



## データ管理と *Magic uniPaaS*

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。

uniPaaS 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 uniPaaS は、Magic Software Japan K.K. の登録商標です。

uniPaaS Studio、uniPaaS Client、uniPaaS Enterprise Server、および uniPaaS RichClient Server は Magic Software Japan K.K. の商標です。

Pervasive.SQL<sup>®</sup> は Pervasive Software, Inc. の商標です。

Microsoft<sup>®</sup> および FrontPage<sup>®</sup> は、Microsoft Corporation の登録商標です。また、Windows<sup>®</sup>、WindowsNT<sup>®</sup> および ActiveX<sup>®</sup> は Microsoft Corporation の商標です。

Oracle<sup>®</sup> は Oracle 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警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。

Copyright 2009 Magic Software Enterprises Ltd. and Magic Software Japan K.K. All rights reserved.

2009年3月19日

## 1 トランザクションとは？

なぜトランザクションが必要なのか .....	1
マルチユーザー環境ではどうでしょうか？ .....	2
いいことばかりではありません。 .....	3
分離レベル .....	3
ロックされたレコード .....	4

## 2 遅延トランザクション

コンセプト .....	5
すべてがうまくいくのでしょうか？ .....	6
更新レコードの識別 .....	6
ネストトランザクションは可能？ .....	9

## 3 他の利点

データの差分更新 .....	10
参照整合性 - 親子間の状況 .....	11

## 4 エラーハンドリング

エラーハンドリング - コンセプト .....	13
Magic uniPaaS のビルトイン動作 .....	14

## 5 プログラミングの考慮点

Web のための開発 .....	16
トランザクションの考慮点 .....	16
論理的シリアライゼーション .....	17
データベース混在プログラミング .....	17
デフォルト記憶型式 .....	17
Magic SQL 句 .....	17

## 6 サマリー



## 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ポ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。

### 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ろうでしょう。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ーが、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

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ポートされるか (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。

## マルチユーザー環境ではどうでしょうか？

二人のユーザー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ー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。トランザクションのない場合にどうなるか、上と同じような簡単な例で見てみましょう。

以下の図のシナリオでは、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。

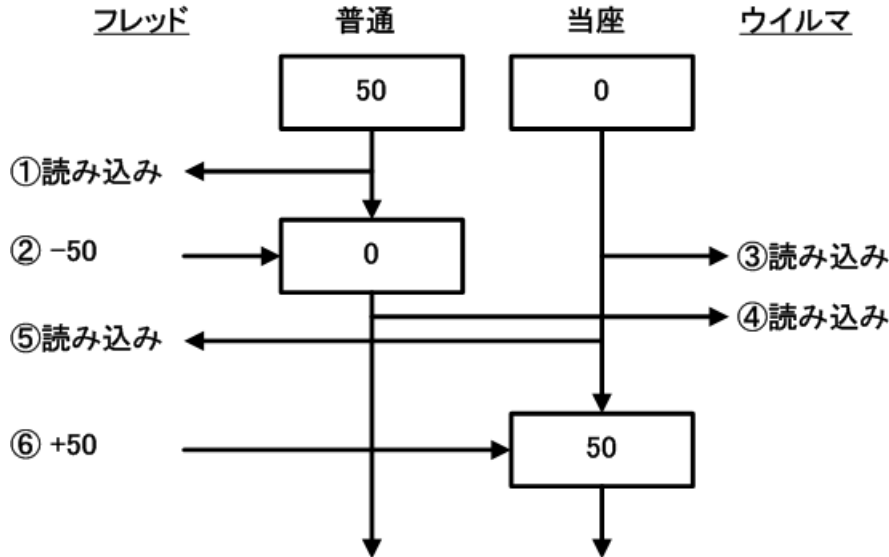


図 1-1 トランザクションのない世界

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 (③)。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完了していないので、当座預金の残金はゼロです。それで、ウィルマが普通預金の残高を照会すると (④)、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。

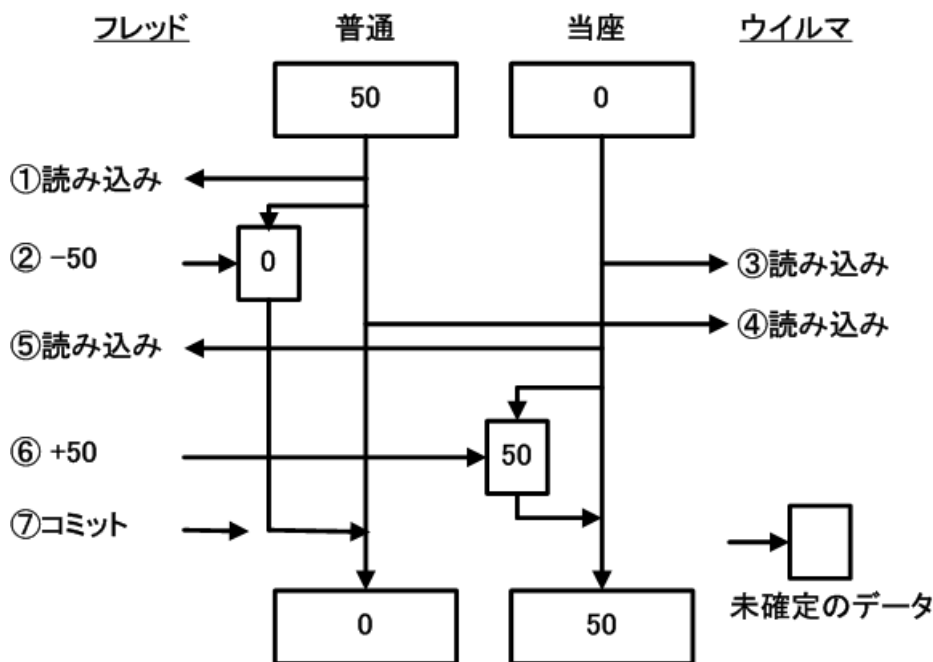


図 1-2 トランザクションのある世界

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 (②と⑥) は、トランザクションがコミットされるまで (⑦)、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 (③)、普通預金の残高を調べると (④) 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すると、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。

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ので、詳細には立ち入らないことにします。

## 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。

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話しました。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ですので、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。これは大変不便です。

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し、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。

ロックについては分離レベルについてのセクションで、もっと詳細に論じますが、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、一人のユーザーが今データを更新している間、同じデータを使用する必要が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のです。これはマルチユーザー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ー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。

以上のシナリオのように、二人以上のユーザー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ー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事です。

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ーが影響を受けてしまうことがあります。前に論じたように、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すべ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。他のユーザーがそのデータにアクセスしよう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論じます。

## 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これについて論じてきましたが、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。

例えば、「未コミット読み取り」という分離レベルでは、あるトランザクションが値を変更し、その変更が実際にコミットされたりロールバックされたりす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照らして見てみましょう。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。

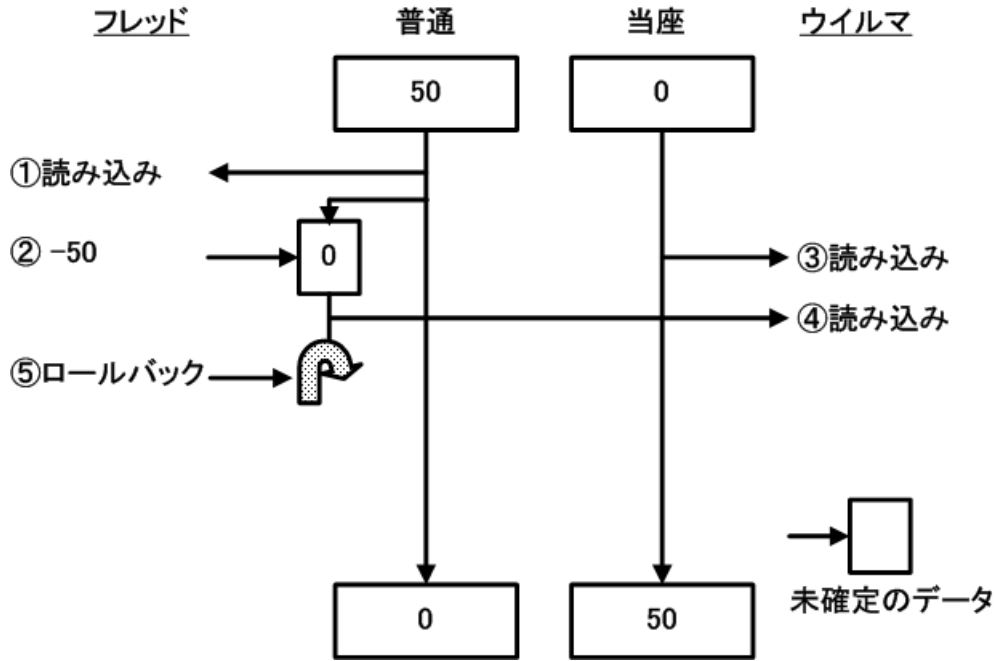


図 1-3 ダーティリードのある世界

ウィルマは当座預金口座を確認しましたが (③)、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 (④) 残高 0 ドルです。ダーティリードがある世界では、フレッドがまだコミットしていないデータ更新の結果が、ウィルマから見えてしまうので、こういうことになります。

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起こるかもしれません。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、明らかにデータ整合性を低下させます。

このため、実際のアプリケーションでは、「未コミット読み取り」よりは整合性の高い分離レベルを用いるのが普通です。

## 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論じました。あるレコードがロックされる時他のユーザー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 番目のユーザー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 uniPaaS では、ユーザーは「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」)。

アプリケーションの要求仕様により、上のどれが適当かを選択し、それから分離レベルの設定や DBMS の選択にも考慮を払う必要があります。

この節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、処理がお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。

## 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？

**M**agic uniPaaSには、遅延トランザクションという独自のトランザクションのメカニズムを備えています。遅延トランザクションは、あらゆる意味で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。

この章と次の章では、これに取り組む方法について手短かに扱うことにします。

### コンセプト



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、他のユーザー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うときには、データベースに対するすべての変更内容が Magic uniPaaS のキャッシュメモリに一旦保持されます。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。このキャッシュメモリの内容は、その Magic セッションでのみ有効で、他の人はそれを見ることができません。

この遅延トランザクションがコミットされるときには、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 RDBMS に対して一括して適用されます。このときに Magic uniPaaS は、RDBMS が提供するトランザクションの機能を利用します。RDBMS に対して行なうトランザクションを、Magic uniPaaS の遅延トランザクションと区別するために、物理トランザクションと呼びます。

遅延トランザクションを使って、フレッドが振替を行う処理を次ページの図に示します。

遅延トランザクションを始めてから (図中①)、フレッドが普通預金から 50 を差し引き (③)、当座預金へ 50 を加える (⑤) と、変更結果はフレッドのキャッシュメモリに一旦格納されます。

ここで遅延トランザクションをコミットすると (⑥)、物理トランザクションの開始 (⑦) から、データ更新、コミット (⑧) まで、ユーザの介入なしに一度に短期間に行われます。このため、遅延トランザクションを利用することにより物理トランザクションの期間は短くなりますのでアプリケーションの同時並行性は非常に向上します。

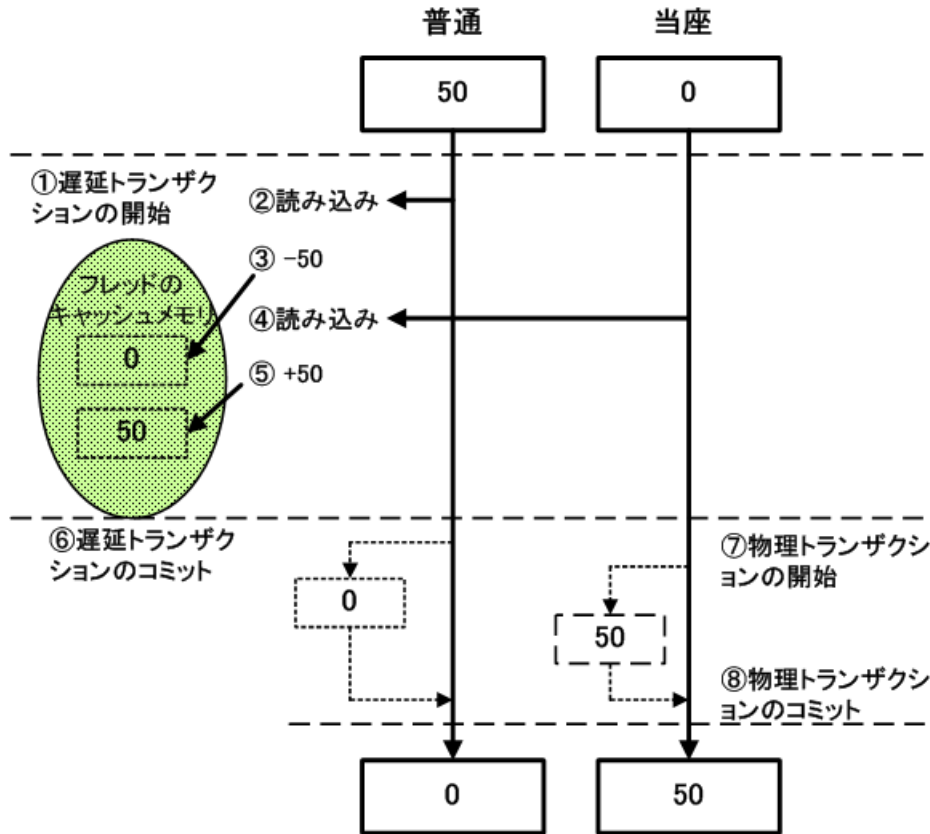


図 2-1 遅延トランザクション

フレッドとウィルマの状況を取って、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ため、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ば、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。この良く起こる問題は「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のでしょうか？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。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のでしょうか？ Magic uniPaaS では、「更新レコードの識別」というパラメータ設定により、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データビューエディタの両方で定義することができます。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）

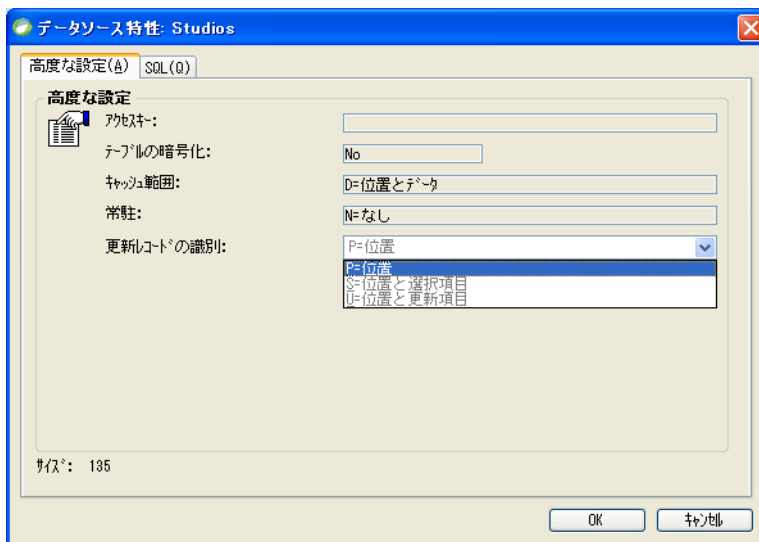


図 2-2 更新レコードの識別

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。

上で示した例は、この特性が「位置」に設定されている場合に起こります。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。

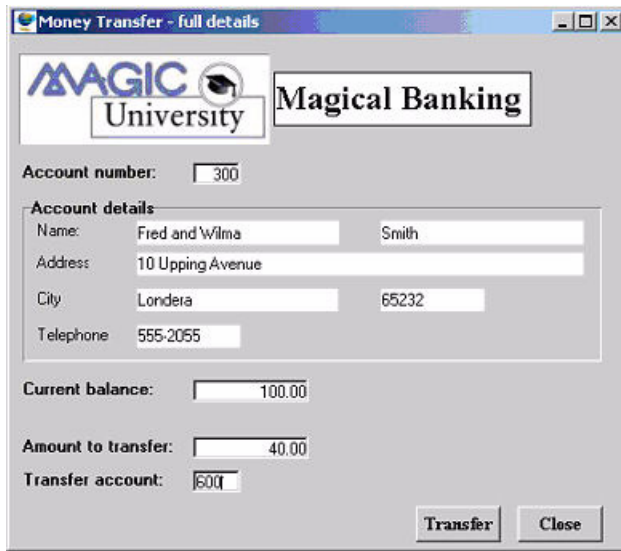
下のフレッドとウィルマの SQL コマンドを見てみましょう(振替の前彼らは普通預金口座に 100 ドル持っていたことを思い出してください)。

```

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
    
```

ここで何が起こるのでしょうか?フレッドのトランザクションは始まり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き残高はもう 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す。それ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
```

他のユーザー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 = 60
WHERE 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う失敗しますから、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 uniPaaS 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せることができます。

## 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 uniPaaS 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。

### 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。

また、トランザクションのネストは、Magic uniPaaS の遅延トランザクションでは可能ですが、RDBMS の提供するトランザクションのメカニズムではネストはできません。

この章では遅延トランザクションのコンセプトと、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。

## 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のでしょうか？

**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ー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。最初に残高が 100 ドルあるときに、ウィルマが 40 ドルを引き出そうとしています。このときには、次のような SQL コマンドが実行されることでしょう。

```
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」の問題が起きてしまいます。

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のでしょうか？

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、テーブルのカラムに定義された数値項目とタスクのセレクトコマンドの特性ダイアログで使用可能です。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。

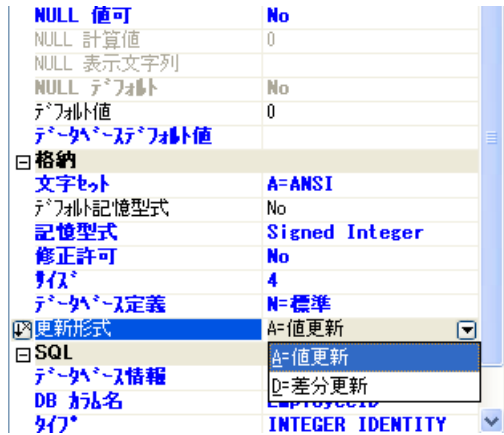


図 3-1 [更新形式] 特性

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？更新形式パラメータが D= 差分更新に設定されると、絶対値、言い換えると一定の値がセットされるのではなく、もとの値と新しい値との差分が SQL 文で使わ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。更新形式がデフォルトの「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、更新形式が「差分更新」にセットされている場合には、次のような UPDATE 文となります。

```

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のでしょうか？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。そして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 uniPaaS 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、受注ヘッダー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。これらのテーブルと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。ここで起こりうる問題点と、Magic uniPaaS の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説明するために、我々が使用してきた銀行のアプリケーションを例に取ります。

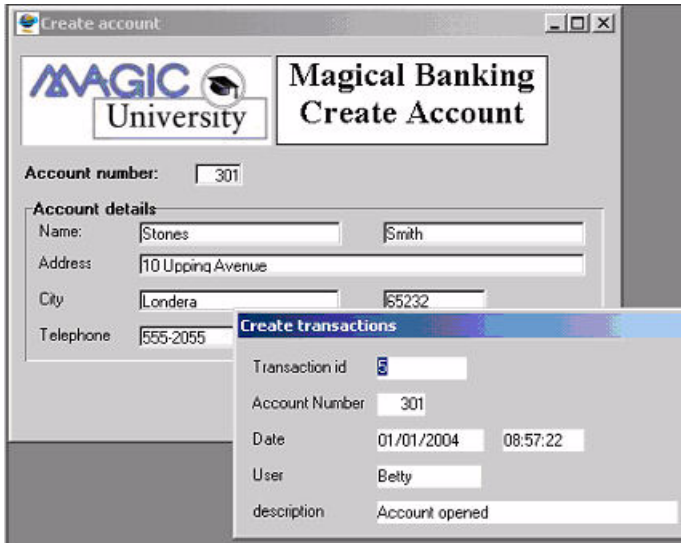
普通預金口座に対して、金銭取引をすべて記録した履歴ファイルを持つ必要があります。銀行の残高に不審があるときに、どのようにしてその金額になったかの履歴が保存されていないのでは、誰も銀行を使わなくなってしまうでしょう。

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のでしょうか？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やるかについては問題が残ります。

そこで、Magic uniPaaS 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を利用できます。このために新しい特性、同期の使用方法を紹介します。

**同期：**

このパラメータはコール（タスク、プログラム、公開）特性ダイアログに含まれます。式を使用可能な論理値です。この特性が Yes と評価されると、子のレコードを書きこむ前に親のプログラムのレコードを書き込みます。

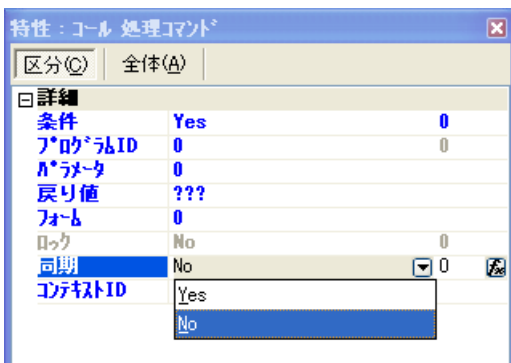


図 3-2 [同期] 特性

我々のシナリオで、これがどのように役立つでしょうか？同期を Yes にすることによって、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 uniPaaS 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。

## エラーハンドリング

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の過ち (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らっしゃいますか？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、RDBM 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。一タビュー定義とリッチクライアントタスクのロジックは、イベントドリブンで SDI 形式のオンラインタスクと同じように構成されています。リッチクライアントタスクは、インタフェース定義がオンラインタスクと異なっています。また、タスクの一般的な動作もリッチクライアントの性質上、異なるものがあります。

### エラーハンドリング - コンセプト



Magic uniPaaS 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ハンドリングのメカニズムがあります。Magic uniPaaS 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 uniPaaS 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ー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うました。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 uniPaaS 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 SQL Server 2000 のエラーメッセージは、「ユニークインデックス : trans を持つオブジェクト : history には、重複キーを持つ行をインサートできません。」といったようになるでしょう。Oracle の場合はまたまったく異なります。Magic uniPaaS ではすべてのデータベースでメッセージは同じです。

さてこれはどのように役に立つのでしょうか？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いてコミットしたら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？ユーザー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 uniPaaS では開発者が独自に定義したエラーハンドリングを提供することが可能です。

開発者が定義したエラーハンドリングメカニズムは Magic uniPaaS 内部のイベントハンドリングメカニズム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。

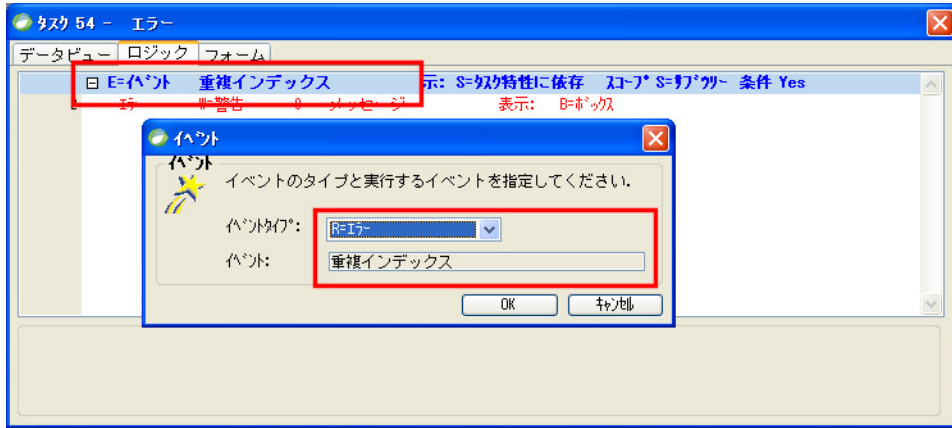
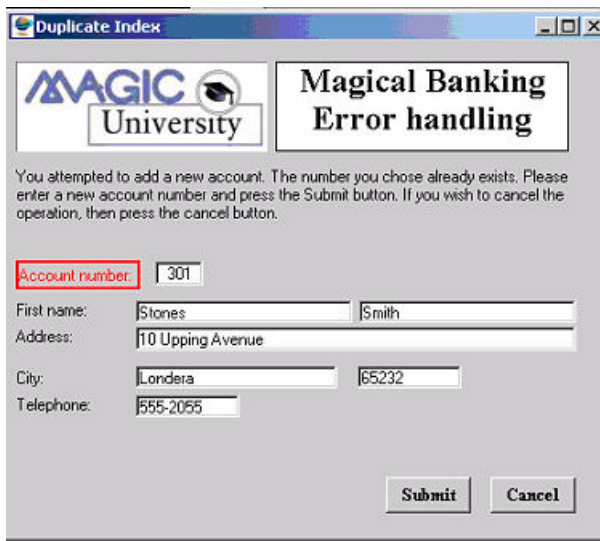


図 4-1 [エラー] ロジックユニット

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起こします。例えば我々の例では「インデックス重複」のエラーイベントです。次にエンジンはこのエラーを処理するハンドラを探します。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ハンドリングメカニズムに戻り、メッセージを出します。イベントハンドリングメカニズム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ーを読んでください。

何が出来るかを示すため重複インデックス状態を例に取りましょう。重複インデックスに対するエラーハンドラを作成することができます。そのハンドラの中でエラーを表示し、ユーザー（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だけの情報を追加することができます。



上の例で、Cancel ボタンはこの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 uniPaaS のビルトイン動作

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ハンドラ、または内部のデフォルトのハンドラを実行した後、その [ロジックユニット特定] 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我々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ーリトライとなり、ユーザー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ーからの入力のない自動リトライになります。

開発者はハンドラ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が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。その場合開発者はハンドラでエンジンの対応に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」のようなエンジンの対応オプションは、エラーによっては不正になります。「重複インデックス」のようなハンドラ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] 特性で定義された処理は、それぞれのエラーハンドラごとに設定されるかもしれませんが、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。

## 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

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っ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。

### Web 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。Web アプリケーションは全世界で同時に何千と言うユーザー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ー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ーと切断されており、その結果データベースサーバーとも切断されます。ユーザーが実際にいつトランザクションをコミットするのかわかりません。また、ユーザーは、ブラウザの閉じるボタンをクリックしたり、別の Web ページを閲覧したりするような通常とは異なるやり方でブラウザを終了させるかもしれません。この場合、Magic uniPaaS 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、ブラウザ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ブラウザタスクは、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、ブラウザクライアントはレコードが修正されるたびにサーバーに接続しにいくということを考慮に入れてください。

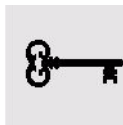
#### タスク全体ですべてのレコードをまとめて処理する

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、ブラウザクライアントがサーバーに接続する回数が減ります。

## 論理的シリアライゼーション



システムの完全性にとって非常に重要であるため、常に一人のユーザー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ー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、という話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おり、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 uniPaaS 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 関数を使います。）、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 DB2 UDB 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はテーブルのカラムごとに定義されます。Magic uniPaaS がデータタイプごとに持っている固有のデフォルトが考慮されます。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 uniPaaS の DBMS ゲートウェイです。

### 注意：

デフォルト記憶型式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 uniPaaS 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 uniPaaS 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ある場合があります。このため Magic uniPaaS 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 では、条件式を SQL 文の WHERE 句に書くとおりに書き、実行時にはそれがそのまま SELECT 文の WHERE 句に付加されます。これは利用するデータベースがサポートする SQL WHERE 句を書けますが、逆に移植性に問題があります。

それに対し、Magic SQL の場合は、条件を Magic uniPaaS の内部関数を使った式で表現します。実行時には、Magic uniPaaS が各 DBMS に固有な等価な WHERE 句に直して SELECT 文を発行します。例えば、ある文字列から文字を取り出す関数は、DB2 UDB と Oracle では Substr ですが MS-SQL では Substring です。Magic SQL を使用すれば、いずれの場合でも、Magic uniPaaS の標準の MID 関数も使用します。DBMS ごとに異なる細かい違いに煩わされることがなくなります。また、文法のチェックも Magic uniPaaS が行ってくれます。

### 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。

## サマリー

### 結びの言葉

この文書が、Magic uniPaaS のデータベースプログラミングについての十分な情報を皆さんに提供できれば幸いです。そして、この文書を読んだ後、皆さんが遅延トランザクションとエラー処理の使用方法についてより精通されるようになることを期待します。

開発者は、Magic uniPaaS 特有の能力を使用すれば、同時並行実行環境で稼動し、各 RDBMS の差が最小化されたより完全で堅牢なアプリケーションを作ることができるでしょう。