して、すぐにトランザクションを開始しようと思い、修正作業に入ります。ここで、口座のデータがロックされてしまいます。このとき、突然電話が鳴り銀行員が電話に出ます。データはロックされたままトランザクションの完了を待ちます。そして、ウィルマはフレッドが当座預金への振り替えを行っていないことに気づき、自分で行おうと思います（後で夫に報告するつもりで）。しかし、データはロックされたままなので、彼女はフレッドのトランザクションが完了するまで待たなければなりません。ロックについては分離レベルについてのセクションで詳細に説明します。

この非常に単純化した状況は、思ったほど単純ではないかもしれません。二人がそれぞれのトランザクションを実行しようとしている瞬間に、誰かが彼らの小切手を現金化しようとしているかもしれません。または、銀行の支店長が口座についての報告書を作成しようとしていて、たまたまフレッドとウィルマの口座がそこに含まれているかもしれません。

ここで明白なことは、データベースの一貫性は維持されてはいるものの、アプリケーションは時間に依存する（利用者が長い間待たされることがある）ということです。つまり、一人のユーザがデータを更新している間、同じデータを使用する必要のある他の人は順番を待つか、古いデータを使用しなければならないことを意味しています。これはマルチユーザ環境ではまったく正しい動作なのですが、電話中の銀行員のようにあるユーザが実行を遮っている場合、利用者は不便を強いられることになります。

ここで解説したシナリオは、並列性です。

**定義：**

二人以上のユーザが同じデータを同時にアクセスしている状態を同時並行性といいます。

アプリケーションは、データの一貫性を保持しつつ、同時並行性を最大限に高めるような形で開発されるべきです。言い換えれば、できるだけ多くのユーザが同じデータを同時にアクセスできるようにするということです。

しかし、実際には両立させるのが難しい課題です。データの一貫性を向上させようとするとき、しばしばトランザクションが長くなって他のユーザが影響を受けてしまうことがあります。前述したように、ロックはトランザクションの中で積み重なります。トランザクションが長いと、全ての修正データはトランザクションが解除されるまでロックされます。他のユーザがそのデータにアクセスしようとすると問題になります。これについては次のセクションで詳しく説明します。

## 分離レベル



ここではデータベースの分離レベルについて少し触れたいと思います。この章の中で間接的には説明してきましたが、まずある種の定義が必要でしょう。

処理の分離レベルとは、ある処理によって読み込まれたり更新されたりしている行（レコード）が、他の同時実行している処理からどの程度利用可能かを指定するものです。

あるトランザクションが値を変更することができる状況があります。そして、その変更が実際にコミットされたりロールバックされる前に別のトランザクションがその変更された値を読むことができます。このような状況は「ダーティリード」と呼ばれています。

これをフレッドとウィルマの話に置き換えてみます。フレッドは 50 ドルを普通預金口座から当座預金口座に振替するトランザクション中です。このトランザクションには 2 つのステップがあります。つまり、普通預金口座を 50 ドル減らすことと、当座預金口座を 50 ドル増やすことです。

ウィルマは当座預金口座を確認しましたが、50 ドルはまだありません。銀行員はまだ電話中です。そして、普通預金口座を確認すると残高は 0 ドルです。フレッドがトランザクションを終了する前に、ウィルマは彼のトランザクション処理中の預金口座をチェックします。

ここでフレッドがトランザクションをアボートすると問題が発生するかもしれません。すべての変更が最初の状態に戻ってしまいます。すなわち、普通預金の口座残高は、トランザクション前の値、すなわち 50 に戻されます。残念ながらウィルマはフレッドが途中でトランザクションをアボートしたことがわかりません。フレッドがトランザクションをアボートしたことがわかるためには、データを読み直さなければなりません。これからわかるように、ダーティリードは同時並行性を向上させますが、データの整合性を低下させます。

## ロックされたレコード

この章の中でロックされたレコード、つまり SQL の世界でいう行について説明しました。あるレコードがロックされるとき、他のユーザが実際にどんなデータを見るかは、定義されている分離レベルと使用している RDBMS によって変わってきますが、可能性としては次の 3 つがあります。

- データはロックされます。2 番目のユーザはメッセージを受け取り、データが開放されるまで待たなければなりません。Magic xpa では、ユーザは「レコードロックの解除待ち」というメッセージを受け取ります。
- 表示されるデータは修正済みのデータとなります（「ダーティリード」の状態）。
- 表示されるデータはロックが開始される以前のデータになります。つまり、トランザクションが開始される前のデータです（いわゆる「前イメージ」）。

このセクションでは分離レベルについて少し触れました。実際には 4 つの分離レベルがあり、処理が互いにどのように影響し合うかを定義できます。分離レベルについてのより詳細な情報については、各 RDBMS のリファレンスガイドを参照してください。

# 第 2 章 Magic キャッシュ

*Magic キャッシュとは何でしょうか ?*

キャッシュ機能の必要性の基本的な背景は、同じデータを何度も利用する場合があるということです。したがって、ディスクの入出力処理を行うことなくデータの再読み込みを行うことができれば、全体的なパフォーマンスが向上します。キャッシュ機能のための一般的なアルゴリズムを使用する汎用的なディスクキャッシュとは異なり、Magic xpa のキャッシュ機能は、物理データに関する知識とプロジェクトの特性に基づいて、また、プロジェクト内でのユーザの意向に基づいてチューニングされます。

Magic キャッシュは、他のハードウェアまたはソフトウェアのキャッシュ機能のように、ファイルマネージャをを呼び出すことができません。したがって、Magic キャッシュは、従来の方法で追加されるファイルマネージャのオーバーヘッドを回避します。

## 何がキャッシュされるのですか？

Magic xpa のキャッシュ機能は、メインソースとリンクテーブルで使用されます。Magic xpa キャッシュは、タスクに実装され、原則としてメインソース／リンクテーブルに対して実行されます。これは、親タスクとサブタスクが同じテーブルをアクセスしている場合、両方のタスクが使用するテーブルをキャッシュに格納することができることを意味しています。この場合 2 つのキャッシュが使用されます。同じことは、タスク内に同じテーブルを別の［リンク］コマンドで読み込む場合も適用されます。この場合も、2 つのキャッシュが使用されます。

## キャッシュされた内容は、いつ使用されるのですか？

Magic xpa がレコードに対してロックを発行させる必要がない場合のみ、データをキャッシュから読み込むことができます。ロックの必要性は、3 つの要因に基づいて決定されます。

- **ロック方式**……［タスク特性］ダイアログの［データ］タブで定義します。
- **キャッシュ範囲**……［タスク特性］ダイアログの［データ］タブで定義します。
- **オープンされるテーブルの［アクセス］特性と［共有モード］特性**……［メインソース］特性で定義します。

ロックは、［アクセス］特性と［共有］特性が「書込」に設定されたテーブルをオープンした場合にのみ発生します。ロックのタイミングは、［ロック方式］特性で定義されます。

照会モードから修正モードへの切り替えや、修正モードから照会モードへの切り替えは、キャッシュを使用して実行されます。モードの切り替え時は、ビュー内のレコードはリフレッシュされません。

ソート処理は自動的にソートファイルに対するキャッシュを作成します。

登録モードでアクセスする場合、あるいは、位置付や範囲指定、ソート処理を実行すると、キャッシュは解放されます。

修正モードと照会モードを切り換えることで、Magic xpa はキャッシュをアクセスします。そして、データベースからデータを再読み込みしないようにします。

## キャッシュサイズの有効化

リンクテーブルのキャッシュを無効にしたり有効にしたりすることができます。この定義は、［リンク］コマンドの［キャッシュ］特性で行います。［キャッシュ］特性を「No」に設定すると、タスク内でのリンクテーブルに対するキャッシュが無効になります。「Yes」に設定されると、テーブルキャッシュが使用されます。

デフォルト値は、「Yes」です。

## プログラム動作の変更

Magic キャッシュを使用するとプログラムの動作も変わります。データがキャッシュ内に見つかった場合、データの内容がデータソースのものと同じでない場合があります。しかし、Magic xpa は、データを更新する前に常にディスクからレコードを読み込みます。

例えば、APG によって作成される簡単なオンラインプログラムを考えてください。挿入位置が 1 レコードから別のレコードに移動した場合は常に、新しいレコードのデータビューが自動的に読み込まれます。

別のユーザがデータを変更した場合、(データがキャッシュにある場合) 変更されたレコードにパークしても変更内容は表示されません、しかし、データを修正しようとすると、現在の値がディスクから読み込まれ次のメッセージが表示されます。

**レコードは変更されました。再開します。**

[キャッシュ範囲] 特性で「位置とデータ」が定義されると、位置に加えて、レコード内の実際のデータも保存されます。レコードを再フェッチすると、キャッシュに保存された古い値を取得します。別のタスクから行われたこのデータに対する修正内容は、現在のタスクで参照することができません。別のタスクによってレコードを更新しようとすると、上記のメッセージが返されます。

プロジェクトの定義によっては、キャッシュの使用が許可されない場合があります。たとえば、価格テーブルの [共有] 特性を「書込」に設定して読み込むようにした受注入力プロジェクトを想定します。あるユーザが価格を変更し、項目の新しい価格がそのユーザのキャッシュに格納されている場合、他のいかなるユーザも新しい価格を参照することができません。

## キャッシュと常駐タスク

Magic キャッシュが常駐タスク内で割り当てられると、タスクを終了してもメニューに戻るまでキャッシュは終了されません。従って、タスクがリンクテーブルから同じデータを読み込むと、データの全てはキャッシュに格納されます。そして、タスクが 2 回目以降に呼び出されるとキャッシュが再利用されます。

## キャッシュとロールバック処理

ロールバック処理によって、トランザクション中に更新されたすべてのキャッシュは無効になります。無効化によって、すべてのデータがキャッシュから削除されます。Update や Delete、Insert 処理によるキャッシュのみが無効になります。

## Magic キャッシュの内部実装

Magic xpa が読み込むあらゆるレコードは、キャッシュに挿入されます。さらに、どんな修正、削除またはレコードの挿入処理もキャッシュに格納されます。すでに存在するレコードが再読み込みされると、前の値に置き換えられます。キャッシュの中に存在しないレコードが読まれると、自動的にキャッシュに格納されます。キャッシュがいっぱいになると、LRU (使用頻度の少なくなるものから置き換わる) アルゴリズムによってキャッシュから廃棄されます。

LRU メカニズムの使用により、頻繁に読みこまれるデータはキャッシュから削除される可能性が低くなります。

# 第 3 章 遅延トランザクション

*遅延トランザクションとは何か?他のトランザクションとはどう違うのか?*

遅延トランザクションは、文字通りトランザクションです。開発者は遅延トランザクションを使用するために新しいコンセプトを学ぶ必要はありません。しかし、開発者はこの新しいメカニズムをいつどのように使用すべきか、利点と起こり得る問題点、そして、それらに対応する方法を知っておくべきです。遅延トランザクションが Magic xpa の内部的な概念であることを知っておいてください。

この章と次の章では、これに取り組む方法について手短に扱うことにします。

## 遅延トランザクションの概念



前章のフレッドとウィルマの例で見たように、トランザクションはできるだけ短くすることが重要です。トランザクションを短くすることによってトランザクションによるロックも短くなり、他のユーザの待ち時間も短くなります。このために明らかに同時並行性が向上します。

遅延トランザクションを使用するときには、データベースに対するすべての変更内容が Magic xpa のキャッシュメモリに一旦保持さます。データベースに対する変更内容とは、INSERT、UPDATE、そして DELETE といったデータ操作ステートメントのことです。

この遅延トランザクションがコミットされると、キャッシュに保存されていたすべてのデータベースの変更内容が 1 つの短いトランザクションで RDBMS に書き込まれます。そして、物理トランザクションとして参照します。次に、実際のトランザクションは短くなります。この種のトランザクションを使用することにより、アプリケーションの並列性は、大いに向上します。遅延トランザクションの別の利点は、サーバ側の動作を減らし、アプリケーションとデータベース間の通信量を減らして、パフォーマンスを向上させることです。

フレッドとウィルマの状況を例に取って、これがどのように彼らの役に立つかを見てみましょう。フレッドは銀行員の前に立って銀行員の会話が終わるのを待っています。銀行員は遅延トランザクションとして実装されたデータを使用しています。そこへウィルマが来てお金を送ろうとしています。驚いたことに何の問題もありません。遅延トランザクションを使っているので、データがロックされていないからです。彼女もやはり遅延トランザクションとしてトランザクションを実行します。データベースは更新されます。当座預金口座にお金が送られ、彼女がもともと引き出したかったお金を引き出すことができます。その後、彼女はフレッドに電話で連絡することでしょう。

ここで見られるのは遅延トランザクションを使用する主な利点です。実際の物理トランザクションはずっと短くなり、データのロック期間も短くなります。ウィルマも電話中の銀行員の影響を受けずに済みます。

## すべてがうまくいくでしょうか?

この新しいトランザクションのメカニズムが同時並行性の問題を解決するなら、我々は皆これを使用してトランザクションを最小に保つべきでしょう。しかし、一つ問題があります。フレッドは電話中の銀行員の前に立っています。ある段階で銀行員は電話を切り、自分が開始したトランザクションを完了しようとします。彼は今トランザクションを完了しようとしますが、ウィルマは既に彼女のトランザクションを完了しています。つまり、ウィルマは既に残高を更新したのです。彼が着手したときの金額は、現在の金額ではありませんので、そのまま更新を行なうと誤った結果となってしまいます。このような問題は「更新の喪失」と呼ばれることがあります。

ここでは何をすべきでしょうか?既にデータが更新されたというメッセージを銀行員が受け取るようにするのでしょうか?それとも銀行員のトランザクションを完了させますか?

## 更新レコードの識別



今挙げた「更新の喪失」の問題をどのようにして打開することができるでしょうか?問題をもう一度見てみましょう。フレッドは普通預金口座からお金を引き出そうとしていますが、ウィルマが金額を更新してしまいました。このような状況に対応するには、更新の前に、元のデータを確認のためにチェックすることが必要になります。

例えば、今度のシナリオでは、二人が普通預金口座に 100 ドル持っていたと仮定します。そのうちフレッドは 50 ドルだけを振替します。トランザクションの後、普通預金口座の残高は 50 ドルになります。ウィルマの方は、シナリオを少し変えて、40 ドルを引き落とすことにします。

フレッドとウィルマが実行する SQL コマンドを見てみましょう。

**ウィルマ :** UPDATE savings SET balance = 60 WHERE id=300

**フレッド :** UPDATE savings SET balance = 50 WHERE id=300

フレッドがトランザクションを完了する前にウィルマがトランザクションを実行した（フレッドはトランザクションを完了していない。）場合、普通預金口座の残高は 50 ドルです。ウィルマのトランザクションは失われます。「更新の喪失」です。

どうすればこれを回避し調整できるでしょうか？ Magic xpa では、［更新レコードの識別］という特性の設定により、更新の喪失が起こらないようにさせることができます。

なぜこのような状況になったのかを考えてみましょう。原因は我々が使用した WHERE id=300 という WHERE 句にあります。これがレコードの位置、つまりユニークな識別子となっていました。しかし、WHERE 句で id=300 としただけでは、レコードが更新されたかどうかを知るための十分な情報がありません。もっと情報が必要です。

**更新レコードの識別：**

このデータ特性は、［データ］リポジトリとタスクの［メインソース］特性、［リンク］特性で定義することができます。ここには次のようなオプションがあります。

- 位置
- 位置と選択項目
- 位置と更新項目（デフォルト）
- データソースに依存（［メインソース］特性と［リンク］特性のみ）

この特性は、［トランザクションモード］特性が「遅延」または「ネスト遅延」、「有効なトランザクション内」の場合のみ有効で、実行時に遅延トランザクションモードが評価された場合に動作します。

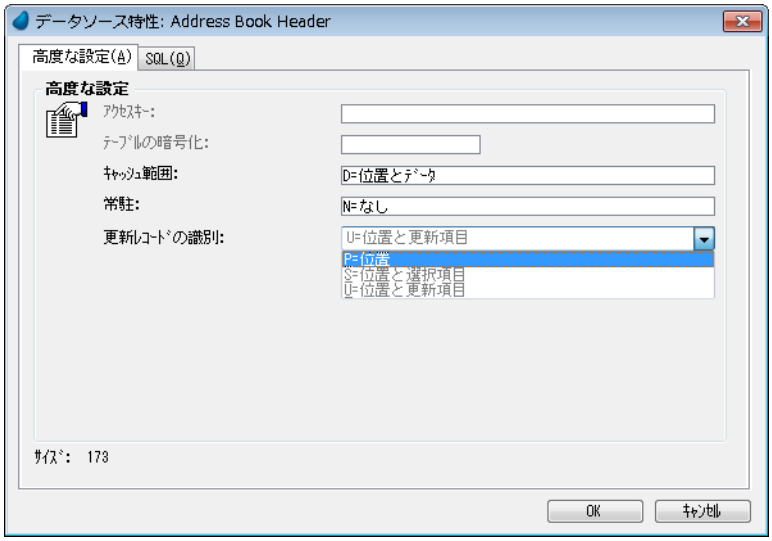



図 3-1 更新レコードの識別

この特性が、「更新の喪失」の問題に対してどのように役立つかを見てみましょう。

上記で示した例は、この特性が「位置」に設定されている場合に発生します。すなわち、更新するレコードを識別するために、id だけが指定され、WHERE id = 300 という WHERE 句となります。

このような指定は、どのような状況で有効でしょうか？もう一度フレッドとウィルマの夫婦の話に戻ってみましょう。彼らが銀行の口座情報を確認していて、自宅の住所の郵便番号が間違っていることに気づいたとします。

彼らは、口座振替を行っている間に郵便番号を変更してもらうよう銀行員に頼むことに決めます。このシナリオではフレッドとウィルマの両方が同時にトランザクションを実行していても、更新後の郵便番号は同じなので、誰が更新するかは重要ではありません。また、フレッドとウィルマの両方が同じ更新を行っても、何も害はありません。この場合は、「位置」のオプションで十分です。

更新レコードの識別の 2 番目のオプション、「位置と選択項目」を設定すると、プログラムで選択されているすべての項目を UPDATE 文の WHERE 句に追加します。

それではこれはフレッドとウィルマにとって何を意味するのでしょうか？残高を表示し普通預金から振替を行うのに使用されているプログラムが、銀行員に次の情報しか表示しないことにしましょう。

- 口座番号
- 口座残高

それから振替金額のための非データベース項目と振替先口座も表示します（下図参照）。


それで結果の WHERE 句には口座番号と口座残高の両方が追加されます。

下のフレッドとウィルマの SQL コマンドを見てみましょう（振替の前彼らは普通預金口座に 100 ドル持っていたことを思い出してください）。

**ウィルマ：** UPDATE savings SET balance = 60 WHERE id=300 and balance = 100

**フレッド：** UPDATE savings SET balance = 50 WHERE id=300 and balance = 100

ここで何が起こるでしょうか？フレッドのトランザクションは開始されますが、銀行員の電話のせいで止められてしまいます。銀行員が電話中の間ウィルマはトランザクションを行って残高を 60 ドルに減らします。それでフレッドがトランザクションを完了しようとすると残高はもう 100 ドルではないため失敗します。データの一貫性は維持されています。これは非常に効率的な更新の確認方法です。

我々が使用した例では理想的な解決方法のように思えますが、本当にそうでしょうか？それでは同じプログラムで口座の詳細情報を追加してみましょう。これらの詳細情報はその口座が誰のものかを示すもので、以下のような情報が表示されます。

- 口座番号（Account number）
- 名前、姓 （Name）
- 住所（Address）
- 住所（City）
- 郵便番号
- 電話番号（Telephone）
- 口座残高 （Current balance）


この場合、ウィルマの SQL コマンドは次のようになります。

> UPDATE savings SET balance = 60 WHERE id=300 AND first_name = “Fred and Wilma” AND surname = “Smith” AND address = “10 Upping Avenue” AND city = “Londera” AND zip = 65232 AND telnum = “555-2055” AND balance = 100

他のユーザが上のデータのどれかを更新すると、トランザクションは失敗してしまいます。

これは非常に精密ですが、すべての場合にここまですることが必要でしょうか？もう一度シナリオを見てみましょう。ウィルマが振替を行っている間、フレッドは郵便番号を 65233 に更新していました。位置と選択項目オプションが使用されていると何が起こるでしょうか？「zip = 65232」ではなくなったのでトランザクションは失敗してしまうのです。残高の更新と郵便番号の更新 はお互いに干渉することがないので、これは必要以上に厳しい制限を課していることになります。

どうしたら良いのでしょうか？そこで 3 番目のオプション、位置と更新項目です。このオプションは更新項目だけを WHERE 句に追加します。それでこの場合には、ウィルマは下のような SQL コマンドを実行することになります。

> UPDATE savings SET balance = 60WHERE id=300 and balance = 100

二つの項目しか持たないプログラムと同じです。

このようにすれば、フレッドがウィルマのトランザクションを妨げずに郵便番号を更新できるので、同時並行性が向上します。一方、二人が残高を同時に更新しようとすると失敗しますから、データの一貫性のチェックも行われることになります。

このように Magic xpa では、「更新レコードの識別」パラメータを適当に設定することにより、必要に応じたレベルで、データの一貫性と同時並行性とを両立されることができます。

## ネストトランザクションは可能？

次に、ネストトランザクション、すなわち、トランザクション内のトランザクションとは何か考えて見ましょう。簡潔に言うと、オープンされたトランザクションがあるときに、そのトランザクションが完了する前にコマンドをデータベースにコミットするような内部のトランザクションを開発者が持ちたい場合です。このトランザクションは他のトランザクションから独立しています。

我々のシナリオをもう一度見てみましょう。ウィルマは普通預金口座から振替しようとしていますが、その間同時に彼女は郵便番号も更新したいと思っています。郵便番号の更新は、振替とは独立した別のトランザクションとして実行するほうが自然です。Magic xpa では、このメカニズムはネスト遅延トランザクションと呼ばれています。上の例では、トランザクションモードをネスト遅延に定義した別のプログラムかタスクを準備して口座の郵便番号情報を更新させることになります。ネストトランザクションの詳細は、第 7 章「付録」を参照してください。

**注意：**

ネストトランザクションでは、内部のトランザクションがコミットされるとそのまま RDBMS にコミットされます。その後、外部のトランザクションがロールバックされたとしても、内部のトランザクションはロールバックできません。両者のトランザクションは独立しており、無関係です。

この章では遅延トランザクションの概念と、トランザクション処理において起こり得る問題が、遅延トランザクションによってどのように解決されるかを見てきました。

遅延トランザクションの詳細は、第 7 章「付録」を参照してください。

# 第 4 章 その他の利点

*遅延トランザクションには他にどのような利点があるでしょうか？*

我々は遅延トランザクションについて、その使い方やそれによってマルチユーザ環境での同時並行性が如何に向上するかを学んできました。

既に説明した利点の他に、Magic 開発者が遅延トランザクションを使用する場合に便利な他の機能があります。

## データの差分更新

次のシナリオを考えて見ましょう。ウィルマが残高を更新しようとしているシナリオを覚えていますか？指定した値、すなわち 60 ドル（ウィルマによって引き出されるため、100-40 ドル）で、残高を更新しました。これは、絶対値として参照します。残高は、固定値で更新されます。

このときには、次のような SQL コマンドが実行されます。

```
UPDATE savings SET balance = 60 WHERE id=300
```

残高は 60 ドルで更新されました。次に、別の観点で見てみましょう。このトランザクションの他の要素を忘れています。当座預金です。ウィルマが振替を行う前に、100 ドルがこのアカウントにもあったと仮定しましょう。次に、振替によって両方がどのようになるか見てみましょう。

```
UPDATE savings SET balance = 60 WHERE id=300

UPDATE checking SET balance = 140 WHERE id=600
```

当座預金口座の残高は 40 ドルという一定の値だけ増えています。これは普通預金口座から振替された金額です。

ところで、ウィルマが作った 30 ドルの小切手を現金化しようとしている人がいたとします。その人のために、次のような SQL コマンドが発行されます。

```
UPDATE checking SET balance = 70 WHERE id=600
```

ところが、これらのトランザクションが同時に実行されると「更新の喪失」の問題が起こってしまいます。

この状況で、データ一貫性を保ったまま、同時並行性を最大限にするためにはどうしたら良いでしょうか？

ここで新しい特性、［更新形式］が登場します。

**数値型項目の更新形式：**

この特性は、［データ］リポジトリにおいて数値型カラムの［カラム特性］で定義することができます。次のオプションがあります。

- A= 値更新
- D= 差分更新

デフォルトは A= 値更新です。

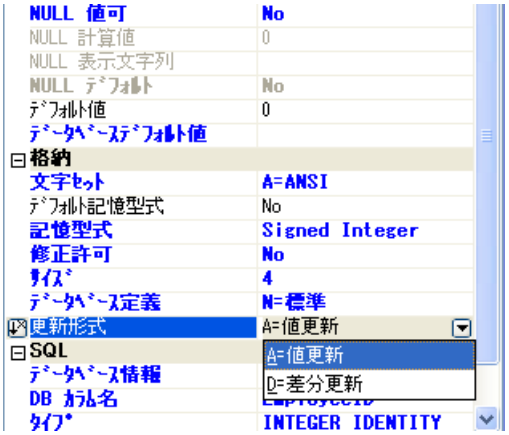



図 4-1 ［更新形式］特性

この特性がどのようにして我々のジレンマの役に立つのでしょうか？［更新形式］特性が「D= 差分更新」に設定されると、絶対値、つまり固定値が設定されるのではなく、もとの値と新しい値との差分が SQL 文で使用されるようになります。

我々の例でいえばもっと理解できるでしょう。［更新形式］特性がデフォルトの「A= 値更新」の場合には、次のような SQL 文が発行されます。

**ウィルマ：** UPDATE checking SET balance = 140 WHERE id=600

**小切手の現金化：** UPDATE checking SET balance = 70 WHERE id=600

一方、［更新形式］特性が「D= 差分更新」に設定された場合には、次のようになります。

**ウィルマ：** UPDATE checking SET balance = balance + 40 WHERE id=600

**小切手の現金化：** UPDATE checkin SET balance = balance - 30 WHERE id=600

では結果はどうなるでしょうか？この場合、どちらの操作が先に行われるかは重要ではありません。結果として当座預金口座の残高は 110 ドルとなります。これは正しい結果です。

このように、Magic xpa では差分更新の機能を用いることで、高いデータの一貫性と同時並行性とを得られます。

## 参照整合性 - 親子間の状況

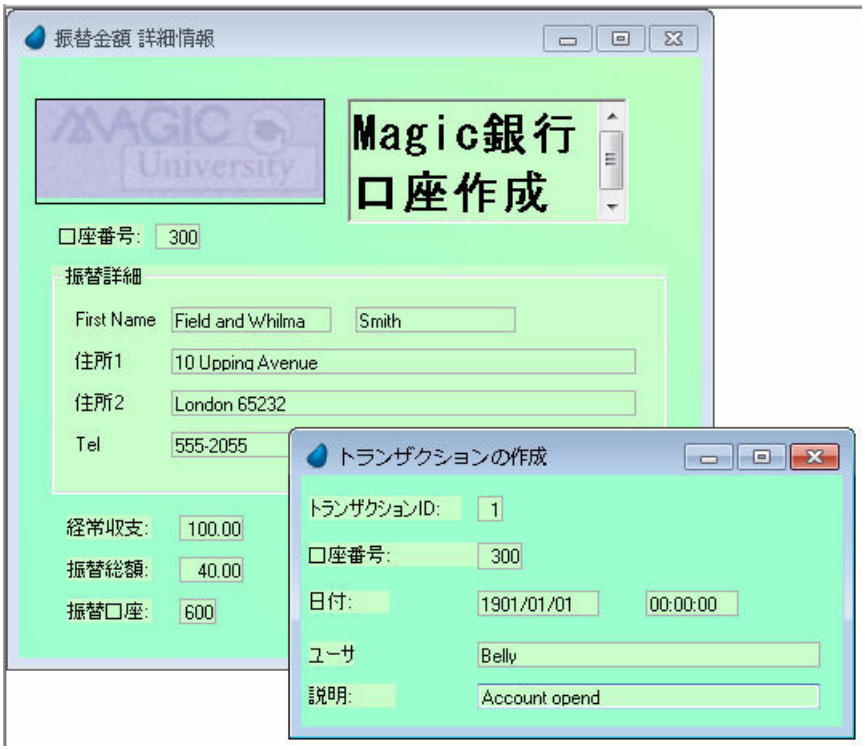
代表的な RDBMS では、受注ヘッダと受注明細の関係のようにテーブル同士が関連付けて定義されることがあります。これらのテーブルが一緒にリンクされるようにデータベースのルールが作成されます。遅延トランザクションは、特定の状況で有利になります。ここで起こりうる問題点と、その解決方法について、我々が使用してきた銀行のアプリケーションを例に説明します。

普通預金口座に対して、金銭取引をすべて記録した履歴ファイルを持つ必要があります。銀行の残高に不審があるときに、どのようにしてその金額になったかの履歴が保存されていないのでは、誰も銀行を使わなくなってしまうでしょう。

この履歴ファイルは口座ファイルにリンクされたファイルです。口座ファイルに対応する口座番号のない履歴レコードにはどんな意味があるでしょうか？履歴ファイルに口座番号 921 の履歴があったとします。しかし、口座テーブルに口座 921 がなかったとしたらその履歴は意味のないものになってしまいます。

これは親子のテーブル関係となります。RDBMS で参照整合性を使用してこのような関係を定義すると、履歴テーブルにはデータがあるが口座テーブルには対応するデータがない、といった状況にならないように RDBMS がチェックしてくれます。同様にリンクするデータが存在しているのに親のデータ（口座）を削除してしまうことがないようにしてくれます。

これを今まで慣れ親しんできたシナリオの中で見てみましょう。ここでは、またシナリオを変えて、フレッドとウィルマが娘のために普通預金口座を開設しようとしていることにします。この銀行のデータベースでは、口座開設時に「口座開設」というレコードを履歴ファイルに登録する必要があります。




ここで何が起こるでしょうか？口座を開設するプログラムは、トランザクションを発行するプログラムを呼び出します。結果の SQL コマンド（日付と時間の変換については無視します。）を見てみましょう。

INSERT INTO history (transid, id, transdate, transtime, clerk, desc) VALUES (5, 301, '2004-01-01','085722','Betty','Account opened')

INSERT INTO savings (id, first_name, surname, address, city, zip, telnum) VALUES (301, 'Stones', 'Smith', '10 Upping Avenue', 'Londera', 65232', '555 2055')

ここで問題があります。履歴ファイル（history）への入力が普通預金ファイル（savings）の入力の前にデータベースに書き込まれようとしているのです。これは参照整合性制約に引っかかってしまいますので、トランザクションが失敗します。

それではこの問題の解決策は何でしょうか？解決策は明白です。履歴レコードは、普通預金レコードがデータベースに書き込まれた後に書き込まれなければなりません。これをアプリケーションでどのように実現するかについては問題が残ります。

Magic xpa の遅延トランザクションでは、すべてのデータ操作コマンドはデータベース自体にコミットされる前にトランザクションキャッシュに保持されるということが利用できます。このための新しい特性、［同期］の使用方法を紹介します。

**同期：**

この特性はコール（タスク、プログラム、公開）特性に含まれています。

式が利用可能な論理値で指定します。この特性が True と評価されると、子レコードを書きこむ前に親プログラムのレコードを書き込みます。

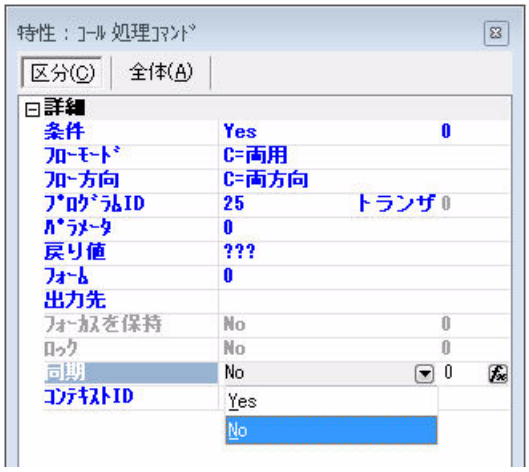



図 4-2［同期］特性

我々のシナリオで、これがどのように役立つでしょうか？［同期］特性を「True」に設定することによって、口座テーブルに新しいデータを追加するためにレコードを書き込もうとしている親プログラム「口座開設」のレコードは、二つ目のプログラム「トランザクション発行」のレコードより先に物理データベースに実際に書き込まれます。このため、参照制約に引っかかることなく、レコードを追加できます。

この機能は、Magic xpa がすべての操作情報をキャッシュの中に格納して最終的な OK を待つために可能になるのです。

# 第 5 章 エラー処理

*このエラーは何を意味しているのか ?私のプログラムにはエラーなんかない!*

有名なことわざで、「To err is human（過ちは人の常）」というものがあります。「しかし大きな過ちにはコンピュータが要る」、ということも皆知っています。しかし、ここではプログラム的な過ち（バグ）のことを言っているわけではありません。ここで言っているのはデータベースのエラーのことです。このエラーは一般的にデータベースの制約に反するようなことを実行しようとした結果として起こります。これはどういう意味でしょうか？参照整合性のことについて述べたのを覚えているでしょうか？トランザクション履歴テーブルを持つ口座テーブルがありました。もし我々が、履歴テーブルに口座に対応するレコードがあるのに口座テーブルからレコードを削除しようとしたら、RDBMS に「おい、それはダメだ。」と止められます。最初に履歴テーブルからレコードを削除して、それから口座テーブルのレコードを削除することができます。

## エラー処理の概念

Magic xpa には、開発者からエラー処理と言う頭痛の種を取り除いてくれるエラー処理の機能があります。Magic xpa によって処理されるデータベースエラーには、いくつかのレベルがあります。一つ目のもっとも重要なレベルはエンジン自体、または DBMS に対応する Magic xpa の内部ゲートウェイによって処理されます。

娘の普通預金口座を開設しようとしていたウィルマを例にとってみましょう。このアプリケーションでは、クライアントが利用可能な番号のリストから自分の希望の口座番号を選択できるようになっています。銀行員はこのリストをウィルマに見せ、彼女は 301 が使用できるとわかります。それで彼女はこの番号を選び、銀行員は詳細を入力します。しかし、彼らはマルチユーザ環境で作業しているため、他の誰かが既にその番号を使ってしまっていました。トランザクションがコミットされると銀行員はエラーメッセージを受け取ります。メッセージには、「インデックスが重複しています。テーブル名：accounts」とあります。これは Magic xpa が生成したエラーです。DBMS からの実際のエラーメッセージは DBMS によって異なります。例えば MS-SQL Server のエラーメッセージは、「一意インデックス 'trans' を持つオブジェクト 'history' には重複するキー行を挿入できません。」といったようになるでしょう。Oracle の場合はまったく異なります。Magic xpa では、すべてのデータベースでメッセージは同じになります。

さてこれはどのように役に立つでしょうか？ほとんどの場合はこれで十分です。しかし、いくつかのタスクを連続した長いトランザクションを行っており、そしてコミットした時にエラーを受け取ってしまったとします。この場合どうでしょうか？ユーザがエラーを訂正できるような専用の画面を用意するか、この場合でしたら新しい口座番号を発行したほうが親切でしょう。Magic xpa では開発者が独自に定義したエラー処理を提供することが可能です。

開発者が定義したエラー処理の機能は、Magic xpa 内部のイベント処理機能と同じように動作します。［イベント］ロジックユニットの［イベントタイプ］特性を「R= エラー」に設定します。

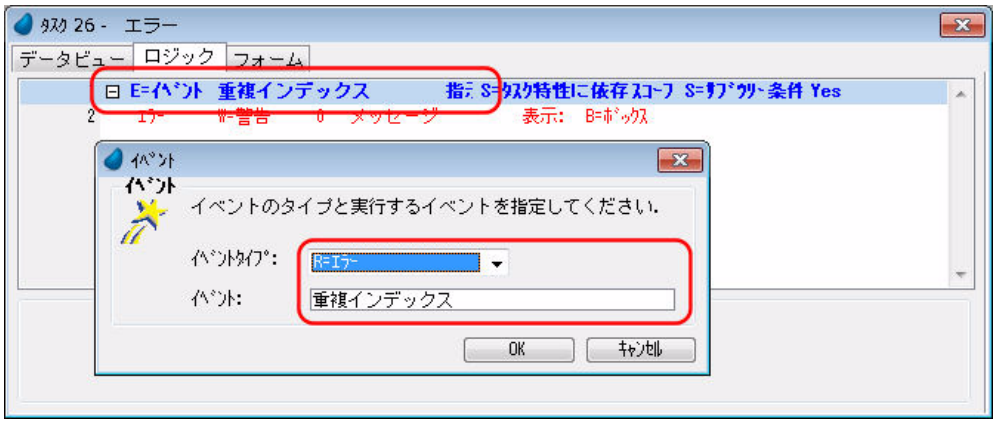



図 5-1 ［エラー］ロジックユニット

では、これはどのように動作するのでしょうか？エンジンが DBMS のエラーに遭遇すると、［エラー］イベントを発行します。例えば、この例では「インデックス重複」の［エラー］イベントになっています。次にエンジンはこのエラーを処理するロジックユニットを探します。現在のタスクに始まり、実行中のタスクツリーをメインプログラムまでサーチします。

ハンドラを見つけられなければ、デフォルトのエラー処理機能に戻り、メッセージを表示します。イベント処理機能についてもっと理解するためには、このメカニズムについて深く掘り下げて説明した『イベントドリブンアーキテクチャ』というホワイトペーパを参照してください。

重複インデックス状態を例に、何ができるかを説明します。重複インデックスに対する［エラー］ロジックユニットを作成することができます。そのロジックユニットの中でエラーを表示し、ユーザ（この例では銀行員）がエラーを修正することができるようにする独自のプログラムを呼び出すことができます。銀行員が確実に何をすべきかわかるように必要な情報を追加することができます。


上の例で［閉じる］ボタンは、エラーになってしまった作業すべてを単にロールバックするものです。

## Magic xpa のビルトイン動作

Magic エンジンがハンドラに遭遇して、それを実行した後どうなるかについては問題が残ります。エンジンがすべての［エラー］ロジックユニット、または内部のデフォルトのハンドラを実行した後、そのロジックユニットの［指示］特性で定義された処理を実行します。この処理は、プログラム自体に定義された［タスク特性］の［エラー発生時］特性に同調して動作します。これには「アボート」の場合と「復旧」の場合があります。このパラメータは、タスクがデータベースエラーに遭遇したときにどのように動作するかを定義するものです。

エラーは、［タスク特性］の［エラー発生時］特性によって、それぞれ異なる動作をします。例えば重複インデックスメッセージを受け取った場合の例を見てみましょう。［エラー発生時］特性が「R= 復旧」として定義されていれば、このエラーに対するエンジンの対応はユーザリトライとなり、ユーザはデータを再入力できます。

一方、もし［エラー発生時］特性が「A= アボート」として定義されていたら、タスクはアボートされます。別な例として最大接続数超過のエラーを見てみましょう。［エラー発生時］特性が「R= 復旧」として定義されると、Magic エンジンの対応はユーザからの入力のない自動リトライになります。

開発者はロジックユニット内で要求された命令を再定義し、新しい命令を定義することでこれらを無効にすることができます。重複インデックスの例をもう一度見てみましょう。重複インデックスに遭遇した場合、エラーは重大なのでタスクはアボートされ、すべての作業はロールバックされなければならない、というように開発者が定義することもできます。その場合、開発者はロジックユニットの［指示］特性でタスクのアボートを設定することになります。

**注意：**

「I= 無視する」のような［指示］特性は、エラーによっては不正になります。「重複インデックス」のようなエラー処理は遅延トランザクションでのみ有効です。

より詳細な情報は『リファレンスヘルプ』をご覧ください。

**重要：**

［指示］特性は、ロジックユニットごとに設定されますが、結果として実行される処理は、実行される最初のハンドラのレベルで定義されます。

## エラー関数

エラーに関する情報は、ハンドラーに入る前に評価される関数を使用することで利用できます。この情報によって、ユーザ定義の［エラー］ロジックユニットを有効にしたり無効にすることができます。

関数は以下の通りです。

- ErrTableName …… エラーが発生したテーブルの物理名を返します。メインソースやリンクテーブルが定義されている場合、この関数によってエラーがどこで発生したかを確認することができます。この関数は、テーブル関連するエラーにのみ対応します。
- ErrDatabaseName …… エラーが発生したデータベース名を返します。
- ErrDbmsCode …… DBMS のエラーコードを返します。これは、致命的で予想外のエラーが発生した場合の対応で利用できます。DBMS 内部のエラーコードについてよく知っていることが前提になります。
- ErrDbmsMessage …… 致命的で予想外のエラーが発生した場合、元の DBMS のエラーメッセージを返します。
- ErrMagicName …… ［エラー］ロジックユニットで「任意のエラー」が定義されている場合に関係します。Magic のエラーが文字列で返ります。この関数は、同じ処理を共有する複数のエラーを処理するために使用することができます。
- ErrPosition …… エラーが発生したレコードの位置を返します。

# 第 6 章 プログラミングの考慮点

*データベースプログラミングの際に考慮すべきことは何でしょうか？*

我々は、トランザクション、特に遅延トランザクションを使用して開発する方法や起こり得るエラーを処理する方法を学んできましたが、それ以外ではどのようなことを考慮すべきでしょうか？この章ではこれについて扱っていきます。

## インターネットのための開発

これまでの章で述べたように、同時並行性を保つためにはロックとトランザクションをできるだけ短くする必要があります。インターネット アプリケーションは全世界で同時に何千と言うユーザが利用できますので、リソースをロックすることでアプリケーションを利用できなくなるユーザが出てくる場合があります。このような環境では、クライアントはサーバと切断されており、その結果データベースサーバとも切断されます。ユーザが実際にいつトランザクションをコミットするのかは不明です。また、クライアント側のアプリケーション処理が異常終了した場合、Magic xpa のコンテキストタイムアウトまでトランザクションはオープンしたままになります。

結果として、リッチクライアントタスクの推奨トランザクションモードは遅延トランザクションとなります。

### トランザクションの考慮点

いつトランザクションをオープンするかについて注意して考慮すべきです。絶対に守らなければならないルールというものはありませんが、基本的な指針をいくつか挙げてみましょう。

#### メニュータスク

メニュータスクからオープンされるリッチクライアントタスクは、通常独立したトランザクションをオープンすることを求められます。このような理由でメニュータスクの［トランザクションモード］特性は「N= なし」に設定すべきです。

#### レコードごとに別々に処理する

レコード処理が終了したらすぐにレコードごとにコミットしたい場合や、それぞれのレコードの修正を別々にロールバックしたい場合、［タスク特性］の［トランザクション開始］特性を「P= レコード前の前」に設定します。

**重要：**

レコードレベルのトランザクションでは、リッチクライアントはレコードが修正されるたびにサーバに接続しにいくということを考慮に入れてください。

#### タスク全体ですべてのレコードをまとめて処理する

すべての修正レコードがタスク終了時のみにコミットされるようにしたい、あるいは、タスク全体のすべての修正レコードをまとめてロールバックしたい場合、［トランザクション開始］特性を「T= タスク前の前」に設定してください。

**注意：**

トランザクションをタスクレベルで定義すると、リッチクライアントがサーバに接続する回数が減ります。

## 論理的シリアライゼイション

システムの完全性にとって非常に重要であるため、常に一人のユーザしかアクセスを許されない処理というものがよくあります。この処理にアクセスしようとする他のユーザは順番を待たなければなりません。

あの夫婦の例を取り上げてみましょう。彼らは、自分たちの当座預金口座に信用限度額が必要かもしれない、ということを話し合っていました。それでウィルマが銀行の支店長に会いに行き、これについて話し合いました。支店長は限度額を 1000 ドルにすることに同意し、システムを更新し始めました。

同時にフレッドは別の支店で（または、同じ支店で）調査部長と話し合って、750 ドルの限度額について同意を得ました。両方のトランザクションが完了すると、実際の信用限度額は 1750 ドルになってしまいます。

この問題を解決するために、Magic xpa の内部関数 Lock をこのプログラムで使用します。信用限度額プログラムの［タスク前］でこの関数を使用した場合（［タスク後］では Unlock 関数を使用します。）、支店長がプログラムをロードするとそれが終わるまで調査部長はプログラムをロードできません。

# データベース混在プログラミング

顧客がどの RDBMS で実行するかを決めて実行するようなアプリケーションを作成する必要がある場合があります。例えばソフトハウスが MS-SQL Server 上で銀行向けアプリケーションを開発して、Oracle データベースを使っているストーンズ・セービング銀行に販売しました。また、同じアプリケーションが Pervasive SQL データベースを使用しているストーンズ・ローン銀行にも販売されるかもしれません。

トランザクションをできるだけ円滑にするためにはいくつかの指針があります。

## デフォルト記憶型式

［デフォルト記憶型式］特性は、データソースのカラムごとに定義されます。Magic xpa がデータタイプごとに持っている固有のデフォルト値が考慮されます。これはおもにアプリケーション自体によって作成されたテーブルに関連したものであり、DBMS によって作成されたテーブルは関係ありません。この場合、データベースに作成される記憶型式を決めるのは Magic xpa の DBMS ゲートウェイです。

**注意：**

［デフォルト記憶型式］特性はゲートウェイに定義されており、開発者は変更できません。

数値型項目を例に挙げてみましょう。これは、ある DBMS では INTEGER として定義され、別の DBMS では NUMBER や PACKED DECIMAL として定義されます。Magic xpa の特定の DBMS 用のゲートウェイは、この違いを内部的に処理します。

## Magic SQL 句

Magic xpa が DBMS に送る WHERE 句に、開発者が複雑な条件を追加する必要が生じる場合があります。このため Magic xpa には、DB SQL と Magic SQL という二つの機能が提供されています。

DB SQL として入力された場合、そのまま DBMS に送られます。クロスデータベースのアプリケーションを開発する場合、関数の中にいくつか異なる点を見つけることがあります。たとえば、特定の文字列から部分文字列を取得する関数は、Oracle では Substr ですが、MS-SQL Server では Substring です。これは、明らかに問題になります。

Magic SQL を使用した場合、Magic xpa で構文チェックを実行したり、通常の Magic 関数を使用することができます。例えば、特定の文字列から部分文字列を取得するには、Magic xpa の MID 関数が使用できます。

**注意：**

Magic SQL で使用可能な内部関数は、Magic 関数のごく一部です。どの関数が利用可能かについては『リファレンスヘルプ』を参照してください。

# トランザクションをクローズする

トランザクションは、オープンしたタスクでクローズされます。

例外として、親と同じトランザクションをもつ（並行実行の）子プログラムが起動される場合があります。トランザクションが親プログラムによってクローズされ、子プログラムがオープン状態のままの時、「トランザクションなし」で残ります。

子プログラムは、トランザクションの一部でなければなりません。最初に呼び出された子プログラムは、トランザクションとトランザクション内の子プログラムの全てを管理します。親タスクがクローズされると、トランザクションの管理はイベントツリーを下って移動します。トランザクションの階層は、開いているタスク順に定義されます。トランザクションがクローズされ、トランザクション内でまだ開いているタスクがある場合、別のトランザクションがすぐにオープンされ、最初の子タスクによって管理されます。

トランザクションの管理が新しいタスクに移動した場合、［タスク特性］で定義された値に関係なく、新しいトランザクションはタスクレベルでオープンされます。

# 第 7 章 付録

## 遅延トランザクションと物理トランザクション

ここでは、トランザクションの 2 つのタイプの違いに関して説明しています。

### トランザクションの開始

遅延トランザクションでは、論理トランザクションがオープンされます。データ操作言語（DML）ステートメントが一時的にキャッシュに保存され、トランザクションが終了した時点でデータベースが更新されます。キャッシュメモリには制限があるため、［タスク前］の前でトランザクションを開始することは推奨されません。

物理トランザクションでは、DML ステートメントが［レコード後］で発行されます。DML ステートメント内の処理コマンドは、すべて物理データベースのロールバックセグメント内に記録されます。

### ロック方式

遅延トランザクションでは、以下のロック方式を設定することができます。

- 入力時 …… 遅延トランザクションのタスクで［ロック方式］特性を「入力時」に設定した場合、データベースにロックを発行しません。これらのロックは、ロック機構によって処理されます。ロックの設定は、必要に応じて使用してください。
- なし

### SQL の範囲ステートメント

遅延トランザクションのタスクで DB SQL を使用すると、要求されたデータソースをアクセスする別のタスクで実行される更新結果が表示されません。

### SQL コマンド

［SQL コマンド］タスクには、物理トランザクションモードのタスクに含めることができます。遅延トランザクションは、SQL コマンドのステートメントによって設定されるタスク用に定義することができます。

**注意：**

遅延トランザクション内のテーブル表示のタスクは、同じトランザクション内の別のタスクで入力されたテーブルの修正を反映します。しかし、［SQL コマンド］タスクは、そのような修正は反映されません。

## ネストトラザクション

親タスクが子タスクを呼び出すと、以下のように動作します。

| 親タスク | 子タスク | トランザクションの動作 |
|---|---|---|
| 遅延<br>ネスト遅延 | 物理 | 子タスクは、物理トランザクションをオープンします。更新処理は、親タスクとは独立してデータベースに送信されます。 |
| 遅延<br>ネスト遅延 | 遅延 | 子タスクは、親のトランザクション内で実行されます。親タスクがコミットする時だけ、子タスクの DML ステートメントが送られます。 |
| 遅延<br>ネスト遅延 | ネスト遅延 | 子タスクは、新しい遅延トランザクションをオープンします。トランザクションは、親タスクから独立してデータベースにコミットされます。 |
| 物理 | 物理 | 子タスクは、各［レコード後］または［タスク後］で親タスクと一緒にデータベースの更新要求を送ります。親タスクと子タスクは、同じトランザクションを共有します。 |
| 物理 | 遅延<br>ネスト遅延 | 実行エラーが発生します。 |

親タスクが物理トランザクションを実際には開いていない場合は、遅延またはネスト遅延の子タスクを含めることができます。

タスクが、別のタスク用に定義された［イベント］ロジックユニットから呼び出された場合、トランザクションのネスト関係は、開発モードの定義ではなく実行時のトランザクションを基に決定されます。

**例：**

- 遅延トランザクションとして定義されているタスク A が、物理トランザクションとして定義されているタスク B を開きます。
- タスク B は物理トランザクションとしてデータベースを更新し、イベントがトリガされます。このイベントのロジックユニットはタスク A 内に定義されており、遅延トランザクションであるタスク C を呼び出します。
- 遅延トランザクションであるタスク C は、物理トランザクションであるタスク B の下では実行できないため、実行エラーが発生します。しかし、タスク C が「親と同じ」として定義されていたとすれば、物理トランザクションとして開くため実行エラーは発生しません。

## データベースにトランザクションをマップする

トランザクション処理機能を実装するため、Magic xpa では、基礎となるデータベースのトランザクション処理機能を活用します。それぞれのデータベースには独自のトランザクション機能があるので、Magic xpa では、さまざまなデータベース内で使用できるトランザクションに内部トランザクションがマップされます。

**注意：**

上記を考慮して、トランザクションはメモリや XML などの RDBMS 以外のデータソースでは実行されません。RDBMS 以外のデータソースを使用している時にトランザクションをロールバックしようとしても失敗します。

メモリデータソースでは、各 DML はコミットされたとみなされ、予測できない動作が行われる可能性があります。データベース・レコードがロールバックされるため、トランザクションにメモリデータと実際のデータベースの両方がある場合は特に当てはまります。しかし、メモリレコードは異なり、データ矛盾を引き起こします。

### ISAM データベース

ISAM データベースがトランザクション処理を実行する方法に基づいて、Magic xpa では内部トランザクションが次の方法でマップされます。

- 読み取りトランザクションとロックトランザクションは、ISAM データベース内で実行されていません。したがって、Magic xpa ではこのトランザクションを ISAM データベースゲートウェイに転送しません。
- 書き込みトランザクションを ISAM トランザクションにマップし、更新を記録してロールバックも実行します。このようなトランザクションでは、テーブルの全てまたは一部をロックすることができます。

### SQL データベース

SQL データベースがトランザクション処理を実行する方法に基づいて、Magic xpa では内部トランザクションが次の方法でマップされます。

- 読み取りトランザクションを R/O（読み取り専用）トランザクションにマップします。このトランザクションでは、トランザクションが開いた場合にデータベースのスナップショットが取得されます。
- ロックトランザクションか書き込みトランザクション内で全てのテーブルのアクセスモードが読み取りである場合は、このロックトランザクションや書き込みトランザクションを R/O（読み取り専用）トランザクションにマップします。アクセスモードが読み取りでない場合は、ロックトランザクションや読み取りトランザクションを R/W トランザクションにマップします。このトランザクションでは、トランザクション範囲内で参照される全てのレコードがロックされます。