RIA オフライン データの同期

Magic xpa

第 2 版　2015 年 9 月 16 日　初版

マジックソフトウェア・ジャパン株式会社

1 はじめに 5
2 準備編 9
2.1 データベース テーブルに定義を追加します 10
2.2 データ リポジトリにローカル テーブルをコピーします 11
2.3 Studio でのローカル テーブルの APG 13
2.4 ローカル データベースについての基礎知識 15
2.5 ローカル テーブルについての制限事項 17
2.6 オフライン起動のメニュープログラム 19
2.7 オフラインで起動するには？ 22
2.8 execution.properties_Lastoffline というファイルは何ですか？ 25
3 データの単純ダウンロード 26
3.1 データのダウンロードをするには？ ①親子タスクにより 27
3.2 データのダウンロードをするには？ ② DataViewToDataSource 関数を使う 28
3.3 DataViewToDataSource 実行時にデータ重複があるとどうなりますか？ 31
3.4 ダウンロード対象以外の項目がタスクで定義されている場合、どうしますか？ 32
3.5 DataViewToDataSource に渡すカラム一覧を簡単に作成する方法がありますか？ 34
3.6 汎用ダウンロードタスクを作れますか？ 36
3.7 コピー元テーブルとコピー先テーブルのカラム名が異なる場合、どうしますか？ 39
3.8 DataViewToDataSource 関数の内部の動作はどのように確認しますか？ 40
3.9 オフライン RIA タスクからダウンロードプログラムを呼び出すには？ 43
3.10 リソースのダウンロードはどう行いますか？ 45
3.11 サーバが応答しない時、どうなりますか？ 49
3.12 オフライン起動時のサーバタイムアウト時間は設定できますか？ 51
3.13 インターフェースビルダでインタフェースファイルを作成 52
4 差分ダウンロード 57
4.1 差分だけダウンロードするにはどうしますか？ 58
4.2「最終更新時刻」はどのようなデータ型にしますか？ 59
4.3 時差のある遠隔地にクライアントがある場合にはどうしますか？ 60
4.4 サーバ時刻とクライアント時刻にずれがある場合にはどうしますか？ 61
4.5 クライアント時刻を使ったときの不具合の例はありますか？ 63
4.6 レコードの削除はどのように取扱いますか？ 65
4.7 デバイス ID はどのように取得しますか？ 66
4.8 最終更新時刻 の作成／設定／取得のプログラムはどう作りますか？ 70
4.9 差分ダウンロードのロジックは最終的にどのようになりますか？ 72

4.10 商品タイプや商品マスタのダウンロードはどうしますか？ 75
5 アップロード 76
5.1 差分アップロードの基本的な考え方は？ 77
5.2 差分アップロードをするにはどうしますか？ 78
5.3 ローカルテーブルの表示・修正プログラムはどうなりますか？ 79
5.4 ローカルテーブルの削除ロジックはどうなりますか？ 80
5.5 アップロード プログラムはどのようになりますか？ 81
5.6 アップロード失敗に対する考慮が必要ですか？ 87
6 選択プログラム 88
7 受注データの扱い 95
7.1 受注データにはどういう特性がありますか？ 96
7.2 受注データのダウンロードはどのように行いますか？ 97
7.3 受注データのアップロードを必要最小限にするには？ 103
7.4 受注入力プログラムはどのように作りますか？ 104
7.5 受注番号の発番はどのように行いますか？ 105
7.6 受注入力プログラムは最終的にどのようになりますか？ 106
7.7 受注データのアップロードはどのように行いますか？ 110
7.8 受注データのアップロードはどのように行いますか？ 112
7.9 受注データの発番とマージはどのように行いますか？ 115
7.10 受注データアップロード後の同期はどのように行いますか？ 118
7.11 例外状況への対応 120
8 データベースの一括転送 121
8.1 データの一括ダウンロードとは？ 122
9 参考情報 123
9.1 ホワイトペーパ > オフラインアプリケーションの開発 > ローカル（オフライン）ストレージ 124
9.2 リファレンスガイド > Magic xpa で使用するデータベース > Local データベース 126
9.3 Magic xpa 2.4c README 127
9.4 Magic xpa 2.5b README 130

1 はじめに

本書の目的

本書は、Magic xpa RIA サーバ製品のオフライン機能を使ってアプリケーションを作成する際に必要となる、ローカルデータと共有データの同期方法について説明することが目的です。

Magic xpa 2.4c から、Magic RIA 機能にオフライン実行機能がサポートされるようになりました。それに伴い、RIA クライアント上に格納するローカルデータベースが利用できるようになりました。これにより、電波の関係などで、クライアントの端末がサーバにネットワーク接続できない環境であっても、Magic の RIA 機能を使ってデータの蓄積を行い、サーバ接続できる場所でそのデータをサーバにアップロードし、同期する、というようなオフラインでの使い方ができるようになりました。

このようなアプリケーションを作成する場合には、サーバ側に存在する共有データベースと、各エンドユーザが利用する端末デバイス上でのローカルデータが、正しく同期するように設計することが重要です。これは一種の「分散データベース」となりますので、設計を誤ると更新のタイミングによるデータ不正などの可能性がありますので、慎重に行う必要があります。

本書ではデータ同期方法の入門として、必要最小限に簡単化したデータ構造を持つサンプルアプリケーションを例として、同期の方法を具体的に見ていきます。

ローカル データベースの同期については、製品添付のチュートリアル「Getting Started for Offline RIA」に基本的なことが記述されていますが、本書ではもう少し掘り下げて解説しています。

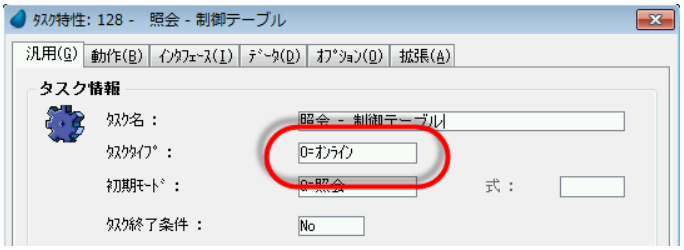
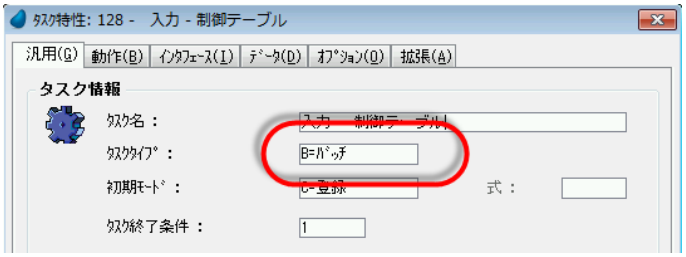
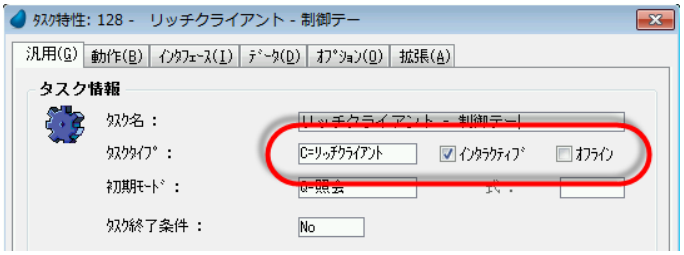
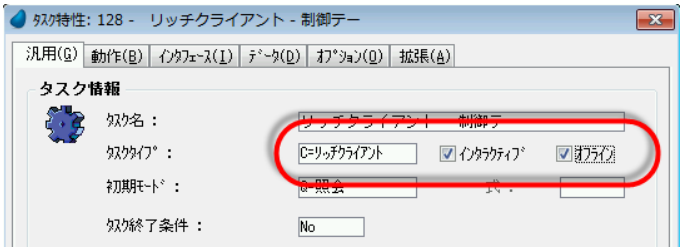
本書で扱わないことがら

本書では、オフライン RIA のローカルデータベースのデータ同期に焦点を絞っていますので、次のような話題は割愛してあります。

- Magic xpa オフライン RIA アプリケーションの基本：読者はすでに、オフライン RIA プログラムの基本的な扱い方（Magic xpa Studio でのタスクの定義、テスト実行など）について習得しているものとします。これはチュートリアルを読むか、弊社トレーニングコース、MOS Web セミナーなどをご利用ください。
- Magic xpa RIA アプリケーションのプログラミングテクニック。紙面の関係で、一つ一つのタスクのプログラミングテクニックの詳細について説明しません。「Mastering Magic xpa」などをご利用ください。
- モバイル RIA での利用：Magic xpa のオフライン機能は、Windows RIA でも Mobile（iOS、Android）の RIA でも、同じ手法により開発することが可能ですので、モバイルに固有な特性による考慮事項については説明しません。「モバイルアプリケーション開発ガイド」などの技術情報をご参照ください。

本書で使う用語

本書では、次のような用語を使います。

種類	用語	説明
タスクタイプ	オンライン	クライアント サーバ アプリケーションでのオンラインタスク 
	バッチ	実行エンジン上で実行されるバッチタスク。 
	RIA オンライン	従来の RIA タスク。実行時にはサーバとの接続を必要とする。 
	RIA オフライン	xpa 2.4 で新しく導入された、オフライン動作が可能な RIA タスク。 
データベース種別	サーバ データベース	従来のデータベース。サーバ上に存在して、一般には複数ユーザにより共有される。
	ローカル データベース	xpa 2.4 で新しく導入された、RIA クライアントデバイス上に存在するデータベース。RIA オフラインプログラムでだけ利用することができる。
テーブル種別	サーバ テーブル	サーバ データベース中に存在するテーブル（データソース）
	ローカル テーブル	ローカル データベース中に存在するテーブル。

本書でのプログラム表記

本書で使うサンプルアプリケーションでは、プログラム名の命名規約として、次のような文字を先頭に付加して、タスクのタイプがわかるようにしています。

種類	表記	説明
タスクタイプ	O	オンライン
	B	バッチ
	R	RIA
タスクモード	Q	照会
	M	修正
	C	登録
	D	削除
フラグ	F	オフライン
	N	非インタラクティブ

例えば、プログラム6番の名前は「RQ_Top」ですが、先頭の「RQ」の意味は以下のようになります。

R → RIA タスク

Q → 照会モード

本書の構成

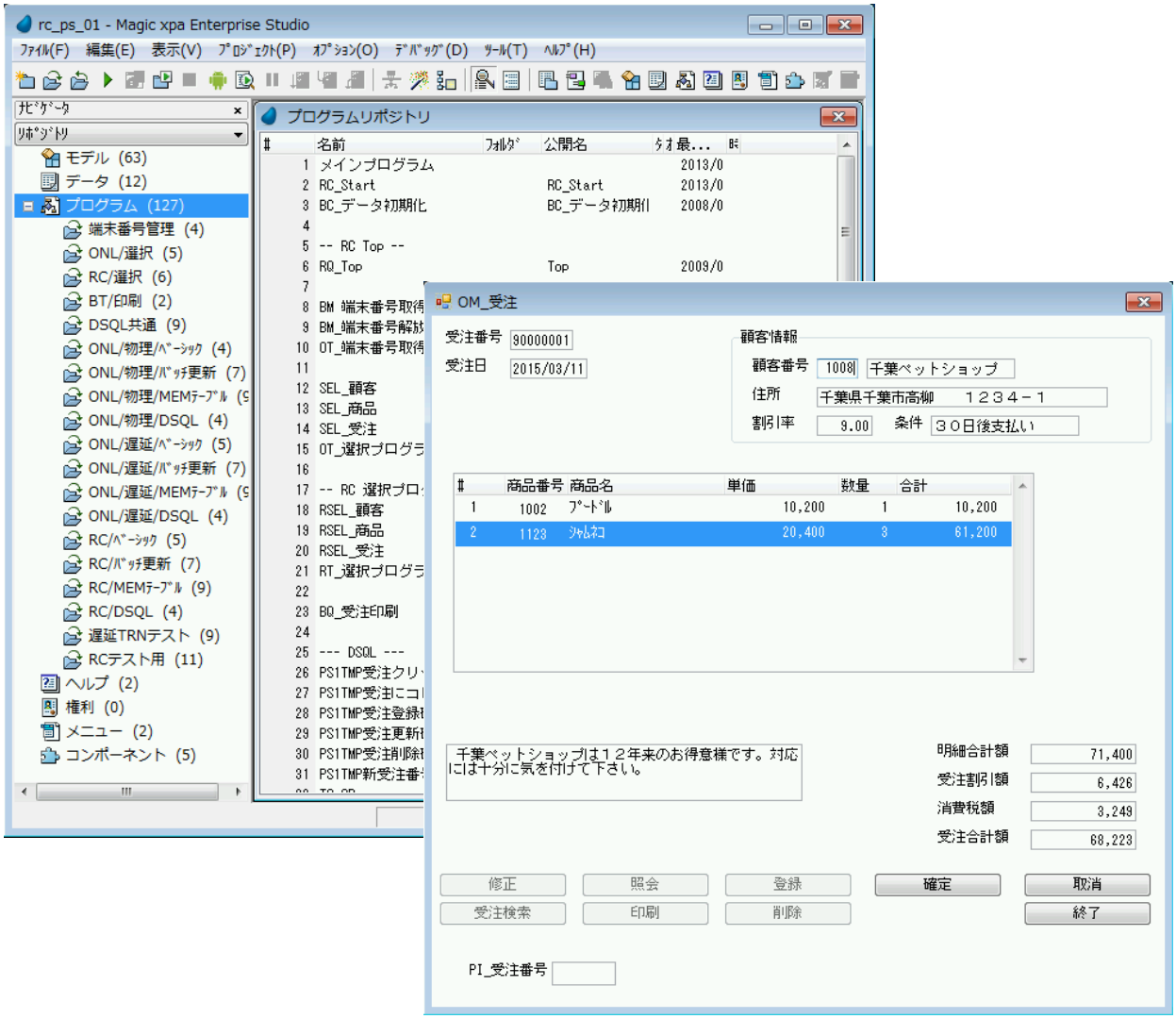
本書は、次のような構成となっています。

- 第 2 章「準備編」では、ローカル テーブルを利用するにあたって必要な準備、設定、および基礎知識について解説します。
- 第 3 章「データの単純ダウンロード」では、サーバ テーブルからローカル テーブルにデータをダウンロードする手法と、留意点について説明します。
- 第 4 章「差分ダウンロード」では、ダウンロードのデータ量を減らすために、前回のダウンロード時の後に変更のあったデータだけをダウンロードする方法を説明します。ここでは、タイムスタンプの利用が重要になります。
- 第 5 章「アップロード」では、端末側で入力したデータを、サーバ上のデータベースに反映する方法を説明します。
- 第 6 章「選択プログラム」では、ローカルテーブルを使った選択テーブルを作成します。これはメインテーブルとしてローカルのテーブルを使う、というだけで、特別なテクニックが必要となるわけではありません。
- 第 7 章 「受注データの扱い」では、二つのテーブルにまたがって親子関係のあるデータをダウンロード、アップロードする方法について説明します。1テーブルだけを対象とするダウンロード・アップロードよりも、関係性を保持したまま反映する必要があるので、少し複雑になります。
- 第 8 章「データベースの一括転送」では、複数のマスターデータなどを一括して効率よくダウンロードする方法について説明します。
- 第 9 章「参考情報」 では、オフラインデータについての参考技術情報を紹介します。

2 準備編

ここでは、オフライン RIA でローカルデータベースを使うにあたり、各種の設定、ローカルデータベースの基礎知識、Studio での操作、制限事項等について説明します。

本書では、サンプルとして、RIA トレーニングコースで使ったペットショップデモ（RIA 対応版）を使います。これをもとにして、オフライン対応化していきます。



2.1 データベース テーブルに定義を追加します

「データベース」テーブルでは、DBMS を「Local」として、ローカルデータベースを定義します。

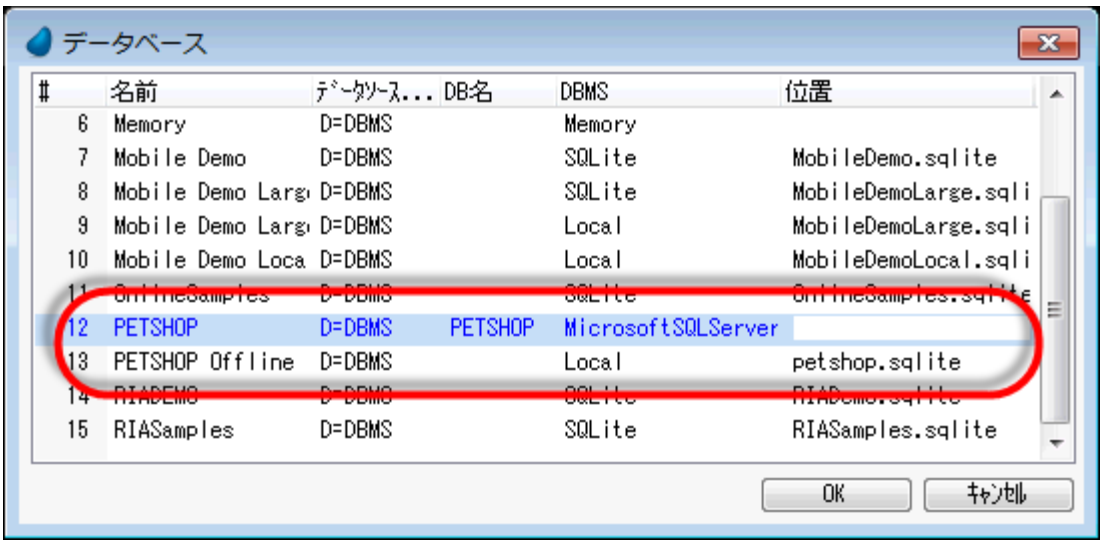
「位置」には、SQLite ファイルの名前を指定します。

ローカル テーブルを使う場合に、まず最初に必要なことは、データベーステーブルにローカル データベースを定義することです。

本書のサンプルでは、ローカル データベース および サーバ データベースとして、データベーステーブルに以下のように定義します。

種類	名前	DBMS	位置
ローカル データベース	PETSHOP Offline	Local	Petshop.sqlite
サーバ データベース	PETSHOP	MicrosoftSQLServer など	

- サーバ データベースの DBMS としては、Pervasive PSQL、Btrieve、Oracle 等、他の DBMS でも OK です。
- ローカル データベースの DBMS は、必ず「Local」でなければなりません。



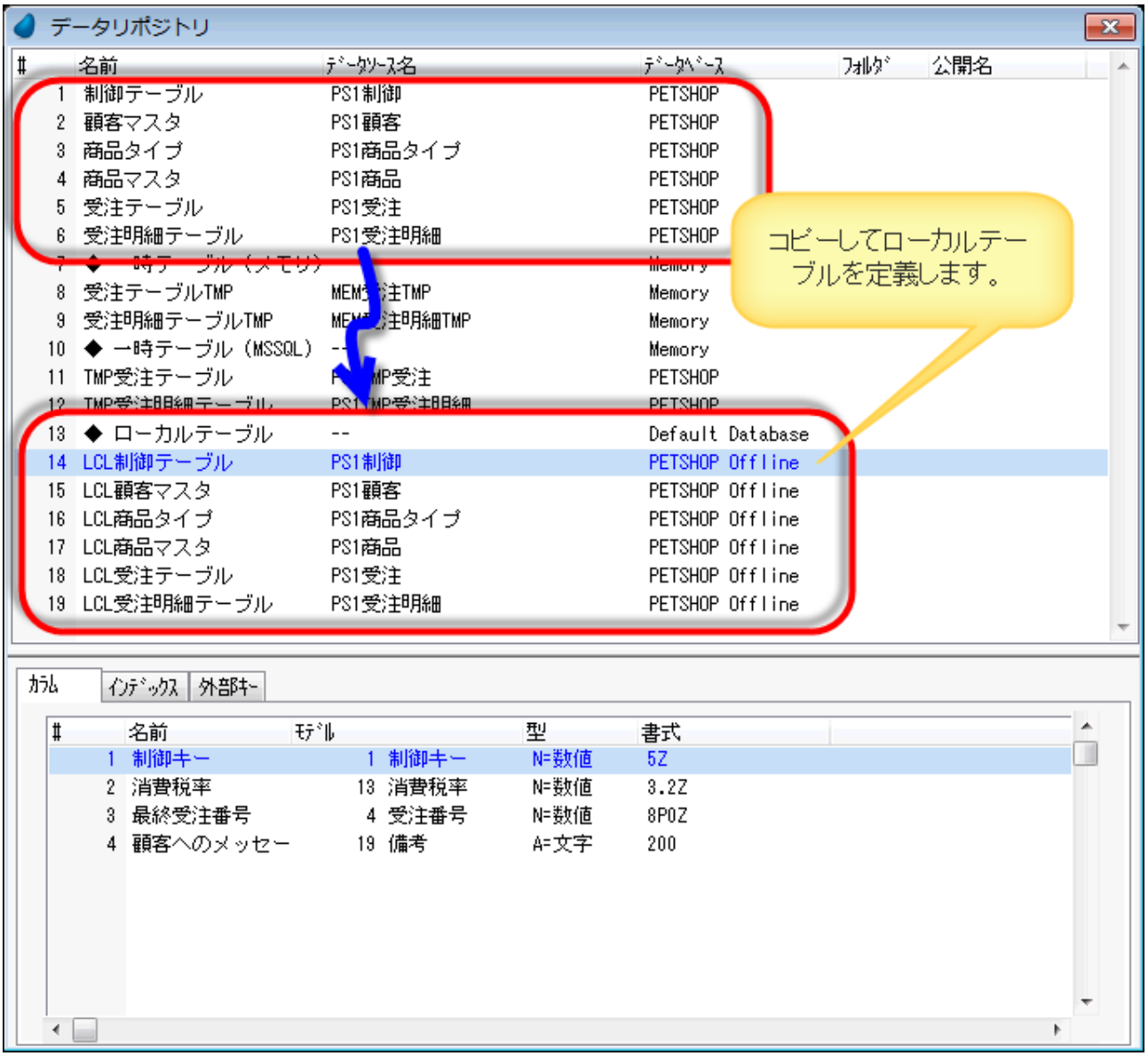
2.2 データ リポジトリにローカル テーブルをコピーします

プロジェクトのデータリポジトリでは、ローカルデータベースを参照して、ローカルテーブルを定義します。

定義としては、サーバのテーブルをそのままコピーして、データベースのみ、ローカル データベースに変更するのが基本です。

ローカル テーブルを利用する次のステップは、テーブルリポジトリにローカルテーブルを定義することです。

ローカル テーブルと、サーバ テーブルは、たとえ全く同じデータ構造をしていたとしても、別々に定義しておく必要があります。ですので、もともと定義されていたサーバテーブル（テーブル1番～6番）をローカル データベースとしてコピーします。データソースとして、ローカル データベースとして定義した「Petshop Offline」を設定します。



後述するように、サーバ、ローカル両方のテーブルには、同期情報としてタイムスタンプや各種フラグ類を追加していきます。

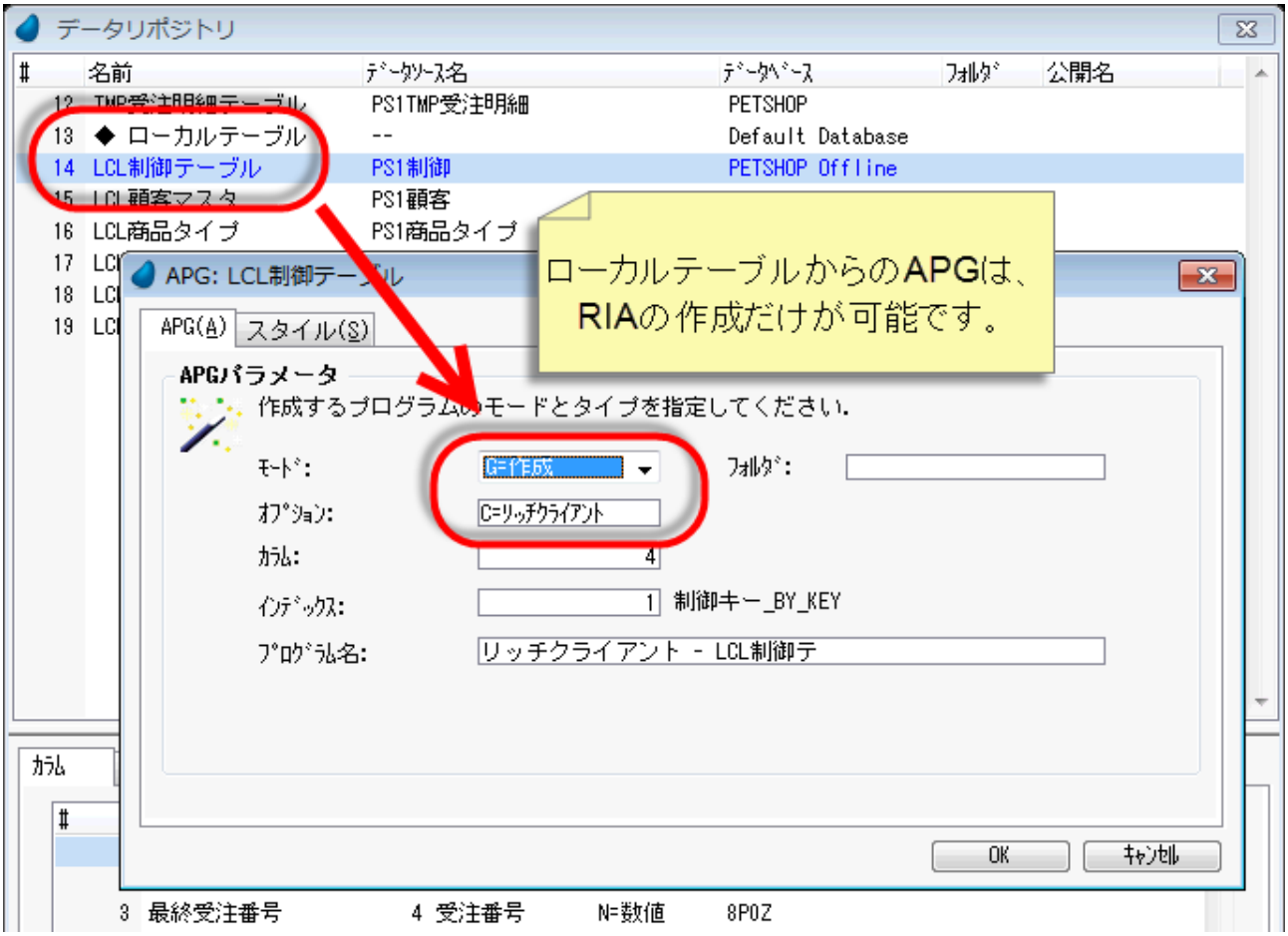
本書では説明の都合上、ここでまずオリジナルのテーブルをそのままコピーしていますが、このようにすると、サーバ側とローカル側の両方のテーブルにそれぞれカラムを追加することになるので、二度手間になります。

これらの追加カラムは、サーバ側、ローカル側両方に同じ定義が必要になりますので、実際のアプリケーション開発においては、サーバ側のテーブルに必要なカラムを追加した後にコピーしてローカル テーブルとする方がいいでしょう。

2.3 Studio でのローカル テーブルの APG

Studio のデータリポジトリでローカル テーブルに対して APG をすると、「G=作成」しか選択できません。
作成したプログラムは、F7 により実行することができます。

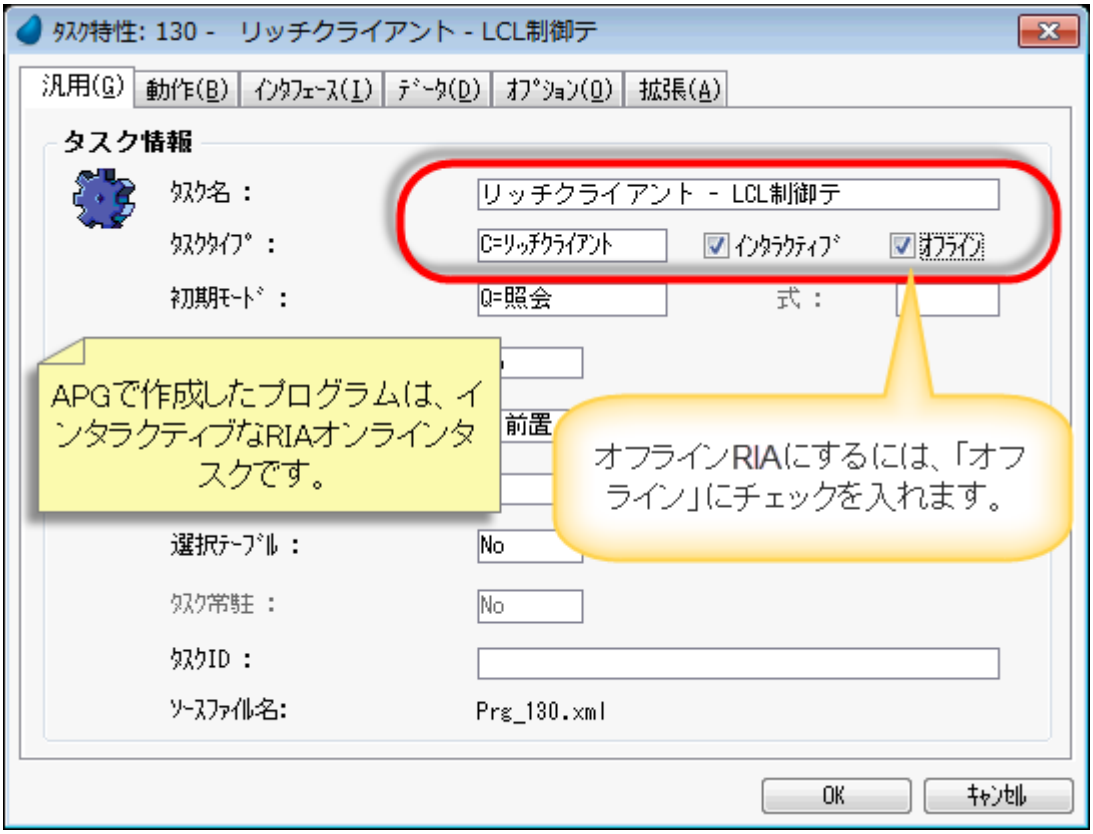
データリポジトリ上で、ローカルテーブルを選択して、Ctrl+G（APG）を行うと、APG ダイアログが出ますが、ここでは「G=作成」モードしか選択できません。ローカル テーブルは RIA プログラムだけで扱うことができるもので、実行エンジン上で直接実行することができないからです。



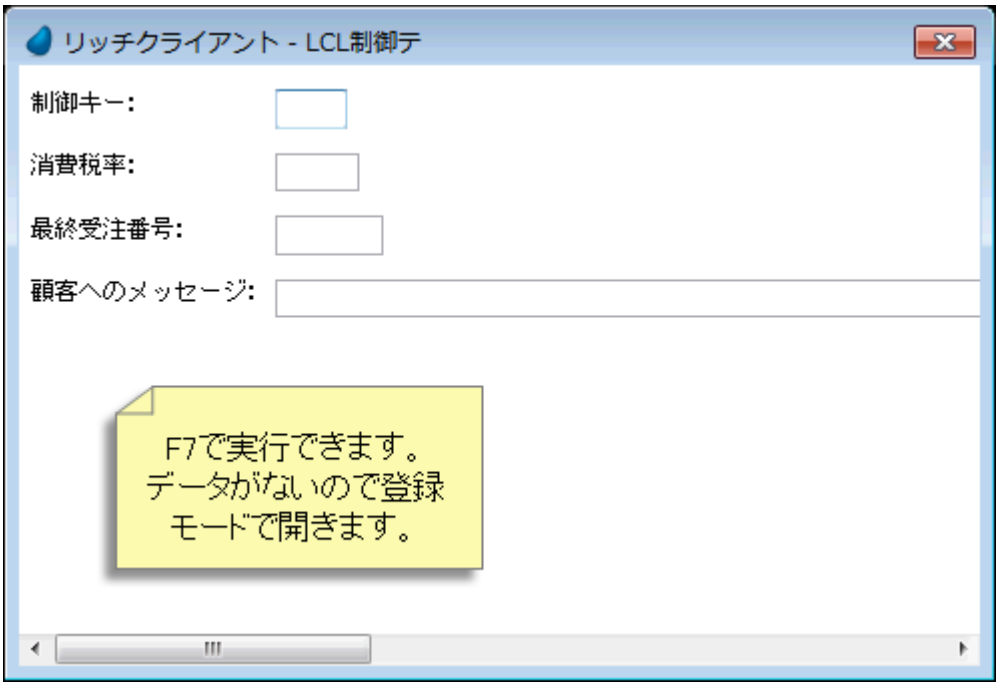
APG で作成されたプログラムは、通常の RIA APG の場合と同じく、次のようなタスク特性となります。

タスク名： リッチクライアント－（テーブル名）
タスクタイプ： C=リッチクライアント、インタラクティブ
初期モード： Q=照会

ローカル テーブルをアクセスするタスクは、オフラインで実行することができますので、ここで「オフライン」フラグにチェックを入れておきます。



これを実行してみましょう。
RIA オフラインタスクも、通常の RIA タスクと同様に、F7 で起動できます。I



2.4 ローカル データベースについての基礎知識

ローカル データベースについての技術的基礎知識があると、プログラム設計時の理解がより深まりますので、プログラムを作る前に、ここでローカル データベースについての基礎知識をいくつか紹介します。

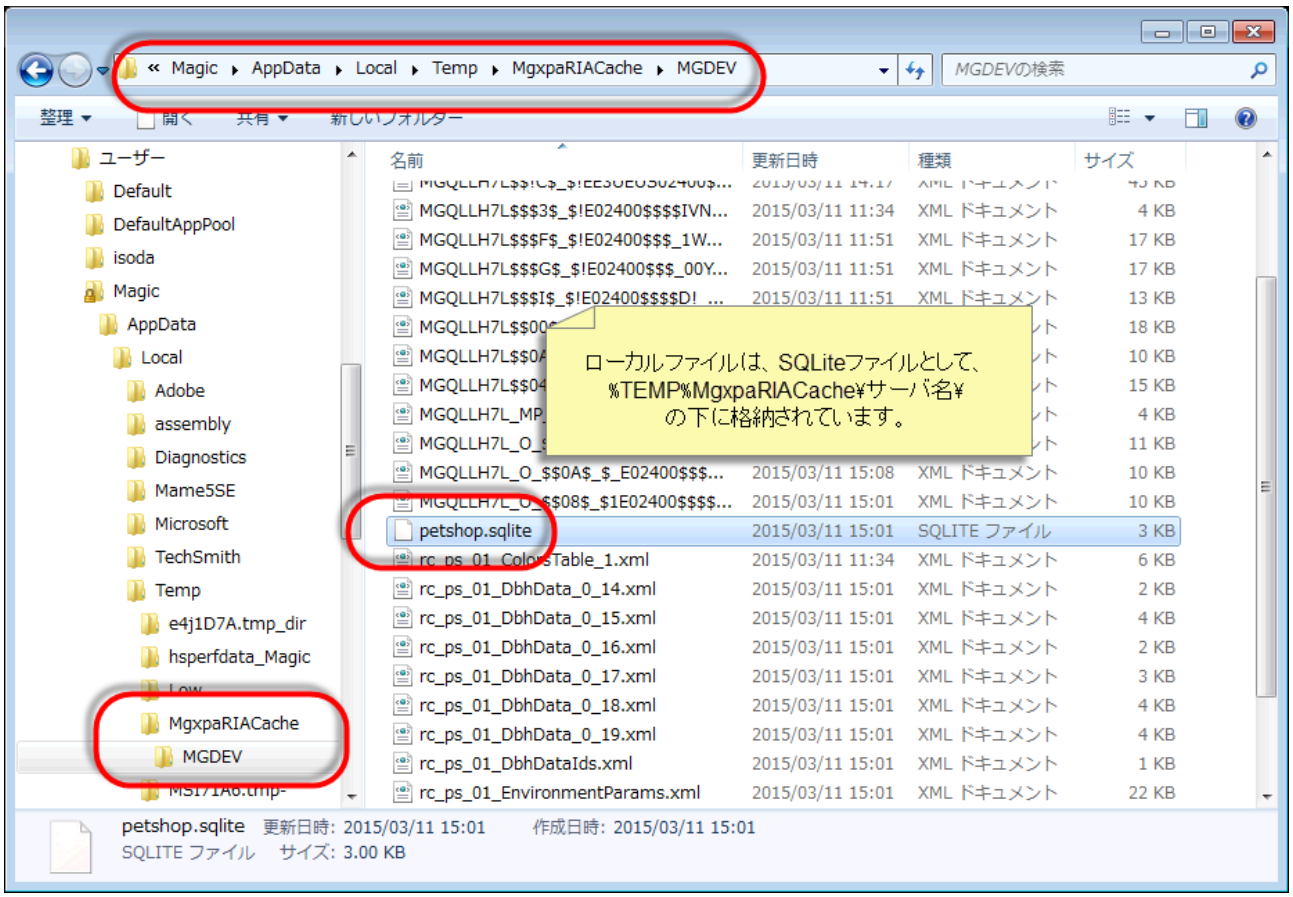
ローカル テーブルというものの実体は、デバイスのローカルストレージに作成される SQLite ファイルです。SQLite というのはオープンソースの個人用の簡易リレーショナルデータベースです。特別なソフトウェアをインストールする必要もなく、Windows 以外のプラットフォームにも対応しており、簡単軽量なものです。

SQLite データベースでは、一つの物理ファイルに全テーブルが格納されています。この物理ファイル名は、データベース テーブルの「位置」欄に設定されたものになります。例えば本書で扱っている「PETSHOP Offline」データベースでは、Petshop.sqlite という名前になります。

このファイルはどこに作成されるのかというと、プロットフォーム毎に規定されているローカルストレージの場所に格納されます。

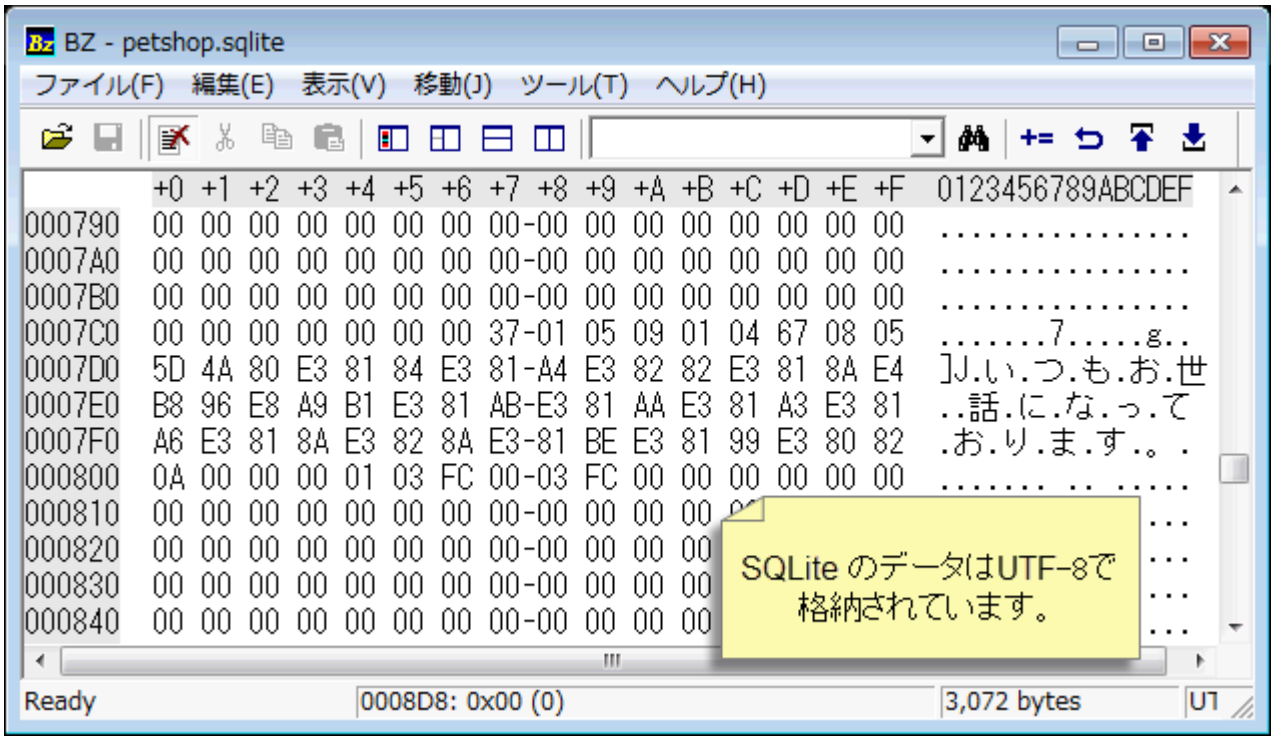
Windows の場合には、ユーザ毎の一時ファイル用フォルダ %TEMP% の下に、「MagicRIACache」という名前のサブフォルダを作り、ここに更にサーバ名でサブフォルダを作ったところに格納されます。すなわち、以下のような名前となります。

%TEMP%¥MagicRIACache¥（サーバ名）¥petshop.sqlite



Mobile の場合には、「Sandbox」と呼ばれている、各アプリ専用のデータ領域に格納され、一般には他のアプリからは参照することができないようになっています。

この SQLite ファイルを、バイナリエディタを使って中身を見てみると、以下のようなデータがあります。



このデータ構造については、SQLite はオープンソースなので SQLite のサイトやソースコードを見ればわかりますが、ここでは立ち入りません。

Magic のローカルデータベースとして利用する際に、覚えておくとちょっと便利なこととしては、次のようなことがあります。

- 定義、データともに一つの物理ファイル（ここでは「petshop.sqlite」ファイル）に格納されています。
- 日本語データは UTF-8 で格納されています。
- データは暗号化されていません。セキュリティ上、注意が必要です。将来のバージョンで対応が予定されています。

2.5 ローカル テーブルについての制限事項

ローカル テーブルを利用するには、いくつかの制限があります。
① 同一タスク内で、ローカルテーブルとサーバテーブルの混在はできません。
② サーバ テーブルはオフライン RIA タスクで利用できません。

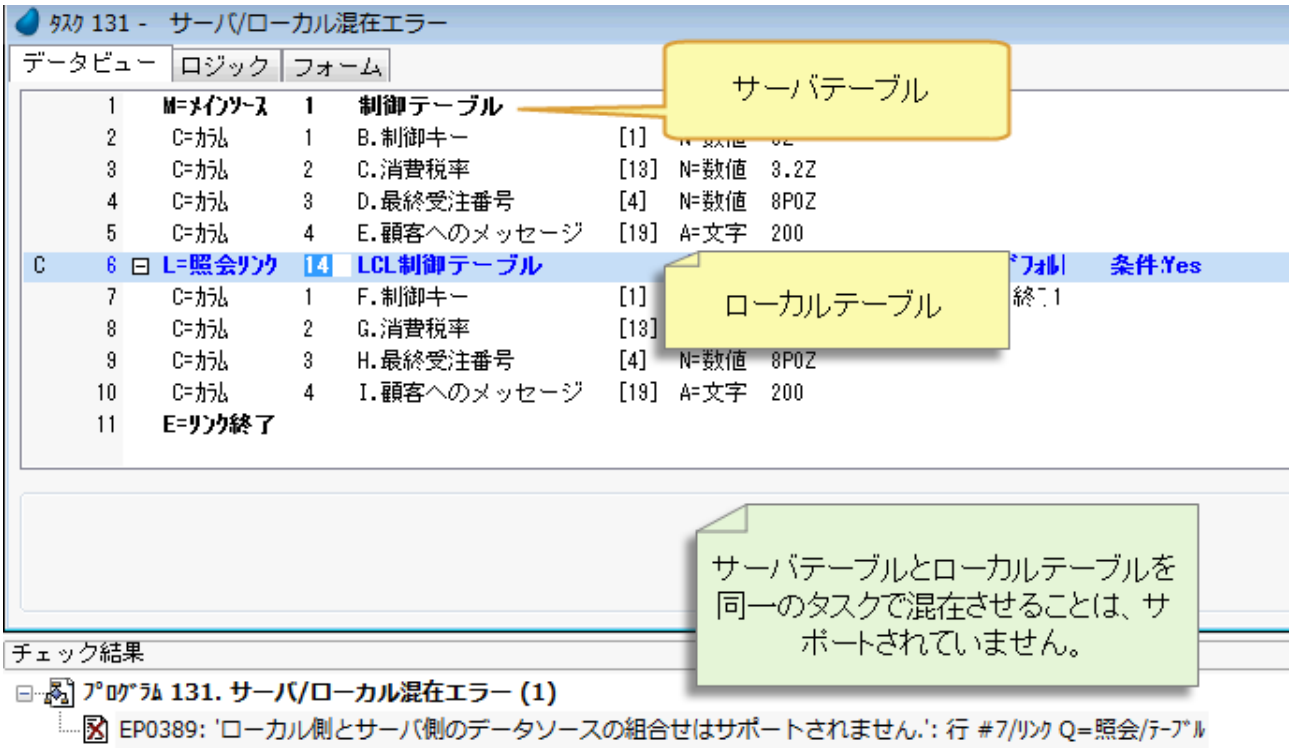
ローカルテーブルはデバイスのローカルファイル上にある、という構造上、いくつかの制約があります。実際にオフライン アプリケーションを開発する際には、ローカル テーブルに関する制約事項を留意しておく必要があります。

2.5.1 サーバ テーブルとローカル テーブルの組み合わせ

同一のタスクで、サーバ テーブルとローカル テーブルの混在はできません。

例えば、メインデータソースがサーバ テーブルであり、リンク データソースがローカル テーブル、というような組み合わせは不可です。

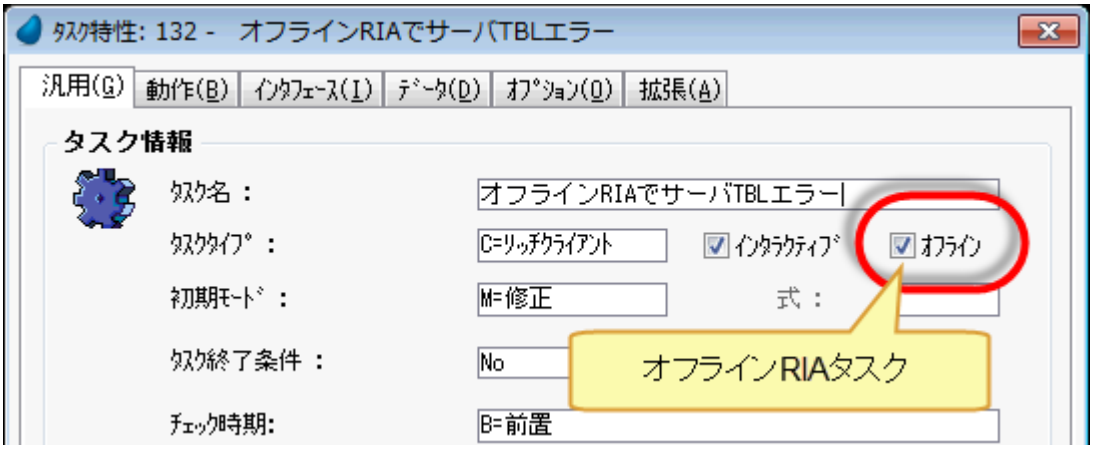
このような組み合わせのタスクを作った場合、F8 のシンタックスチェックでエラーが報告されます。



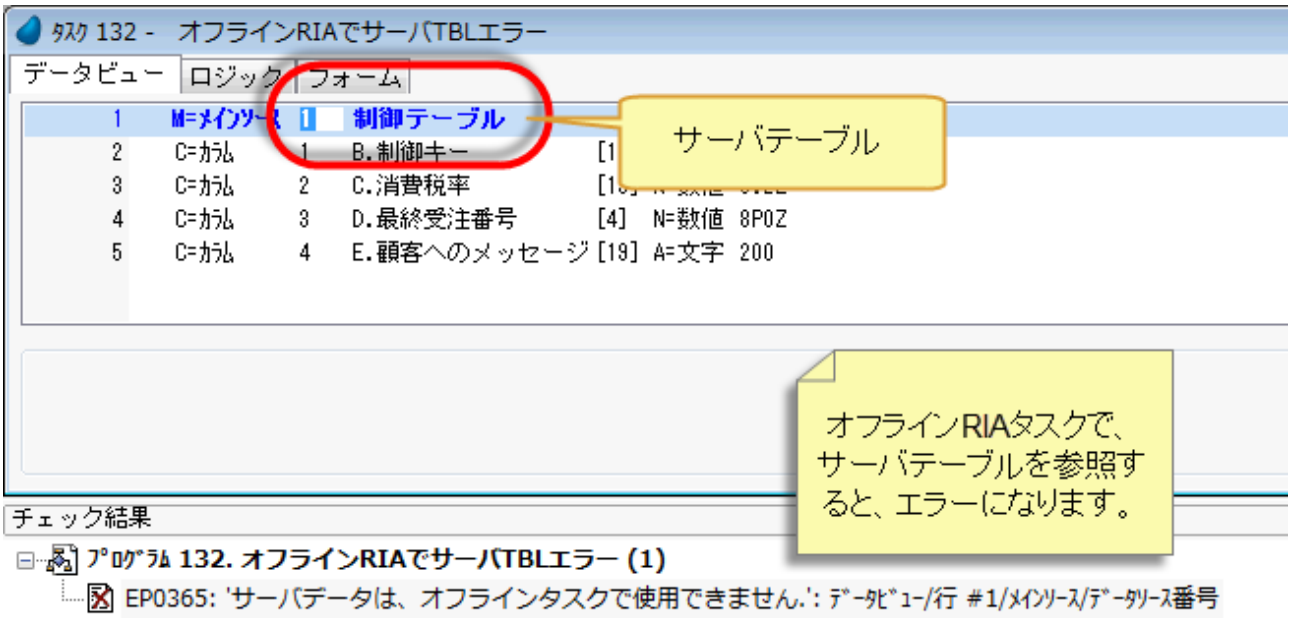
2.5.2 タスクの種類とテーブルの種類の組み合わせ

Magic RIA タスクには、タスクの種類として、RIA オンライン と RIA オフラインとの 2 種類があります。そして、テーブルの種類としては サーバ テーブル、ローカル テーブルの 2 種類があります。

これの組み合わせとしては、２×２＝４通りあるのですが、このうち、RIA オフラインとサーバテーブルの組み合わせは不可です。オフラインタスクというのは、サーバへの接続がなくても実行できるのが特徴ですが、サーバ テーブルはサーバへの接続がなければアクセスできないので、矛盾するからです。



このような設定を行った場合、F8 のシンタックスチェックでエラーが報告されます。



2.6 オフライン起動のメニュープログラム

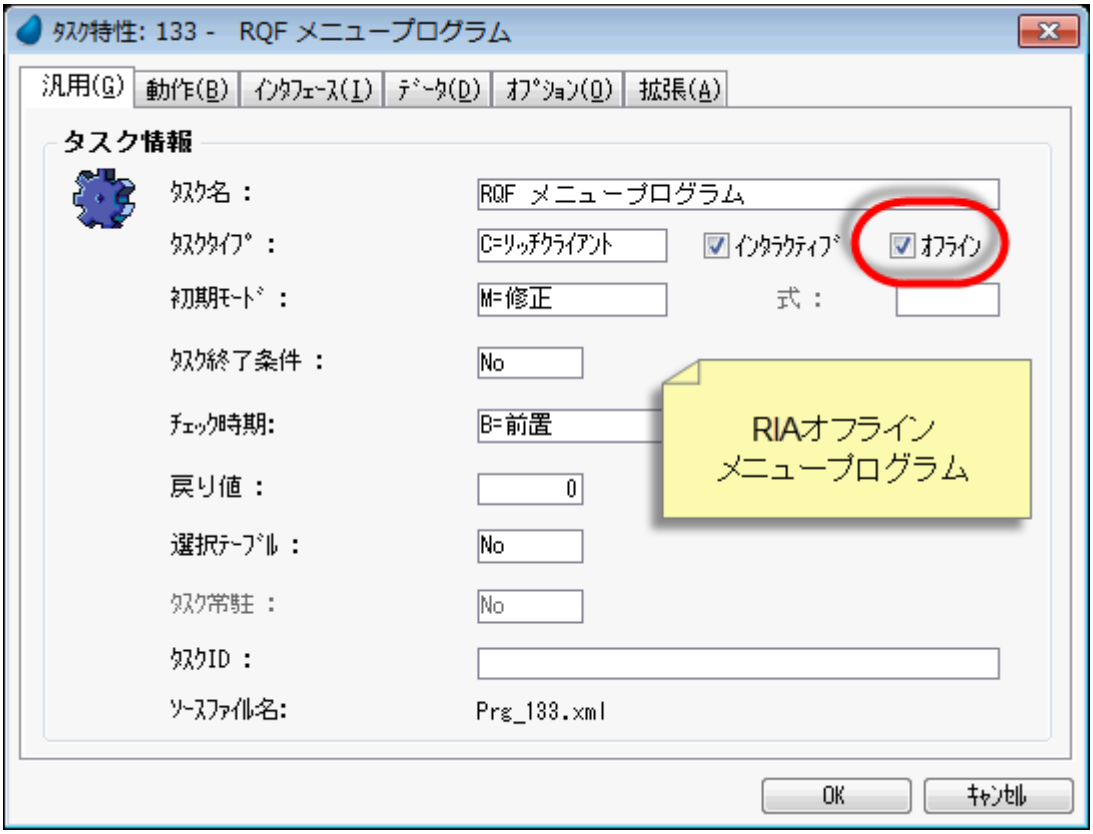
オフラインで起動するメニュープログラムを作っておきましょう。

オフライン プログラム開発の第一歩として、テーブルを一つも使わない、メニュー プログラムを作ってみましょう。

2.6.1 メニュープログラムの作成

(1) オフライン プログラムの定義

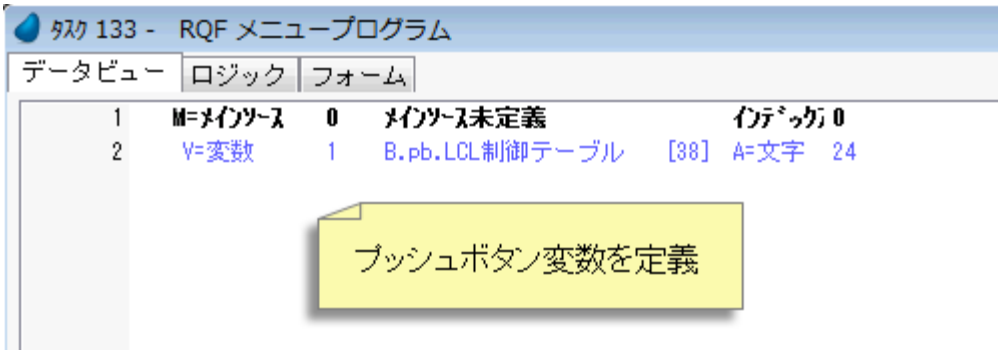
オフラインタスクを定義するのは簡単で、タスク特性の「オフライン」チェックボックスをオンにするだけです。これだけでこのタスクはオフラインでも実行できるタスクになります。



(2) データビューの定義

このほかは、通常の RIA タスクの開発と同じです。

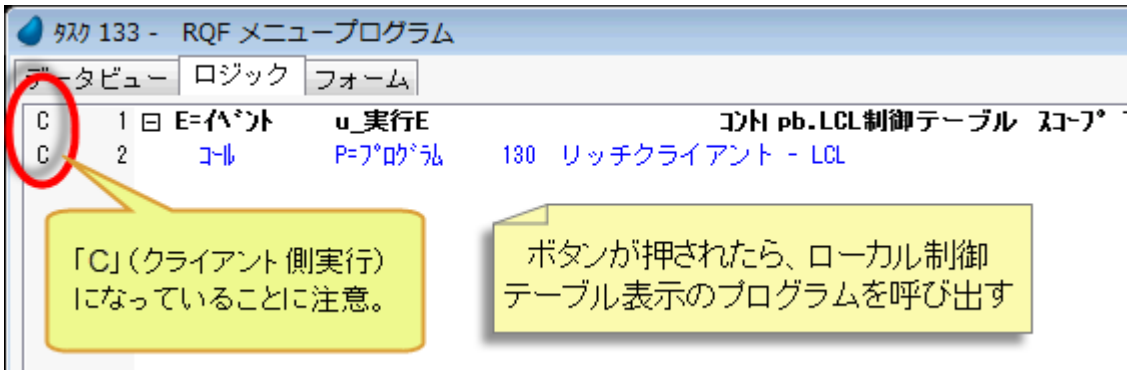
メニューに使うプッシュボタンを定義します。



(3) ロジックの定義

このプッシュボタンが押されたときに実行すべきロジックを、ロジック エディタで定義します。ここでは別のプログラム（プログラム番号 130 番「リッチクライアント – LCL」）をコールプログラムで呼び出します。

このプログラムは、先に APG で作成したものです。



ここで、ロジックの左端に表示されるインジケータが「C」になっていることに注意してください。これは「クライアント側での実行」を意味します。オフラインタスクの実行にあたっては、一切サーバへのアクセスを行いません。そのため、別タスクへの呼び出し（コールコマンド）も、クライアント側での実行となります。

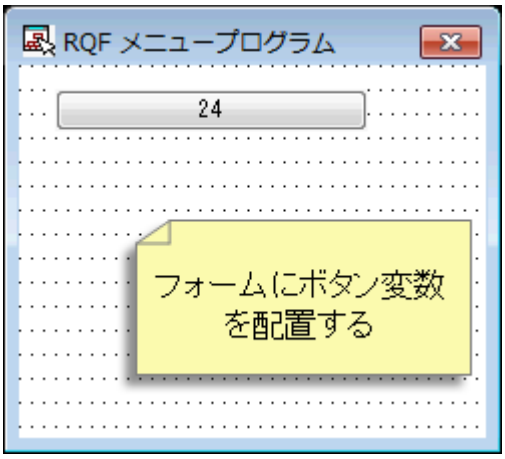
オフラインタスクではすべてクライアント側での実行となるので、逆に言うと、サーバへのアクセスが必要となるようなオペレーションを行うことができません。

上記の例では、呼び出すタスクがオフライン RIA タスクなので、「コール」コマンドの実行がクライアント側だけで行うことができます。しかし、呼び出すタスクがオフライン RIA タスクでない場合には、サーバ側へのアクセスが必要になるため、実行することができません。

このようなプログラムを作った場合には、F8 シンタックスチェックで、エラーが報告されます。

(4) フォームの定義

最後に、メインフォームにこのボタンを張り付ければ、一番簡単なメニュープログラムの出来上がりです。

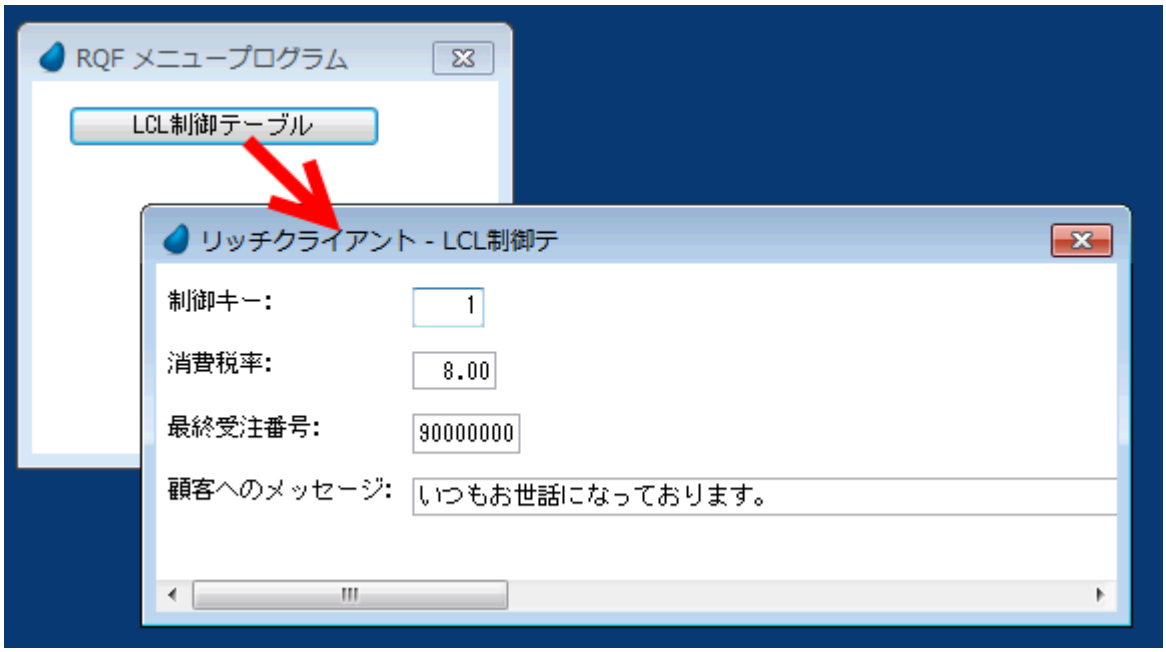


2.6.2 メニュープログラムの実行（F7）

ではこのプログラムを早速実行してみましょう。

オフラインタスクでも、Studio からの起動は普通の RIA タスクと同じで、F7 キーを押して実行できます。

ボタン「LCL 制御テーブル」を押してみましょう。APG で作成した、「リッチクライアント – LCL 制御テ」という名前のプログラムが起動されるはずです。



（注: このイメージでは、データが表示されていますが、実際にはこの時点ではローカル テーブルが空のはずなので、データは表示されないはずです）

2.7 オフラインで起動するには？

RIA サーバに全く接続せず、オフラインで起動するには、アプリケーションの実行特性で ConnectOnStartup 特性を N に指定します。

オフラインのアプリケーションでは、最初にインストールした後、次回以降起動する場合には、サーバにアクセスすることなく、オフラインのみで起動したい、という要望があると思います。このような場合には、RIA クライアントの実行特性で ConnectOnStartUp 特性を N に指定します。

2.7.1 ConnectOnStartup 特性

ConnectOnStartup 特性の説明を、リファレンスファイル（リファレンスガイド > Web 開発 > リッチクライアントアプリケーション > リッチクライアントインタフェースビルダ > アプリケーションの実行特性）から引用します。

特性	内容	モバイル
ConnectOnStartup	アプリケーションが実行毎にサーバに接続するか、メタデータを更新する時だけ接続するかを定義します。 Y …… （デフォルト）起動する度にサーバに接続します N …… メタデータが変更された（メタデータが変更されたというエラーがサーバから返された後の、次の実行時を意味します）場合だけサーバに接続しようとします。クライアントが起動時にサーバにアクセスする必要はない場合は、この特性を N に設定すると効果的です。 開始プログラムがオフラインプログラムの場合のみ、この特性は関連します。 サポートバージョン: 2.4	Android, iOS

この特性値をどこで設定するかは、ClickOnce を使って起動するか、あるいは mgxpaRIA.exe を直接起動する方法を使うか、によって異なります。以下に説明します。

2.7.2 ClickOnce を使ってインストールする方法

ClickOnce を使って RIA クライアントモジュールをインストールする方法では、インターフェースビルダで HTML 等を作成します。この方法は、普通の RIA の場合と同じなので、説明は省略します。

出来上がった（アプリケーション名）.publish.html ファイルの xml 部分で ConnectOnStartup プロパティを追加します。

例えば、次のようになります。ここで、最後の property に指定してあるのが、追加した行です。

```
<body onload="initialize()">
   <xml id="rcExecProps">
      <properties>
         <property key="protocol" val="http"/>
         <property key="server" val="MyServer"/>
         <property key="requester" val="/Magic25Scripts/MGrqispi.dll"/>
         <property key="appname" val=" rc_ps_ofl_25 "/>
         <property key="prgname" val="Top"/>
         <property key="arguments" val=""/>
         <property key="envvars" val=""/>
         <property key="UseWindowsXPThemes" val="Y"/>
         <property key="InternalLogLevel" val=""/>
         <property key="InternalLogFile" val=""/>
         <property key="InternalLogSync" val="Session"/>
         <property key="LogClientSequenceForActivityMonitor" val="N"/>
         <property key="DisplayStatisticInformation" val="N"/>
         <property key="HTTPCompressionLevel" val="Normal"/>
         <property key="FirstHTTPRequestTimeout" val="5"/>
         <property key="LogonWindowIconURL" val=""/>
         <property key="LogonImageURL" val=""/>
         <property key="LogonWindowTitle" val="ログオン"/>
         <property key="LogonGroupTitle" val="ログオンパラメータ"/>
         <property key="LogonMessageCaption" val="ユーザ ID とパスワードを入力してください．"/>
         <property key="LogonUserIDCaption" val="ﾕｰｻﾞ ID"/>
         <property key="LogonPasswordCaption" val="ﾊﾟｽﾜｰﾄﾞ"/>
         <property key="LogonOKCaption" val="OK"/>
         <property key="LogonCancelCaption" val="ｷｬﾝｾﾙ"/>
         <property key="ConnectOnStartup" val="N"/>
      </properties>
   </xml>
```

このようにすると、実行時には次のような動作になります。

1. まだクライアント PC に RIA クライアントモジュールがインストールされていない状態では、最初だけ Web ブラウザを使って、RIA サーバに接続し、ウィザードで作成・公開された .publish.html ファイルを開きます。従ってこの状態では、当然、オンラインでなければなりません。
2. 「起動」ボタンを押します。これにより、RIA クライアントモジュール等がダウンロード・インストールされます。
3. 更に、RIA クライアントモジュールが RIA サーバに接続し、RIA アプリケーションが開始されます。このときに、アプリケーションのすべてのオフラインタスクの定義が自動的にダウンロードされ、クライアントのキャッシュに格納されます。これで、オフラインのみで動作できるようになります。
4. 次回にこの RIA アプリケーションを起動するには、Windows のスタートメニューからこのアプリケーションのショートカットを選択します。
5. このときには、ConnectOnStartup が N に設定されているので、サーバへのアクセスは起こらず、アプリケーションはクライアントのキャッシュに格納されている情報だけで、オフライン動作するようになります。従ってこの時には、ネットワーク接続は切れていても大丈夫です。

2.7.3 MgxpaRIA.exe を直接起動する方法

ClickOnce を用いずに、mgxpaRIA.exe を直接起動する方法では、MgxpaRIA.exe と共に提供する execution.properties ファイルに、ConnectOnStartup 情報を設定します。

```
<properties>
   <property key="protocol" val="http"/>
   <property key="server" val="MyServer"/>
   <property key="requester" val="/Magic25Scripts/MGrqispi.dll"/>
   <property key="appname" val=" rc_ps_ofl_25 "/>
   <property key="prgname" val="Top"/>
   <property key="arguments" val=""/>
   <property key="envvars" val=""/>
   <property key="UseWindowsXPThemes" val="Y"/>
   <property key="InternalLogLevel" val=""/>
   <property key="InternalLogFile" val=""/>
   <property key="InternalLogSync" val="Session"/>
   <property key="LogClientSequenceForActivityMonitor" val="N"/>
   <property key="DisplayStatisticInformation" val="N"/>
   <property key="HTTPCompressionLevel" val="Normal"/>
   <property key="FirstHTTPRequestTimeout" val="5"/>
   <property key="LogonWindowIconURL" val=""/>
   <property key="LogonImageURL" val=""/>
   <property key="LogonWindowTitle" val="ログオン"/>
   <property key="LogonGroupTitle" val="ログオンパラメータ"/>
   <property key="LogonMessageCaption" val="ユーザ ID とパスワードを入力してください。"/>
   <property key="LogonUserIDCaption" val="ユーザ ID"/>
   <property key="LogonPasswordCaption" val="パスワード"/>
   <property key="LogonOKCaption" val="OK"/>
   <property key="LogonCancelCaption" val="ｷｬﾝｾﾙ"/>
   <property key="ConnectOnStartup" val="N"/>
</properties>
```

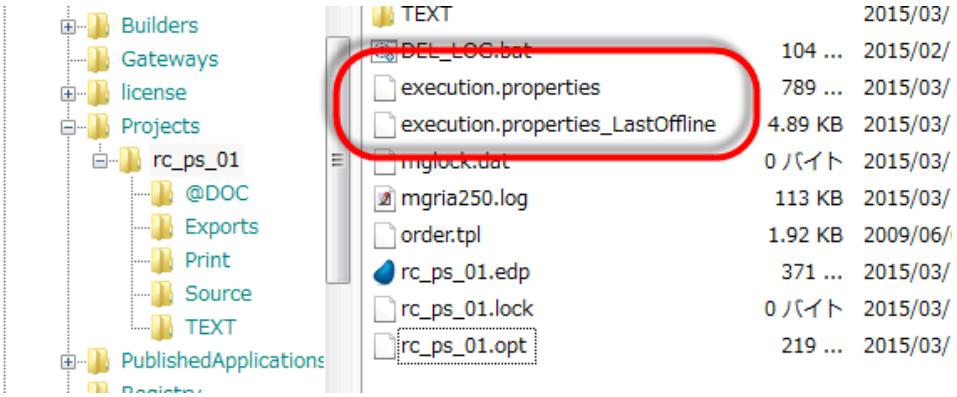
このファイルの形式は、.publish.html の xml 部分の内容とまったく同じです。

この方法でも、最初に1回だけ、RIA サーバへ接続し、環境情報やオフラインタスクの情報をダウンロードしてクライアントのキャッシュに格納しておく必要があります。

2.8 execution.properties_Lastoffline というファイルは何ですか？

オフラインの状態を記録しておくファイルです。

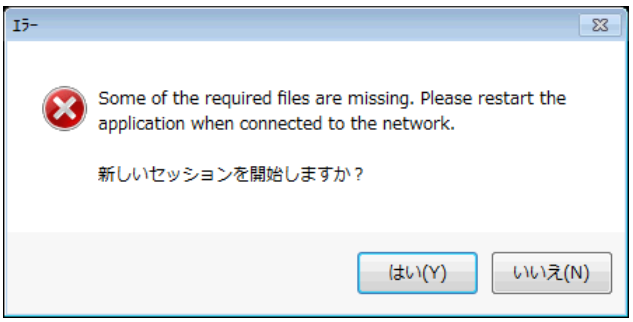
オフライン プログラムを実行すると、execution.properties と同じフォルダに、execution.properties_Lastoffline という名前のファイルができます。これはオフラインプログラムの状態を記憶しておくための内部データが暗号化されて格納されています。



運用時にはあまり起こることではありませんが、ClickOnce による起動、MgxpaRIA.exe の直接起動による起動、および F7 起動を混同して使うことはできません。それぞれの方法ごとに、execution.properties_Lastoffline ファイルが別のフォルダに作成され、実行のたびに状態が変更されるので、以前の状態と矛盾した状態になるからです。このような場合には、下のようなエラーが表示されます。

このエラーが出た場合には、execution.properties_Lastoffline の情報が古くなってしまっていることを意味するので、このファイルを削除して、再度起動してください。

この場合には、ConnectOnStart の設定に関わらず、サーバへの接続が必要となります。



3 データの単純ダウンロード

ここでは、サーバ テーブルのデータをローカル テーブルにダウンロードする際の技法および設計方針、注意事項等について説明します。

データのダウンロードには、本章で説明する方法の他に、データベース全体を一括でダウンロードまたはアップロードする方法があります。
この手法は非常に効率が良い方法ですが、制限も多いものなので、初期化時のデーター括ダウンロードなどの場合に用います。「第 6 章 データベースの一括転送」の章で説明します。

3.1 データのダウンロードをするには？ ①親子タスクにより

親タスクでサーバデータを読み込み、子タスクでローカルデータに書き込みます。

1レコードづつの処理になるので、時間がかかります。

クラサバのアプリでテーブルのレコードをコピーするには、バッチタスクを使って、メインデータソースとしてコピー元のテーブルを指定し、リンクコマンドでコピー先のレコードを作成する、という手法が一般的です。

しかし、ローカル テーブルに関する制限事項の一つに、「同一タスクでサーバ テーブルとローカル テーブルを混在させることができない」というものがありますので、オフライン RIA アプリでこの手法をそのまま使うことができません。

この制限を回避するために、コピー先テーブルのレコードを作成するためのサブタスクを作り、親タスクの1レコードごとに子タスクを呼び出すようにします。

この手法では、親タスクはサーバ テーブルを参照するので、オンライン RIA タスクである必要があります。一方、サブタスクはローカル テーブルを参照するので、オフライン RIA タスクである必要があります。いずれも、ユーザとのインタラクションは必要ないので、「インタラクティブ」フラグはオフにします。

これで制限事項を回避し、実行してコピーできますが、1レコードごとにサブタスクを呼び出すようになるので、パフォーマンスは悪いです。コピーレコード数が少ない場合にだけ採用すべき方法です。

3.2 データのダウンロードをするには？ ② DataViewToData-Source 関数を使う

より効率よくデータコピーを行うために、DataViewToDataSource という組み込み関数が用意されていますので、これを使います。

3.2.1 DataViewToDataSource 関数とは？

DataViewToDataSource() 関数は、一つの関数の実行だけで、複数のレコードのアップロード・ダウンロードを効率よく実行することができます。この関数は、ダウンロード、アップロード 両方の方向のデータ転送に用いることができます。

この関数の説明を、以下にリファレンスから転記します。（リファレンスガイド > 式エディタ > 関数ディレクトリ > DataViewToDataSource）

DataViewToDataSource	
現在のデータビューをデータソースに追加します。	
構文:	DataViewToDataSource (世代番号, タスクの項目名, 出力データソース番号, 出力データソース名, 出力カラム名)
パラメータ:	世代番号 …… タスクの階層位置を表す番号。カレントのタスクが 0、親タスクが 1、その親タスクが 2 などとなります。 タスクの項目名（文字） …… 出力する項目の名前をカンマ区切りでリストアップした文字列。現在のタスクの項目のみ有効です。大文字小文字を区別します。空白を指定した場合、すべてのタスクの項目が出力されます。 出力データソース番号（数値） …… [データ]リポジトリ上の通番を表す数値（例: '3'DSOURCE）。このパラメータは必須です。 出力データソース名（文字） ……このパラメータは、ソース番号の代わりに使用することができます。必要ない場合は、空白('')で指定してください。必要な場合のみ、データソース名を表す文字列を指定してください。文字列には、パスを含めることもできます。パスが含まれない場合、現在のディレクトリとして扱います。 出力カラム名（文字） …… 出力先のデータソースで更新されるカラムのすべての名前をカンマ区切りで指定します。大文字小文字を区別します。 これは、Magic のカラム名（DB カラム名ではない）です。 空白は、タスク項目名で定義された名前と同じ値が使用されることを意味します。
戻り値:	論理値 :以下の値が返ります。 True …… 現在のデータビューがデータソースに追加された場合 False …… データビューが追加されなかった場合
用途:	以下のような場合にこの関数を使用することができます。 データを表示するインタラクティブなリッチクライアントタスクで、データビューを出力したい

	場合 非インタラクティブなリッチクライアントタスクで、データを表示することなく出力したい場合この場合、関数そのものがデータビューを作成してスキャンするため、データビュー定義を保持するタスクはレコードをスキャンすることなく終了することができます。このタスクは、以下のように定義することができます。 メインデータソースを定義し、出力したいデータに基づいて、カラムに範囲条件を指定します。 [タスク後]ロジックユニットで関数を呼び出します。 [タスク終了条件]特性を「Y」に設定し、[チェック時期]特性を「前置」に設定します。この場合は、タスクは全てのレコードのスキャンするわけではありません。
注意事項:	この関数は、リッチクライアントでのみサポートされます。 メインデータソースと出力先のデータソースを同じ側(ローカル側とサーバ側)にすることはできません。 出力先のデータソースが存在しない場合、データソースが作成されます。 レコードが出力先のデータソース内に存在する場合、更新されます。 この関数を使用する場合、ユニークなインデックスを出力先のデータソースに定義しておかなければなりません。そして、ユニークインデックスのセグメントが全て選択されていなければなりません。 サーバ側のデータソースを更新するとき、更新は新しい物理トランザクションで行われます。 [NULL 値可]特性が「No」に設定されたカラムは、必ず更新するようにしてください。 ローカルのデータソースを更新する場合: 出力先のデータソースのデータベース上でオープンされたトランザクションがない場合、トランザクションは挿入が開始されるオープンされ、関数の処理が終了するとクローズされます。 出力先のデータソースのデータベース上でトランザクションがオープンされている場合、変更はトランザクションの一部で行われトランザクションがクローズされる時に一緒にコミットされます。

3.2.2 実際の利用例

DataViewToDataSource() を使ったデータダウンロードの例として、顧客マスタ のダウンロードを行うプログラムを作ってみます。

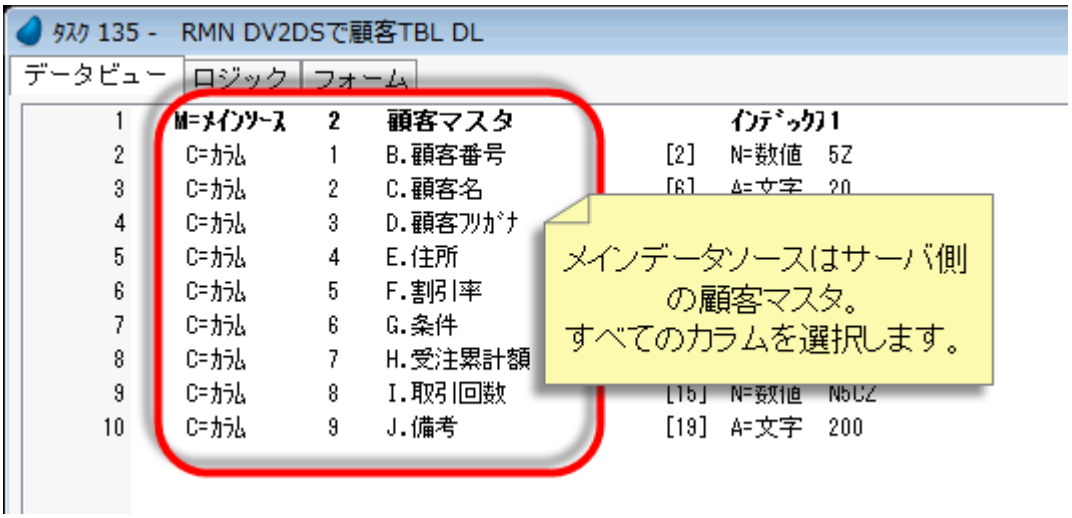
タスク特性:

タスクタイプ:	RIA
オフライン:	No (チェックを外す): メインデータソースがサーバ テーブルなので、オフラインにすることができません。
インタラクティブ:	No (チェックを外す)
終了:	Yes
評価 :	A=後置: すなわち、レコードサイクルを1回だけ実行して、終了します。

データビュー:

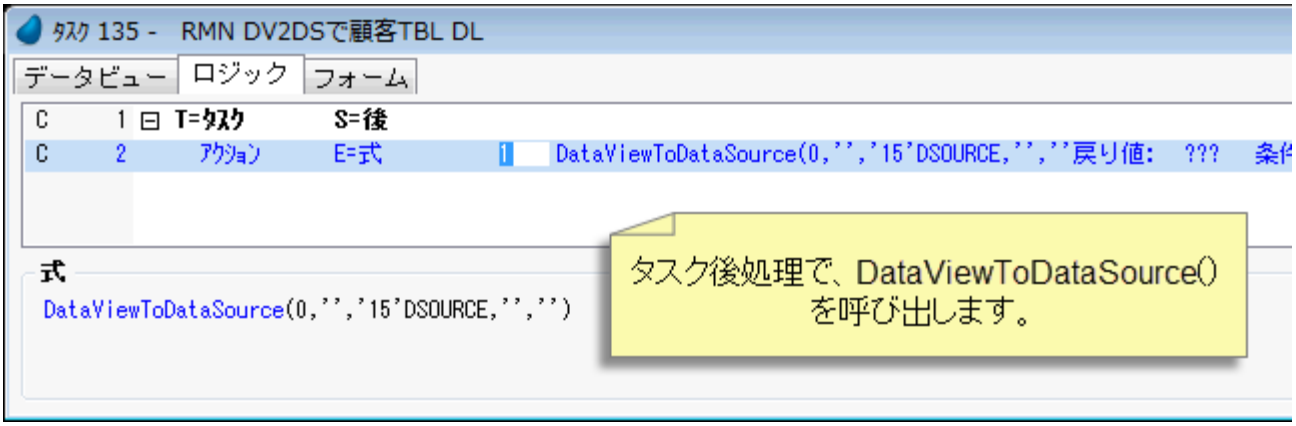
メインデータソース:　TBL#2「顧客マスタ」テーブル

データビュー:　すべてのカラムを選択します。



ロジック:

タスク後処理:　アクションコマンドで DataViewToDataSource 関数を実行。



3.3 DataViewToDataSource 実行時にデータ重複があるとどうなりますか？

一意キーをキーとして、上書き（UPDATE）されます。

DataViewToDataSource() 関数でレコードをダウンロードすることを繰り返した場合、データ重複が起こる可能性があります。このような場合には自動的に、一意キーをキーとして、上書き更新するようになります。

具体的には内部の処理の流れは次のようになります。

1. 最初に INSERT ステートメントを実行する。これで成功すれば、データ重複はないということなので、次のレコードに進みます。
2. 「重複キー」で失敗した場合には、
   (ア) SELECT ステートメントを実行して、既存レコードを確認します。
   (イ) UPDATE ステートメントを実行して、上書き更新します。

実際の実行内容を、ゲートウェイログで確認すると、次のようになります。ログファイルには実際にはもっといろいろな情報が記録されますが、必要な部分だけを抜粋しています。INSERT が失敗した後、SELECT、UPDATE 文が発行されているようすがわかります。

STMT: INSERT INTO PS1 顧客 (顧客番号 , 顧客名 , 顧客ﾌﾘｶﾞﾅ , 住所 , 割引率 , 条件 , 受注累計額 , 取引回数 , 備考) VALUES (1008,'千葉ペットショップ','ﾁﾊﾞﾍﾟｯﾄｼｮｯﾌﾟ','千葉県千葉市高柳　１２３４－１',9,'30日後支払い',68223,1,' 千葉ペットショップは12年来のお得意様です。対応には十分に気を付けて下さい。')

LastError(): <<<<< Abort due to constraint violation

columns 顧客ﾌﾘｶﾞﾅ, 顧客番号 are not unique

STMT: SELECT 顧客番号 ,顧客名 ,顧客ﾌﾘｶﾞﾅ ,住所 ,割引率 ,条件 ,受注累計額 ,取引回数 ,備考 ,rowid FROM PS1 顧客 WHERE rowid = 1 AND 顧客番号 = 1008 ORDER BY rowid ASC

STMT: UPDATE PS1 顧客 SET 顧客番号 = 1008, 顧客名 = '千葉ペットショップ', 顧客ﾌﾘｶﾞﾅ = 'ﾁﾊﾞﾍﾟｯﾄｼｮｯﾌﾟ', 住所 = '千葉県千葉市高柳　１２３４－１', 割引率 = 9, 条件 = '30日後支払い', 受注累計額 = 68223, 取引回数 = 1, 備考 = ' 千葉ペットショップは12年来のお得意様です。対応には十分に気を付けて下さい。' WHERE rowid = 1

3.4 ダウンロード対象以外の項目がタスクで定義されている場合、どうしますか？

以下の二つの方法が考えられます。
- 対象外の項目は親タスクに移動する。
- DataViewToDataSource の第 2 パラメータに項目名を明示的に指定する。

DataViewToDataSource() 関数の第 2 パラメータ（タスクの項目名）に空白文字を指定すると、当該タスクの「データビュー」に定義されている項目すべてがダウンロードの対象となります。
この機能は、いちいち項目名を列挙する必要がなくなるので便利な機能なのですが、時には、すべての項目を対象にしたくない場合もあります。例えば、パラメータ項目や変数項目、リンク項目がある場合などです。

対象となる項目を絞りたい場合には、次の二つの方法が考えられます。

3.4.1 パラメータや変数だけの場合

DataViewToDataSource() の対象から除外したい項目がパラメータや変数だけの場合には、タスクを親子に分けて、親タスクにはダウンロード対象にしたくない項目（パラメータ、変数など）を定義し、子タスクには対象となる項目だけを定義します。こうすれば、DataViewToDataSource() の第 2 パラメータを空白にして使うことができるようになります。

例えば、「最終更新日時」を範囲指定のパラメータとし、それ以降のレコードのみをダウンロード対象とする、というような場合には、次のようにします。

親タスク：

データビュー:	パラメータ項目定義「pi.最終更新日時」
タスク特性	終了＝Yes
	インタラクティブ＝No
タスク後処理	コール サブタスク

子タスク:

データビュー:	メインデータソース ＝ 顧客マスタ
	カラム: 顧客マスタのすべての項目。
	範囲: pi.最終更新日時 ＜ 最終更新日時
タスク特性:	終了＝Yes
	チェック時期＝A=後置
タスク後処理	DataViewToDataSource() 実行

3.4.2 リンク項目や、再計算の必要な代入式の設定してある変数などがある場合

DataViewToDataSource() から除外したい項目が、リンク項目とか、再計算の必要な代入式の設定してある変数などの場合には、親タスクで定義する、という手法は使えません。

このような場合には、DataViewToDataSource() の第 2 パラメータに、ダウンロード対象とする項目名を明示的に指定する必要があります。

この方法をとるとき、項目数が多いと全項目名を列挙するのが大変です。また、テーブル定義を変更した際に、それに合わせて変更する必要が出てくるので、メンテナンスの点でも不利です。

この問題を軽減するために、次の「3.5 DataViewToDataSource に渡すカラム一覧を簡単に作成する方法がありますか？」の手法を使うことができます。

3.5 DataViewToDataSource に渡すカラム一覧を簡単に作成する方法がありますか？

DataViewVars() 関数を使います。DataViewVars() と DataViewToDataSource() を一度に行う、汎用のダウンロードタスクを作ると便利です。

DataViewToDataSource() の第二パラメータ（タスクの項目名）に、明示的に項目名を列挙する必要のある場合、項目数が多いと手で書くのは大変です。また、テーブル定義が変更になった場合に、それに合わせて項目名も追加・修正しなければならず、メンテナンス性が悪くなります。

このために、DataViewVars() 関数を使って、項目名を列挙した文字列を自動的に作成してくれるようなタスクを定義しておくと大変に便利です。

3.5.1 DataViewVars() とは

DataViewVars() は、特定のタスク上に定義されたデータビューの項目名を、ベクトルデータとして返します。以下にリファレンスガイドからの抜粋を転記します。（リファレンスガイド > 式エディタ > 関数ディレクトリ > DataViewVars）

DataViewVars	
データビューの取得: タスクに定義されたデータビューやフォーム上に配置されたデータビューの項目名を取得します。	
構文:	DataViewVars (世代番号, オプション)
パラメータ:	世代番号 …… タスクの階層位置を表す番号。カレントのタスクが 0、親タスクが 1、その親タスクが 2 などとなります。 オプション（数値） …… 取得する項目の範囲を定義する数値。このパラメータは必須です。以下の値が指定できます。 0 …… タスクのデータビュー全体 1 …… フォーム上の全ての項目 2 …… 表示されたフォームの項目（表示される項目のみを対象とします。）
戻り値:	ベクトルデータ …… 文字型項目を含むベクトル型の値が返ります。ここには、データビューの項目名が格納されます。 オプションが「0」でない場合、ベクトル内の項目の順序は、フォームに表示されている順番になります。 オプションが「0」の場合、ベクトル内の項目の順序は、[データビュー]エディタに定義されている順番になります。 [世代番号]で定義されたタスクの定義項目のみ出力されます。
注意事項:	この関数は、フォーム項目を取得するためにどのような段階でも使用できます。 2番目のパラメータが「2」と評価された場合、表示されているかどうかは、現在のレコードで判断されます。データビューを取得できない[タスク前]で実行された場合、オプション指定は無視され、全てのフォーム項目を取得します。この場合、オプションの

	「1」と「2」では、同じ動作になります。 2番目のパラメータが不正な値と評価された場合、関数は空のベクトルデータを返します。 2番目のパラメータが「1」または「2」と評価され、フォーム上に項目が定義されていなかったり、タスクの［ウィンドウ表示］特性が「No」に設定されている場合、関数は空のベクトルデータを返します。

3.5.2 DataViewVars() から項目一覧を作成するには？

DataViewVars() 関数の戻り値は、項目名（文字型）をセルとするベクトルデータです。この戻り値から、項目一覧（項目名をカンマで区切って連結した文字型データ）を作成するには、「ブロック While」コマンドなどを使ってループを作り、ベクトルの全セルについて項目名を連結するようにします。
これの具体的な例は、次節「汎用ダウンロードタスク」で説明します。

3.5.3 対象項目を限定するには？

この関数のデフォルトでは、指定されたタスクで定義されているデービューの全項目が対象となります。全項目ではなく、対象となる項目を限定するには、次のような方法が考えられます。

1. 第 2 パラメータに 1（フォーム上のすべての項目、の意味）を指定し、対象カラムに含めたい項目をフォームに配置します。このフォームは実行時には表示されませんが、DataViewVars() 関数のためにだけ作成します。
2. 名前の命名規則で判断します。例えば、v から始まるのは変数（除外）、p から始まるのはパラメータ（除外）、x から始まる名前は実項目だが除外したいもの、などと言った命名規約を設けてアプリケーションを開発します。DataViewVars() の結果から項目一覧（文字データ）を作成する際に、項目名をチェックして、このような v、p、x で始まる項目名は除外する、というようなアルゴリズムにします。

3.6 汎用ダウンロードタスクを作れますか？

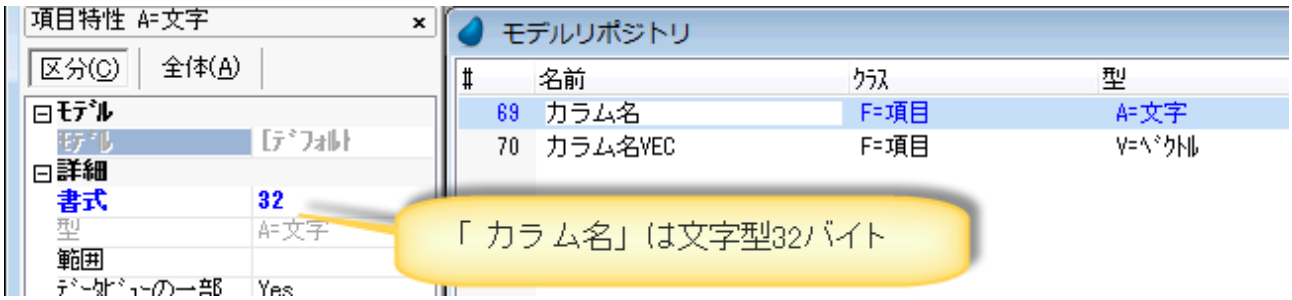
DataViewVars() と DataViewToDataSource() 関数とを組み合わせて、汎用のダウンロードタスクを作ることができます。

DataViewVars() 関数と DataViewToDataSource() 関数を組み合わせれば、汎用のダウンロード用タスクを作ることができます。以下に作り方を説明します。

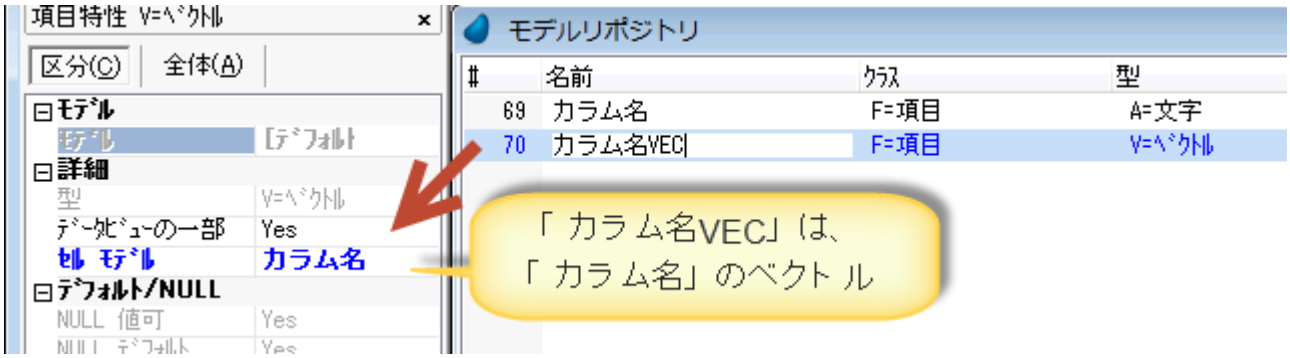
3.6.1 ベクトルモデルの定義

DataViewVars() 関数の戻り値は、文字型をセルとするベクトル型なので、あらかじめモデルリポジトリで定義しておかなければなりません。これには、次のような二つのモデルを定義します。

①「カラム名」モデル：項目名を格納するものです。項目名は最大32文字なので、文字型、書式は32として定義します。



②「カラム名 VEC」モデル：「カラム名」をセルとするベクトル型のモデルです。

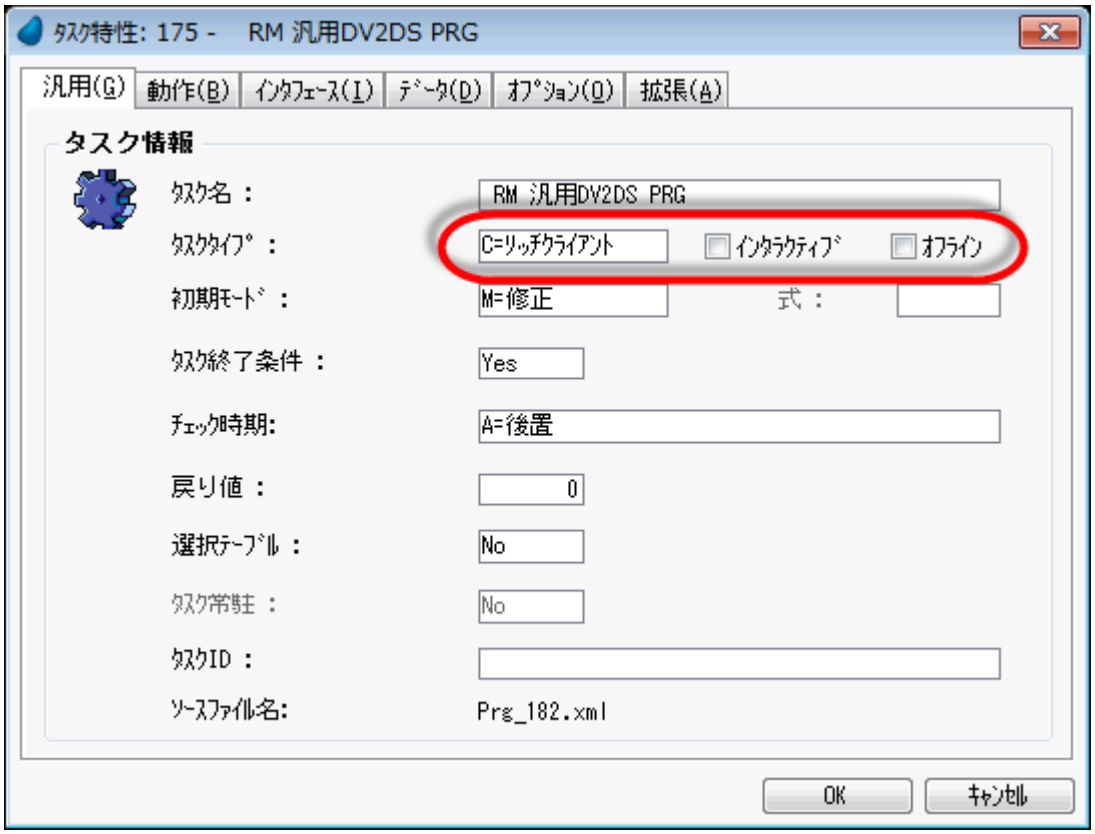


3.6.2 タスクの定義

タスク特性：

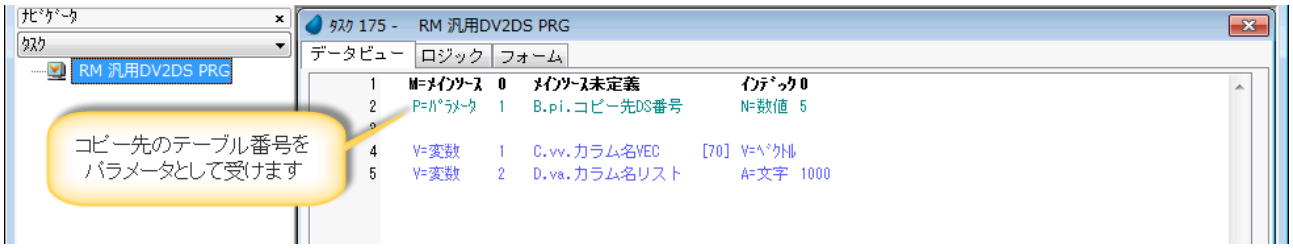
タスクタイプ	C=リッチクライアント
インタラクティブ	No（ユーザとの対話が必要ないので）

オフライン　　No（DataViewToDataSource() がサーバとの接続を必要とするので、オフラインにはできません）



データビュー:

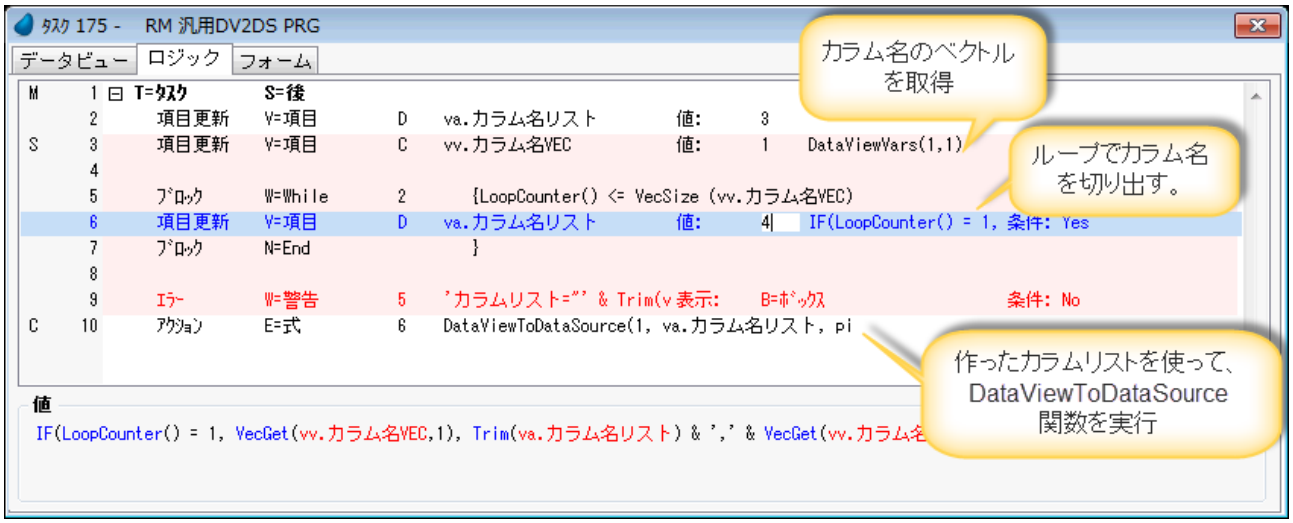
パラメータ　　コピー先のデータソース番号は、パラメータで受け取ります。



ロジック:

タスク後処理　　カラム名一覧は、DataViewVars(1,1) を使って作成します。

その一覧を使って、DataViewToDataSource (1, カラム名一覧, 出力データソース番号（パラメータ),'', '') を実行します。



この汎用ダウンロードタスクの利用例は、7.2.3 節「ダウンロードのロジック ②」で取り扱います。

このタスクで使っている DataViewToDataSource() 関数は、ダウンロード、アップロード共に双方向で使えるので、この汎用タスクもダウンロード、アップロード双方向に使えます。

3.7 コピー元テーブルとコピー先テーブルのカラム名が異なる場合、どうしますか？

DataViewToDataSource() の１番目のパラメータにコピー元テーブルのカラム一覧を指定し、５番目のパラメータに、対応するコピー先テーブルのカラム名の一覧を指定します。

大変面倒なので、コピー元とコピー先はできるだけ定義を一致させるのが一番です。

3.8 DataViewToDataSource 関数の内部の動作はどのように確認しますか？

ゲートウェイログ、および、RIA ログを使います。

DataViewToDataSource 関数は便利で、一つの関数で内部では多くのオペレーションを実行します。

逆に、レコードがちゃんと意図したとおりに処理されているかは、実行結果を見る以外に確認することができません。

意図したとおりになっていないとしたら、何が原因なのかを追及するにはログを追うのが一番確実です。

3.8.1 クライアント側ログ

RIA クライアント側の活動は、すべてクライアントのログファイルに記録されます。これには、ローカル データベースのゲートウェイのログも含まれます。

RIA クライアントのログを制御するパラメータは、実行特性に指定します。

特性名	値	意味
InternalLogLevel	NONE	ログを出力しません。
	BASIC	HTTP リクエストの受信と送信についてのみ記録されます。
	SERVER	HTTP リクエストと応答のみ出力します。
	SUPPORT	SERVER 指定の内容に加え、サーバ間のデータの内容を出力します。
	GUI	SUPPORT 指定の内容に加え、クライアントによって記録された GUI メッセージの一部を出力します。
	DEV	クライアントによって記録されたすべてのメッセージを出力します。
InternalLogFile	（ファイル名）	ログを出力するファイル名を指定します。設定されない場合、クライアントのデスクトップに以下のファイル名で保存されます。 Magicxpa_YYYY_MM_DD[.Process ID].log
InternalLogSync	None	.NET フレームワークのデフォルトにもとづいてログファイルが自動的に作成されます。
	Session	ログファイルは自動的には作成されません。このレベルは最も処理が早くなります。
	Message	ログファイルは、各メッセージ毎にオープン/クローズされます。このレベルは、ログの保全性には優れていますが動作が遅くなります。

この特性を設定する箇所は、どのように RIA アプリケーションを起動するのかによって変わります。

F7 実行時:

Studio から、F7 で実行する場合には、MAGIC.INI ファイルの [MAGIC_RIA] セクションに指定します。

実行時に作成される execution.properties ファイルにこれらのパラメータがコピーされます。

```
[MAGIC_RIA]
InternalLogLevel=S
InternalLogFile=mgRia25.log
InternalLogSync=Message
```

ClickOnce で起動する場合:

リッチクライアント インターフェース ビルダ で作成された publish.html ファイルの中の xml 部分に設定します。以下に例を記載します。

```
<body onload="initialize()">
   <xml id="rcExecProps">
      <properties>
         <property key="protocol" val="http"/>
         <property key="server" val="MGDEV"/>
         <property key="requester" val="/Magic25Scripts/MGrqispi.dll"/>
         <property key="appname" val="rc_ps_01"/>
         <property key="prgname" val="Menu"/>
         <property key="arguments" val=""/>
         <property key="envvars" val=""/>
         <property key="UseWindowsXPThemes" val="Y"/>
         <property key="InternalLogLevel" val="S"/>
         <property key="InternalLogFile" val="mgxpa.log"/>
         <property key="InternalLogSync" val="Session"/>
         <property key="LogClientSequenceForActivityMonitor" val="N"/>
         <property key="DisplayStatisticInformation" val="N"/>
         <property key="HTTPCompressionLevel" val="Normal"/>
         <property key="FirstHTTPRequestTimeout" val="5"/>
        …
      </properties>
   </xml>
```

この特性値の設定は、.publish.html ファイルを作成した後に、テキストエディタで直接編集して追加することができます。

ウィザードで作成する際には、いつもこの設定を入れたい、という場合には、実行プロパティ用テンプレートに記述しておくこともできます。実行プロパティ用テンプレート ファイルは、以下にあります。

<Magic インストールフォルダ>\Add_On\Builders\Templates\ExecutionProperties.tpl

MgxpaRIA.exe を直接起動する場合

MgxpaRIA.exe を直接起動する場合には、execution.properties ファイルに、これらの特性を設定します。

設定方法は、上記 ClickOnce の場合と同じです。

3.8.2 サーバ側ログ

サーバ側のログは、「ロギング」ダイアログ、または MAGIC.INI の [MAIGC_LOGGING] セクションおよび [MAGIC_DBMS] セクション（ゲートウェイログの設定）で行います。

これについては、通常の RIA タスクなどと同じなので、詳しい説明は省略します。

3.8.3 ログの読み方

ログには非常に多くの情報が記録されます。

データベースへの活動を確認するには、ログをテキストエディタで開いて、「STMT:」をキーワードにして検索すると、データベースに対して発行されている SQL 文を拾うことができます。

DataViewToDataSource() で、何件のレコードが対象になったのかを確認したい場合には、INSERT をキーにして検索すると良いでしょう。また、上書きされたレコード件数は UPDATE をキーにすればよいでしょう。

3.9 オフライン RIA タスクからダウンロードプログラムを呼び出すには？

イベントを介して呼び出します。

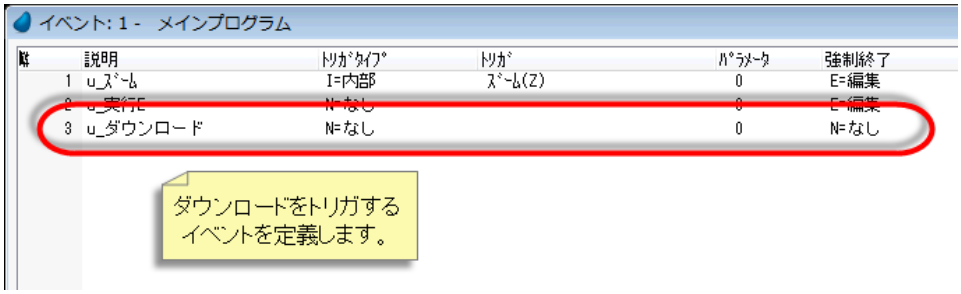
オフラインアプリケーションでは、サーバとの接続がない環境でも使いたいのですから、最初のメニュー画面などは、RIA オフラインタスクとして作成するのが普通です。一方、ダウンロードプログラムというのは、サーバテーブルを参照するものなので、必然的に RIA オンラインタスクになります。

ここで、オフライン RIA タスクの制限事項があり、オフライン RAI タスクから、オンライン RIA タスクを直接呼び出すことはできません。そのようなプログラムを作って実行するとエラーになります。
ではどうするかと言うと、グローバルイベントを定義して、イベント経由で呼び出すようにします。

次のように、プログラムを作成します。

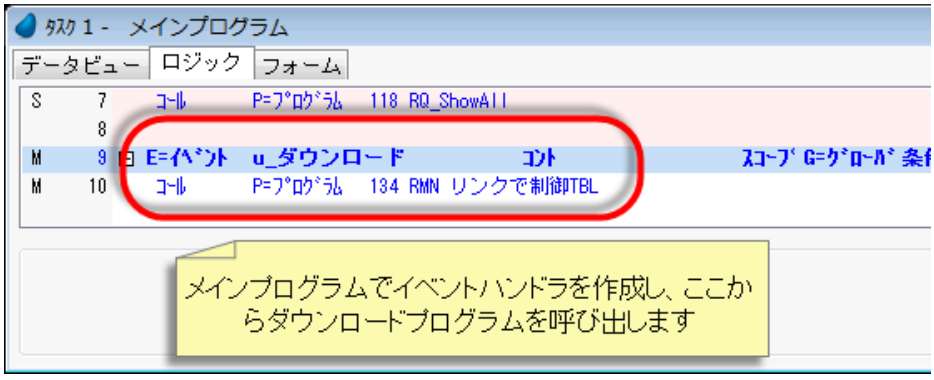
① グローバル イベントの定義

「メインプログラム」で、グローバルなユーザ定義イベント「u.マスターダウンロード」を定義します。「トリガタイプ」、「強制終了」ともに「N=なし」とし、これ以外には特別なことは必要ありません。



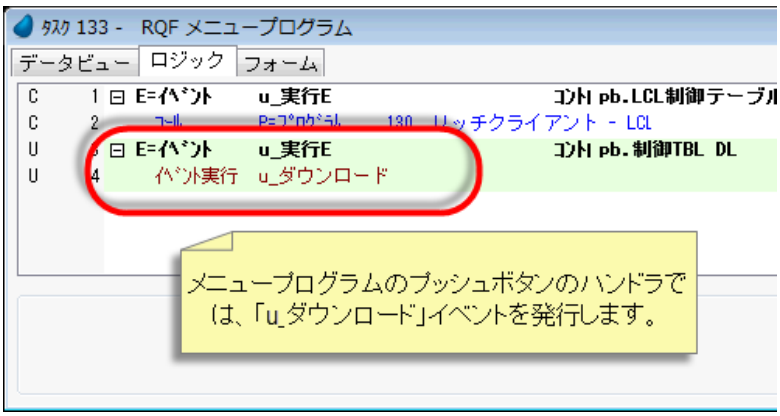
②「メインプログラム」でイベント ハンドラを定義

メインプログラムのロジックで、新規イベントハンドラを作成します。トリガとしては、上で定義したユーザイベントを指定します。そして、ハンドラ中でダウンロードプログラムを呼び出します。

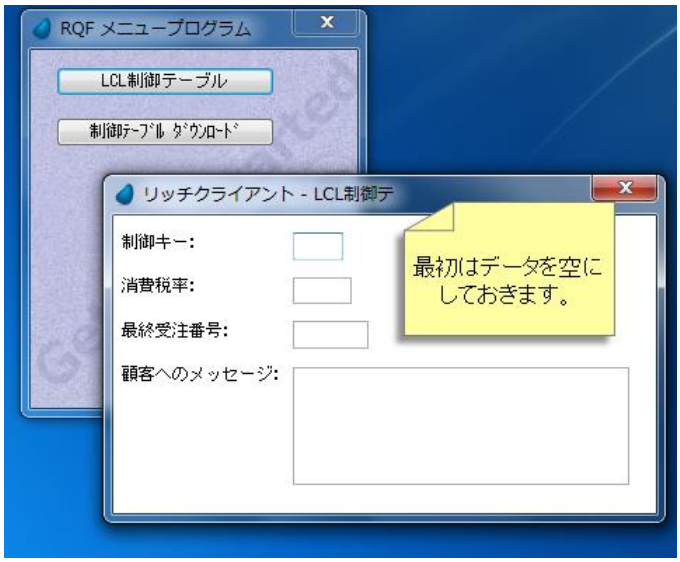


③ オフライン RIA タスクからは、「u.マスターダウンロード」イベントを実行させます（ウェイト＝Yes）。

右の例は、「メニュープログラム（RIA オフライン）」で「イベント実行」コマンドを実行させているものです。

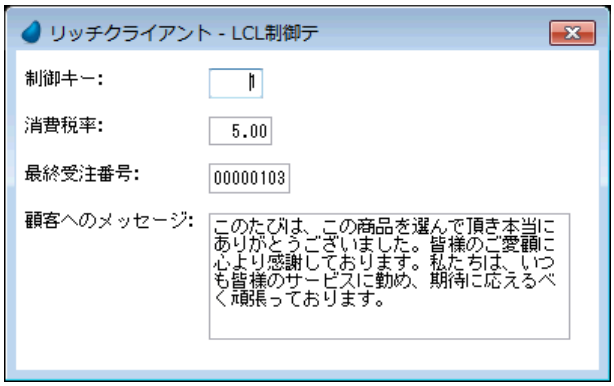


実行してみて、動作を確認します。
最初は、データが空です。



ダウンロードを実行します。

再度テーブル内部を表示させてみると、データがダウンロードされたことが確認できます。



3.10 リソースのダウンロードはどう行いますか？

イメージファイルなどのファイル類を一括して「リソース」と呼びます。

リソースは ServerFileToClient() 関数を使ってあらかじめダウンロードしておきます。

プログラムの背景やボタン、アイコンなどで参照するイメージファイルなどを、一般に「リソース」と呼びます。ここではリソースをどのようにサーバからローカルストレージにダウンロードするかを説明します。

リソースを使う例として、メニュープログラムの背景イメージを見てみると、最初は背景が指定されていないので、真っ白になっています。JPG イメージファイルを背景に設定してみることにしましょう。

背景の設定は通常の RIA タスクと同じく、フォームの「背景」プロパティに JPG ファイル名を設定するだけなのですが、オフライン RIA タスクの場合には、この JPG ファイルもまた、ローカル ストレージに置いておいて、サーバへのアクセスをしないようにする必要があります。

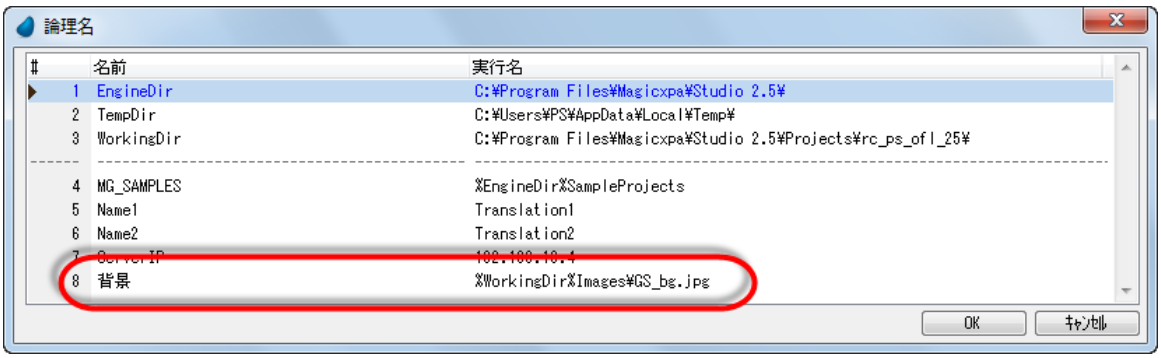
3.10.1 フォームに背景を設定する

背景となる JPG ファイルは、直接プログラムで指定することも可能ですが、システム構成の柔軟性を考慮に入れて、背景のイメージファイル名は論理名で設定するようにします。

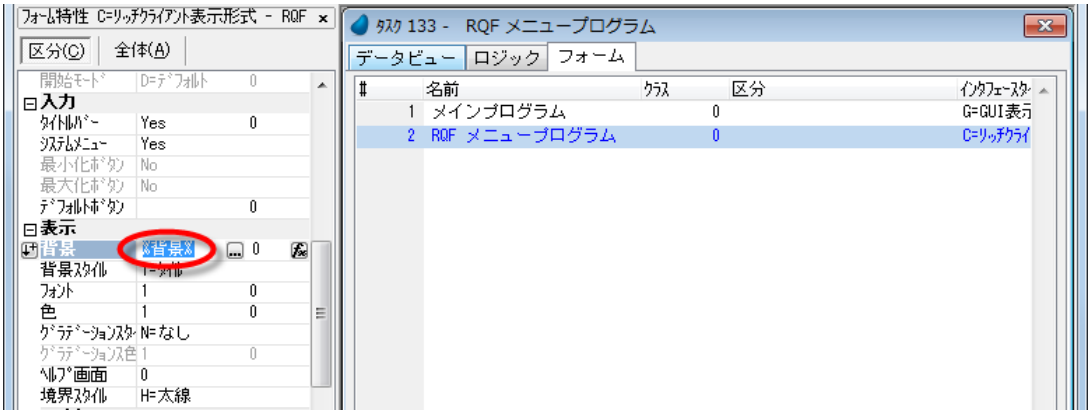
① 論理名テーブルを開いて、次の行を追加してください。

論理名：背景

実行名：%WorkingDir%Images\GS_bg.jpg

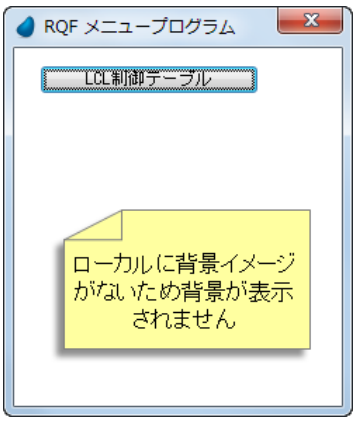


② ここで設定した論理名を、メニュープログラムの「背景」プロパティに設定します。



Studio のフォームエディタ上では、右図のように、設定された背景が表示されます。

メニュープログラムを F7 で実行してみると、この時点では、背景イメージとなる JPG ファイルがローカル ストレージに存在しないため、背景となる JPG ファイルが見つからず、背景が描画されません。



3.10.2 ServerFileToClient 関数

背景イメージが正しく表示されるためには、イメージ JPG ファイルをローカルストレージにあらかじめダウンロードしなければなりません。リソースファイルは自動的にはダウンロードされません。

リソースの同期をするには ServerFileToClient 関数を使ってリソースとなるファイルをコピーするのが便利です。

以下に、ServerFileToClient 関数の説明を リファレンスガイド（リファレンスガイド > 式エディタ > 関数ディレクトリ > ServerFileToClient） から抜粋します。

ServerFileToClient	
リッチクライアントのサーバ側からクライアントのキャッシュフォルダに指定されたファイルをコピーします。	
構文:	ServerFileToClient(ファイル名)
パラメータ:	ファイル名 …… サーバ側のファイルの名前。パスが指定されなかったり相対パスが指定された場合、アプリケーションのフォルダがカレントとして扱われます。 ワイルドカードやフォルダ名（末尾に'\'を設定）を指定することで、クライアントに複数のファイルを送ることができます（サポートバージョン:2.4）。フォルダを指定した場合は、フォルダ内のファイルだけで、サブフォルダはコピーされません。
戻り値:	文字型 …… コピーされたクライアント側のファイル名。サーバ側のパス名をもとにした名前になるため、変数項目に保存する場合はデータ長を長く定義してください。 （ファイルが見つからないなどの）エラーが発生した場合、空白が返ります。

	パス名やファイル名の「\」や「:」は、「_（アンダーバー）」に変換されます。
例:	ServerFileToClient ('c:\images\my.bmp') my.bmp ファイルをサーバ側の C:\image フォルダからクライアント側にコピーし、コピー先のフォルダ名+ファイル名を返します。
利用目的:	サーバ上の非公開フォルダ上のイメージまたは PDF ファイルを表示させたい場合 サーバからクライアントへファイルをコピーしたい場合 .NET メソッドにサーバリソースを渡したい場合
注意事項:	この関数は、リッチクライアントタスク、またはリッチクライアントとして使用するメインプログラムでのみ有効です。 ファイルをクライアントへコピーする前に、関数はサーバ上のファイルがクライアント側のキャッシュ内のファイルより新しいかどうか調べるために、リクエストを送ります。ファイルが同じタイムスタンプの場合、関数はファイルをクライアントにコピーしません。 関数が呼び出される度にタイムスタンプの確認を行うため、クライアントに同じファイルを何度も送ろうとすると、低帯域幅のネットワークではパフォーマンスの低下につながるかもしれません。このため、関数は 1 回だけ実行し、ワイルドカードやフォルダ名を使用することを推奨します。この場合、タイムスタンプのチェックは、1 つのリクエストに対して全てのファイルで行われます。更新されたファイルは、個別のリクエストを使用して、後で読み込まれます。 Windows クライアントの場合、キャッシュフォルダは、%TEMP%\MagicxpaRIACache\(ServerName) です。 接続中のデータベースファイルをコピーしようとすると失敗します。DbDiscnt 関数や ClientDbDiscnt 関数を実行して対応するデータベースを予め切断するようにしてください。
モバイル仕様:	この関数は、モバイル用の代替イメージを検索するメカニズムを利用して、サーバからクライアントにファイルをコピーします。 クライアントがモバイルの場合、Unicode の名前はサポートされません。

ServerFileToClient()はワイルドカードも指定できるため、複数ファイルをいっぺんにコピーして、パフォーマンスを改善することができます。

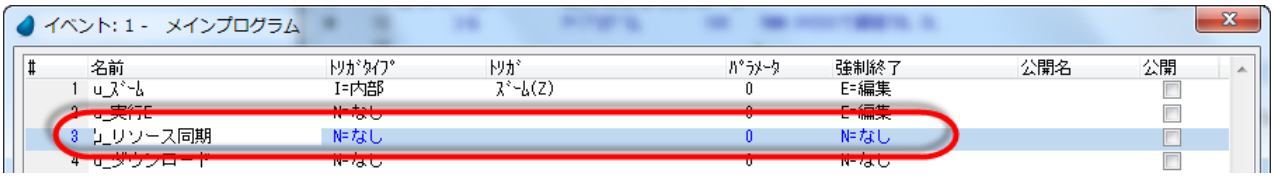
3.10.3 リソースの同期処理

ServerFileToClient() 関数を使う際には、これがサーバを呼び出す関数であり、オフライン RIA タスクの中で直接呼び出すことができない、という制約に留意する必要があります。

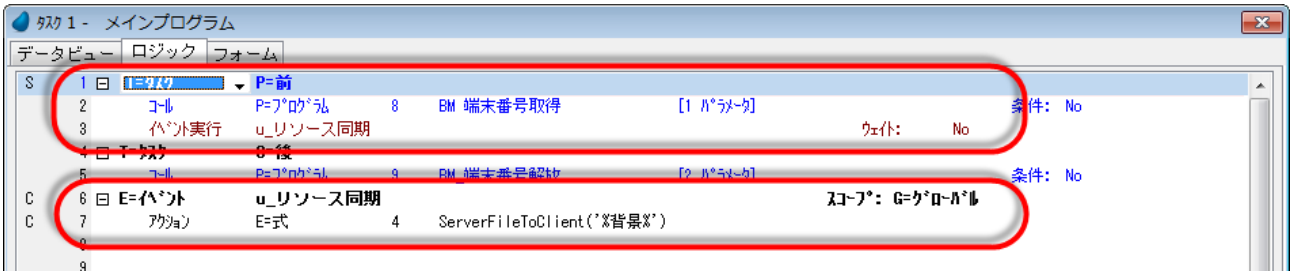
そのため、「3.9 オフライン RIA タスクからダウンロードプログラムを呼び出すには？」で説明したのと同様に、この関数の呼び出しはイベントを使って間接的に行います。

具体的には、次のようにします。

① メインプログラムのイベントテーブルに、ユーザイベントを新規追加します。このイベントに「u_リソース同期」という名前を付けます。



② このイベントに対するイベントハンドラをメインプログラムのロジックとして定義します。



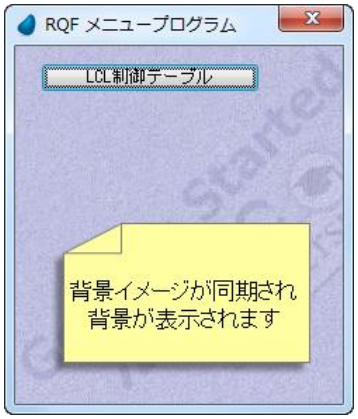
③ 最後に、このイベントをどのタイミングで発行するかですが、背景イメージなどのリソースは、プログラムが始まる前に一通り全部そろっていなくてはなりませんから、「メインプログラム」の「タスク前処理」で実行するのが良いでしょう。

注意：実際には、この「イベント実行」コマンドは条件付けをする必要があります。「タスク前処理」は、アプリケーションが開始されるたびに毎回実行されるものですが、リソースのダウンロードは1回だけ実行すれば済むものであり、アプリ起動のたびに毎回行う必要はありません。特に、オフライン対応のアプリケーションの場合には、アプリケーション起動時に毎回、タスク前処理でサーバへのアクセスを行う ServerFileToClient() 関数が実行されるのは困ります。オフラインアプリは完全にオフライン状態でも起動できるようにしておくべきだからです。

従って、条件としては、「ファイルが存在しない場合」となり、ClientFileExist() 関数を使うのが良いでしょう。

3.10.4 メニュープログラムを再実行して、背景を確認

このようにメインプログラムを変更して、再度 F7 実行すると、今度は JPG ファイルがダウンロードされるので、背景が正しく表示されるようになります。



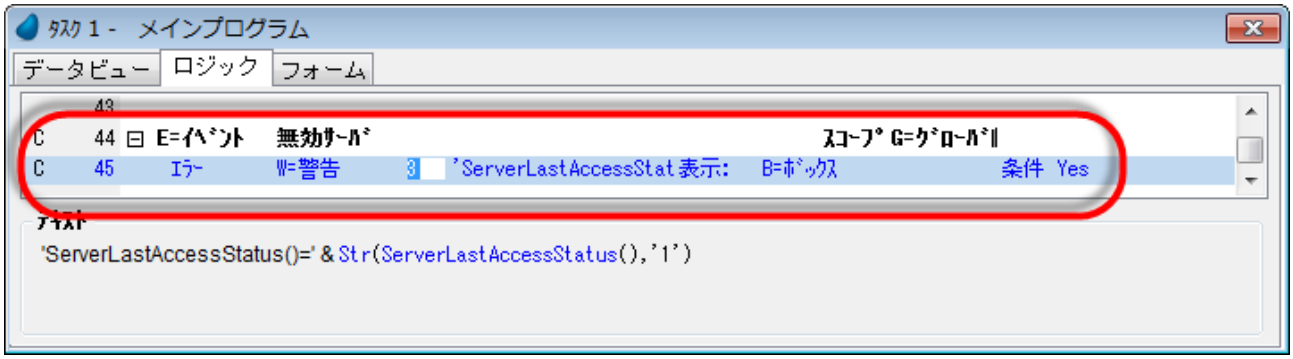
3.11 サーバが応答しない時、どうなりますか？

タイムアウトで処理は中断します。
「無効サーバ」イベント（内部イベント）が発生します。
オフラインタスクは実行を継続します。
ステータスが ServerLastAccessStatus() でステータスを取得できます。

実際に「無効サーバ」イベントが発生する状態で確認してみましょう。

3.11.1 「無効サーバ」イベントハンドラ設定

メインプログラムのロジックに、「無効サーバ」イベント のハンドラを定義しましょう。
ハンドラの中身としては、ServerLastAccessStatus() の結果をメッセージボックスで表示します。



ここでは、単に ServerLastAccessStatus() の結果を表示するだけですが、実用的には、関数の結果によって、エラー時の処理を分ける条件式に使用したりします。

3.11.2 ServerLastAccessStasus 関数とは

ServerLastAccessStatus() の説明を、リファレンスガイド（リファレンスガイド > 式エディタ > 関数ディレクトリ > ServerLastAccessStatus）から以下に抜粋します。

ServerLastAccessStasus	
サーバへの最後のアクセス状態: サーバに最後にアクセスしたときの状態を返します。	
構文:	ServerLastAccessStatus()
パラメータ:	なし
戻り値:	数値 0 …… 成功

	1 …… メタデータファイルが同期していない 2 …… サーバは、利用できません（起動時の場合や、リトライ確認のダイアログでユーザが'N'をクリックしたため)。 3 …… サーバは、リクエストを処理できません（ライセンスエラーのようなサーバエラーのため) 4 …… コンテキストは、利用できません 5 …… Magic xpa Studio で、リクエストを処理できません（Magic xpa Studio がオンラインモードで起動しているか、アプリケーションがオフラインのモードで開始している場合)。 6 ……ConnectOnStartup の特性が N に設定されているためにクライアントがサーバへの接続をスキップした場合。
用途:	この関数は以下のように使用することができます。 [コール]処理コマンドが失敗した後でメッセージを表示するような場合 [無効サーバ]イベントでユーザに問題発生を通知する場合
注意事項:	戻り値は、次のサーバアクセスが実行されるまで残ります。 メタデータが同期しない場合、将来のサーバアクセスは無視されるため、関数は常に同じエラーコードを返します。

3.12 オフライン起動時のサーバタイムアウト時間は設定できますか？

RIA アプリケーションの実行特性 FirstHTTPRequestTimeout で指定します。デフォルトは2秒です。

オフラインアプリケーションでも、データの同期などのために、起動時にサーバにアクセスしたいと思うことはよくあると思います。しかし、ネットワークが切れていてサーバに接続できない場合もあるので、そのような場合にはタイムアウト時間は必要最小限（例えば数秒）にしておきたい、と思うでしょう。このような場合には、FirstHTTPRequestTimeout というプロパティを使います。

以下にリファレンスガイド（Web 開発 > リッチクライアントアプリケーション > リッチクライアントインタフェースビルダ > アプリケーションの実行特性）から、FirstHTTPRequestTimeout について抜粋します。

	内容		モバイル
FirstHTTPRequestTimeout	起動時に Magic xpa エンジンに接続するためのタイムアウトを定義します。 クライアントがサーバから HTTPReuestTimeout の値を受け取るまで、この値がクライアント側のデフォルトの HTTP タイムアウトになります。 低速のネットワーク環境での実行時における非オフラインが使用可能なアプリケーションの場合、このキーにより高い値（例えば、通信の失敗の影響を最小にするために、10 秒に設定します）を設定することを推奨します。 デフォルト値：2 秒		Android
	注意：	IP アドレスが正しくない場合の接続時のように、この特性が利用できない場合があります。	
	サポートバージョン：	2.4	

このパラメータは、RIA の実行特性に指定します。

① ClickOnce を使う場合には、.publish.html ファイル中の <xml> セクションに、以下のような設定をします。

```
<property key="FirstHTTPRequestTimeout" val="5"/>
```

② ClickOnce を使わず、直接 MgxpaRIA.exe プログラムを起動する場合には、execution.properties ファイルに上と同じ設定を入れておきます。

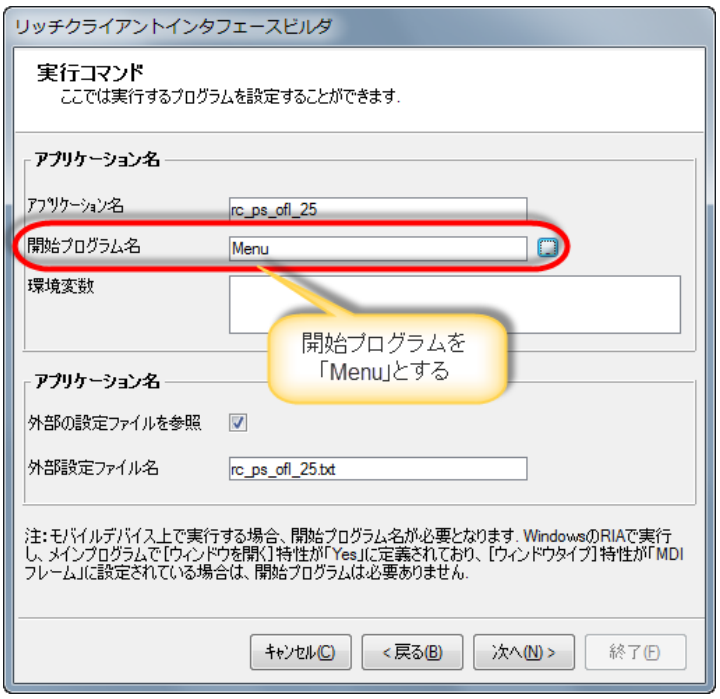
3.13　インターフェースビルダでインタフェースファイルを作成

本章の最後に、リッチクライアント インターフェースビルダで、インターフェースファイルを作成しましょう。

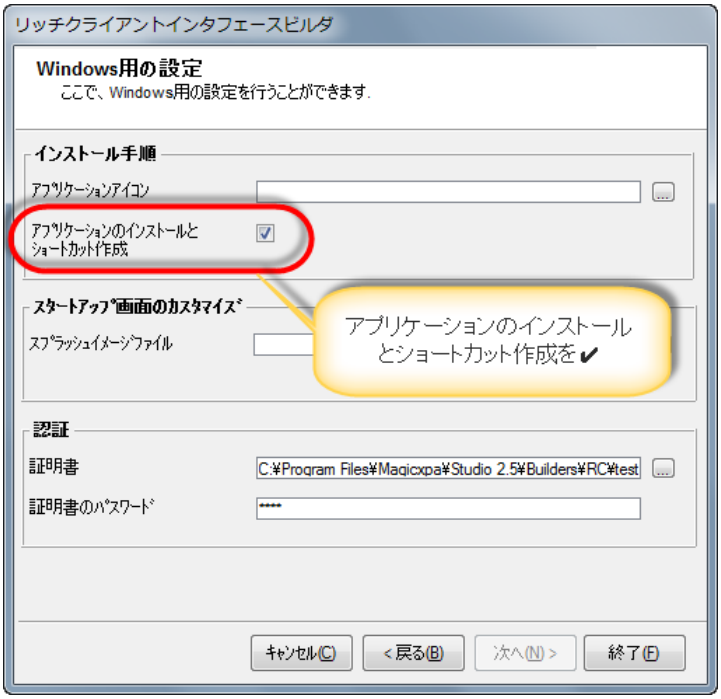
3.13.1 ウィザードの起動

① Studio で、メニュー「オプション＞インターフェースビルダ＞リッチクライアント」を選び、リッチクライアント インターフェースビルダを起動します。

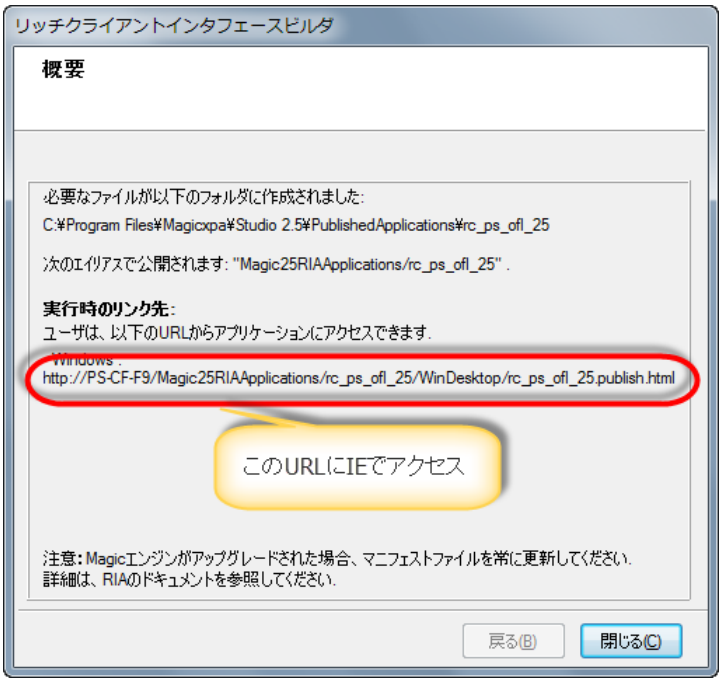
②「開始プログラム」は、Menu とします。これは前の節で作成したメニュープログラムの公開名です。



③「アプリケーションのインストールとショートカット作成」にチェックを入れておきます。

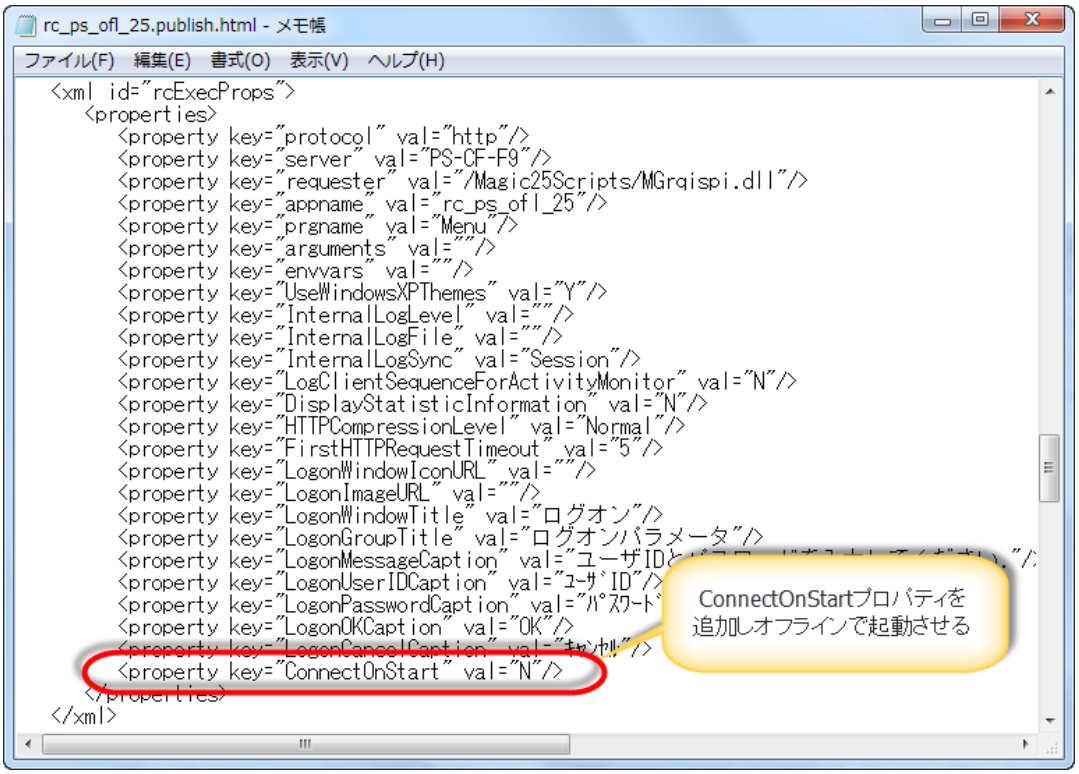


作成し終わったら、URL が表示されますので、クライアント PC からは Internet Explorer からこの URL を入力し、RIA プログラムを起動することができます。



3.13.2　アプリケーション起動用 html 修正

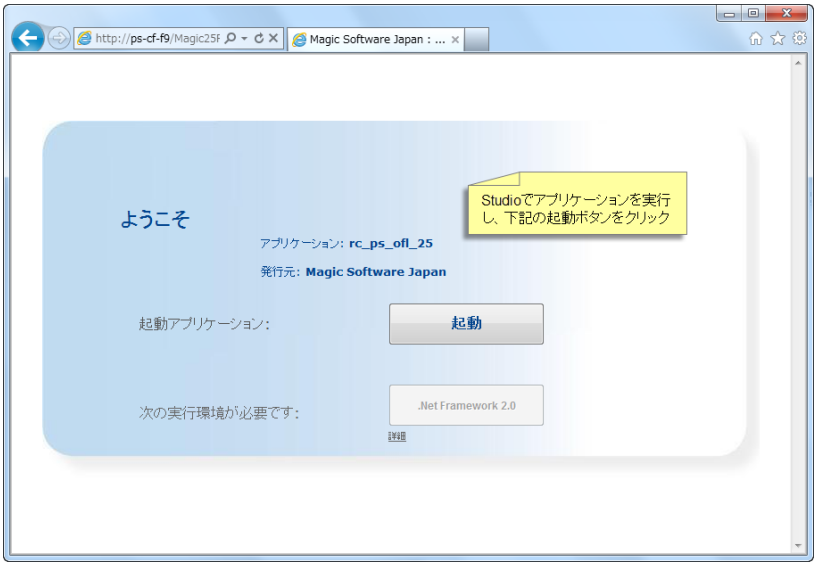
オフラインで起動させるため、「ConnectOnStart」プロパティを起動用 html に追加します。



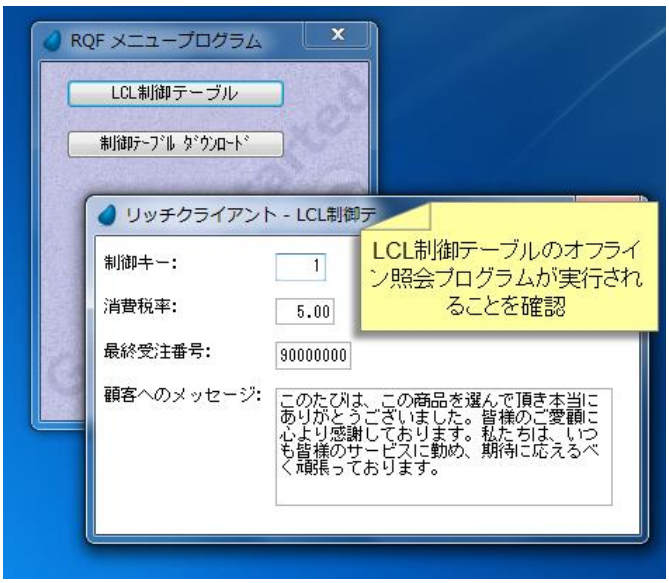
3.13.3　アプリケーションのインストール・起動

Studio をバックグラウンドモードでプロジェクトの実行をします。

起動用 html に IE でアクセスし、起動ボタンをクリックしてアプリケーションのインストールを行い起動します。



最初の起動でオフラインプログラムを
ローカルにキャッシュします。



いったん ClickOnce 機能で起動すると、必要なプログラム等がすべてダウンロードされてクライアント PC にインストールされます。また、スタートメニューにもこのアプリケーションの起動メニューが登録されます。（スタート＞Magic Sotfware Japan＞rc_ps_opl_25 ）。この辺は、普通の RIA アプリケーションと同じです。

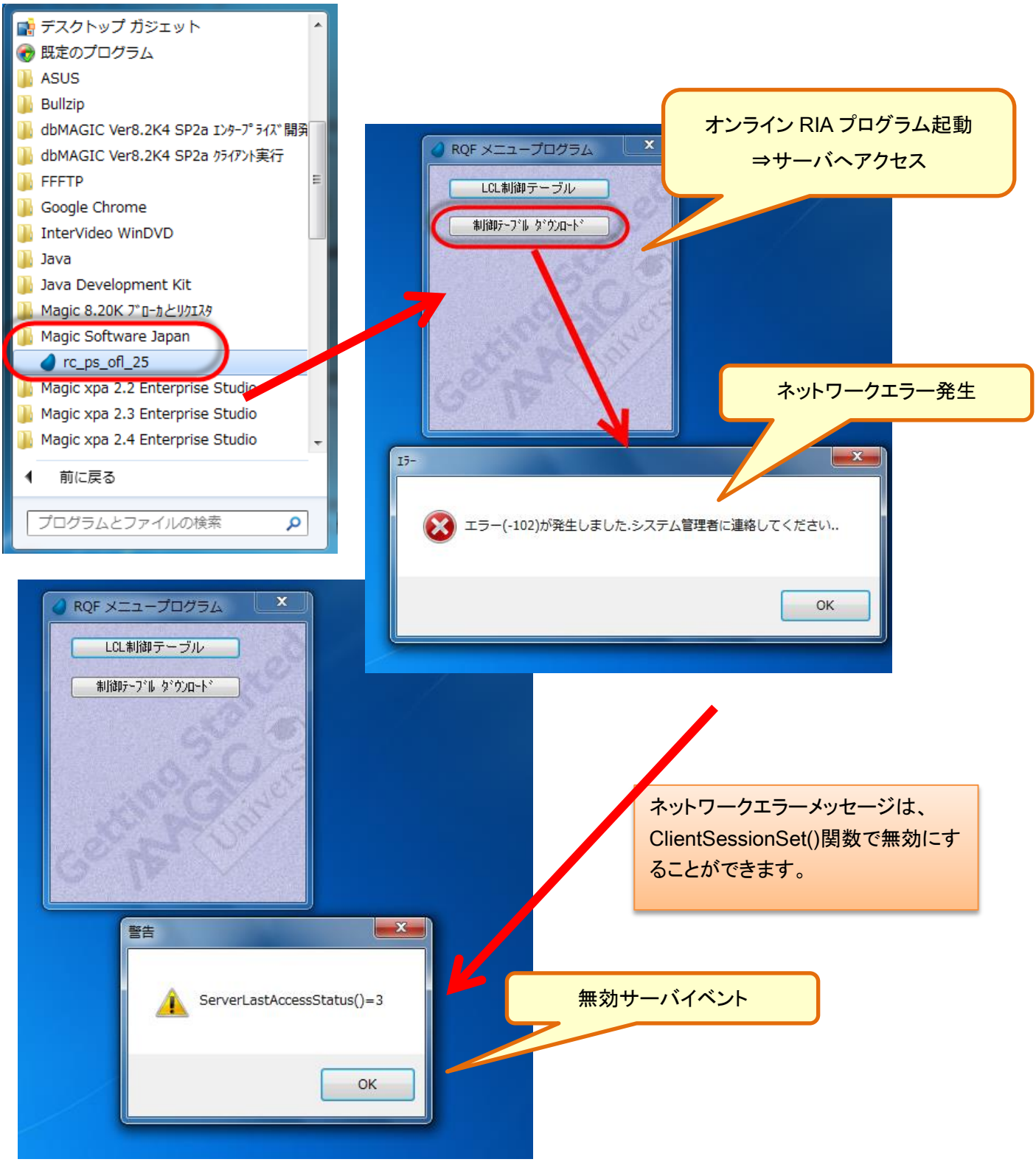
3.13.4　Studio、Broker を停止してオフライン RIA を実行

では、実際にオフラインで（サーバへの接続なしで）、アプリケーションが実行できるかを試してみましょう。

ここでは、サーバ停止状態を作るために、Studio および MRB をすべて停止します。

そうしておいて、Windows のスタートメニュー「スタート＞Magic Sotfware Japan＞rc_ps_opl_25」を選んで、RIA アプリケーションを起動します

すると次のように接続不能の旨エラーメッセージが表示された後も、アプリは通常通り動作することがわかります。すなわち、オフラインで動作することが確認できました。



4 差分ダウンロード

いままでは、サーバ テーブルの全レコードをダウンロードしていました。

マスターレコードなどは常に更新（追加、変更、削除など）が入るのですから、ローカルデータもそれに応じて適時更新しなければ、正しいデータで仕事をすることができません。

マスターデータを含むサーバのデータは非常にレコード数が多くなることがありますので、できれば「差分」だけ、すなわち、最後にダウンロードした時から、変更のあったレコードについてだけダウンロードして、転送データ量を最小限にしておきたくなります。

本章では、差分レコードだけをダウンロードする方法について、顧客マスタを例にして説明します。

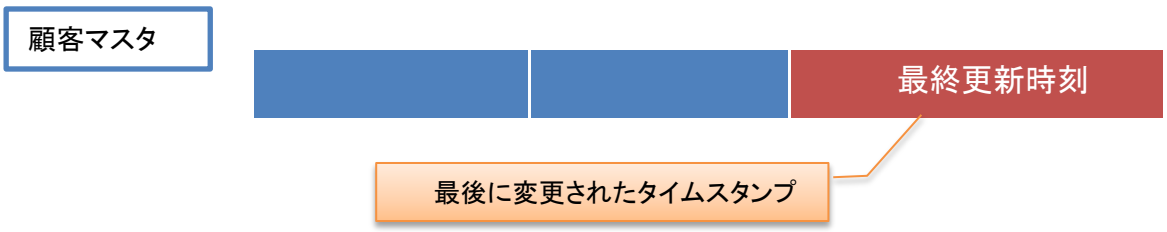
4.1 差分だけダウンロードするにはどうしますか？

レコード毎に「最終更新時刻」を記録しておき、「最終ダウンロード時刻」と比較して、範囲づけをします。

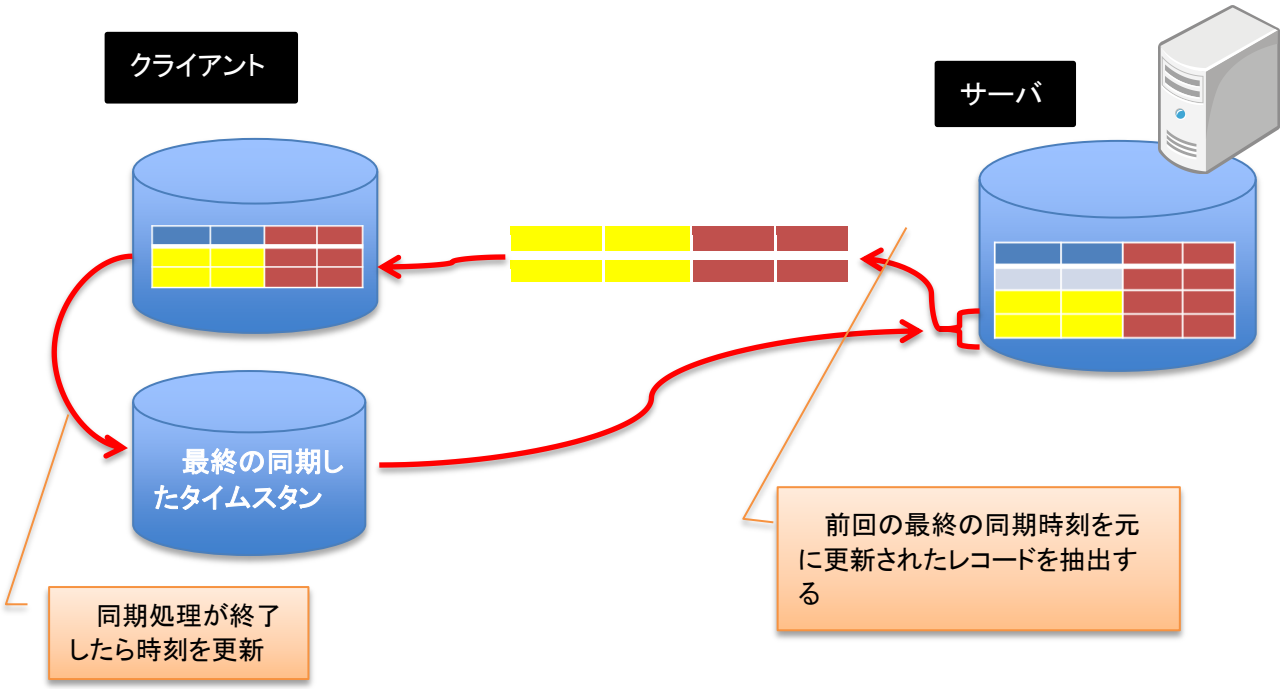
差分ダウンロードは、タイムスタンプを利用して実現できます。例えば、「顧客マスタ」テーブルでは、次のようにします。

1. 「顧客マスタ」テーブルに、「最終更新時刻」というカラムを追加し、それぞれのレコードが更新された時刻を記録しておきます。
2. 特定のデバイスが最後にデータをダウンロードした時刻（「最終ダウンロード時刻」）を記録しておきます。
3. そのデバイスから「差分ダウンロード」のリクエストが来たら、「最終更新時刻が、そのデバイスの最終ダウンロード時刻よりも大きい」という範囲づけをして、顧客マスタのデータを DataViewToDataSource 関数でダウンロードします。

基本方針としてはこれで良いのですが、「最終更新時刻」として、どの時刻を採用すべきか？ということが重要な問題になってきます。これについて以下の節で説明します。



- ローカルデータソースには、最後の同期時間の経過を記録
- 最終の同期したタイムスタンプ（YY/MM/DD HH:MM:SS フォーマットの文字列）
- 新しく更新されたレコードを見つける範囲条件として使用



4.2 「最終更新時刻」はどのようなデータ型にしますか？

日付と時間とを組み合わせた一つの文字列型として扱うのが便利です。

最終更新時刻は、日付と時間（秒単位）との組み合わせで表現します。Magic のデータ型としては、日付型と時刻型という二つのカラムに分けると比較が面倒になるので、YYMMDD 書式の日付文字列と、HHMMSS 書式の時刻文字列とを連結した12文字の文字型データひとつとして使うのが便利でしょう。
このデータは、プログラムで頻繁に必要とされるものなので、モデルリポジトリでモデルとして定義しておくのが良いでしょう。

名前：最終更新時刻 A=文字型

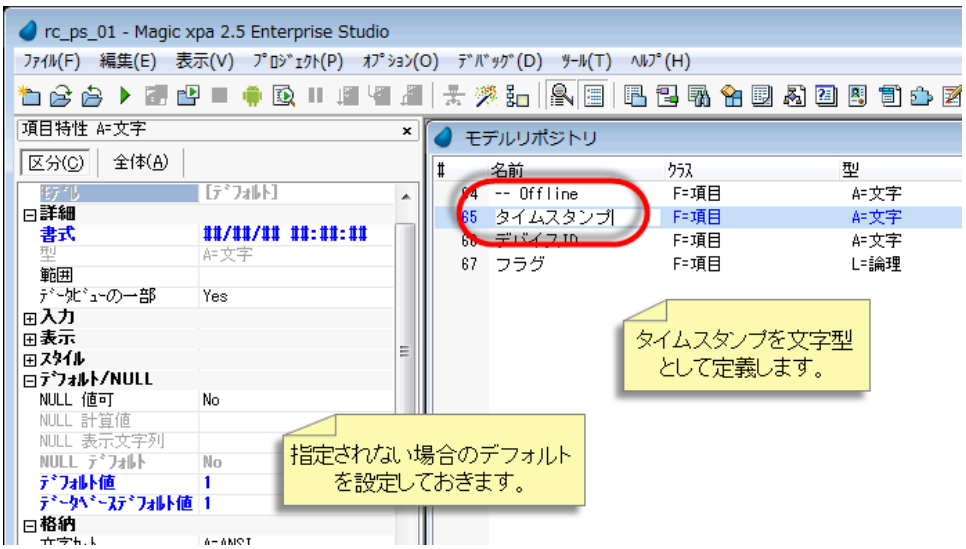
格納するデータとしては、日付6桁と時刻6桁の合計12桁になります。

表示を見やすくするために、書式は ##/##/## ##:##:## にしておくと良いでしょう。

あまり必要ないとは思いますが、秒単位では間に合わない場合には、ミリ秒単位まで記録しておくことができます。

デフォルト値、および データベース デフォルト値を設定しておくのが良いでしょう。

追加カラムなので、既存のプログラムでは選択されないため、レコード作成時にエラーになる可能性があります。



デフォルト値として、「000000000000」（12 桁の数字「0」）を設定するのは、若干問題があります。ローカルテーブルは SQLite ファイルとして実装されるのですが、SQLite は文字型と数値型の区別が曖昧なところがあり、「000000000000」は「0」として格納されてしまいます。Magic は文字データとして扱うので、「000000000000」と「0」とは違いますから、デフォルト値の扱いで矛盾が起こり、期待したように動かないことがあります。

これを防ぐために、デフォルト値として、先頭の 0 はつけないようにしてください。一桁だけの「0」あるいは「1」などが良いでしょう。

4.3 時差のある遠隔地にクライアントがある場合にはどうしますか？

クライアントとサーバが遠距離にあり、時差がある場合には、タイムスタンプとして、時差設定に依存しない UTC（協定世界時）を返す UTCDate、UTCTime、UTCmTime 関数を使います。

タイムスタンプとして、時刻のローカルタイム（日本であれば、日本標準時）を使うと、時差のある遠隔地からアクセスするクライアントをサポートすることができません。例えば、日本の標準時とアメリカ東海岸の標準時とでは、－14時間の時差があります。つまり、日本で午後2時のときにはニューヨークでは午前0時です。また、サマータイムに移行する際には、その日の午前2時に時刻調整が行われます。このようなローカル時刻を使ってタイムスタンプの比較を行うことはできません。

このため、タイムスタンプとしては、ローカル時刻の設定に関わらず、世界共通の UTC (協定世界時）を返す UTCDate、UTCTime、UTCmTime 関数を使うべきでしょう。

日本標準時は、UTC より 9 時間進んでいますから、UTCTime が返す値は、Time() 関数で返すよりも -9 時間となります。
この関数を使えば、タイムスタンプは

DStr(UTCDate(),'YYMMDD') & TStr (UTCTime(), 'HHMMSS')

となります。
もしミリ秒まで必要ならば、

Str(UTCmTime() – UTCTime() * 1000, '3')

を使います。

4.4 サーバ時刻とクライアント時刻にずれがある場合にはどうしますか？

タイムスタンプは常にサーバの時刻を基準として使うようにします。

サーバマシンとクライアントデバイスとは、クロック発生のハードウェアの異なる別々のハードウェアですから、時刻を正確に一致させることは技術上非常に困難です。また、クライアントデバイスの時刻は、サーバ管理者が管理できるものではありません。クライアントのデバイスの時刻設定は所有者が管理するものですが、一般的なユーザは大きなずれがない限り、普通はそう気にしないものです。

そのため、サーバとクライアント間には、秒単位、あるいは分単位のずれが発生することも想定して設計しておかなければなりません。極端には、西暦の下 2 けたと、平成の年号とを混同して設定して使っている、などのことも発生しないとは限りません。

たとえわずかな秒単位の違いであっても、コンピュータの処理は非常に速いので、多くのデータ更新が発生する環境では、タイムスタンプを正しく取り扱わなければ、不正データ更新が発生する可能性があります。

基本的に、クライアントが「自己申告」したデバイスの時刻というものは、サーバでの処理の基準として信頼することができません。理由としては、以下のようなことがあります。

- デバイスの時刻がサーバと合っているとは限らない。（普通ずれている）
- デバイスの時刻を、昨日から今日の間にユーザが変更したかもしれない。キャリアへの接続ができるスマートフォンでは、時刻の自動校正機能が有効になっていますので、定期的に自動的に変わる可能性があります。
- デバイスがリクエストを出してから、Magic xpa RIA サーバにリクエストが届くまでに、ネットワークの遅延、MRB による待ちなどがあるので、リクエストがサーバに届いた時点で、すでに時刻が異なっていることがある。

従って、時刻の比較をする際には、このようなずれによる処理への悪影響が起こらないよう、常にサーバの時刻を使って比較するべきだということになります。

前述の差分ダウンロードの場合には、正確には次のようにすべきです。

1. 「顧客マスタ」テーブルに、「最終更新時刻」というカラムを追加し、それぞれのレコードが更新された時刻を記録しておきます。この「時刻」は、サーバの時刻です。
2. 特定のデバイスが最後にデータをダウンロードした時刻（「最終ダウンロード時刻」）を記録しておきます。この「時刻」もサーバの時刻です。デバイスの自己申告の時刻ではありません。
3. そのデバイスから「差分ダウンロード」のリクエストが来たら、「最終更新時刻が、そのデバイスの最終ダウンロード時刻よりも大きい」という範囲づけをして、顧客マスタのデータを DataViewToDataSource 関数でダウンロードします。（この部分は変更ありません。）

ここで、ステップ 2 あるいはステップ 3 で使う「最終ダウンロード時刻」、はどのように記録しておくのが良いでしょうか？

- 一つの方法は、サーバに別途「最終ダウンロード時刻」テーブルを作成しておき、そこにデバイス ID と最終ダウンロード時刻（サーバ時刻）とを組にしたレコードを格納しておくことです。
- もうひとつの方法は、デバイス側に「最終ダウンロード時刻」を記憶しておくのですが、この時刻として、デバイスの時刻ではなく、サーバから提示された時刻を格納しておく、という方法です。

いずれの方法でも、正しい結果が得られます。

クラサバのアプリケーションでも、監査の目的のために、更新の記録として、最終更新時刻と更新者（ログイン ID など）を記録しておくことがよくあります。RIA オフラインアプリケーションにおいても同じことをするとすれば、ユーザがレコードを更新した時刻は、デバイスの時刻を記録することになります。

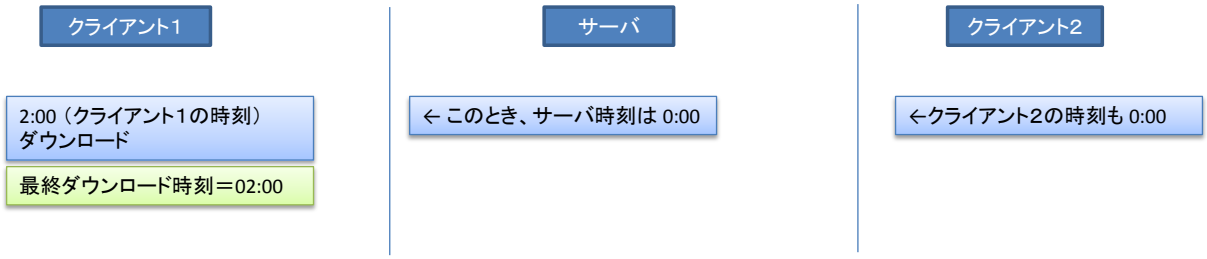
このような記録は監査の目的のためのものとして使うのは全くかまいませんが、本書で対象としている、データの同期の目的のためには信頼して使うことはできませんので、別途カラムを追加する必要があります。

4.5 クライアント時刻を使ったときの不具合の例はありますか？

複数のデバイスで更新されたレコードの同期が正しくとれないことがあります。

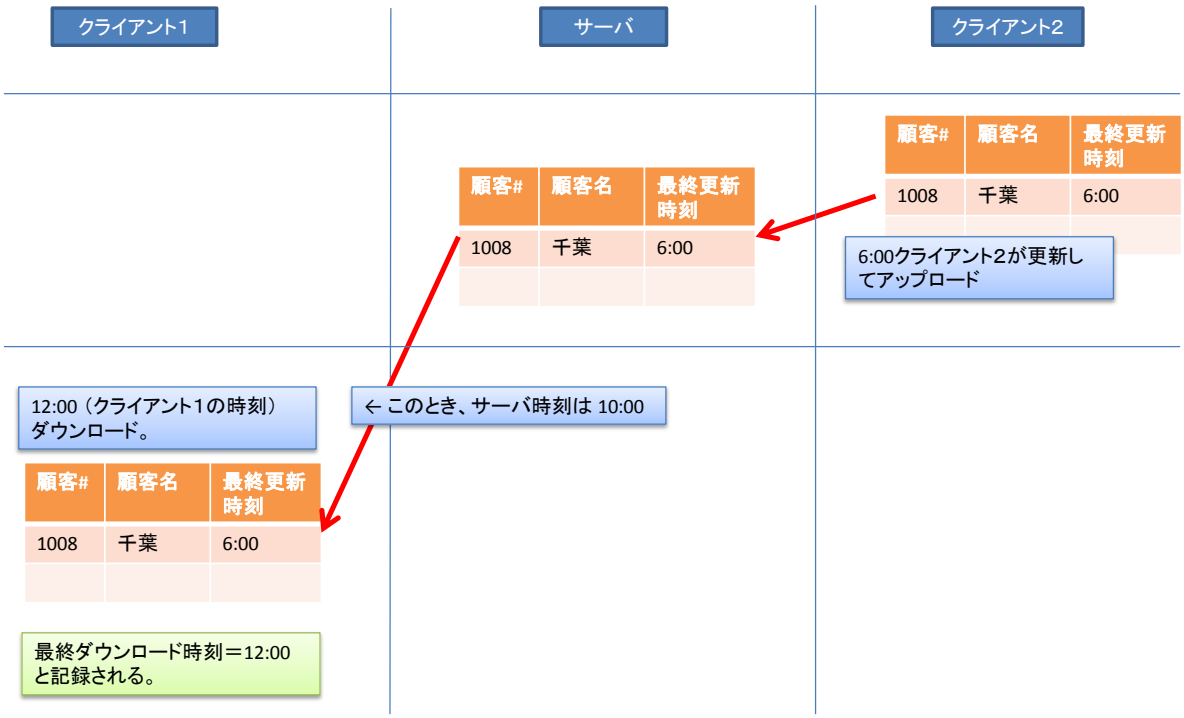
最終更新時刻として、クライアントの時刻を使うと、レコードの同期が正しくとれないことがあります。以下に、クライアントが2つ（デバイス1 とデバイス2）あり、デバイス1の時刻だけが2時間進んでいるとします。

① 真夜中の 0:00 に、クライアント1がサーバからデータをダウンロードします。このとき、クライアント1の時刻は2時間進んでいるので、2:00 と認識するため、クライアント1の最終ダウンロード時刻は 02:00 と記録されます。



② 06:00 に、クライアント2が顧客番号#1008 のレコードを更新します。
レコードの「最終更新時刻」には、クライアント2の時刻 06:00 が記録されます。

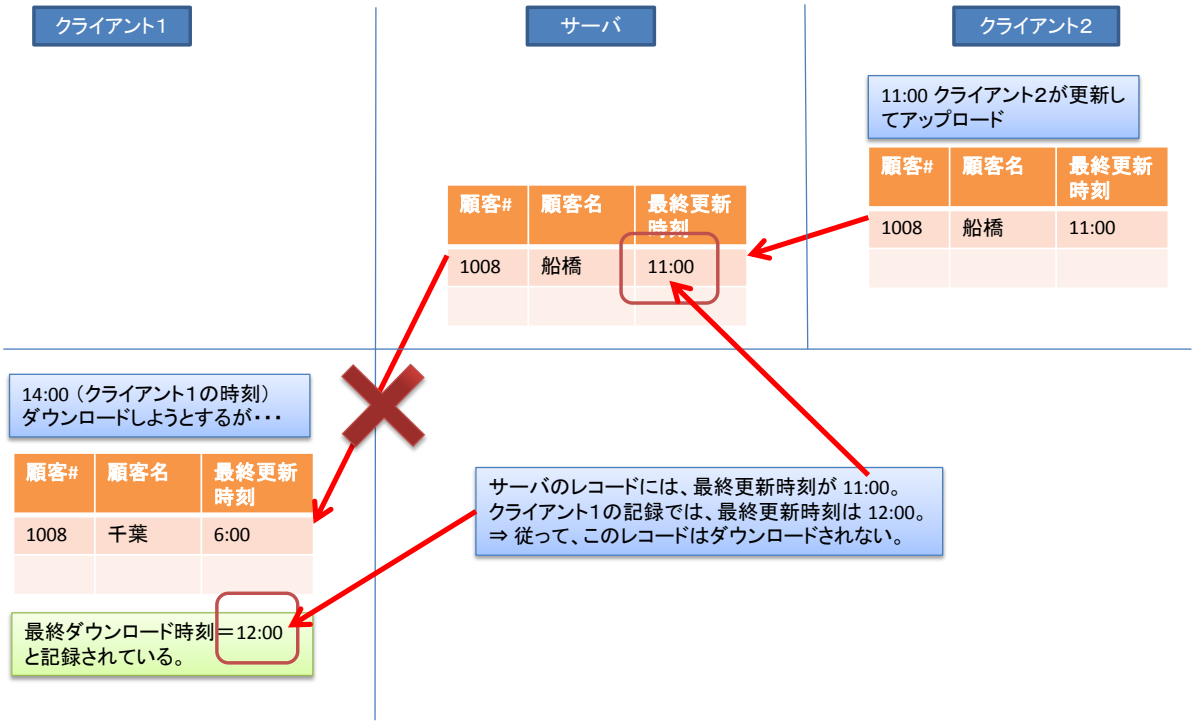
③ 10:00 に、クライアント1がこの顧客レコードをダウンロードします。クライアント1は2時間進んでいるので、このときのクライアント1の時刻は 12:00 ですから、「最終ダウンロード時刻」は 12:00 と記録されます。



④ 11:00 に、クライアント2が、再度同じレコード（顧客＃1008）を変更します。クライアント2の時刻はずれていないので、顧客#1008 の「最終更新時刻」は 11:00 と記録されます。

⑤ 次に、12:00 にクライアント1が顧客レコードをダウンロードしようとします。

デバクライアントイス1に記録されている「最終ダウンロード時刻」は 12:00 です。一方、顧客#1008 に記録されている最終更新時刻は 11:00 です。このため、このレコードは「最終ダウンロード時刻より前に更新されている」と判断されるので、ダウンロードの対象から外れることになってしまいます。



これは正しくありません。結果として、顧客#1008 のレコードは古いデータのままでいつまでもダウンロードされない、ということになってしまいます。

4.6 レコードの削除はどのように取扱いますか？

削除フラグを利用します。本当にレコードを削除してしまうと、同期のための情報もなくなってしまうからです。

レコードを削除したいとき、本当のそのレコードを削除してしまうと、同期をとる際に、「このレコードは削除された」という情報まで失われてしまうので、レコードは削除することはできません。

その代りに、「削除フラグ」あるいは「無効フラグ」をレコードに追加し、削除レコードはそのフラグを TRUE にすることにより識別するようにします。

ダウンロードの際には、この「削除フラグ付き」のデータもダウンロードの対象とする必要があります。

つまり、「削除フラグ」＝FALSE という範囲づけ条件により、除外してしまってはいけません。

「削除された」という情報も、ローカル テーブルに反映させておく必要があるからです。

削除フラグにより対応する方法は、ダウンロードの時だけでなく、アップロードの時にも必要です。

レコードの削除例は、5.4 節「ローカルテーブルの削除ロジック」で説明されています。

4.7 デバイス ID はどのように取得しますか？

プログラムで生成し、ローカルファイルに格納しておくのが一番簡単です。

4.7.1 デバイス ID とは？

RIA プログラム、特に RIA オフラインを利用するアプリケーションでは、各利用者の端末デバイスを特定するための ID を使うことが必要になります。ローカルデータの同期の時にも、どのデバイスがいつデータをダウンロード、あるいはアップロードしたかを記録しておくことが必要になります。このようなデバイス ID は、どのようにしたら取得できるでしょうか？

Windows にせよモバイル デバイスにせよ、ハードウェアにはそれぞれのデバイスを一意に識別できる ID が何等かの形で実装されています。しかし、個人情報保護の観点から、そのようなデバイス ID を利用することは推奨されません。多くの場合、ID を取得すること自体、OS が禁止しています。それぞれのプラットフォームで、ハードウェアのデバイス ID の代りになる、ソフトウェア的に生成される ID が提供されていることもありますが、これを利用しようとするとクライアントモジュールのカスタマイズなどが必要になり、また、環境やユーザ設定により利用できなくなっていることもあり、簡単ではありません。

Magic xpa のクライアントモジュールにも、そのような ID を取得する手段は提供されていません。従って、デバイス ID はアプリケーションで自作するのが一番汎用性があり確実です。

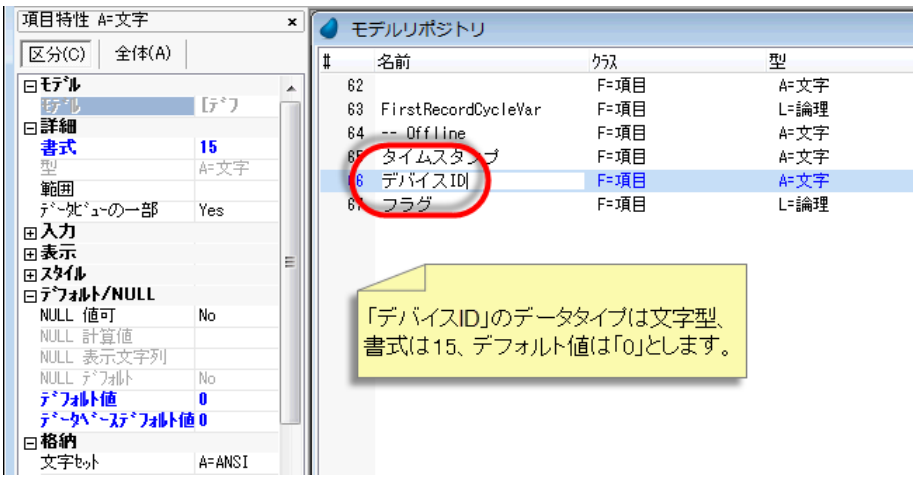
4.7.2 デバイス ID のデータ型は？

デバイス ID の目的としては、複数のユーザが使うデバイスを区別できて、ある程度の永続性のあれば良いのですから、一番簡単には、日時をもとに ID を作って、ローカルファイルに格納することで永続性を確保します。

ID としては、アプリを使っているユーザを一意に区別できればよいのですから、秒単位までの生成時の日時を利用すればたいてい間に合うでしょう。例えば、150302173343 というようなものです。西暦の 2000 年は省略してあります。

秒単位で衝突する可能性、すなわち、複数のユーザがそれぞれのデバイス上でたまたま同時刻に ID を生成する確率というのはまずゼロと言ってもいいと思いますが、念には念を入れて、ミリ秒単位までとってもいいかもしれません。この場合には 150302173343325（15 桁）のようになります。ミリ秒までとれば、よほどユーザ数が多くない限り、たまたま同じ ID となる確率は絶無と言ってよいでしょう。

デバイス ID はアプリケーション中の多くの箇所で使われるので、モデルとして定義します。ここでは、ミリ秒単位まで入れた、15 桁の文字型として定義しています。



4.7.3 デバイス ID の保存方法

この値を ID として、ファイルに保存するには、以下の二つの方法が考えられます。

- ローカル データベースの「LCL 制御テーブル」に「デバイス ID」カラムを追加して保存します。
- 文字列 → BLOB として、ClientBlb2File() で保存しておきます。次回以降は、このファイルから読み出すようにします。

いずれの方法でも、ローカルファイルをリセットしてしまうと、ID も同時に無くなってしまうという欠点があります。

アプリ削除して再インストールするとローカルファイルもリセットされますので、やはり ID はなくなります。

また、Windows の場合には、%TEMP% フォルダのサブフォルダに格納されますが、このフォルダの内容はいろいろなクリーナーユーティリティで削除の対象となるフォルダですので、削除されやすいです。

一旦削除されてしまった ID は修復するのが難しいです。再度生成したら、別の ID になってしまいます。

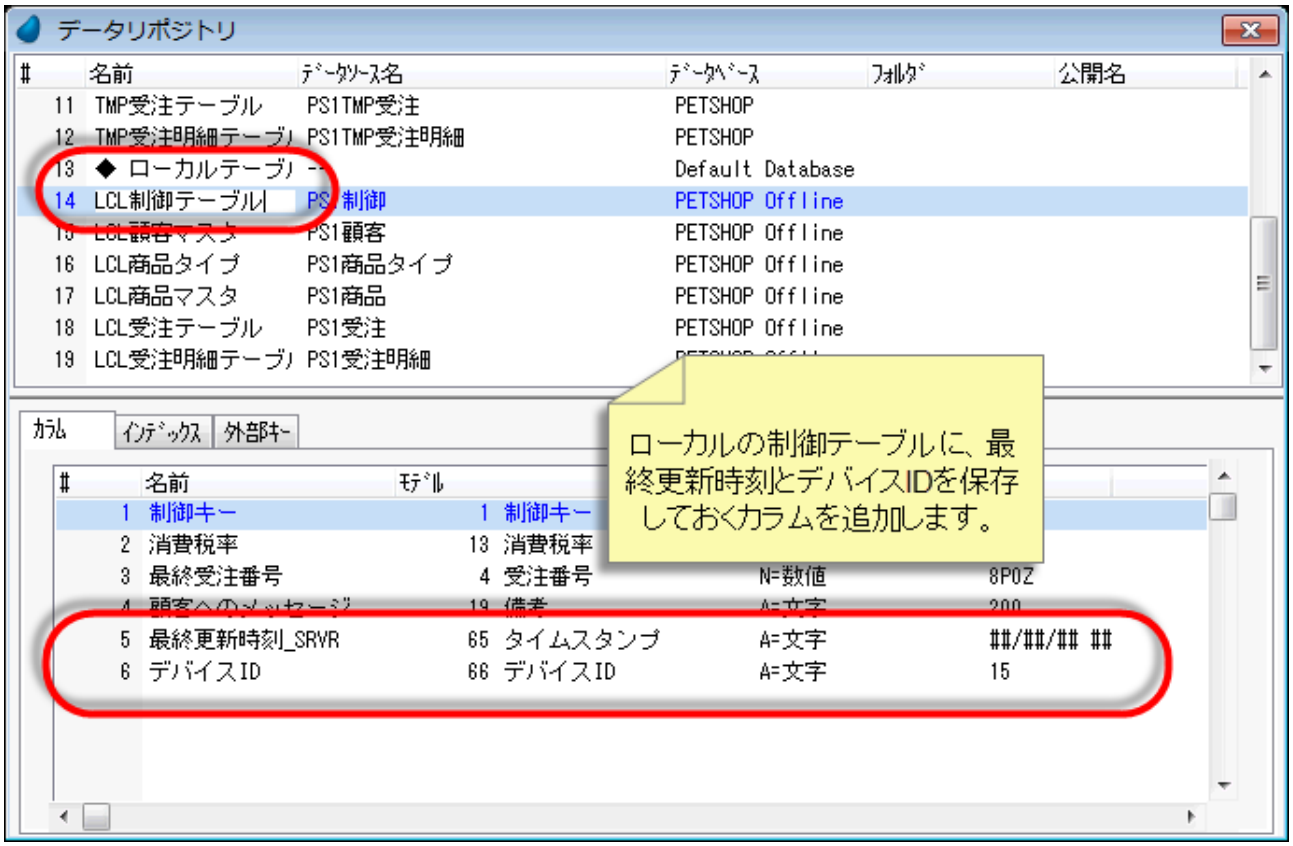
もっと固い（失われにくい）ID を使いたい場合には、ファイルの格納場所を工夫する必要がありましょう。

4.7.4 デバイス ID 取得プログラムの例

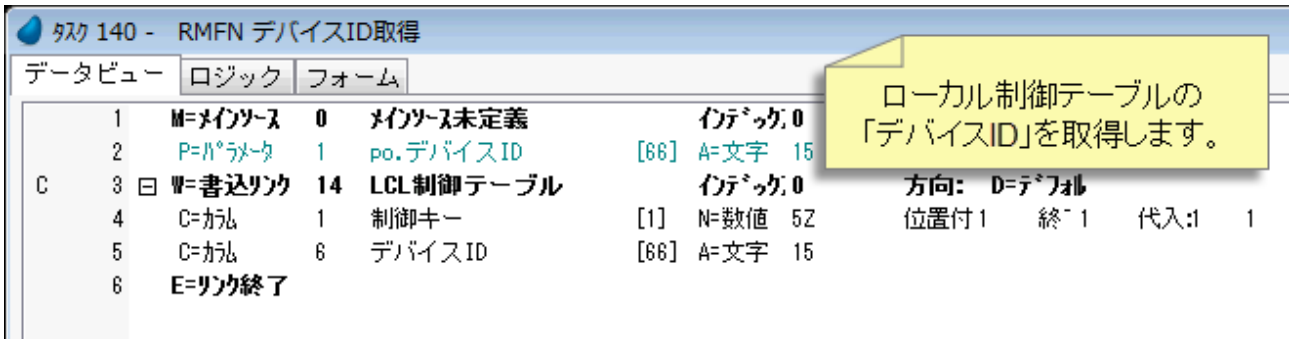
制御テーブルに「デバイス ID」カラムを追加する方法を使った例です。

① ローカル データベースの「LCL 制御テーブル」に、「デバイス ID」カラムを追加します。

ついでに、最終更新時刻を格納しておく「最終更新時刻_SRVR」カラムも追加しておきます。



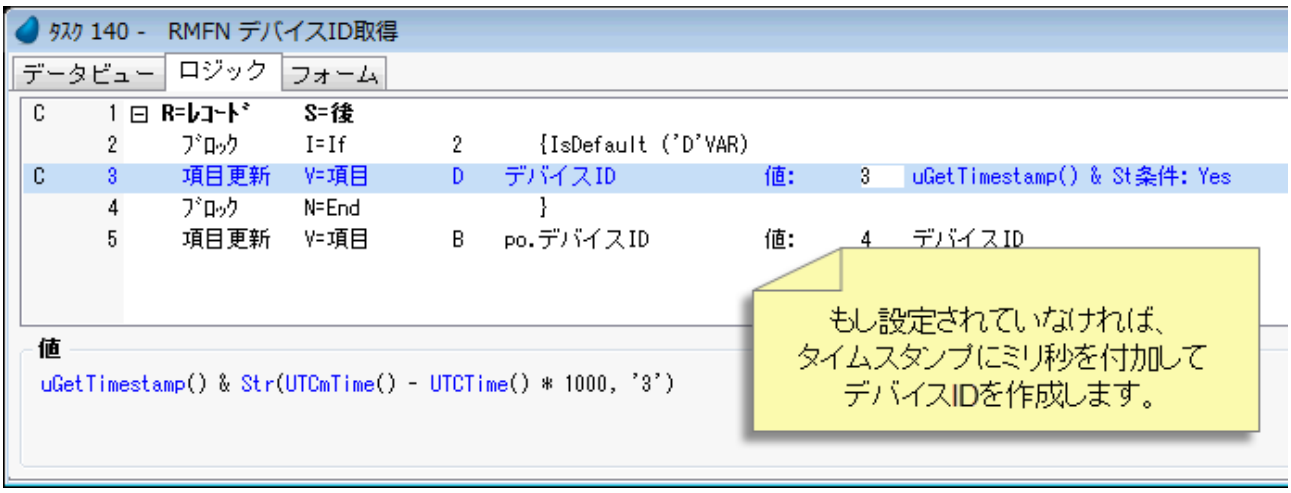
② デバイス ID 取得 プログラムでは、このローカル 制御レコードをリンクでアクセスします。



もし、デバイス ID が設定されていない（デフォルト値のまま）ならば、ここで生成して格納しておきます。

これは、最初に「デバイス ID 取得」を呼び出したときに行われます。

2回目以降は、作成されたデバイス ID を利用することになります。



③ 最後に、このデバイス ID をパラメータとして返します。

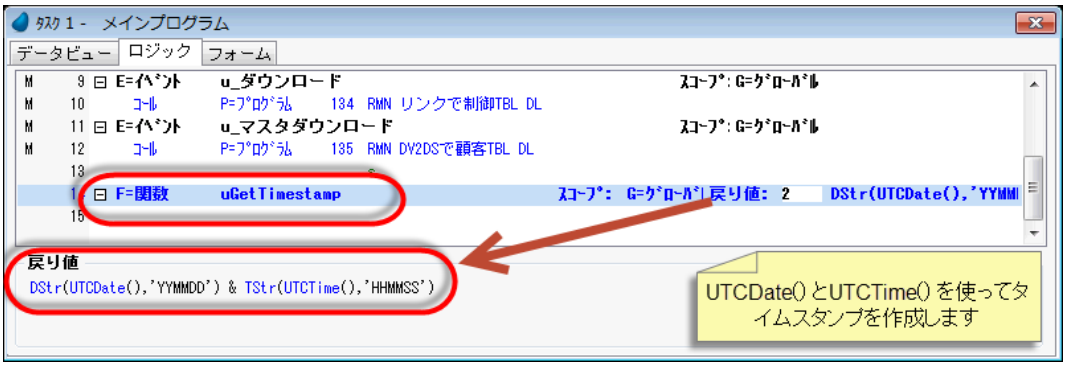
4.8 最終更新時刻 の作成／設定／取得のプログラムはどう作りますか？

ローカルの制御テーブルに格納しておいたものを使うのが一番簡単です。

4.2 節「「最終更新時刻」はどのようなデータ型にしますか？」で説明したように、タイムスタンプは秒までの日時を 12 桁の数値で表した文字型データを使うことにしました。また、日時は、ローカル日時ではなく、UTC の日時を使うことにしました。

4.8.1 タイムスタンプの生成

現在のタイムスタンプを取得するため、グローバルなユーザ定義関数としてメインプログラムで定義しておくと便利です。

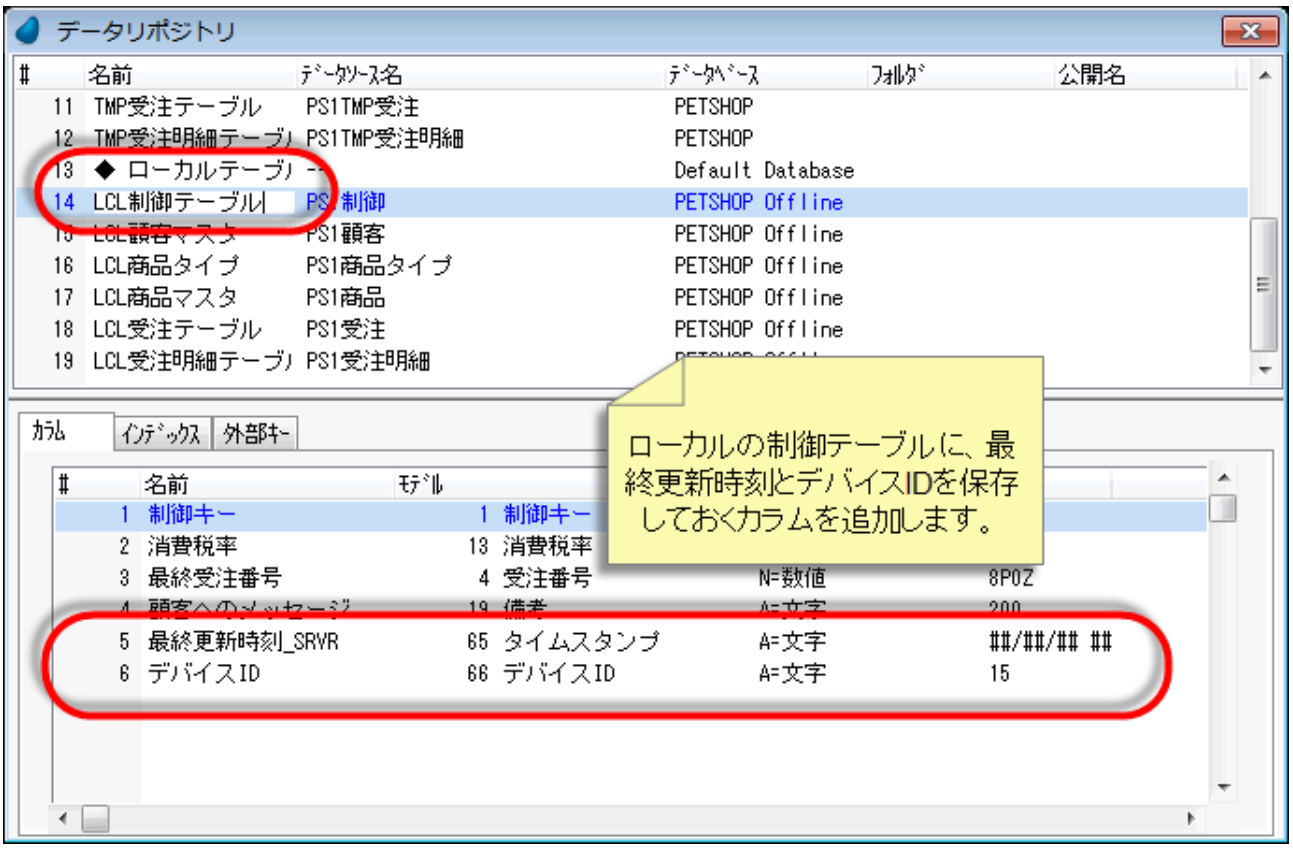


ここでは、日時を取得するのに、UTCDate() および UTCTime() を使っています。

4.8.2 タイムスタンプの記憶

タイムスタンプは、ローカルの制御テーブル（LCL 制御テーブル）に格納します。

このテーブルは、4.7.4 「デバイス ID 取得プログラムの例」で説明したように、「最終更新時刻_SRVR」と「デバイス ID」カラムとが追加されています。



4.8.3 最終更新時刻取得、設定プログラム

最終更新時刻の取得および設定プログラムは、非インタラクティブな RIA オフラインプログラムとして作ります。

LCL 制御テーブルのレコードにリンクして、取得の場合には「最終更新時刻_SRVR」の値を返し、設定の場合には、パラメータとして受け取った最終更新時刻を「最終更新時刻_SRVR」カラムに代入します。

いずれも非常に簡単なプログラムなので、細かな説明は省略します。

4.9 差分ダウンロードのロジックは最終的にどのようになりますか？

今まで説明してきたことをまとめると、こんなものになろうと思います。

4.9.1 「顧客マスタ」テーブルの定義拡張

次のカラムを追加します。

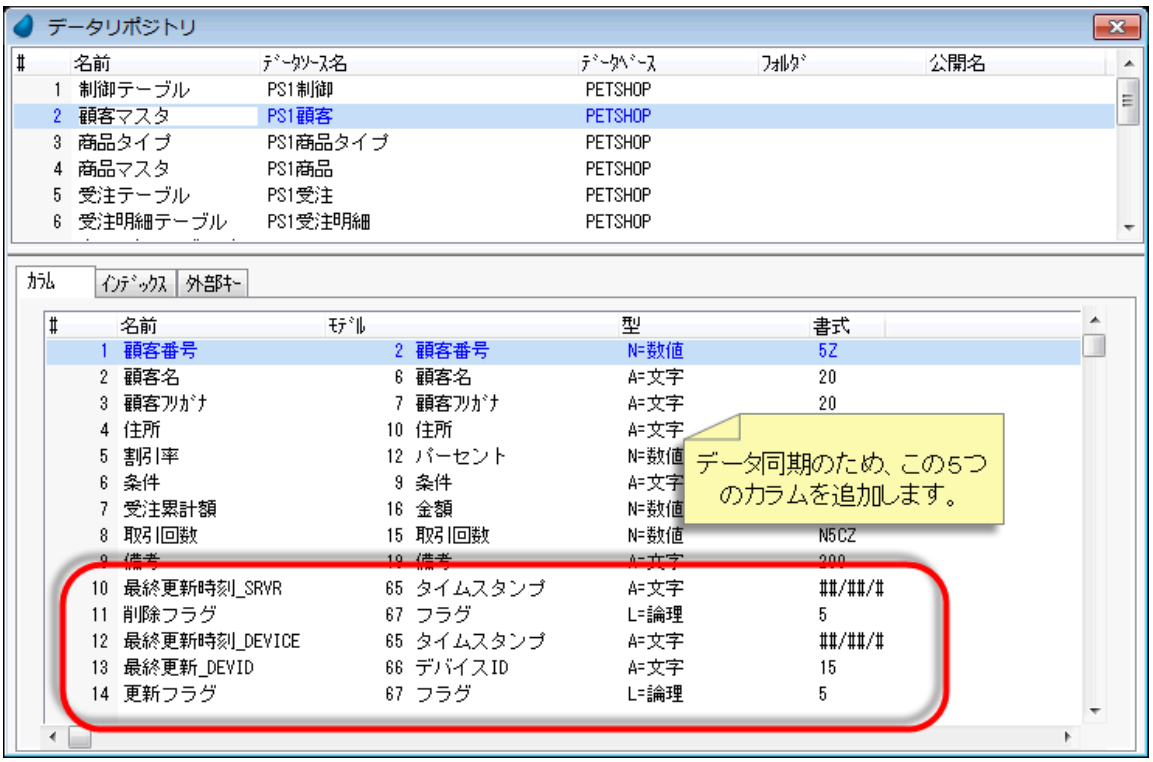
- 最終更新時刻_SRVR （タイムスタンプ型。レコードの最終更新が行われた時刻を、サーバ時刻で格納しておきます。）
- 削除フラグ（論理型。4.6 「レコードの削除はどのように取扱いますか？」参照）

ついでに後で必要になってくるので、ここで以下のカラムも追加しておきます。

- 最終更新時刻_DEVICE（クライアント デバイス上での、最終更新時刻。記録のため）
- 最終更新 DEVID

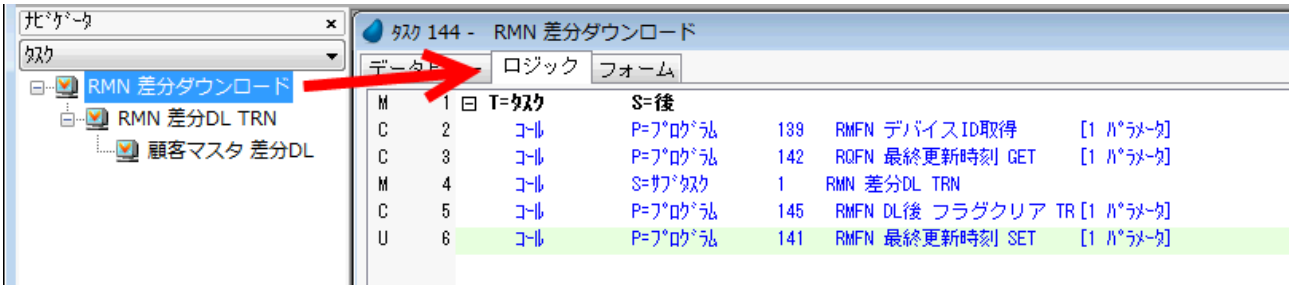
更新フラグ（アップロード時に必要となります。5.1 「差分アップロードをするにはどうしますか？」参照）

これらのカラムは、サーバ側テーブル（テーブル＃2「顧客マスタ」）およびローカルテーブル（テーブル#15「LCL 顧客マスタ」）の両方に追加する必要があります。

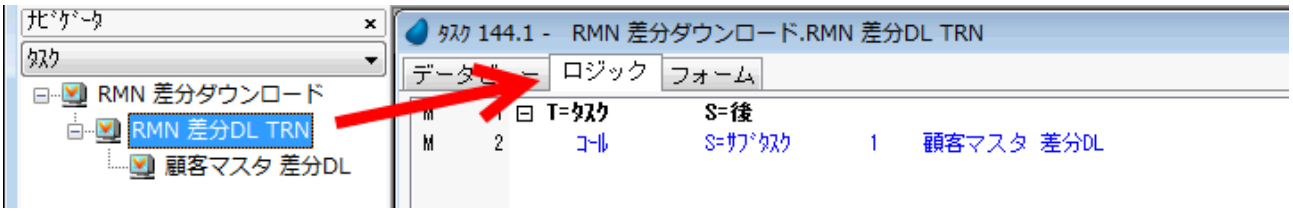


4.9.2 差分ダウンロードロジック

親タスクで、デバイス ID、最終更新時刻を取得します。

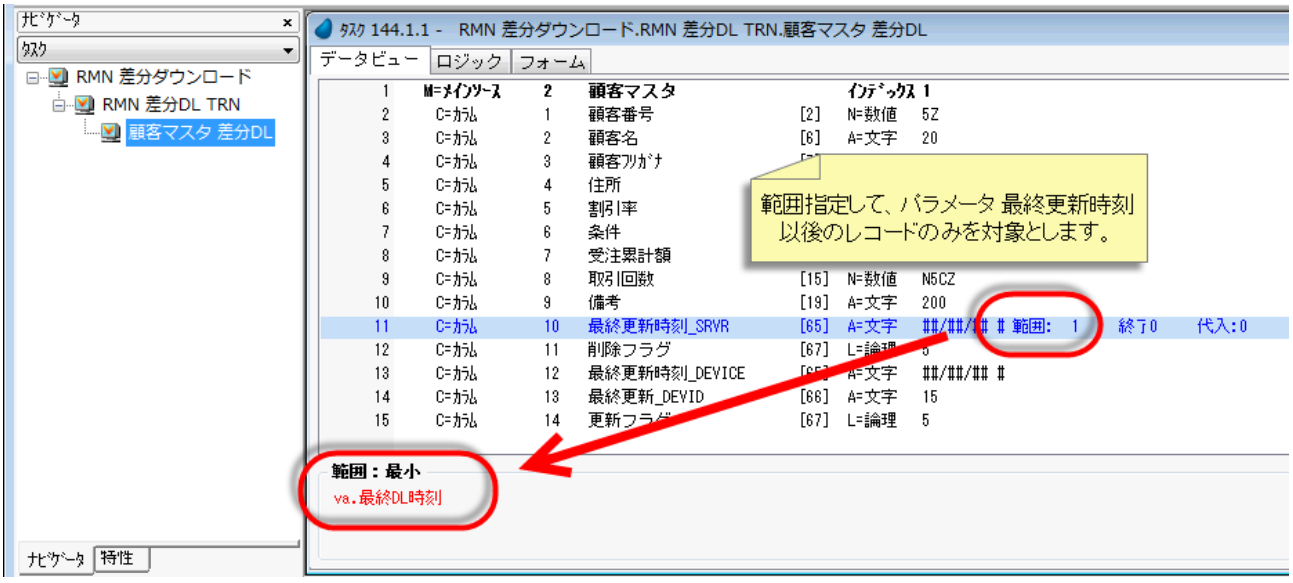


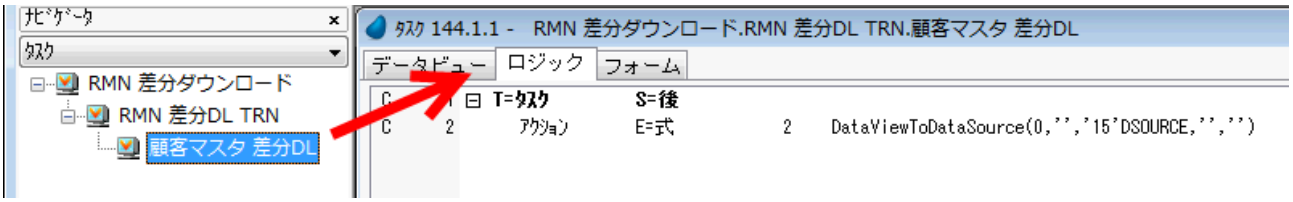
子タスクで、トランザクションを囲みます。



孫タスクで、DataViewToDataSource() を実行します。

データビューとして、「最終更新時刻_SRVR」カラムが、ローカルの最終更新時刻よりも新しいもの、という範囲づけをします。





最後に、「更新フラグ」をクリアするプログラムを呼び出します。

これは非インタラクティブな RIA オフラインプログラムで、別プログラムとして定義されています。


ﾅﾋﾞﾀﾞｰﾀ
ﾀｽｸ
RMFN DL後 フラグクリア TF
LCL顧客 フラグクリア
ﾀｽｸ 145.1 - RMFN DL後 フラグクリア TRN.LCL顧客 フラグクリア
データビュー
ロジック
フォーム
1 R=ﾚｺｰﾄﾞ S=後
2 項目更新 V=項目 E 更新フラグ 値: 2 'FALSE'LOG

4.10 商品タイプや商品マスタのダウンロードはどうしますか？

商品マスタも同様のアルゴリズムで差分更新します。

商品タイプはレコード数が少ないので、毎回全レコードダウンロードで問題ありません。

先のダウンロードプログラムに、「商品タイプ」ダウンロード および「商品マスタ」差分ダウンロードのプログラムを追加します。

更新フラグをクリアするプログラムにも、ローカルの「LCL 商品マスタ」のフラグをクリアするプログラムを追加します。

「商品タイプ」 テーブルには「更新フラグ」カラムを設けていないので、設定する必要はありません。

5 アップロード

アップロードには、ダウンロードと同じく、以下の三つの方法が考えられます。

1. 親子タスクで行う
2. DataViewToDataSource 関数を使う
3. SQLite ファイル全体をアップロードする、

本章では、いちばん一般的な2番目の DataViewToDataSource() 関数を使う方法について説明します。

他の方法についても、ダウンロードの時と同様の考え方で、データ転送の方向を変えるだけで実装することができます。

5.1 差分アップロードの基本的な考え方は？

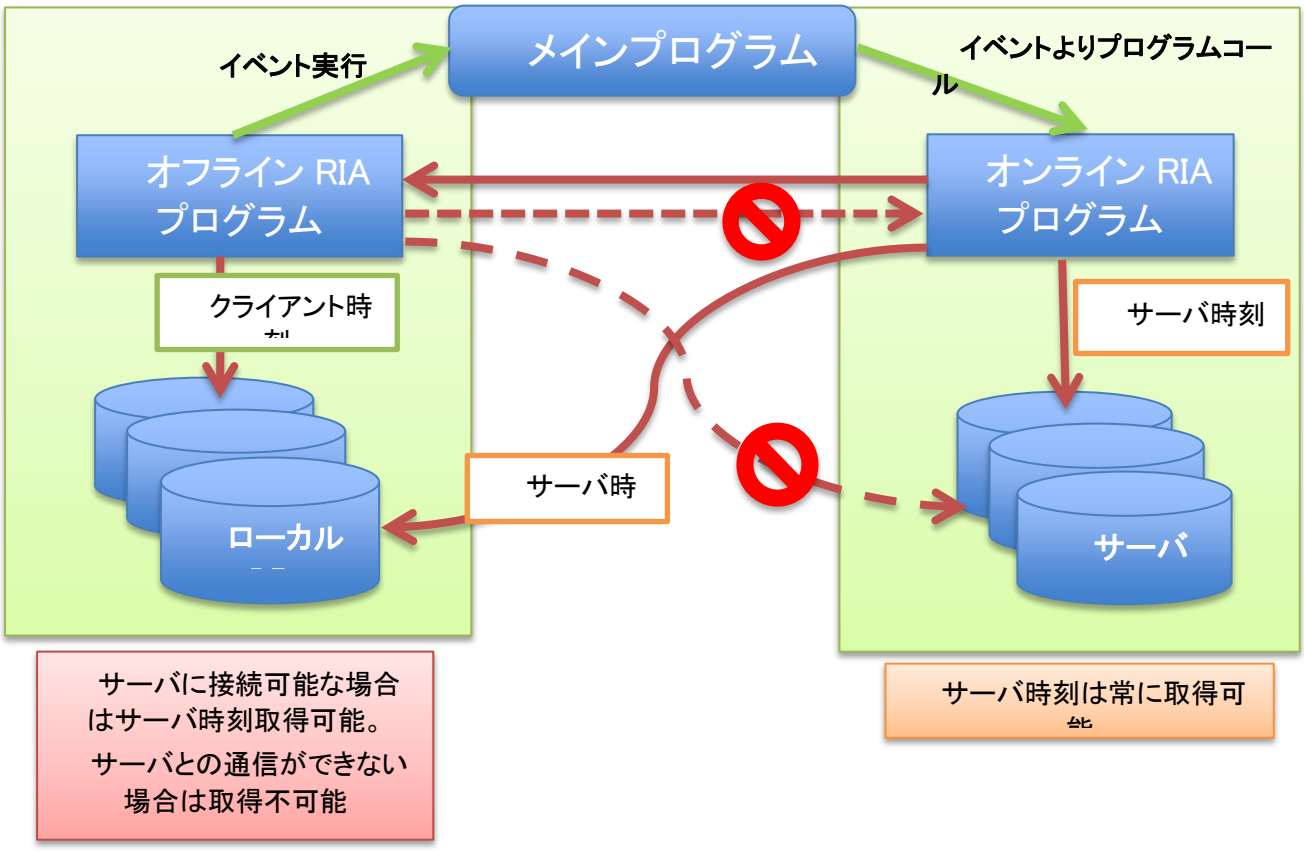
基本的には、差分ダウンロード・差分アップロードともに同じ考え方で処理することができます。ただし、最終更新時刻に更新するタイムスタンプに関して注意が必要です。

4.4 節 「サーバ時刻とクライアント時刻にずれがある場合にはどうしますか？」 で説明されているように、更新時刻は常にサーバ時刻で更新する必要があります。

サーバデータの更新は、オンライン RIA タスク、もしくはバッチタスクで更新されるため、サーバ時刻を取得することが可能です。

一方、ローカルデータの更新は、オンライン RIA タスクの場合はサーバ時刻の取得が可能ですが、オフライン RIA タスクの場合は、サーバ時刻の取得が可能かどうかわかりません。

⇒通信ができない状態の場合、オフライン RIA タスクからイベント経由でオンライン RIA タスクがコールできない（無効サーバ）ため、サーバ時刻が取得できません



ここでは、更新したローカルデータは全て同期してアップロードするという考え方で、「更新フラグ」を使用した方法でアップロード処理を行っていきます。

5.2 差分アップロードをするにはどうしますか？

ローカル テーブルに「更新フラグ」を追加し、更新のあったレコードを識別します。

5.2.1 処理の概略

ローカル テーブルに対してユーザが変更を行い、その結果をアップロードするには、変更レコードを識別するために「更新フラグ」を使います。

この更新フラグを利用した同期処理の概略は、次のようになります。

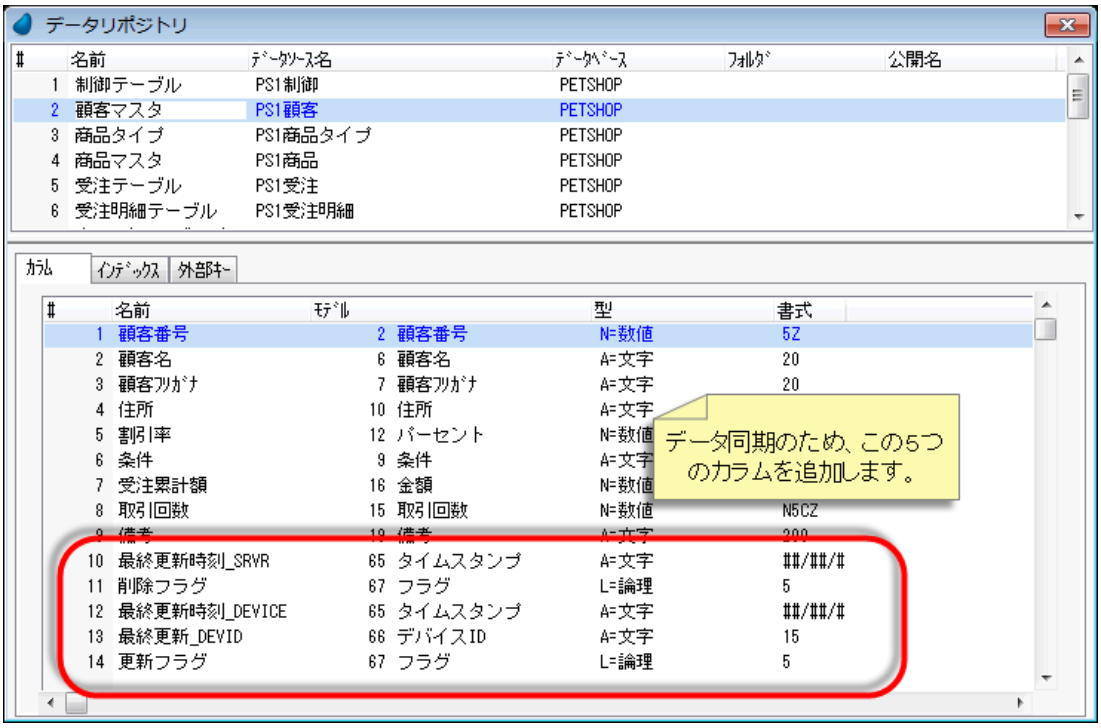
1. 更新フラグは、データをダウンロードしたときに、すべてクリアします。
   → これは、ダウンロードプログラムにおいてすでに行っています。4.9.2 節『差分ダウンロードロジック』を参照してください。
2. 表示・修正プログラム：このテーブルをクライアント端末上でユーザに表示するプログラムは、インタラクティブな RIA オフラインプログラムになります。ここでユーザがデータを更新した場合には、レコード後処理で更新フラグを TRUE に設定します。
3. アップロード プログラム：データのアップロードには、この「更新フラグ」＝TRUE、という範囲づけを行って、処理対象レコードを絞り込みます。データのアップロードが正常に終了した後には、更新フラグをすべてクリアします。

以下に、詳しく説明していきます。

5.2.2 顧客テーブルへはどんなカラムを追加しますか？

この「更新フラグ」は単純な論理型でよいのですが、サーバの途中処理にも使えるので、ローカル テーブル、サーバ テーブル 両方に追加します。

このカラムはすでに、4.9.1 節『「顧客マスタ」テーブルの定義拡張』で追加しました。以下はサーバ テーブルですが、ローカル テーブルにも同じカラムを追加しました。



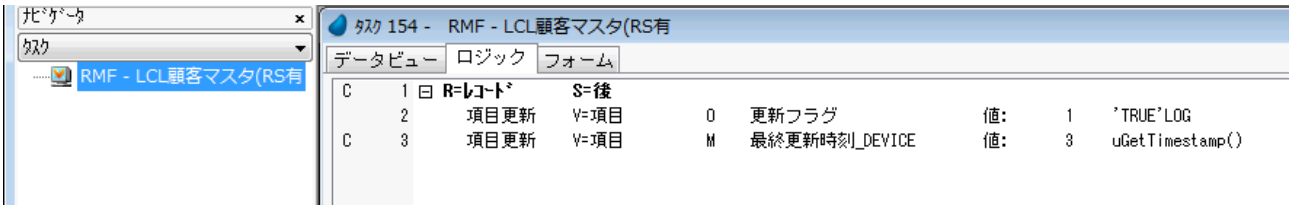
5.3 ローカルテーブルの表示・修正プログラムはどうなりますか？

レコード後処理で、最終更新時刻と更新フラグをセットします。

このテーブルをクライアント端末上でユーザに表示するプログラムは、インタラクティブな RIA オフラインプログラムになります。ここでユーザがデータを更新した場合には、レコード後処理で更新フラグを TRUE に設定します。

データビューでは、「最終更新時刻_DEVICE」と、「更新フラグ」も選択します。

レコード後処理で、この二つのカラムを更新します： 更新フラグは TRUE にし、最終更新時刻_DEVICE は現時刻（クライアント時刻）のタイムスタンプをセットします。



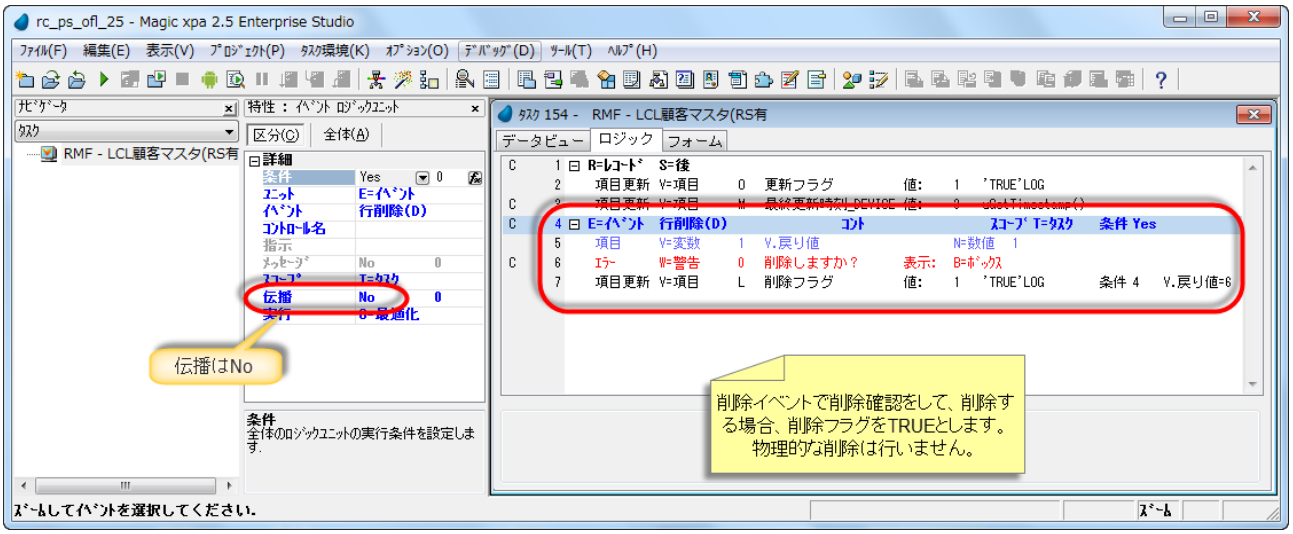
5.4 ローカルテーブルの削除ロジックはどうなりますか？

行削除イベントハンドラで、削除確認を行い、削除フラグを TRUE に更新します。

オフラインプログラムにおいて、レコードを単純に削除するとサーバとクライアントのデータの同期を正しくとれなくなってしまうことがあります。クライアント側のテーブルで、実際にレコードを削除してしまうと、「レコードを削除した」という情報も削除されてしまうので、サーバ側のテーブルを削除するタイミングがなくなってしまいます。

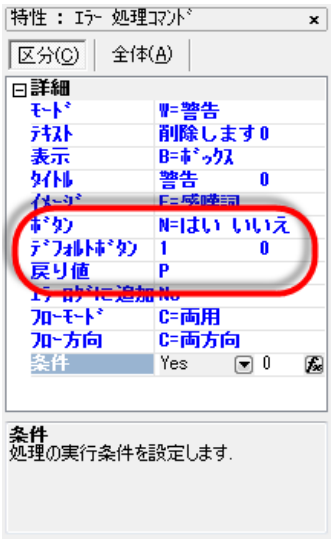
このため、削除ロジックでは、レコードの物理的な削除は行いません。削除フラグを更新して削除情報を残し、削除レコードもサーバ側と同期できるようにします。

このようなロジックでデータを扱う必要があるので、「削除」操作は、内部的には「行削除イベント（内部イベント）ハンドラで、削除フラグを TRUE に更新する。」という処理になります。このハンドラを実行したら、実際のレコード削除は行ってはいけませんので、ハンドラの「伝播」特性は No に設定します。



このイベントハンドラでは、削除フラグ設定に先立ちユーザに確認ダイアログを表示して、Yes と答えた場合にだけフラグを設定するようにしています。

エラーコマンドの特性は右図のようになっています。



5.5 アップロード プログラムはどのようになりますか？

以下のような処理となります。

① DataViewToDataSource() 関数で、更新フラグ＝TRUE のデータをアップロードします。

② サーバ テーブルのフラグを更新します。

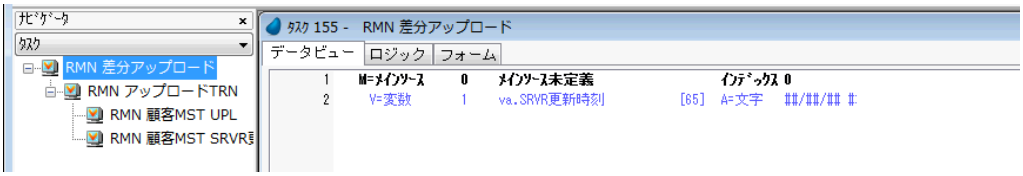
③ ローカル テーブルのフラグをリセットします。

3階層のタスク構造で作りますが、いずれも、RIA オンラインの非インタラクティブ タスクです。

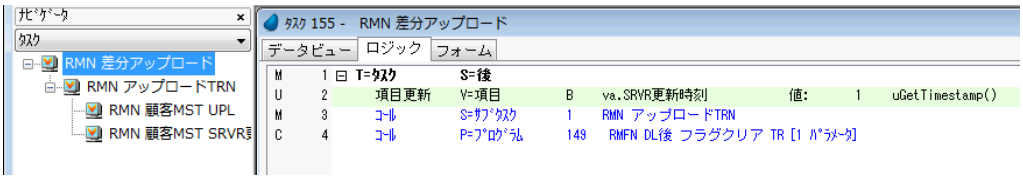
5.5.1 トップレベルのタスク

トップレベルのタスクは、全体のとりまとめを行います。トランザクションは設定しません。

サーバ更新時刻として、現在のタイムスタンプを取得します。



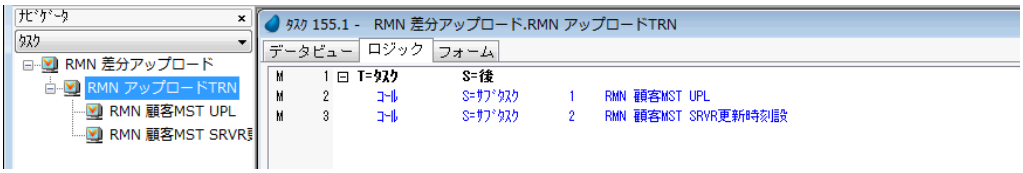
レコード処理は行いません。タスク後処理で、アップロードタスクと、ローカルテーブルのフラグをクリアするタスクとを呼び出します。



5.5.2 サブタスク: トランザクションを実行するタスク

サブタスクでは、タスクレベルの遅延トランザクションを行います。このタスクでトランザクションを設定することにより、中途半端なレコードアップロードを防止します。

これもレコードレベル処理はありません。タスク後処理で、アップロードタスクとサーバテーブルのフラグ更新プログラムをよびだします。

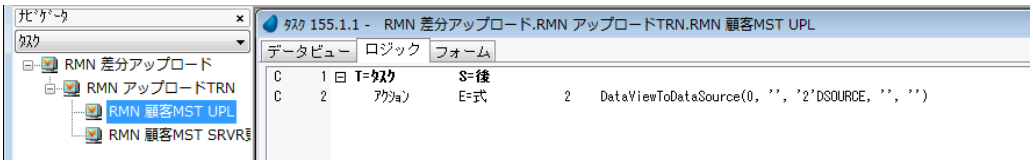


5.5.3 孫タスク1： データアップロードタスク

孫タスク1では、DataViewToDataSource() 関数を使って、更新レコードをアップロードします。

データビューで、「更新フラグ」＝TRUE の範囲指定をしています。

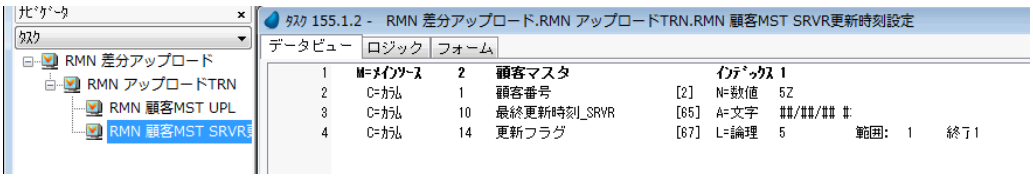
タスク後処理で、DataViewToDataSource() を実行します。



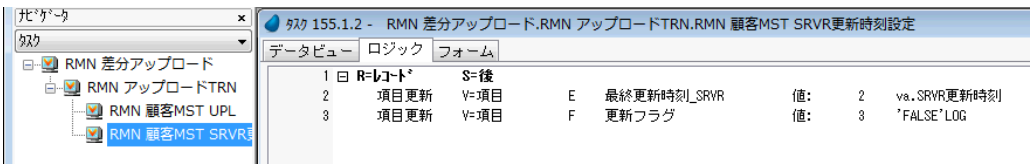
5.5.4 孫タスク2：サーバテーブルのフラグ更新

サーバ側の顧客テーブルで、「最終更新時刻_SRVR」カラムに現在のタイムスタンプを設定し、「更新フラグ」は FALSE に設定します。

この処理の対象となるのは、ローカル テーブルからアップロードされたレコードだけです。これを判別するため、データビューで「更新フラグ」＝TRUE という範囲づけを行っています



レコード後処理で、設定をしています。

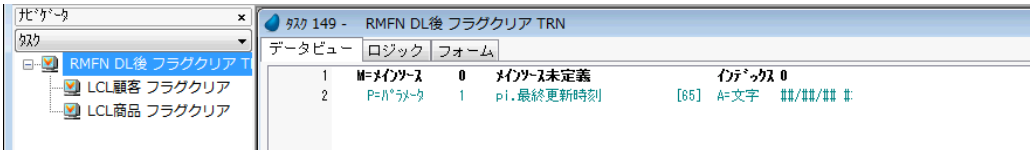


5.5.5 ローカル テーブルのフラグ リセット プログラム：親タスク

ローカルテーブルの更新フラグをリセットするプログラムは、ローカルテーブルだけをアクセスするので、RIA オフラインの非インタラクティブ タスクとして実装します。

このため、別の独立したプログラムとして実装します。

親タスクでは、全体のとりまとめとして、パラメータ（最終更新時刻）を受け取り、タスク前処理で子タスクを呼び出します。



5.5.6 ローカル テーブルのフラグ リセット プログラム: 子タスク

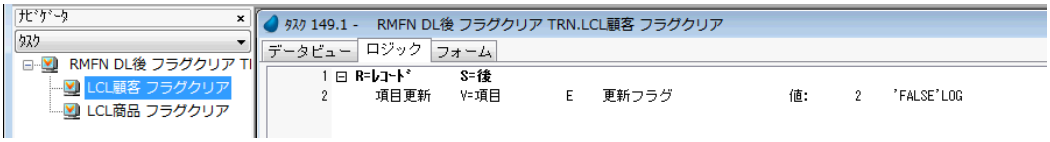
子タスクでは、それぞれのテーブルの「更新フラグ」を FALSE にリセットします。

ローカルテーブルごとに、子タスクを作成します。

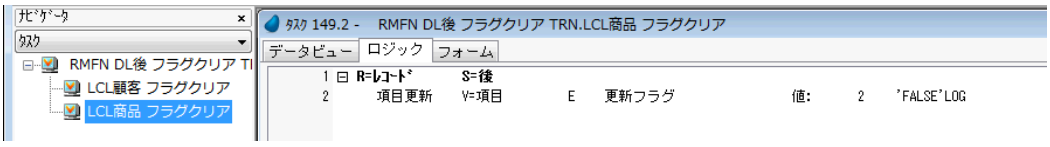
ここでは、ローカル顧客テーブルと、ローカル商品テーブル用に二つの子タスクを作りました。

ローカル顧客テーブル用のデータビューです。範囲として、パラメータ（最終更新時刻）以後、かつ、更新フラグ＝TRUE という条件を付けています。

レコード後処理でフラグをクリアします。



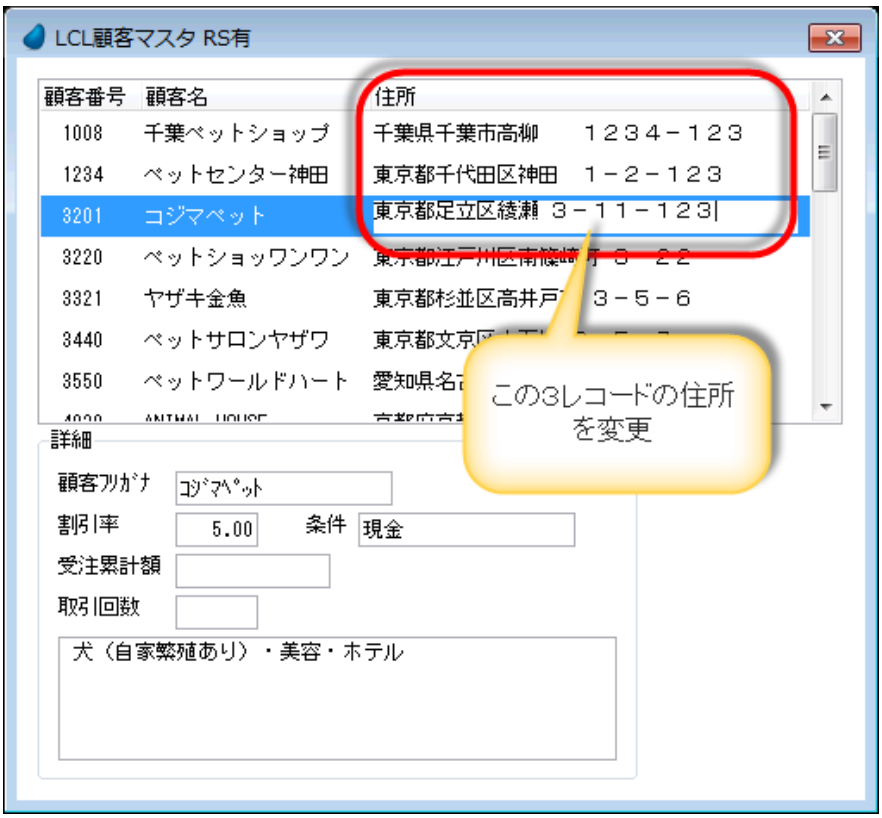
ローカル 商品マスタについても同様です。



5.5.7 実行例

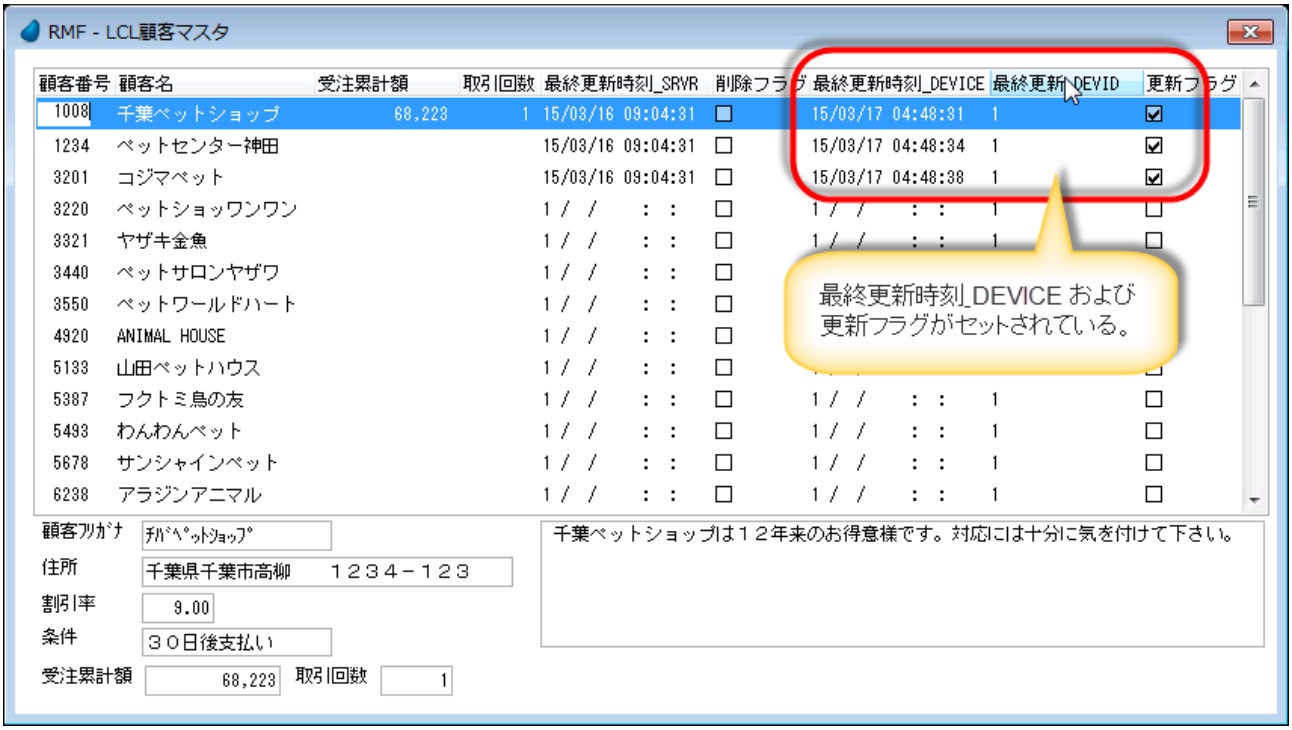
まず、ローカル 顧客マスタのレコードをいくつか変更します。

下図の例では、上の3レコードの住所を変更しました。



ローカル顧客テーブルの、データ同期関係のカラムの値を見てみると、「採取更新時刻_DEVICE」と「更新フラグ」とが設定されています。

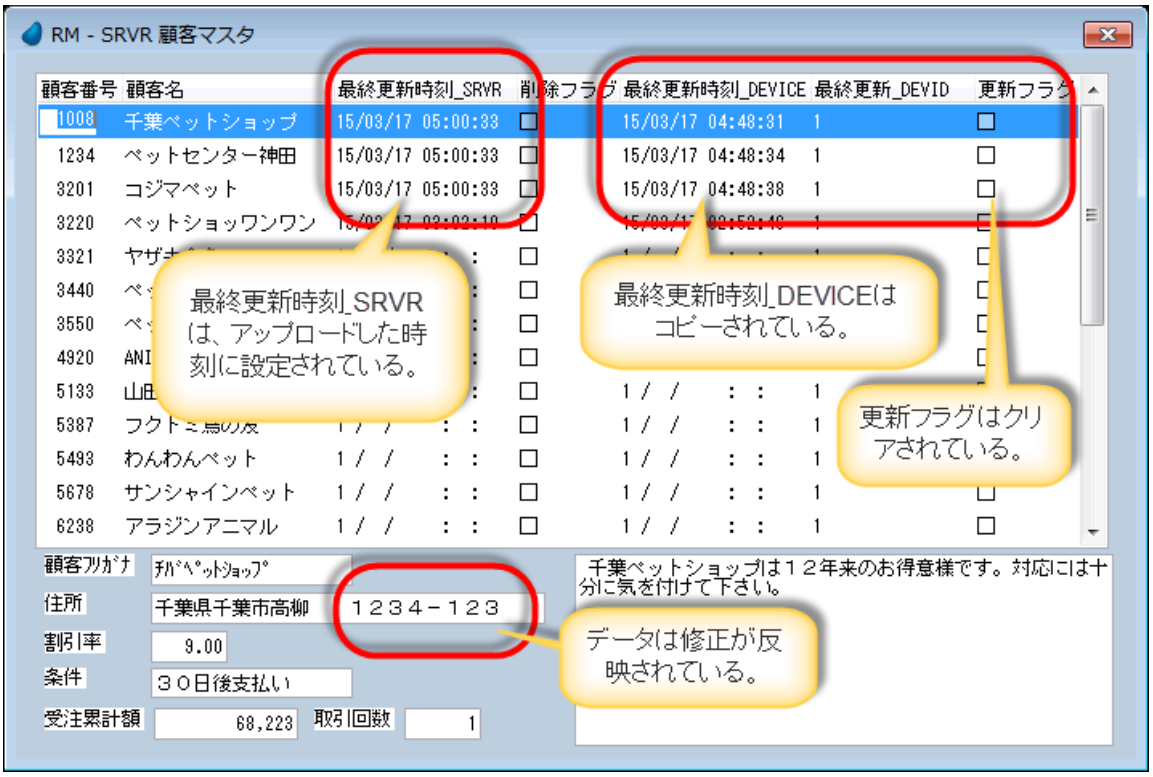
この時点では、まだサーバにデータはアップロードされていません。



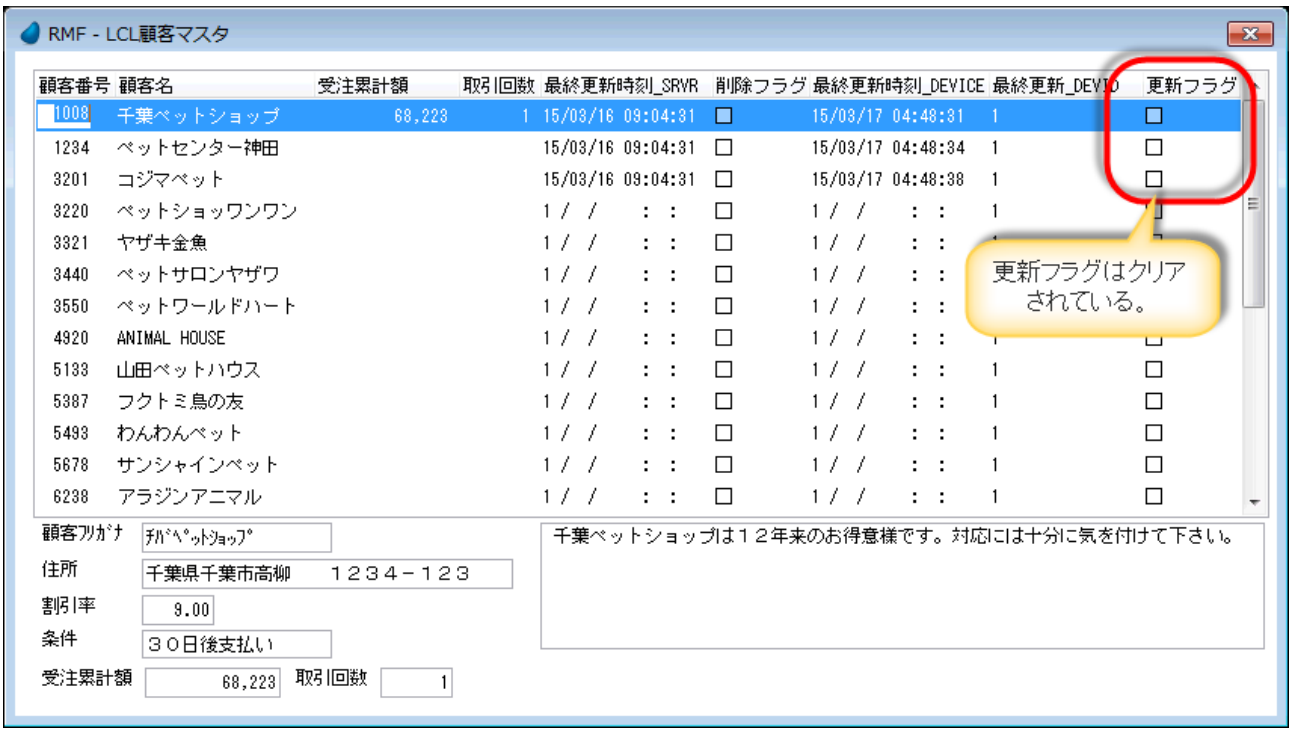
次に、差分アップロードプログラムを実行します。ここでは、Studio から F7 で実行しました。

その後、サーバ側の顧客テーブルのレコードを見てみます。
先頭の3レコードについて、以下の結果を確認してください。

- 「最終更新時刻_SRVR」カラムは、アップロード処理を実行した時のタイムスタンプに設定されています。
- 「最終更新時刻_DEVICE」カラムは、ローカルテーブルからそのままコピーされています。
- 「更新フラグ」カラムは、クリアされています。



最後にローカルテーブルの内容を確認してみます。更新フラグがクリアされているはずです。


RMF - LCL顧客マスタ
顧客番号 顧客名 受注累計額 取引回数 最終更新時刻_SRVR 削除フラグ 最終更新時刻_DEVICE 最終更新_DEVID 更新フラグ
1008 千葉ペットショップ 68,223 1 15/03/16 09:04:31 15/03/17 04:48:31 1
1234 ペットセンター神田 15/03/16 09:04:31 15/03/17 04:48:34 1
3201 コジマペット 15/03/16 09:04:31 15/03/17 04:48:38 1
3220 ペットショッワンワン
3321 ヤザキ金魚
3440 ペットサロンヤザワ
3550 ペットワールドハート
4920 ANIMAL HOUSE
5133 山田ペットハウス 1
5387 フクトミ鳥の友 1
5493 わんわんペット 1
5678 サンシャインペット 1
6238 アラジンアニマル 1
更新フラグはクリアされている。
顧客ﾌﾘｶﾞﾅ ﾁﾊﾞﾍﾟｯﾄｼｮｯﾌﾟ
住所 千葉県千葉市高柳 １２３４－１２３
割引率 9.00
条件 ３０日後支払い
受注累計額 68,223 取引回数 1
千葉ペットショップは１２年来のお得意様です。対応には十分に気を付けて下さい。

5.6 アップロード失敗に対する考慮が必要ですか？

必要です。RIA アプリケーションでは、ネットワーク接続が突然切れる可能性があり、そのような場合にもデータ不正が起こらないように考慮が必要となります。

RIA オフラインアプリケーションが想定するような環境では、無線 LAN などの不安定なネットワーク接続が大前提であり、クライアントデバイスとサーバの間のネットワークが確実につながっているという保証はありません。このため、ネットワーク接続が突然切れる、という事態も想定した設計を行う必要があります。

データ同期処理、特にアップロード処理においては、アップロードデータが中途半端に更新されて不正データが発生すると大きな問題になりますし、また、せっかく入力したデーローカルデータが失われたりすると作業の効率に影響があります。

アップロードをしようとして、オフライン状態からオンライン状態に移行しようとしたときに、サーバに接続できなかった。

→ この場合には、タイムアウトにより「無効サーバ」イベントが発生し、オフライン状態で処理が継続します。

アップロードは始めから行われないので、データ更新不正ということは起こりません。

最初はサーバに接続できてアップロード処理が始まったのだが、処理の途中でネットワーク接続が切れてしまった。

→ これが一番困るケースです。

中途半端にデータ更新されて、データ不整合が発生する可能性があります。

これに対しては、トランザクションを適宜設定して、データの不整合が起こらないようにします。

RIA オフラインの場合、データ更新は、ローカル テーブル、サーバ テーブル 両方に渡って行われるので、本質的に「分散トランザクション」になります。分散トランザクションを完璧の行うのは非常に難しい技術であり、Magic xpa RIA サーバでは厳密な意味での分散トランザクションは実装されていません。

プログラム上では、全体を遅延トランザクションで設定することは可能ですが、実際の処理においては、サーバ側のトランザクションと、ローカル側のトランザクションとが別々に走っています。

このような場合、サーバ側のトランザクションは無事にコミットされて終了するが、ローカル側は失敗する、あるいはその逆のケースが起こる可能性があります。

このような事態を完全に防止することが現実的ではありません。従って、設計方針としては、そういうこともありうると想定して、サーバ、ローカル両方で、データの状態（最終更新時刻）を確認できるようにしておく必要があります。そして、不一致があれば補正し、同じデータアップロードを重複して 2 回行わないようにします。もし可能であれば、万一同じデータアップロードを重複して行ってしまったとしても、同じ注文が二つ出るなどの不正な結果がおこらないようにできればより良いでしょう。

6 選択プログラム

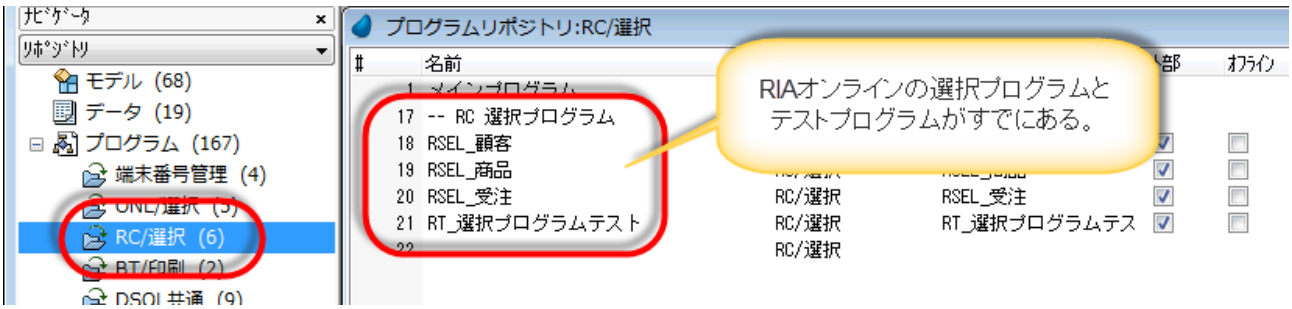
受注データの取り扱いに入る前に、選択プログラムを作っておきましょう。

実はこれは非常に簡単で、タスクをオフラインにして、ローカルテーブルを参照するように変更するだけでほとんど終わりです。

以下に詳しく説明していきます。

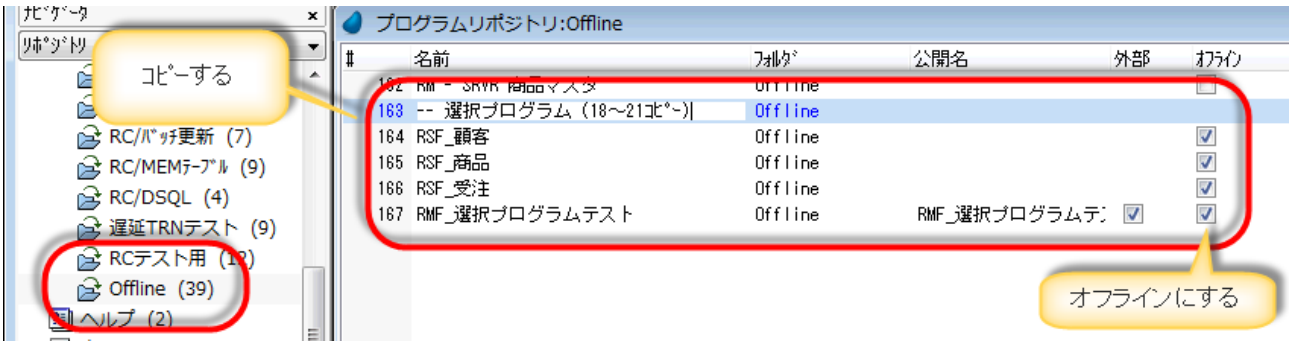
6.1.1 RIA オンライン選択プログラムのコピー

サンプルプロジェクトには、クライアント・サーバ アプリケーションから RIA 化した際に、すでに RIA の選択プログラムが作られています。



これをそのままコピーします。ついでに、テストプログラムもあるので、これもコピーします。

コピーした後、「オフライン」のチェックボックスをチェックして、RIA オフラインタスクとします。

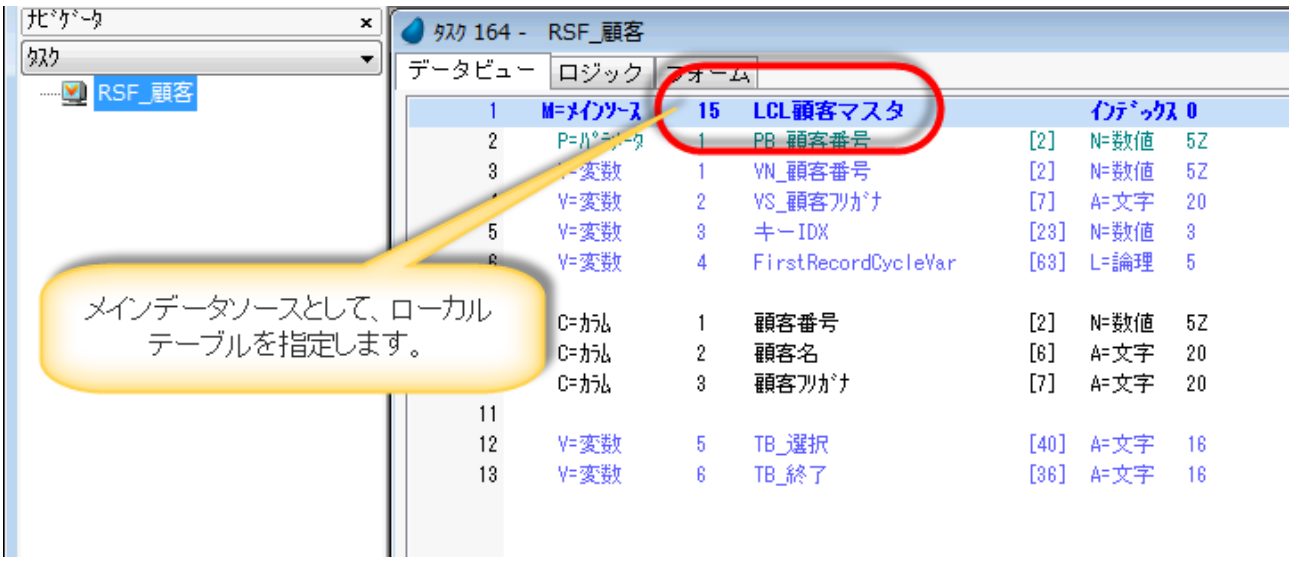


6.1.2 顧客選択プログラムのデータソース変更

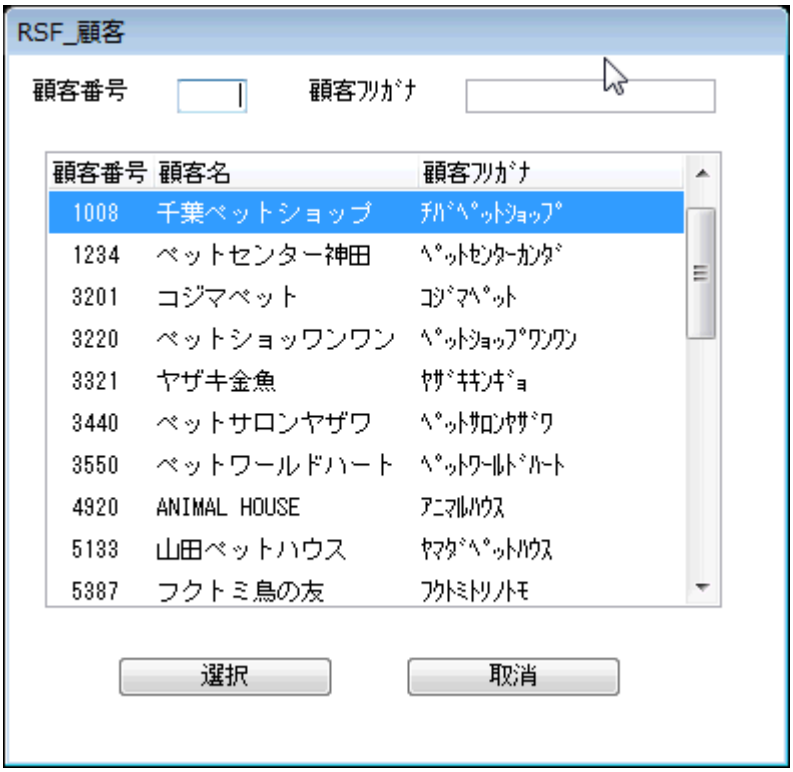
顧客選択プログラムにおいて、もともとのメインソースは テーブル＃2「顧客マスタ」でした。これはサーバ テーブルです。

オフラインとして使うために、これに対応するローカル テーブル ＃15「LCL 顧客マスタ」に設定しなおします。

これだけで出来上がりです。

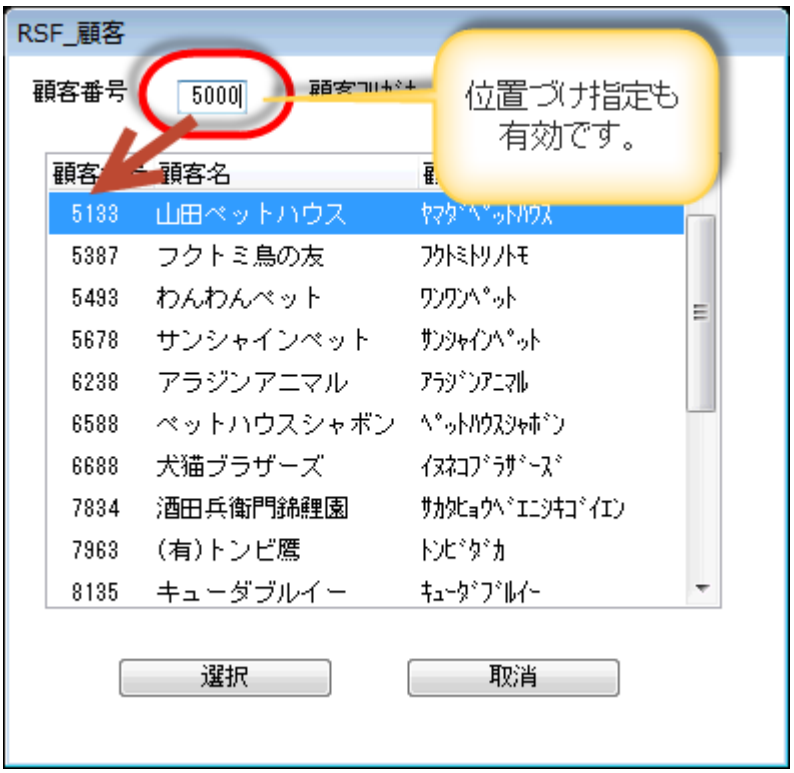


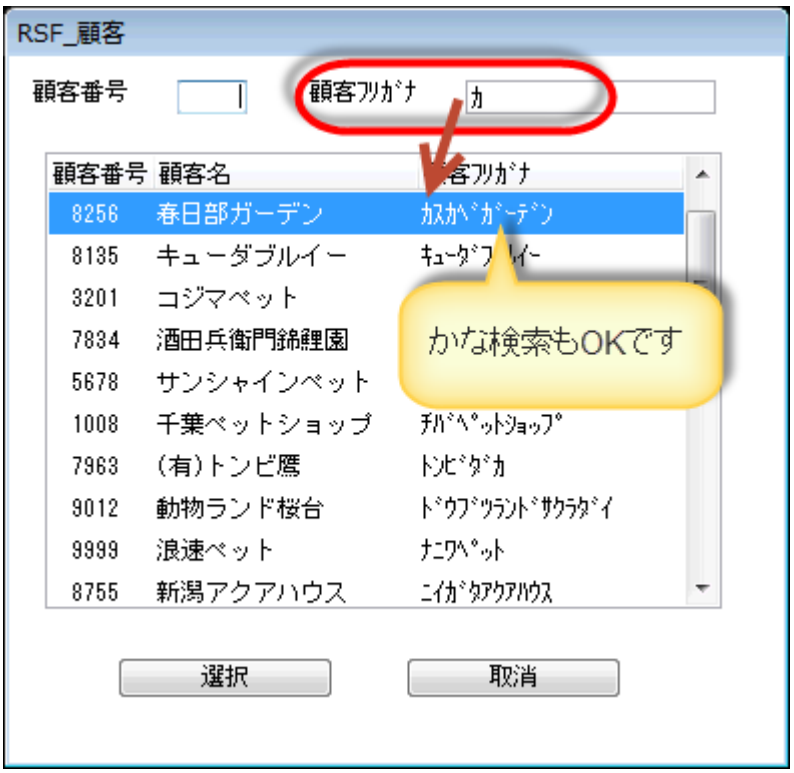
実際に実行してみます。



この選択プログラムには、顧客番号および顧客フリガナによる位置づけ指定ができるようになっています。すなわち、ユーザが顧客番号を入力すると、その番号以上の最初のレコードの位置づけされて表示されます。顧客フリガナも同様で、ユーザが半角フリガナを入力すると、それをキーとして位置づけされます。

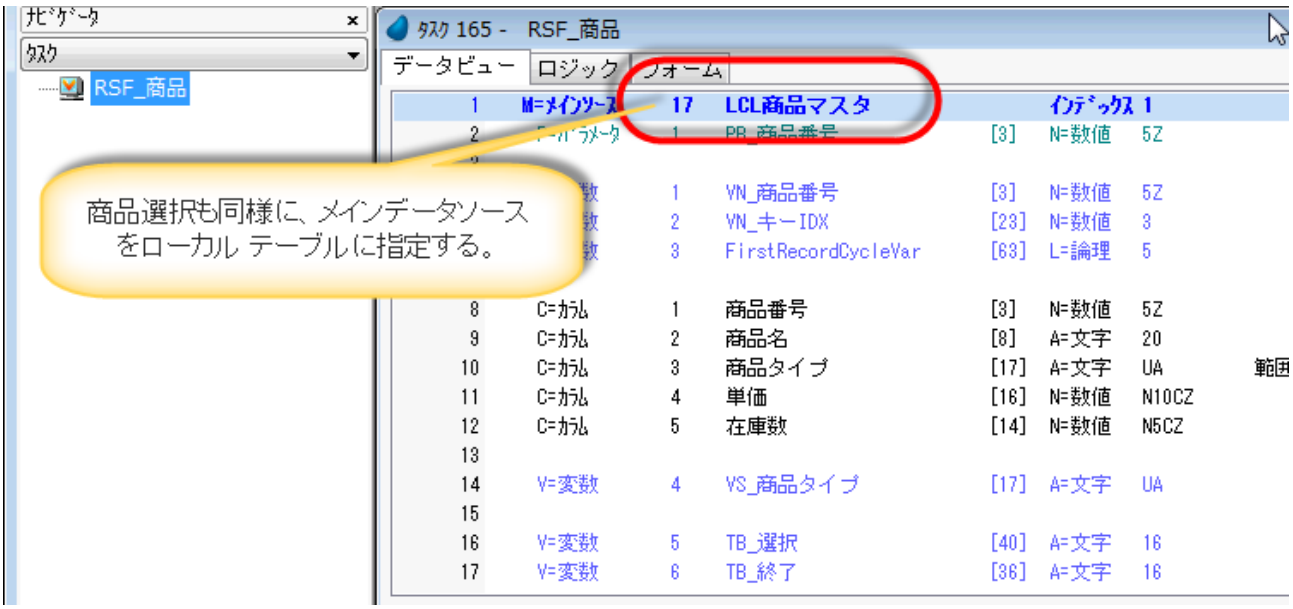
オフライン化した後のプログラムでも、この機能がそのまま有効になっていることが確認できます。





6.1.3 商品選択プログラム

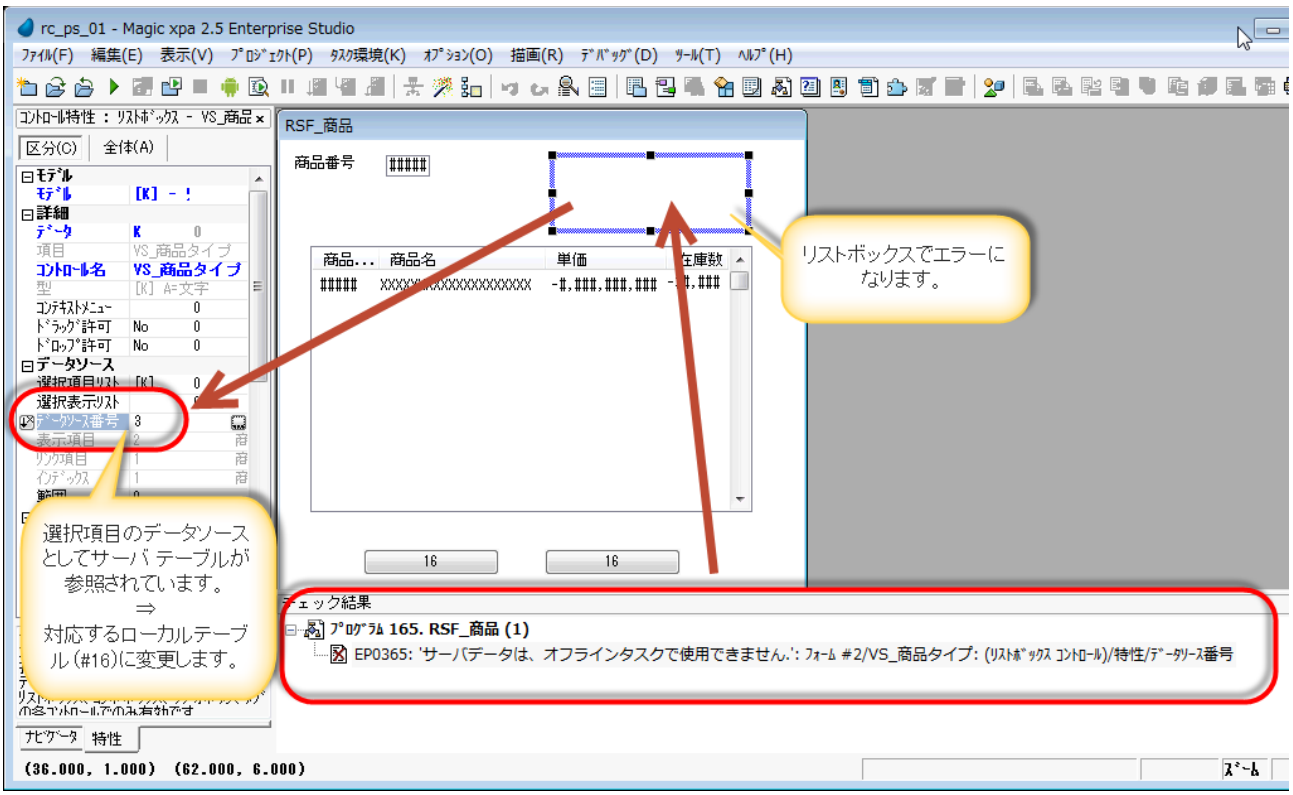
商品選択プログラムでも、同様に、メインソースをローカルテーブル（＃17 LCL 商品マスタ)に変更します。



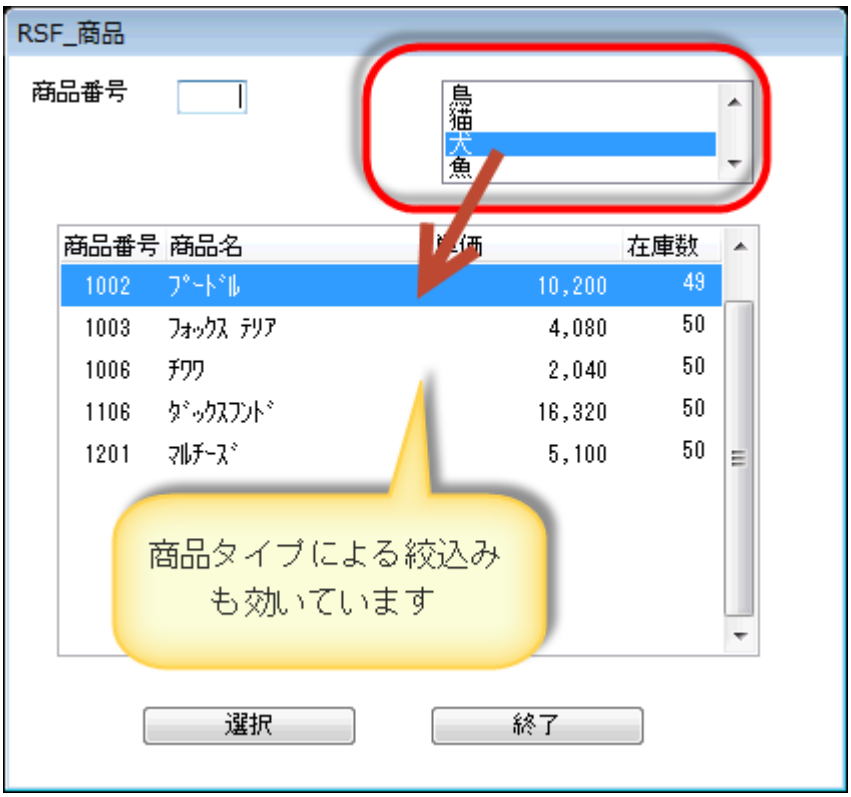
ここで F8 でシンタックスチェックを行うと、エラーが出ます。エラーはフォームのリストボックスの「データソース番号」プロパティです。ここにはサーバ側のテーブルである#3「商品タイプ」が指定されているので、オフラインでは利用できません。

このプロパティも、対応するローカルテーブル（＃16 LCL 商品タイプ）に変更します。

これで F8 も通るようになります。



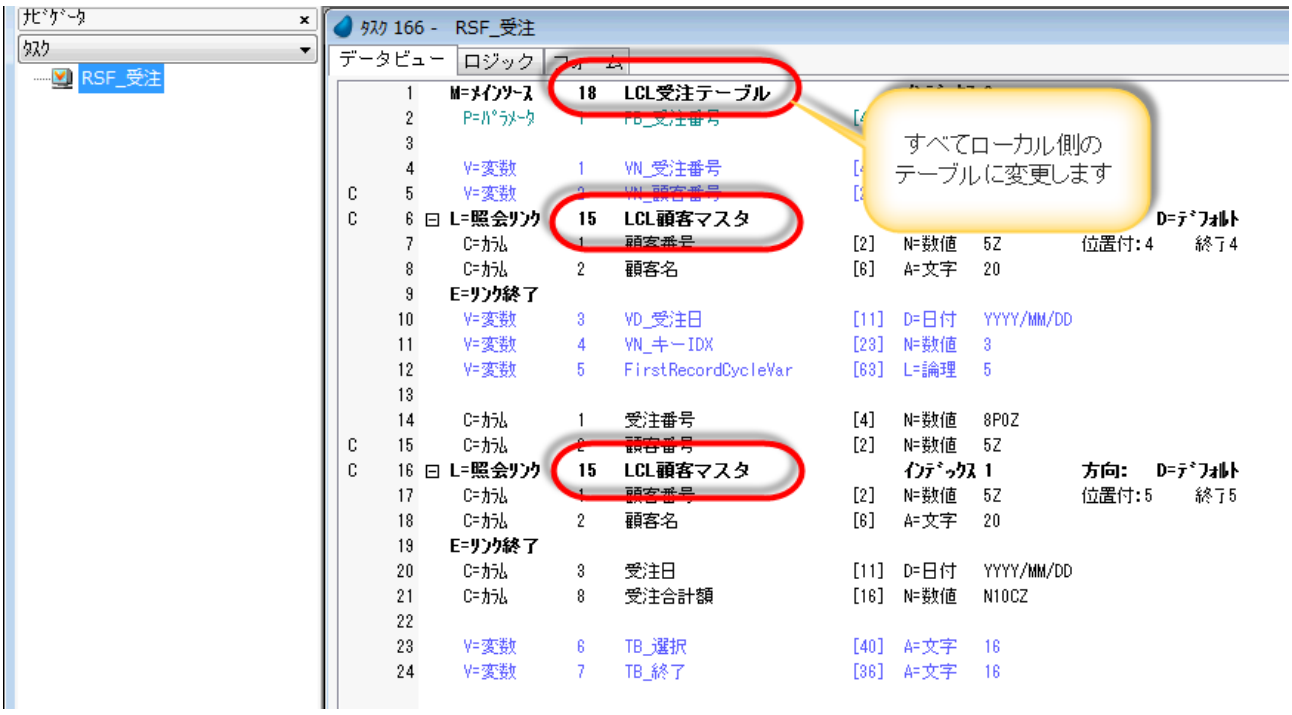
この商品選択プログラムでは、商品番号による位置づけ機能の他に、商品タイプをリストボックスから選んで、商品一覧表示を絞り込むことができます。実行してみると、この機能も、オフライン化した後にも有効であることが確認できます。



6.1.4 受注選択プログラム

受注選択プログラムも同様に、ローカル テーブルを指定するようにします。

このプログラムには、メインソースの他に、データリンク（照会リンク）が 2 か所あるので、これも対応する「LCL 顧客マスタ」に変更します。



このプログラムも問題なく動作しますが、現時点ではまだ受注データが 1 件もないので、実行しても「レコードがない」というエラーになります。受注データが作られれば、このエラーは出なくなります。

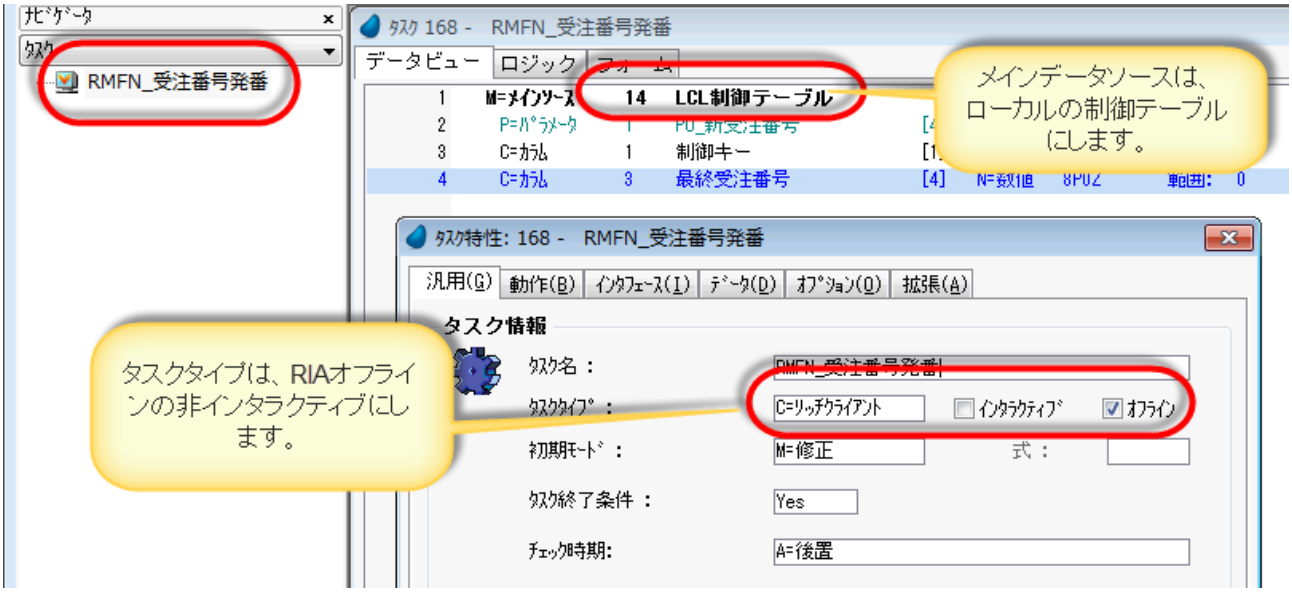
6.1.5 受注番号発番

選択プログラムではありませんが、受注番号の発番プログラムも、ここでオフライン化しておきましょう。

受注番号の発番プログラムは、もともとはプログラム39番で、バッチタスクとして実装されていました。

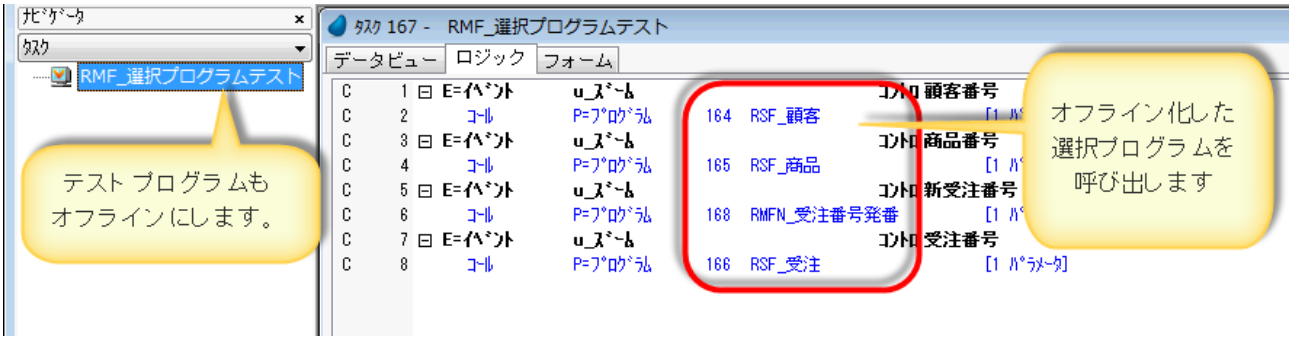
オフライン化するために、このプログラムをコピーし、次の点を変更します。

- バッチタスクから、RIA オフライン、非インタラクティブタスクにします。
- メインソースを、サーバ側の＃2「制御テーブル」から、ローカル側の＃14「LCL 制御テーブル」に変更します。



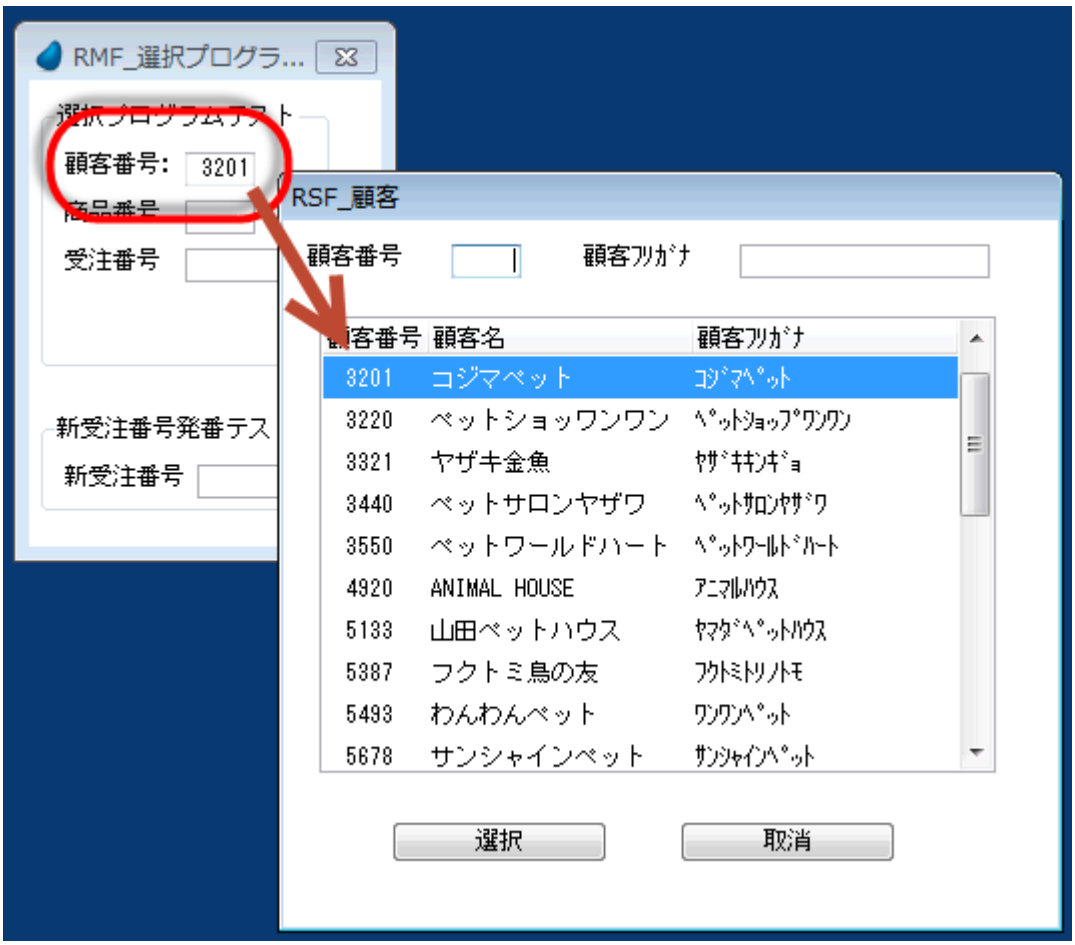
6.1.6 テストプログラム

テストプログラムも、RIA オンラインから RIA オフラインタスクとし、それぞれで呼び出すプログラムも、上記でオフライン化したものを呼び出すようにします。



実行してみて、選択プログラムおよび受注番号発番プログラムが正しく動作することを確認してください。

下図は、顧客番号 フィールドから F5 ズームして、顧客選択プログラムを呼び出しているところです。


RMF_選択プログラ...
顧客番号: 3201
受注番号
新受注番号発番テス
新受注番号
RSF_顧客
顧客番号
顧客ﾌﾘｶﾞﾅ
顧客番号 顧客名 顧客ﾌﾘｶﾞﾅ
3201 コジマペット ｺｼﾞﾏﾍﾟｯﾄ
3220 ペットショップワンワン ﾍﾟｯﾄｼｮｯﾌﾟﾜﾝﾜﾝ
3321 ヤザキ金魚 ﾔｻﾞｷｷﾝｷﾞｮ
3440 ペットサロンヤザワ ﾍﾟｯﾄｻﾛﾝﾔｻﾞﾜ
3550 ペットワールドハート ﾍﾟｯﾄﾜｰﾙﾄﾞﾊｰﾄ
4920 ANIMAL HOUSE ｱﾆﾏﾙﾊｳｽ
5133 山田ペットハウス ﾔﾏﾀﾞﾍﾟｯﾄﾊｳｽ
5387 フクトミ鳥の友 ﾌｸﾄﾐﾄﾘﾉﾄﾓ
5493 わんわんペット ﾜﾝﾜﾝﾍﾟｯﾄ
5678 サンシャインペット ｻﾝｼｬｲﾝﾍﾟｯﾄ
選択
取消

7 受注データの扱い

今までは、顧客マスタなどの、一つのテーブルを単位としてダウンロード、アップロードすることのできるデータのみを扱ってきました。

ここでは、受注データのように、テーブルの依存関係のあるデータのダウンロードとアップロードについて説明します。

7.1 受注データにはどういう特性がありますか？

受注データは、明細データ、顧客マスタ、商品マスタなど、他のテーブルとの間に密接な依存関係があり、ダウンロードやアップロード時には、レコードをコピーするほかに付随処理が発生します。

受注データは、顧客マスタのようなデータに比べ、他のテーブルとの依存関係が強い、という特徴があります:

1. 受注ヘッダ・受注明細というような1:N の親子関係があります。このため、同じ受注番号を持つヘッダレコードと明細レコードとは、常にひとかたまりの情報として取り扱う必要があります。
2. 顧客マスタ テーブルに、受注回数、受注額、というような統計情報があります。この統計情報は、受注が発生するたびに更新する必要があります。また、実際のアプリケーションでは、与信限度額のチェックなどのチェックも入るでしょう。
3. 商品マスタに対し、在庫の確認や引当の処理があります。
4. それぞれのチェックが通らなかった場合(例えば、在庫が足らなかった場合)には、処理を中断して、クライアント側にその旨通達する必要があります。

このような特性があるので、アップロードの際には、個々のレコードをコピーするだけでは済みません。それに付随する処理が発生しますので、そのためのロジックを作る必要があります。

このような付随処理や、処理の中断の可能性があるデータをアップロードする際には、レコードを直接本番データベースに書き込むのは好ましくありませんし、また付随処理のロジックが難しくなります。このような場合には、一旦アップロードデータを一時テーブルに書き込んで、一時テーブルのデータを確認しながら、本番テーブルへのマージ処理を行うのが安全でわかりやすいので良いでしょう。

一方、ダウンロードの場合には、クライアント側での付随処理というものがないので、単純なコピーでたいてい間に合うでしょう。ただ、上の 1 で書いたように、同一受注番号のヘッダと全明細とを、ひとかたまりの情報として扱うところに気を付ける必要があります。

7.2 受注データのダウンロードはどのように行いますか？

受注データのダウンロードは、参照専用のデータとして扱って良いとすると、それほど複雑ではありません。ダウンロードすべきヘッダを決めたら、付随する明細データもともにダウンロードするようにすれば良いです。

7.2.1 ダウンロードレコードの選択

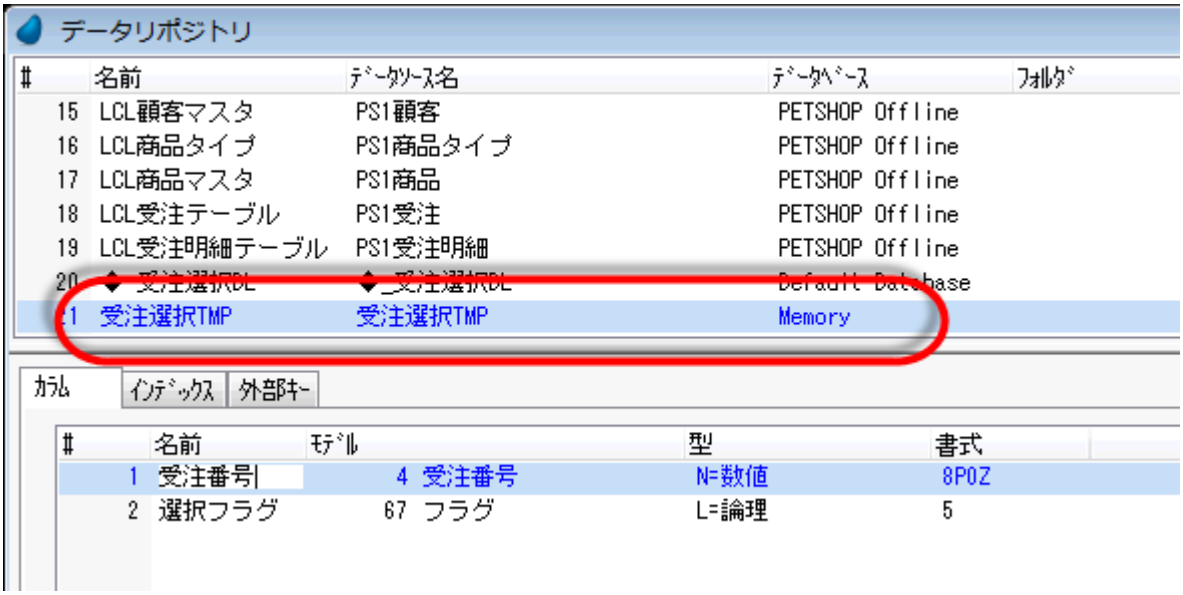
普通、すべての注文情報をダウンロードする必要はありません。各自が自分のデバイスにダウンロードしたい注文番号を選択できるよう、受注一覧タスクを作成しましょう。

このタスクは、サーバ側の受注データを一覧表示して、その中から選択する、というものですから、オフラインでは実行することができません。そのため、オンライン RIA タスクとして定義します。

次に、ユーザが「選択した/しない」を記録しておくために、別途一時テーブルをテーブルリポジトリに定義します。ここで必要なカラムは、次の二つだけです。

1. 受注番号
2. 選択 FLG（論理型）

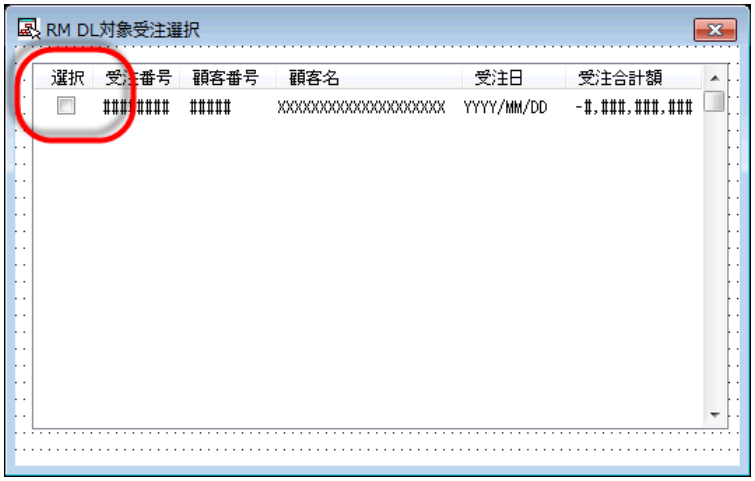
このテーブルは、セッションの中でだけ一時的に使うものなので、メモリテーブルでよいでしょう。



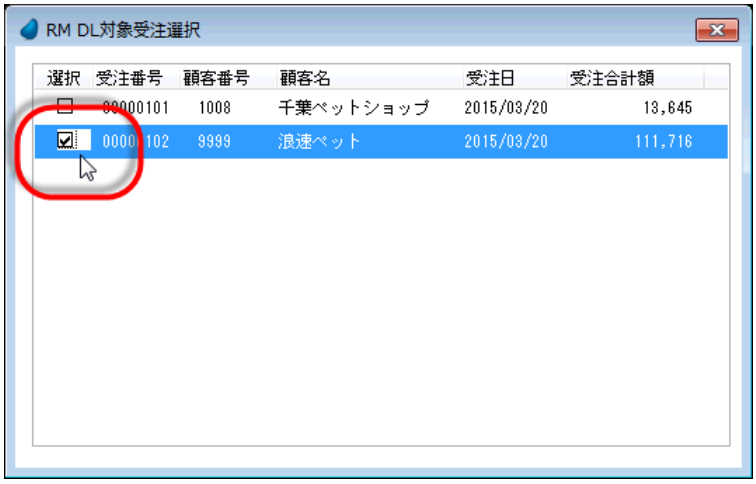
これをリンクして、選択 FLG をチェックボックスとして表示し、ユーザが選択できるようにします。

データビュー

フォーム



実行時画面

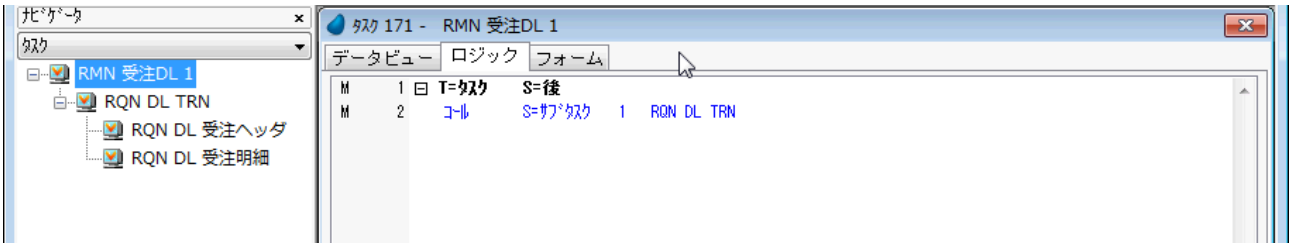


7.2.2 ダウンロードのロジック ①

この選択タスクでダウンロードすべき受注レコード番号を決定したら、ダウンロードのロジックは、だいたい次のようになります。

注：このロジックそのままでは、実行時にエラーになります。その理由と対応方法については、節を追って説明します。

全体は、以下に示すような3段階のタスクになります。



これらのタスクはすべて、

① ユーザとのインタラクティブタスクな対話を必要としない（非インタラクティブ）

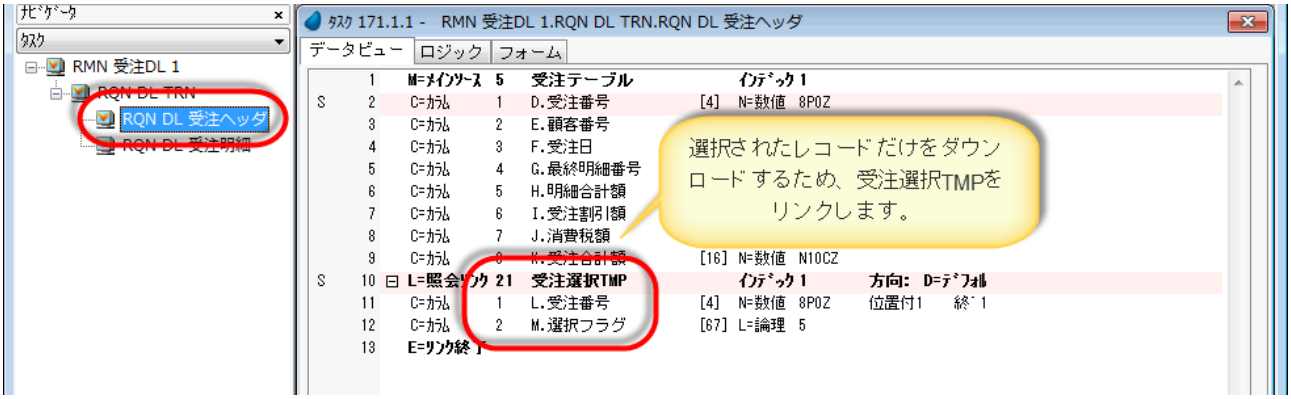
② サーバへのアクセスを必要とする（オンライン RIA）

ということから、非インタラクティブのオンライン RIA タスクとなります。

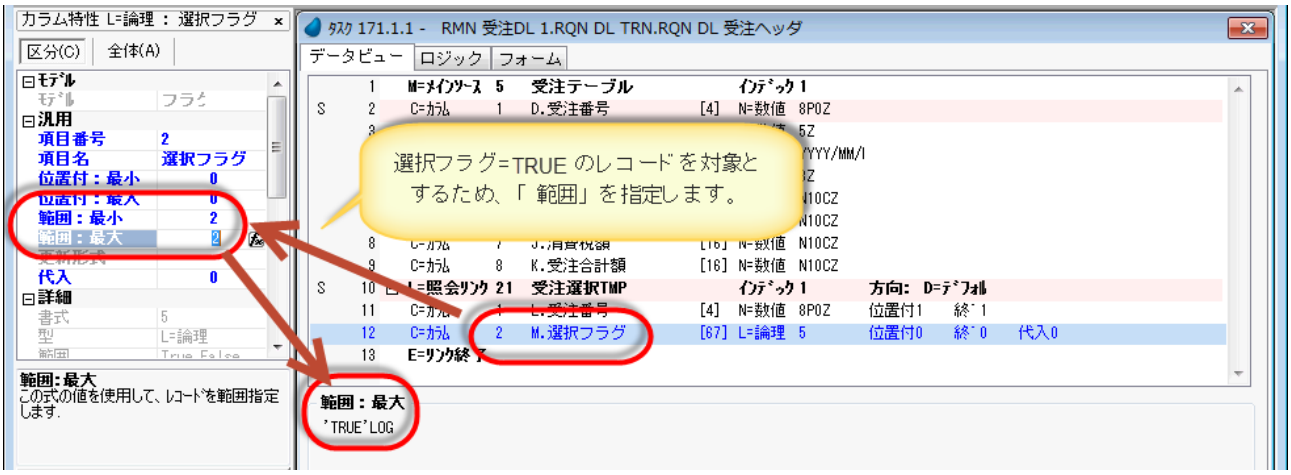
子タスクでは、トランザクションを実行します。子タスクおよび孫タスクで行われるすべてのデータ処理は、このトランザクションでカバーされることになります。これは、途中で処理の中断があった場合に、処理をすべてロールバックして、データの整合性を確保するためです。モバイルデバイスなどでは特に接続が切れやすいので、このようなデータ整合性への考慮は重要です。

孫タスクではそれぞれ、受注ヘッダレコードと受注明細レコードのダウンロードを行います。前節の「選択プログラム」で選択した受注情報だけをダウンロードの対象にしたいので、この一時テーブルをリンクし、「選択フラグ」が TRUE であるレコードだけを対象とするようにします。

第一の孫タスクでは、メインテーブルは「受注テーブル」です。リンクコマンドで「受注選択 TMP」テーブルをリンクします。

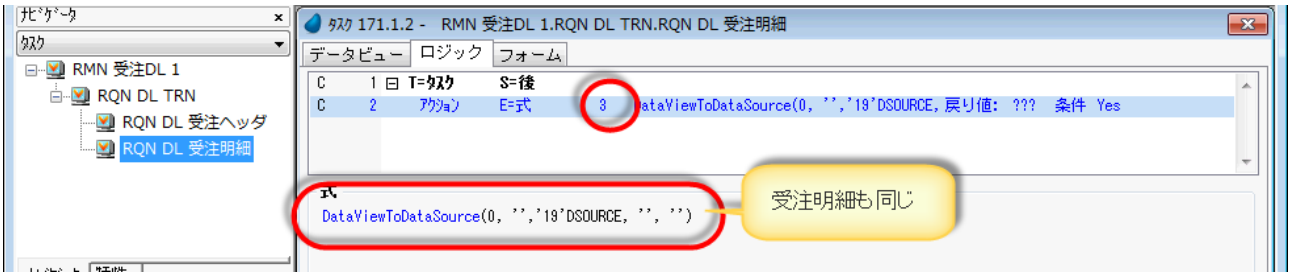


「選択フラグ」＝TRUE のレコードだけを対象とするため、範囲の指定を行っています。



タスク後処理で、DataViewToDataSource() 関数を使い、レコードのダウンロードを行います。

第 2 の孫タスクも同様にして、明細データ（受注明細テーブル）のダウンロードを行います。



さて、このようにして作ったプログラムを実行してみると、エラーになります。

```
<556925768686691392> 20/03/2015 19:32:15.151    [Error ]
 - DataViewToDataSource - Illegal destination columns specified : 選択フラグ,
 program: RMN 受注 DL 1.RQN DL TRN.RQN DL 受注ヘッダ
<556925768686691392> 20/03/2015 19:32:15.308    [Error ]
 - DataViewToDataSource - Illegal destination columns specified : 選択フラグ,
 program: RMN 受注 DL 1.RQN DL TRN.RQN DL 受注明細
```

なぜエラーが出たのでしょうか？エラーメッセージは、「Illegal destination columns specified : 選択フラグ」で、訳すると「不正な転送先カラムが指定されました: 選択フラグ」ということになります。

DataViewToDataSource() 関数を実行する際、「タスクの項目名（第 2 パラメータ）」、「出力カラム名（第 4 パラメータ）」に、いずれも「' '」空文字を指定しました。このように指定した場合、

- タスクの項目名は、タスクのデータビューに定義されているすべての項目、
- 出力カラムは、「タスクの項目名」と同一

として実行されます。

ここで、「タスクのデータビューに定義されている項目」としては、メインテーブルの他にリンクテーブルや変数項目も含まれます。今の例では、リンクテーブルの項目「受注番号」および「選択フラグ」も含まれることになります。

一方、出力データ先はテーブル 18 番 「LCL 受注データ」となりますが、これには「選択フラグ」というカラムは存在しません。

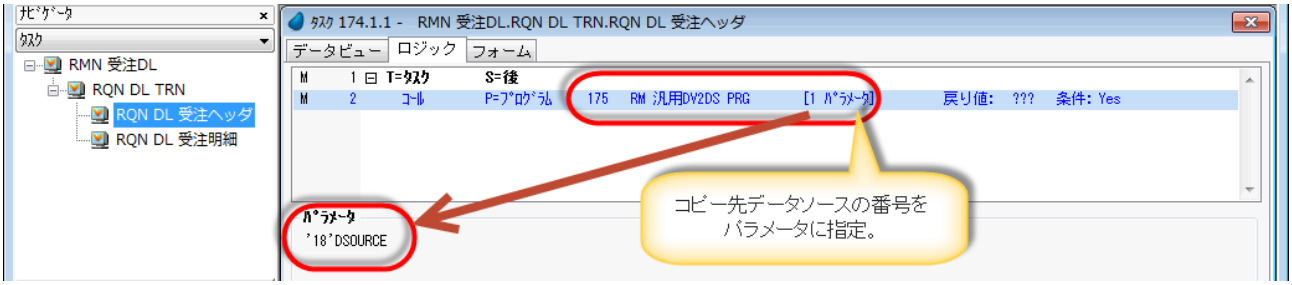
従って、存在しない不正なカラム名「選択フラグ」が指定された、ということになってしまいますので、このエラーが発生した、ということです。

7.2.3 ダウンロードのロジック ②

これに対応するには、「受注選択 TMP」テーブルのリンク項目（受注番号、選択 FLG）はダウンロードの対象から除外しなければなりません。

ここでは、「3.5.4 対象カラムを限定するには？」の方法2 に従い、「3.5.3 汎用ダウンロードタスク」 で説明したタスクを使って、ダウンロードするようにします。

受注ヘッダプログラムでは、タスク後処理で、DataViewToDataSource() 関数を直接呼び出す代わりに、汎用ダウンロードタスクを呼び出します。このとき、コピー先データソースの番号（今の場合、ローカルテーブル「LCL 受注テーブル」の番号 18 ）を指定します。



この方法では、フォームにはメインデータソースの項目のみを配置するようにして、対象カラムを限定するようにします。以下のように、メインソースの項目のみを配置し、リンクデータの項目は配置しません。

受注明細レコードのダウンロード:

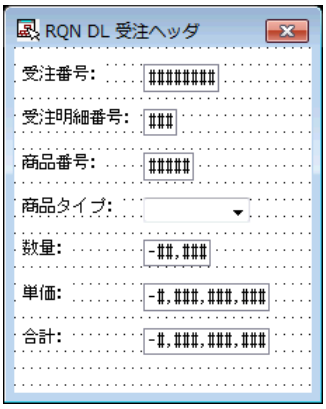
受注明細レコードも、受注ヘッダレコードとほぼ同じロジックでダウンロードすることができます。

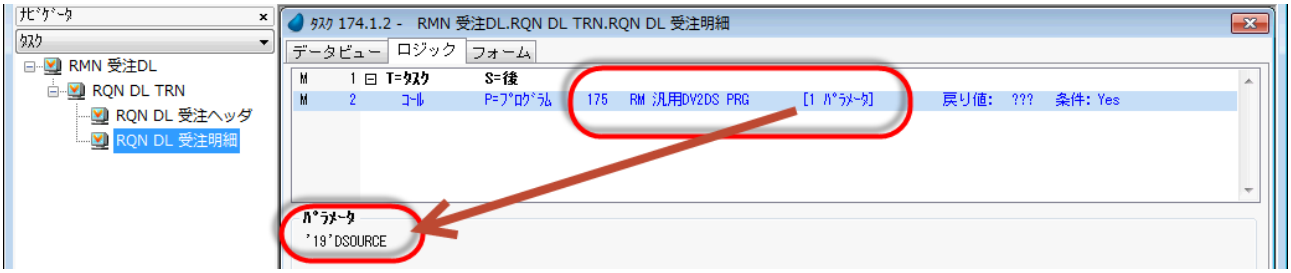
すなわち、

- RIA オンラインタスクで、インタラクティブ ＝ No、終了＝Yes、チェック時期＝A=後置

- メインデータソース ＝ 受注明細テーブル（サーバ テーブル）
- リンク データソース ＝ 選択 TMP テーブル
- 範囲づけとして、選択 TMP テーブルの「選択 FLG」＝ TRUE とします。

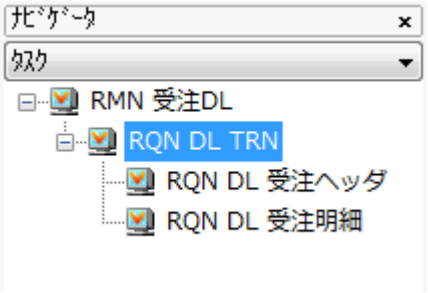
タスク後処理でダウンロード処理を呼び出します。ここでも「3.5.3 汎用ダウンロードタスク」で説明した汎用のダウンロードプログラムを呼び出します。





7.2.4 トランザクション

以上の処理は、一つのトランザクションでまとめて行わなければなりません。このため、一段「トランザクション」用のタスクを作成しておきます。このタスクでは、タスクレベルの遅延トランザクションを設定します。



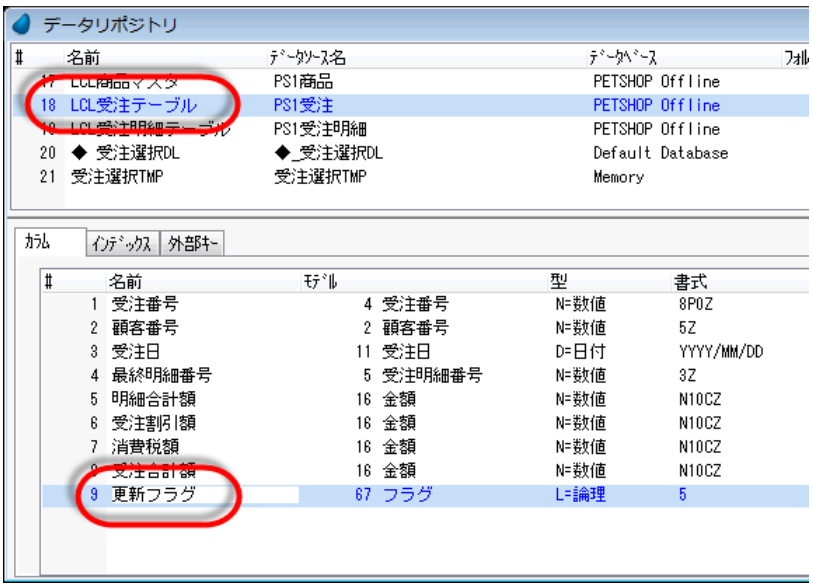
更新フラグのクリア：ローカル テーブルにダウンロードした後には、更新フラグをクリアしておきます。ロジックとしては、サーバで「更新フラグ」が TRUE になっていることはないと思いますが、念のためです。今回は省略します。

7.3 受注データのアップロードを必要最小限にするには？

ローカルの受注ヘッダ レコードで、アップロード対象であるかどうかを区別するために、「更新フラグ」カラムを追加します。

ローカルに受注データをダウンロードした後、ユーザがデバイス上でデータを登録あるいは修正して、サーバにアップロードする必要があるデータ量は、全体からすれば一般には小さい割合になるでしょう。このため、ローカルテーブルの受注データのうち、アップロードすべきものと、する必要がないものを区別する必要があります。

このため、ローカルにある受注ヘッダテーブル（テーブル#18「LCL 受注テーブル」）に、「更新フラグ」カラムを追加します。タイプは L=論理型です。



これに対応して、サーバテーブル（テーブル#5「受注テーブル」）にも、同じカラムを追加します。

「更新フラグ」のカラムは、受注ヘッダのテーブルにだけ追加します。明細テーブルには追加しません。ヘッダと明細とは1対 N の関係にあり、ヘッダをアップロードした場合にはそれに付随する明細レコードもすべてアップロードしなければならないので、明細レコードに個別に更新フラグを設ける必要がないからです。

7.4 受注入力プログラムはどのように作りますか？

RIA オフラインでの受注入力は簡単化することができます。

RIA オフライン アプリケーションで受注入力を行うときには、

1. オフライン状態で、ローカル テーブルに受注データを登録しておく。
2. 適当なタイミングで、サーバにアップロードして、注文を確定する。

という2段階を経ることになります。

ここでは、第 1 段階の、ローカルテーブルに受注データを登録していくためのプログラムについて説明します。第 2 段階のデータアップロードについては、次節で悦明します。

7.4.1 受注入力プログラムの概要

RIA オフラインで受注入力プログラムを作るときには、通常の RIA オンラインの場合よりも簡単化することができます。

その理由の第一は、RIA オフラインはローカル テーブルだけにアクセスするので、他ユーザとのレコードを共有することがないので、ロックとトランザクションとによるレコードの排他制御に悩まされる必要がないからです。

一般に、Magic のプログラムで複雑になるのは、排他制御の考慮をしなければならないからです。元祖 PetShop の受注入力プログラムは非常に簡単なのに、複数ユーザの共有による排他制御について考え始めた途端にプログラムが何倍も複雑になっていまいます。

それに対し、RIA オフラインでは排他制御を行う必要がないので、元祖 PetShop のような、非常に簡単なつくりで十分になります。

理由の第 2 は、在庫の確認や引き当ては、第2の段階でオンラインになってデータをアップロードするときに、付随処理として行うので、第1の段階で行う必要はないため、そのロジックを省略できるからです。

元祖 PetShop では、受注明細レコードのレコード後処理で、商品マスタの在庫数を更新し、受注ヘッダのレコード後処理で顧客マスタの取引回数および取引額の更新を行っていました。

このとき、「項目更新」コマンドで「加算＝Yes」のオプションを使うことにより、ロジックを簡単化することができましたが、RIA タスクでは「加算＝Yes」の「項目更新」コマンドがサポートされていません。このため、RIA タスクで同様のことをしようとすると、ロジックが少し複雑になります。

一方、RIA オフラインの受注入力プログラムでは、顧客マスタも商品マスタも、ローカル データベースにあるものを参照します。このレコードの在庫数とか取引回数とかをオフライン状態で更新しても、実際にはあまり意味はありません。

その理由は、

- オフライン テーブルのデータは、サーバにあるデータと乖離している可能性があります。サーバデータは常に更新されているのに対し、オフラインテーブルのデータは同期した以後の変更が反映されていないからです。
- オフライン テーブルの商品の在庫数を更新しても、サーバの商品 在庫数に反映されるわけではありません。

などです。

従って、顧客マスタや商品マスタの更新は、第 2 段階のデータアップロード時に行うようになります。逆に言うと、第 1 段階では行う必要がありません。そのため、このロジックも省略することができます。

このような理由から、プログラムはかなり簡単になります。

簡単には、登録モードでどんどん登録させていくこと、明細データはサブフォームで表示すること、明細レコードのレコード後処理で、ヘッダ レコードの「受注合計」を更新すること、などが主な処理となりましょう。

7.5 受注番号の発番はどのように行いますか？

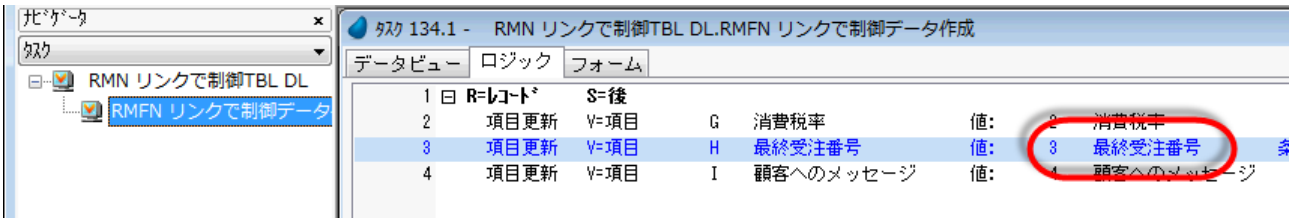
受注番号の発番では、仮の受注番号を使います。

受注番号は、仮の受注番号を使います。例えば、90000000 以上を仮番号として定義し、ローカルの「OF_制御ファイル」の「最終受注番号」に登録します。受注番号を発番する際は、このカラムを1づつ増加させていくようにします。

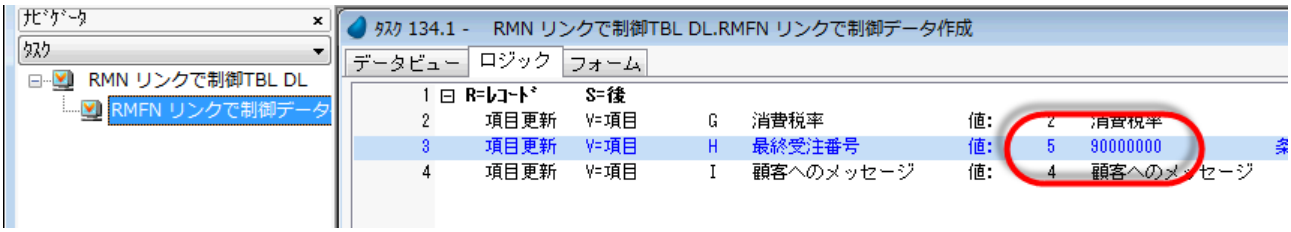
このようにするので、仮の受注番号は、90000000 から始まるものである必要があります。このため、ローカルテーブル「LCL 制御テーブル」の「最終受注番号」の初期値は 90000000 にしておきます。

これは、制御テーブルのダウンロードプログラムを変更して、ダウンロード後に強制的に 90000000 に設定するようにします。

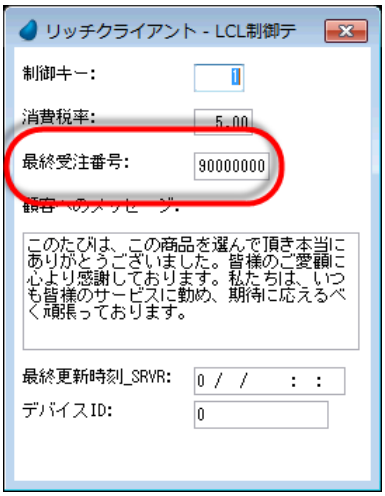
変更前「RMN リンクで制御 TBL DL」プログラムは、次のようになっていました。



変更後は、次のようにします。「最終受注番号」に 90000000 を設定するところが異なります。



このプログラムを実行した直後の「LCL 制御テーブル」の初期状態はこのようになります。



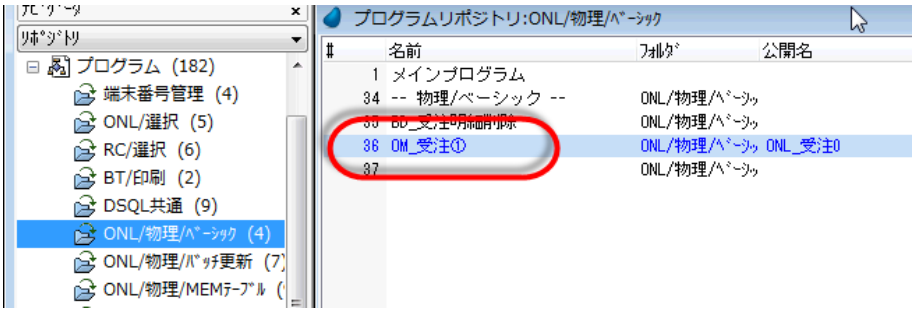
7.6 受注入力プログラムは最終的にどのようになりますか？

受注入力プログラムは、一番簡単なオンラインの受注入力プログラムからコピーして変更します。

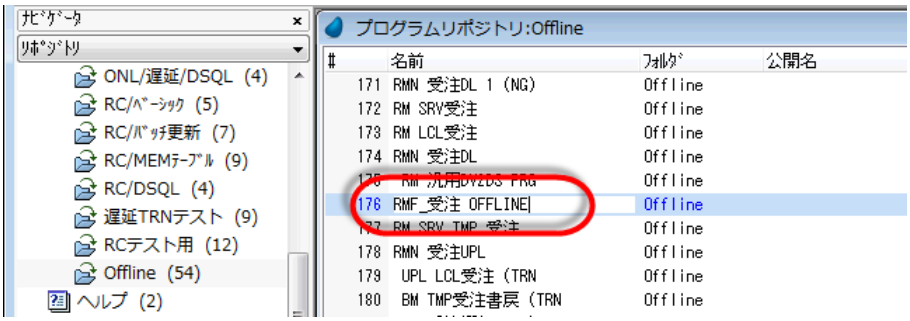
オフラインの受注入力プログラムは、前述のように、複数ユーザの同時実行に対する考慮が不要なので、簡単化することができます。具体的には、オンラインの一番簡単な、オリジナルのペットショップデモの受注入力のようなもので十分です。

7.6.1 受注入力プログラムのコピー

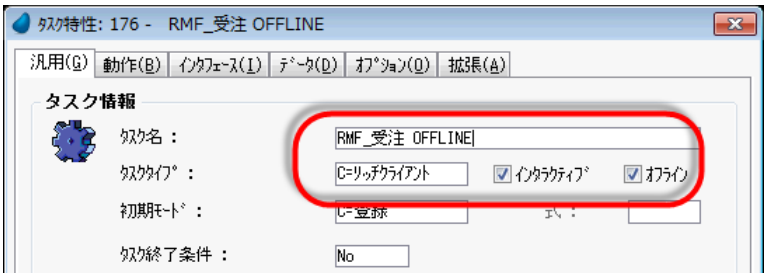
オンラインの受注入力プログラムの一番簡略なものは、プログラム 36 番「OM_受注①」です。



コピーして名前変更します。



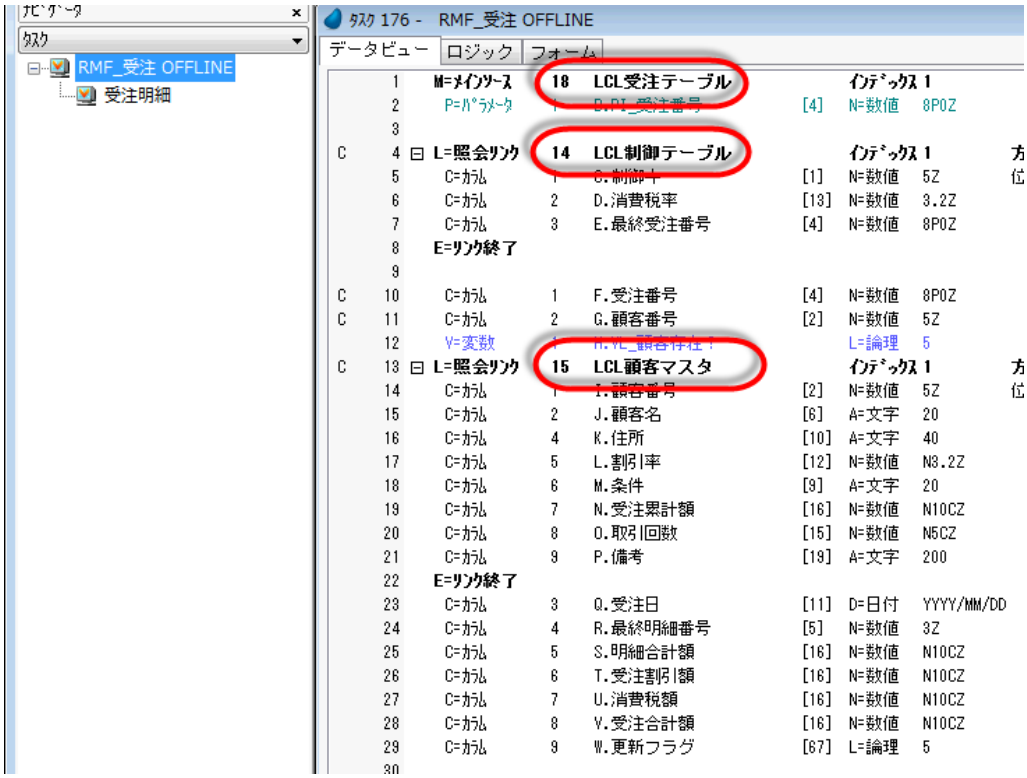
RIA オフラインに変更します。



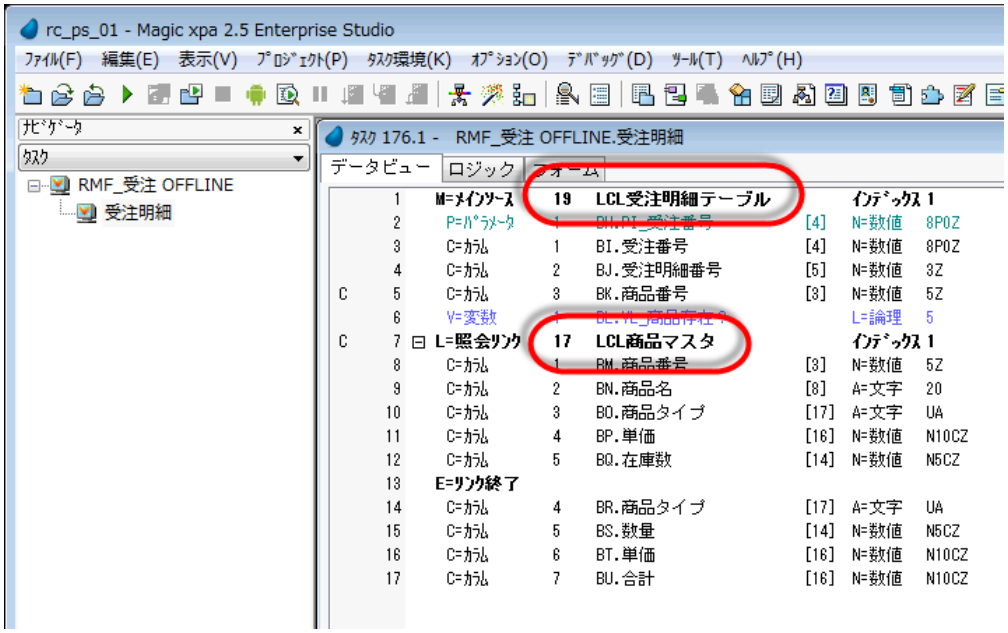
7.6.2 データソースの変更

親タスク、子タスクともに、メインソース、リンクデータのデータソースを、対応するローカルテーブルの番号に付け替えます。

親タスク：メインテーブル、リンクテーブル共に、サーバテーブルから、対応するローカルテーブルに変更します。

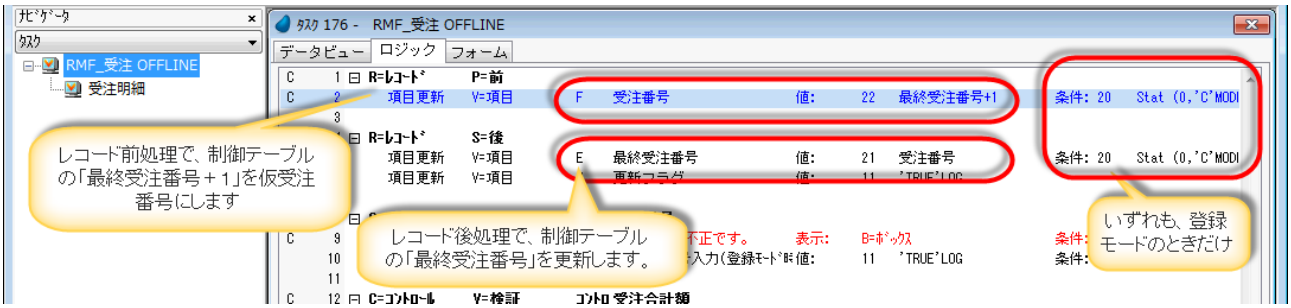


子タスクも同様に、サーバテーブルをローカルテーブルに置き換えます。



7.6.3 受注番号の発番

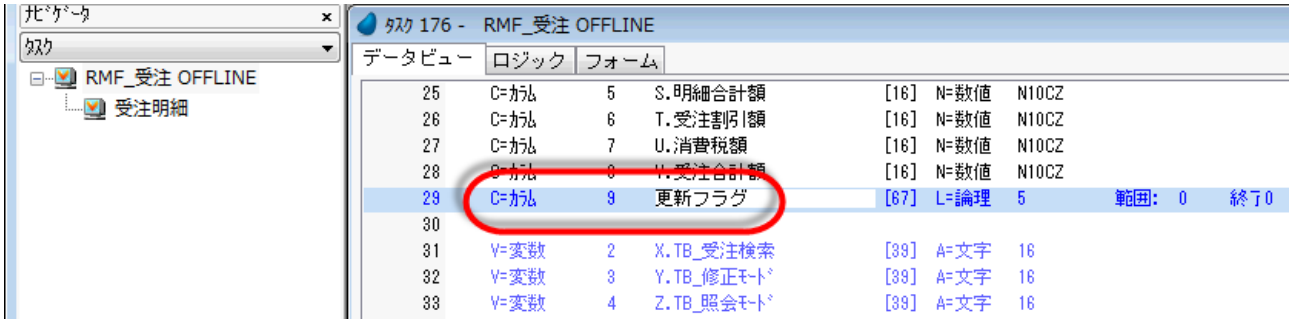
受注データ入力（登録モード）の場合には、新しい受注番号を発番します。このときには、前節で説明したように、ローカルの「LCL 制御テーブル」にある「最終受注番号」をもとに、仮の受注番号を発番します。



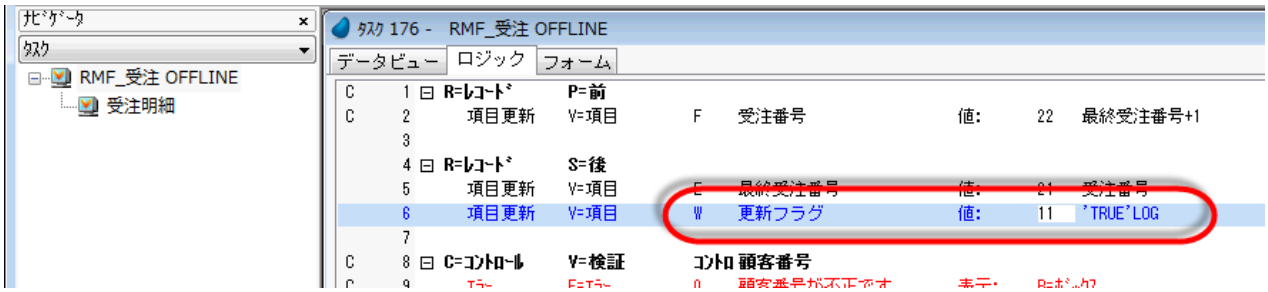
7.6.4 更新フラグの設定

「更新フラグ」カラムは、アップロードの対象となるかを区別するために、受注ヘッダテーブル（ローカル、サーバ、サーバ側一時テーブルの3本）に新規に追加したカラムです。データに変更を加えたときに、「更新フラグ」を TRUE に設定します。

まず、データビューに「更新フラグ」カラムを追加します。



レコード後処理で、TRUE に設定します。レコード後処理はレコードに修正があった場合（登録の場合も含め）だけ実行されるので、データ修正の加えられたレコードだけ、更新フラグが TRUE になります。



7.6.5 顧客マスタへの更新処理は不要

オンラインの受注入力プログラムでは、受注ヘッダのレコード後処理で、顧客マスタへの「受注回数」および「受注合計額」をリアルタイムで更新していました。

しかし、オフラインでの受注入力プログラムでは、顧客マスタへの更新をデータアップロード時に行います。クライアント上での受注データ入力時には行いません。

従って、レコード後処理での、顧客マスタへの更新処理は削除します。

7.6.6 印刷ロジックは実現できない

オンラインの受注入力プログラムでは、印刷ボタンがあって、これを押すと印刷プログラムが起動し、印刷結果が出力されました。

RIA オフラインプログラムでは、印刷機能（バッチ実行）がありません。従って、この機能は実装できないので、プッシュボタンおよびハンドラのロジックを削除します。

7.6.7 明細タスクでは、商品マスタへの更新処理が不要

受注ヘッダのタスクのレコード後処理で、顧客マスタへの更新処理は行わないようにしました。

全く同じ理由により、明細のタスクのレコード後処理で、商品マスタへの更新処理を行う必要はありませんので、削除します。

7.7 受注データのアップロードはどのように行いますか？

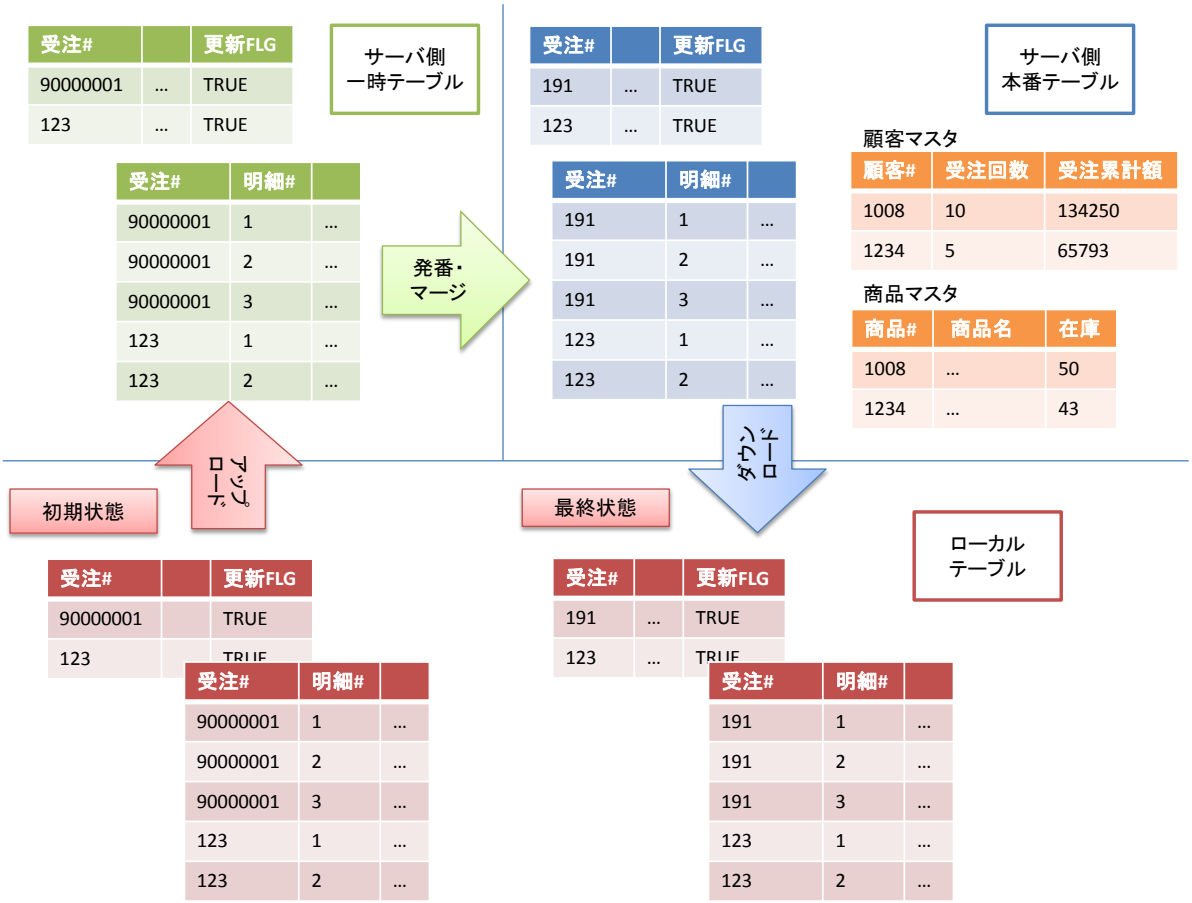
受注データのアップロードは、いったん一時テーブルにアップロードしてから、本番テーブルにコピーするようにします。その間に種々の付随処理を行います。

7.7.1 アップロード処理の概要

前述のように、オフライン状態でクライアントデバイス上に入力された受注データはまだ仮のものであり、本番のサーバ側テーブルに登録されて、始めて正式なものになります。そして、そのアップロード処理に伴って、様々な付随処理を行う必要があります。

受注データのアップロード処理は、大きく分けて次の3段階からなります。

1. アップロード: ローカルデータベースに蓄積された受注データを、サーバ側の一時テーブルにアップロードします。
2. 発番・マージ: サーバ側の一時テーブルのデータを、本番受注データにマージします。新規受注レコードの場合には受注番号を発番します。この処理の中で、顧客マスタや商品マスタへのデータ更新も行います。
3. 同期: サーバ側でマージされた最新の受注データを、ローカルテーブルにダウンロードして、データの同期をとります。

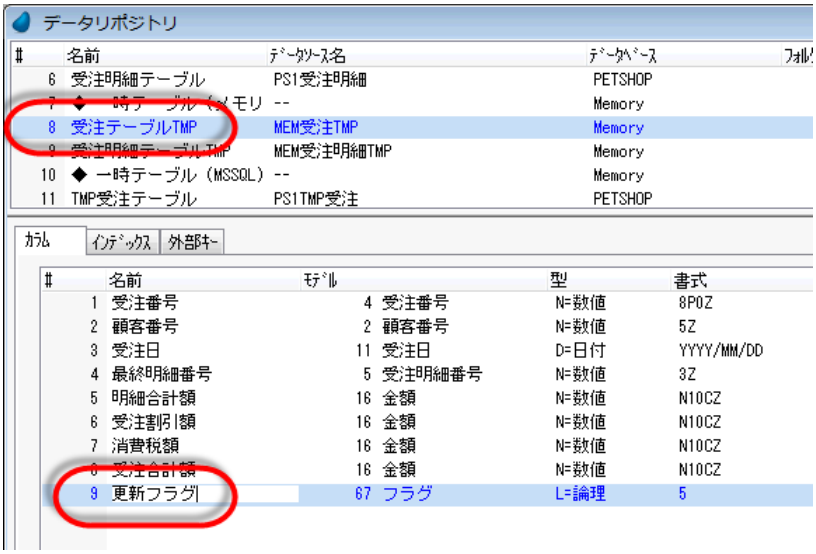


7.7.2 一時テーブルの定義

プログラム 52 番「OM_受注③」は、オンライン受注プログラムの一つで、一時テーブルを利用します。この一時テーブルは、データソース#8 番「受注テーブル TMP」、およびデータソース#9 番「受注明細テーブル TMP」として、いずれも Memory テーブルとして定義されています。

データのアップロードの際にも、この一時テーブルを流用するようにします。

但し、受注ヘッダには「更新フラグ」カラムを追加したので、一時テーブルにも同じカラムを追加します。

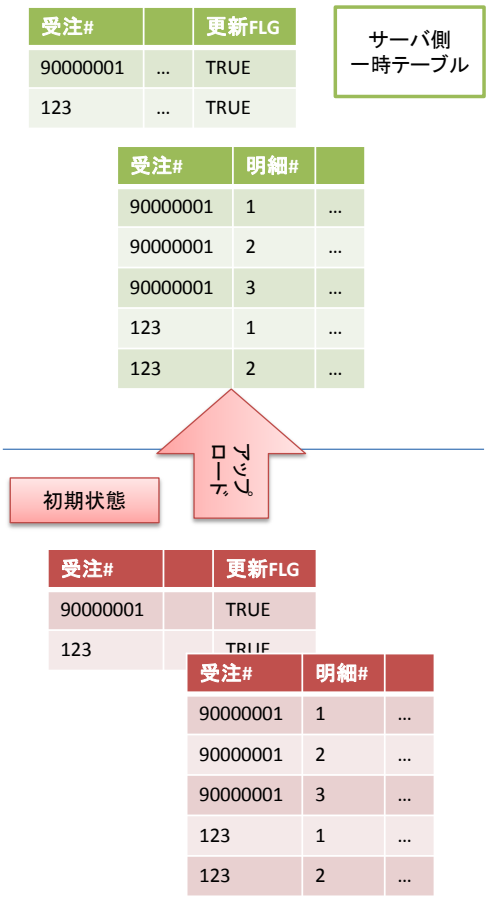


7.8 受注データのアップロードはどのように行いますか？

DataViewToDataSource() を使います。変更のあったレコードのみを対象とするため、「更新フラグ」で範囲指定を行います。

7.8.1 アップロード処理の概要

1. 一時テーブルの全レコードを削除します。
2. ローカル データベースの受注データを、サーバ側の一時テーブルにアップロードします。これは、DataViewToDataSource() 関数などを使って行います。
   アップロードデータの量にもよりますが、一般的な使い方では、一時テーブルとしてメモリテーブルで十分でしょう。
   このとき、変更のあったレコードだけをアップロードするので、「更新フラグ」で範囲づけします。
3. 同様に、受注明細データを DataViewToData-Source() 関数を使って一時テーブルへとアップロードします。
   このときも同様に、変更のあったデータだけをアップするのですが、受注明細レコードには「更新フラグ」がありません。このため、受注番号で「受注テーブル」をリンクし、受注テーブルの「更新フラグ」で判断します。



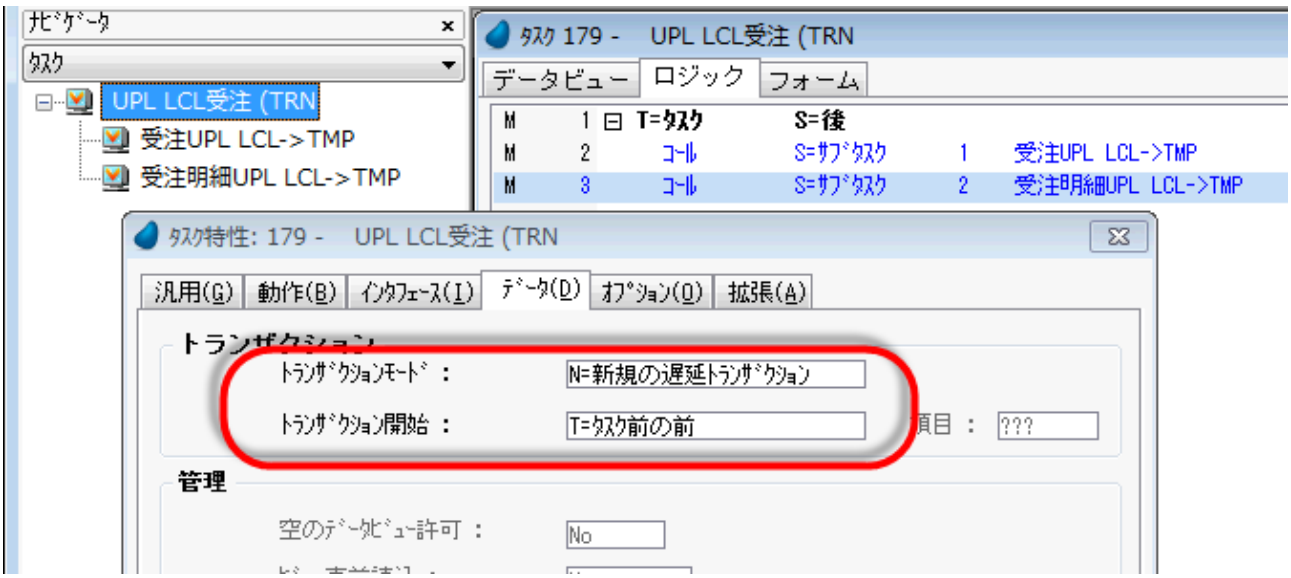
7.8.2 一時テーブルのデータ削除

一時テーブルの削除は、通常通り、DbDel() 関数で行います。これをヘッダ、明細それぞれに実行します。

7.8.3 一時テーブルへの受注データアップロードプログラム

ローカルの受注データ(ヘッダ、明細とも)を、サーバの一時テーブルにアップロードするために、プログラム「UPL LCL 受注(TRN)」を作ります。ここでは、データアップロードに DataViewToDataSource() 関数を使います。これをヘッダ、明細両方のテーブルに対して行います。

プログラムの構造は、次の図のようになります。



二つのサブタスクは、それぞれ、ローカルの受注テーブル、ローカルの受注明細テーブルを、サーバの一時テーブルへとアップロードします。

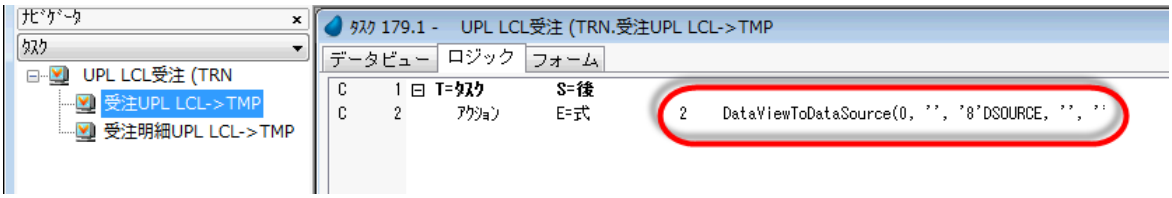
親タスクはサブタスクをコールするだけです。

アップロードの処理は一括して行う必要があるので、親タスクでタスクレベルのトランザクションを設定しておきます。

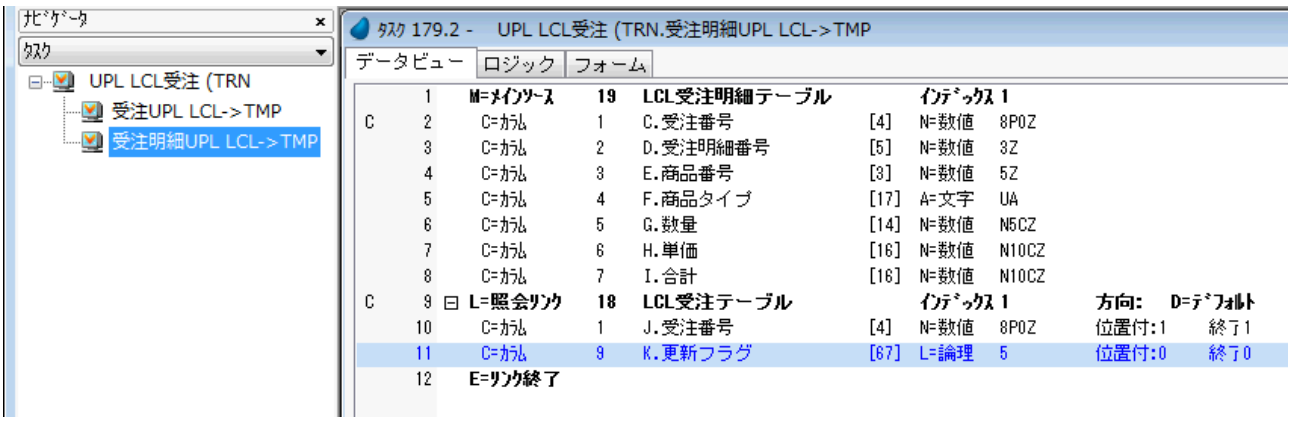
サブタスク1の「受注 UPL LCL->TMP」では、メインソースとしてローカルの「LCL 受注テーブル」を指定し、データビューを展開します。

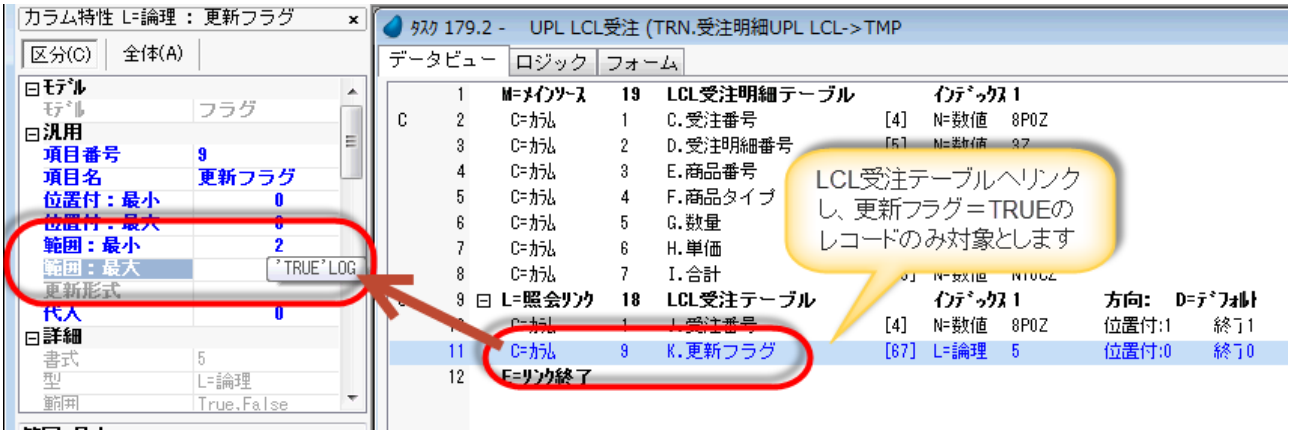
データビューへの範囲指定として、「更新フラグ」＝TRUE のものだけを選択するようにします。

タスク後処理で、DataViewToDataSource() 関数で、データソース#8（受注テーブル TMP）にアップロードします。



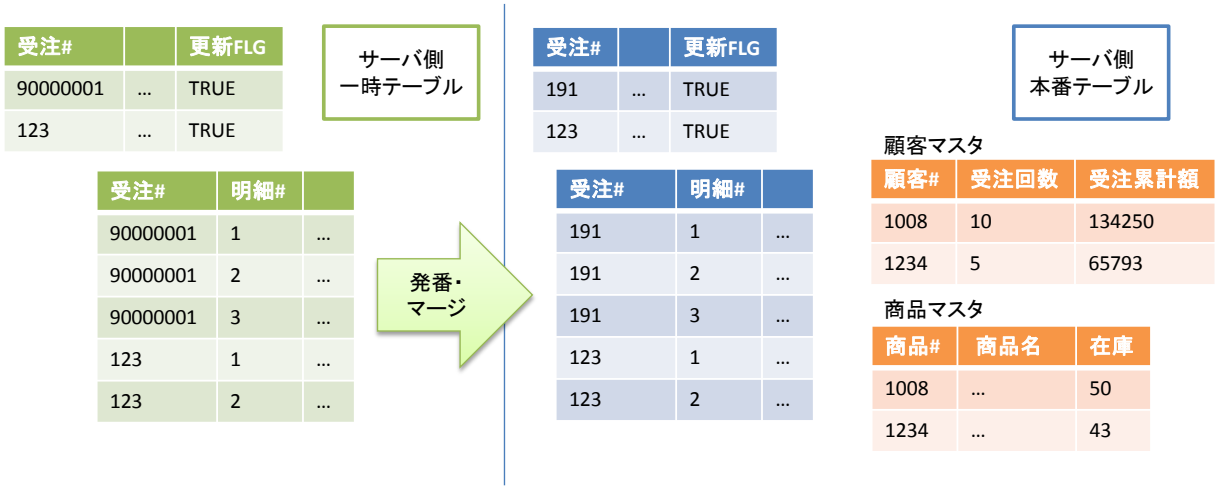
サブタスク2の「受注明細 UPL LCL->TMP」は明細レコードをアップロードするもので、プログラムの作りは同様ですが、明細レコードには「更新フラグ」のカラムがないので、ヘッダレコードにリンクをはり、ヘッダレコードの「更新カラム」が TRUE であるもの、という範囲指定を行います。


ﾅﾋﾞｹﾞｰﾀ
ﾀｽｸ
UPL LCL受注 (TRN
受注UPL LCL->TMP
受注明細UPL LCL->TMP
ﾀｽｸ 179.2 - UPL LCL受注 (TRN.受注明細UPL LCL->TMP
データビュー
ロジック
フォーム
1 M=ﾒｲﾝｿｰｽ 19 LCL受注明細テーブル ｲﾝﾃﾞｯｸｽ 1
C 2 C=ｶﾗﾑ 1 C.受注番号 [4] N=数値 8P0Z
3 C=ｶﾗﾑ 2 D.受注明細番号 [5] N=数値 3Z
4 C=ｶﾗﾑ 3 E.商品番号 [3] N=数値 5Z
5 C=ｶﾗﾑ 4 F.商品タイプ [17] A=文字 UA
6 C=ｶﾗﾑ 5 G.数量 [14] N=数値 N5CZ
7 C=ｶﾗﾑ 6 H.単価 [16] N=数値 N10CZ
8 C=ｶﾗﾑ 7 I.合計 [16] N=数値 N10CZ
C 9 L=照会ﾘﾝｸ 18 LCL受注テーブル ｲﾝﾃﾞｯｸｽ 1 方向: D=ﾃﾞﾌｫﾙﾄ
10 C=ｶﾗﾑ 1 J.受注番号 [4] N=数値 8P0Z 位置付:1 終了1
11 C=ｶﾗﾑ 9 K.更新フラグ [67] L=論理 5 位置付:0 終了0
12 E=ﾘﾝｸ終了


カラム特性 L=論理 : 更新フラグ
区分(C)
全体(A)
モデル
ﾓﾃﾞﾙ フラグ
汎用
項目番号 9
項目名 更新フラグ
位置付：最小 0
位置付：最大 0
範囲：最小 2
範囲：最大 'TRUE'LOG
更新形式
代入 0
詳細
書式 5
型 L=論理
範囲 True,False
ﾀｽｸ 179.2 - UPL LCL受注 (TRN.受注明細UPL LCL->TMP
データビュー
ロジック
フォーム
LCL受注テーブルへリンクし、更新フラグ＝TRUEのレコードのみ対象とします
C=ｶﾗﾑ 9 K.更新フラグ [67] L=論理 5 位置付:0 終了0
E=ﾘﾝｸ終了

7.9 受注データの発番とマージはどのように行いますか？

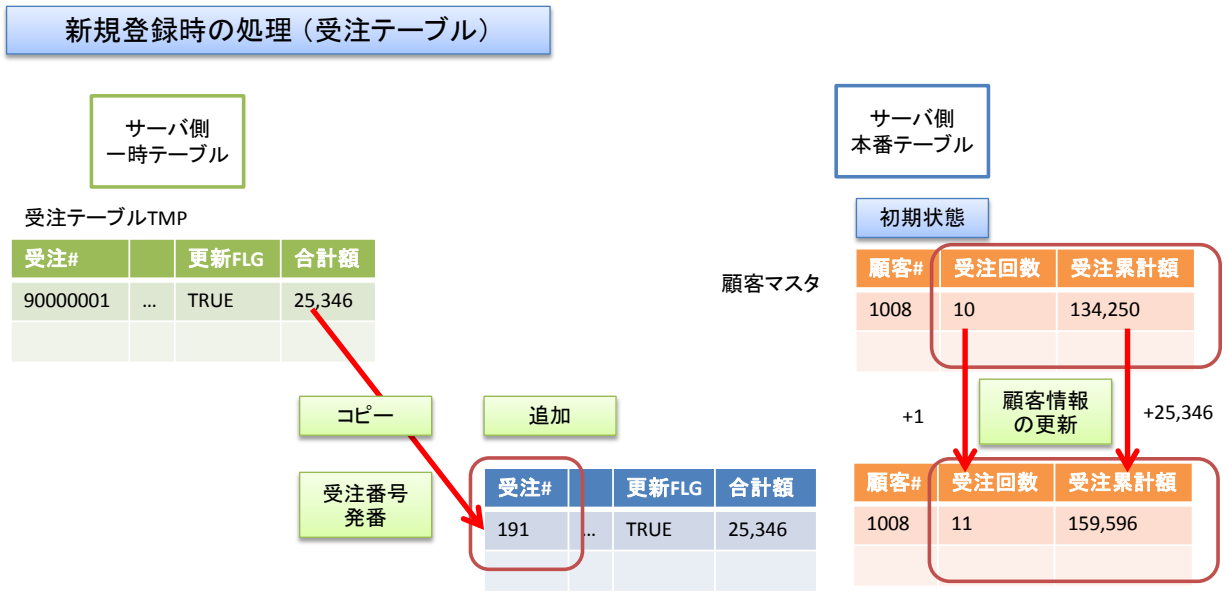
新規登録の場合と、既存のデータ更新の場合で異なります。

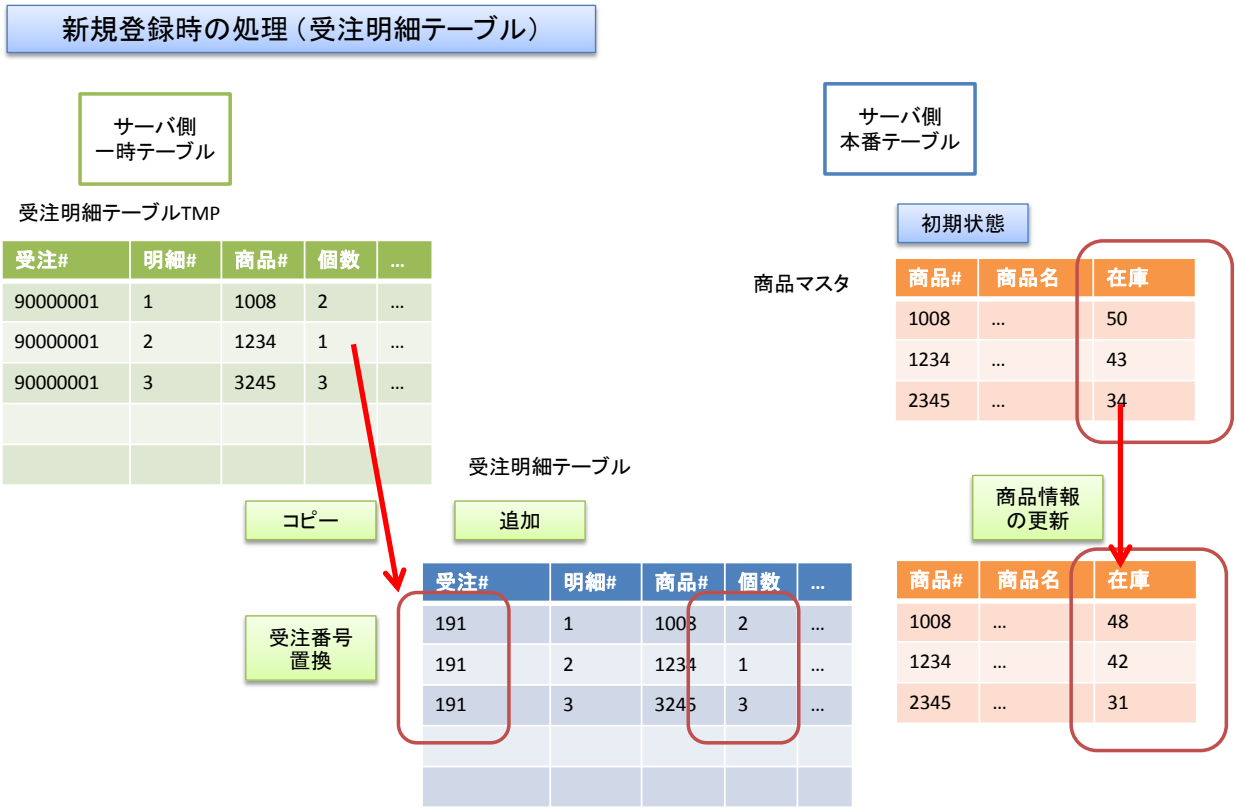


実際のアプリケーションでは、アップロードしたデータのチェックが必要でしょう。例えば、在庫数の確認や、顧客の与信管理などです。このチェックは、アップロードされた一時ファイル中のレコードを使って、本番テーブルへのマージを行う前に行うのが良いでしょう。
一時テーブルも本番テーブルもすべてサーバ上にあるので、チェックプログラムはバッチタスクとして実装して、高速に処理できます。

7.9.1 新規データを登録する場合の処理:

1. 新規登録の場合には、正式受注番号を発番し、一時テーブル上のデータの受注番号を更新します。こととときに、ついでに「更新フラグ」もクリアしておきましょう。
2. 明細レコードをもとにして、商品ごとに、商品マスタの「在庫数」を調整します。また、受注ヘッダレコードをもとにして、顧客マスタの「取引回数」と「受注累計額」を更新します。
3. 一時テーブルのレコードを、本番の受注テーブル、受注明細テーブルに登録します。



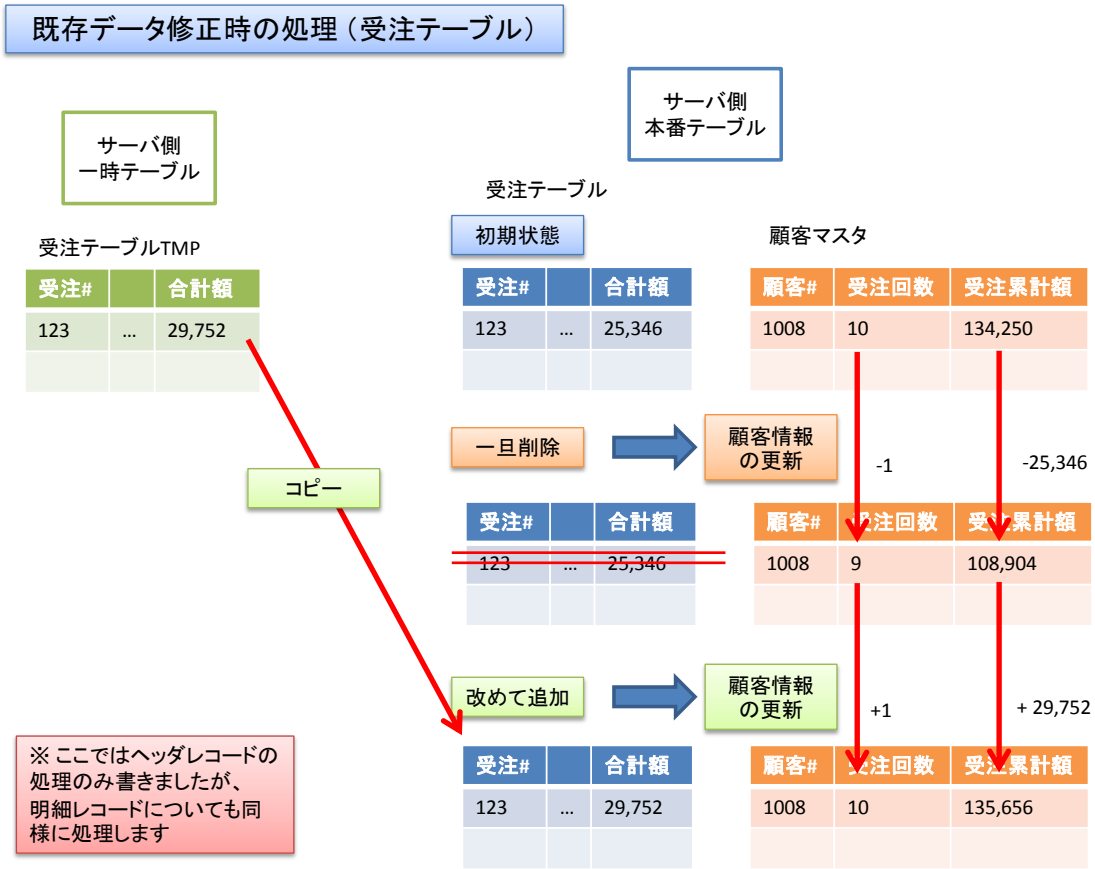


7.9.2 既存データを修正する場合の処理:

既存データを修正する場合には、

1. 一旦、既存データを削除する。このときに、同時に顧客マスタや商品マスタの調整を行う。
2. その後、改めて修正後のデータを登録する。これは、上述の新規登録時の処理と同じ。

という2段階を踏んで行います。



以上で本番テーブルへの登録は終わり、受注は確定しました。

しかし、ローカル テーブルの方のデータは更新されていません。具体的には、受注番号が新しく発番されましたが、ローカルテーブル上ではまだ仮の受注番号のままです。そこで、次にはサーバで更新したデータを、クライアントにダウンロードして、最終的にデータを同期させることが必要になります。これは次節で説明します。

7.10 受注データアップロード後の同期はどのように行いますか？

7.2 「受注データのダウンロード」で使った「受注選択 TMP」テーブルとダウンロードプログラムを使って、データのダウンロードを行います。

アップロード後にはサーバデータとローカルデータとに乖離があるので、同期処理を行う必要があります。その理由は、次のようなものです。

1. 新規データは仮番号から正式番号が発番されているので、正式番号でのデータを取得する必要がある。
2. 既存データをクライアント側で更新した場合、アップロード後にサーバ側でも付随処理が発生してデータ変更が行われた可能性があり、ローカルにあるデータと異なる可能性がある。
3. チェックロジックにより新規登録や既存データの変更が拒否された場合には、ローカルデータとサーバデータに乖離が起こる。

上の2や3のケースは、本書で説明した単純なアップロード・ダウンロードでは発生しないのですが、一般的なアプリケーションでは起こる可能性がありましょう。

データの同期は、基本的にはサーバからローカルでのダウンロードになりますので、7.2 「受注データのダウンロード」で行った手法をそのまま利用することができます。

そこでは、受注データをダウンロードするために、「受注選択 TMP」テーブルを使い、このテーブルにダウンロード対象となる受注番号を記録しておきました。今回もこのテーブルにダウンロード対象とする受注番号を記録しておくことにします。

7.10.1 今回のアップロードで変更のあった受注番号を特定します。

最初にダウンロード対象となる受注番号を特定し、「受注選択 TMP」テーブルに記録しておく必要があります。このような受注番号には、次の 2 種類があります。

- 新規登録された受注データの受注番号: 新規登録された受注データは、ローカルでは仮番号で振られており、サーバで正式番号が発番されます。従って、正式発番を行った時点で、「受注選択 TMP」テーブルに登録しておきます。
  この処理は、7.9.1 「新規データを登録する場合の処理:」の中で、追加しておきます。
- 既存データに修正を行った受注番号: 既存データに修正を行った受注番号。これは 7.9.2 「既存データを修正する場合の処理:」の処理の一環として、行うようにします。

7.10.2 ローカルの対象データを一旦削除

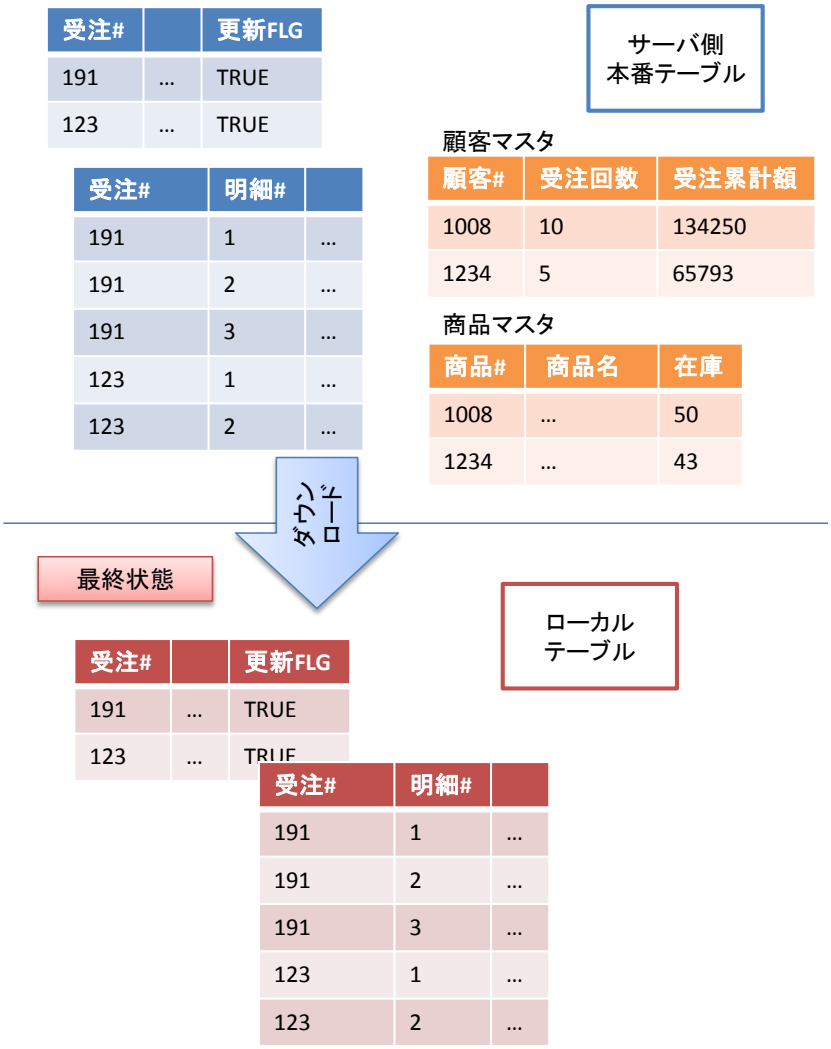
次に、サーバデータをローカルデータにマージします。マージは一旦ローカルの対象データを削除した後に、サーバからダウンロードする、という方法をとります。

削除対象となるローカルデータの受注番号は、アップロードを行った受注番号と同じになります。これはローカルテーブルで「更新フラグ」が TRUE になっているものなので、これを目印にしてレコードの削除を行います。当然、受注ヘッダレコードを削除する場合には、それに付随する明細レコードも一緒に削除しなければなりません。

ただし、サーバ側のチェックなどで正しく登録・更新ができなかったものは除外しなければなりません。厳密にしようとすると、面倒な処理となります。本書ではそこまで細かな処理については省略します。

7.10.3 受注データのダウンロード

以上でダウンロードする準備が整いましたので、7.2「受注データのダウンロード」で使ったのと同じプログラムを使って、ダウンロードを行います。



受注#		更新FLG
191	…	TRUE
123	…	TRUE

受注#	明細#	
191	1	…
191	2	…
191	3	…
123	1	…
123	2	…

顧客#	受注回数	受注累計額
1008	10	134250
1234	5	65793

商品#	商品名	在庫
1008	…	50
1234	…	43

受注#		更新FLG
191	…	TRUE
123	…	TRUE

受注#	明細#	
191	1	…
191	2	…
191	3	…
123	1	…
123	2	…

7.11 例外状況への対応

さて、すべての処理がつつがなく終了すれば、以上のアルゴリズムで正しくアップロードができます。

しかしながら、実際にはさまざまな例外処理に対応しなければなりません。例を挙げれば、次のようなことが考えられます。

データの論理的エラー:

上のステップ3で在庫数の確認をしていますが、ここで在庫不足が発覚した場合には、処理を中断して、クライアント側にエラーの旨を通知する必要があります。

もし受注データが複数あったならば、ステップ3で在庫不足が見つかったときに、それ以外の受注データはどうするかを決めておかなければなりません。

もし受注が一連のものであり、一部だけ成功し一部は失敗する、というようなことができないのであれば、全体の処理を中断することになります。

一方、受注データがそれぞれ独立しているのであれば、在庫不足となった受注についてだけ、処理を中断し、そのほかのデータはそのまま処理を続ける、ということが考えられます。

この場合には、成功した受注データと、失敗したものとをどこかで記録しておく必要があり、クライアント側では、その情報をもとにして、適切にユーザに状況を報告しなければなりません。これは、一時テーブルに「失敗」フラグと「失敗理由」カラムとを設けて、失敗したレコードにはフラグを立て、理由を何等かの形(ステータスコード、あるいは文字列)で記録しておけば良いでしょう。

また、ステップ8において、仮受注データを削除しますが、失敗したものは削除してはいけません。データはそのままにしておいて、ユーザに判断(発注数を減らすか、発注自体をキャンセルするか)を仰がなければなりません。

通信接続エラー:

これは「4.3 アップロード失敗に対する考慮が必要ですか?」で説明したことと本質的には同じことですが、受注データは処理がより複雑なので、確実にするための難度が高くなりましょう。

8 データベースの一括転送

大量のデータのコピーが必要な場合、データベース全体を一括してダウンロードする技法もあります。効率は良い方法ですが制限もあるので、初期化時などに使うと便利な方法です。

8.1 データの一括ダウンロードとは？

ローカル データベースは SQLite ファイルであることを利用し、サーバで作成した SQLite ファイルをクライアントにダウンロードします。

ローカルデータベースの実体は、SQLite ファイルです。これは個人用 DBMS 向けの簡易な SQL データベースです。SQLite ファイルのフォーマットは公開されており、プラットフォームが異なっても互換性があります。従って、Windows 上で作った SQLite ファイルは、Windows で使えるのはもちろん、Android や iOS 上でもそのまま使うことができます。
この性質を利用して、RIA クライアントと RIA サーバ間の大量のデータ転送を効率よく実行することができます。

この方法では、だいたい次のような手順になります。

1. サーバ側で、必要なテーブルをすべて SQLite ファイル上に作成（コピー）。
2. その SQLite ファイルをサーバからクライアントにコピー。ServerFileToClient 関数を使う。
3. クライアント側（オフライン RIA タスク）で、このファイルにアクセス。

注意事項: この手法を使う場合には、次のような点に注意する必要があります。

① SQLite ファイルをオープンしている状態では、コピーを実行することができません。Magic がつかんでいる SQLite をクローズするためには、DBDiscnct 関数を使って、データベースとの接続をクローズする必要があります。これは、サーバ側、ローカル側両方に必要となります。

② ローカルのファイルは上書きされるので、既存のローカルデータはすべて失われます。

③ サーバ側とローカル側の SQLite のファイル名を同一にしておく必要があります。これは、「データベース」テーブルの「位置」特性で、同一の名前を指定することになります。

④ SQLite テーブルの一般的制限として、複数ユーザでの共有はできません。サーバ側では複数のクライアントが接続し、同時接続する可能性があるので、SQLite ファイルを作成するにあたっては、一時には一人のユーザだけしかアクセスされないように、プログラムの排他制御を行う必要があります。

9 参考情報

ここでは、本文中に説明しなかった、ローカル データベースに関する情報を集めました。

ここに示した情報は、Magic xpa 2.4c におけるものです。Magic のバージョンアップにともない、制限事項が緩和されることがありますので、正確なところはリファレンスヘルプや README などにより確認してください。

9.1 ホワイトペーパ > オフラインアプリケーションの開発 > ローカル(オフライン)ストレージ

製品添付のホワイトペーパ「オフラインアプリケーションの開発 」にローカル ストレージについての記述がありますので、転載します。

9.1.1 ローカル(オフライン)ストレージ

RIA クライアントは、ローカルデータソースをサポートします。ローカルデータソースを使用することで、サーバにほとんどアクセスしないオフラインプログラムでクライアントにデータを保存したり使用することができます。

ローカルストレージを使用することで、データをローカルのキャッシュに格納してパフォーマンスを向上させることができます。サーバデータのサブセットをローカルに保存して、後でサーバと同期することで、オフラインの処理を完結させることができます。

9.1.2 ローカルデータベース定義

ローカルデータベースは、たとえば、DBMS が MS-SQL Server または Memory に設定される場合と同じように)[データベース]テーブルの[DBMS]カラムで「Local」と指定することで定義することができます。

「Local」DBMS のデータベースを使用したデータソースは、クライアントのデータソースで、クライアントのキャッシュに保存されます。

9.1.3 ローカルデータベースとデータソースの制限

ローカルデータベースとデータソースには以下の制限があります。

- ローカルデータベースは、ダイレクト SQL ステートメントで使用することができません。
- ローカルデータソースは、リッチクライアントタスクでのみ使用できます(オンライン、バッチの各タスクで使用することはできません)。
- ローカルデータソースのトランザクションは、物理トランザクションとして動作します。Local データソースを含むすべてのタスクは、同じトランザクション内にあります。
- ローカルデータのコミットは、トランザクションをオープンしたタスク従って実行されます。トランザクションが並行タスクでも開始されている場合、これらのタスクのデータもコミットされます。
- ローカルデータソースを持つタスクをロールバックすると、以下もロールバックされます
  - ローカルデータソースを持つすべてのタスク。
  - 親タスクの遅延トランザクション。
- ローカルデータソースは、[ツリー]コントロールが定義されたタスクで使用することができません。
- タスクには、サーバとローカルのデータソースを混在させることができません。

以下の機能は、ローカルデータソースではサポートされません。

- ユニークインデックスまたは RowID SQL ポジションの値
- ローカルデータソース間の[結合リンク]コマンド
- [コール]処理コマンドの[同期]特性
- 位置の範囲

- SQL 範囲
- インデックスの最適化
- ユーザソートと Magic ソート
- 遅延トランザクション
- RangeAdd()、LocateAdd()と SortAdd()関数を使用した範囲、位置付、ソート処理
- 定義取得
- [データベース特性]の[開発モードでのテーブル変換]特性
- [エラー]ロジックユニット …… ローカルデータソースのエラーに対応するエラー処理を定義することはできません。
- チャンクサイズ

9.1.4 ローカルデータソースのデータベース構造を変更する

ローカルデータソースの構造（例えば、フィールドの削除や、書式の修正、フィールドの追加）が変更された場合、プロジェクトのメタデータがクライアントと同期した後で、クライアントのローカルデータソースに自動的に反映されます。

既存のローカルデータは、可能な限り新しい構造のデータソースに自動的に変換されます。

ローカルデータソースの構造の変更が、幾つかのデータ（例えば、ユニークなインデックスの追加や文字列フィールドを数値に変更した場合）を無視した場合、このデータは失われます。したがって、ローカルデータを必要とする場合、そのような変更は避けることを推奨します。

古い構造から新しいものへの変換が、データソースの ISN に基づいて行われることに注意すべきです。これは、ローカルデータソースが Studio 環境で名前を変更した場合、それがローカルデータベースにも反映されることを意味します。最初の Local データソースが変更されないままにしたい場合、新しい構造に対応した新しいデータソースを作成する必要があります。

9.2 リファレンスガイド > Magic xpa で使用するデータベース > Local データベース

リファレンスヘルプの「Magic xpa で使用するデータベース」>「Local データベース」のページに、ローカル データベースについての記述があるので、転載します。

9.2.1 Local データベース

local データベースは、RIA クライアントでローカルクライアントにデータを保存、アクセスする場合に使用することができます。このデータベースを使用することで、サーバにアクセスしないオフラインプログラムで、クライアント側にデータを保存したり、利用したりすることができるローカルデータソースを定義することができます。

ローカルストレージを使用することで、データをローカルのキャッシュに格納することができ、パフォーマンスを向上させることもできます。

以下は、Local データベースで動作する上での関連情報です。

埋め込み SQL	Local データベースは、埋め込み SQL のステートメントで使用することができません。
トランザクション	ローカル・データソースに対するトランザクションは、物理トランザクションとして動作します。ローカル・データソースを含むすべてのタスクは、同じトランザクション内で動作します。
コミット	ローカルデータのコミットは、トランザクションをオープンしたタスクによって行われます。トランザクションが兄弟関係のタスクでも開始される場合、これらのタスクのデータもコミットされます。
ロールバック	ローカル・データソースを持つタスクをロールバックすると、以下の内容がロールバックされます。 ローカル・データソースを持つすべてのタスク 親タスクの遅延トランザクション

9.3 Magic xpa 2.4c README

Magic xpa にオフライン RIA 機能がサポートされるようになった 2.4c の README に、ローカル データベースも含めた、オフライン RIA 一般についての説明がありますので、転載します。

リッチクライアント

9.3.1 オフライン機能とクライアント側のストレージ

Magic xpa は、サーバに接続しない場合でも RIA のクライアントアプリケーションを実行させることができるようになりました。そして、クライアントのローカルにデータベース情報を保存できるようになりました。Windows 用と、iOS と Android デバイス用に非オフライン／オフラインの両方に対応したアプリケーションを開発することができるようになりました。オフラインアプリケーションは、インターネットへの接続が断続的であったり、制限があったり、利用できないような場所でも操作を継続させることができるようになります。オフラインで動作している間、データは Local データベースに保存されます。そして、インターネットへの接続が再開された場合にサーバのデータと同期させることができます。

オフラインプログラムとローカル・ストレージを使用すると、通常使用されるデータ（例えば選択プログラムで使用されるもの）をローカルにキャッシュすることによってパフォーマンスを向上させることもできるため、データを取得するためにサーバにアクセスする必要がなくなります。

この新しい機能の利便性を最大限に利用するには、ホワイトペーパーの『オフラインアプリケーションの開発』を参照していただく事を推奨します。

既存の Magic xpa RIA Server のライセンス（MGRIA12）では、オフライン機能は利用できません。

9.3.2 アプリケーションの起動

Magic xpa がオフライン機能をサポートしたことにより、サーバが利用できない時にリッチクライアントアプリケーションを実行すると、アプリケーションはオープンされたままになり、自動的には終了しなくなります。

そのような状況でアプリケーションを閉じたい場合、[無効サーバ]イベントに対応するイベントロジックユニットをメインプログラムに追加して、このロジックユニット内で[終了]イベントを発行することで実現できます。

9.3.3 関数の追加

オフライン処理用に以下の関数が追加されました。

- ClientDBDiscnt() …… ローカルデータベースとの接続をクローズします。ServerFileToClient()関数を使用してローカルデータベースをサーバからコピーする際に必要になります。
- ClientDBDel() …… ローカルデータベース上でデータソースを削除します。サーバからクライアントへデータソースの全てのデータをコピーする際に必要になります。
- DataviewToDataSource() ……サーバ側のデータソースとローカル・データソースの間でデータをコピーする処理のパフォーマンスを向上させるために追加されました。

9.3.4 ネットワークが利用できる場合もオフラインモードで実行

ネットワーク接続が利用できる場合も、サーバに接続することなくクライアント側でアプリケーションを開始するための設定が追加されました。サーバへの接続がないため、このオプションを利用することで起動時のパフォーマンスが改善されます。このモードは、RIA の実行特性ファイル（execution.properties）で、"ConnectOnStartup"の値を'N'に設定することで有効になります。

9.3.5 非インタラクティブタスクでの削除モード

非インタラクティブなリッチクライアントタスクの[初期モード]特性で「削除」をサポートするようになりました。

このモードは、特にクライアント側のストレージ・データソースからレコードを削除する場合に利用できます。

9.3.6 ネットワークエラーでの再実行用ダイアログボックス表示の無効化

ClientSessionSet()という名前の関数が追加され、現在のセッションパラメータを制御することができるようになりました。

この関数は、現在、"EnableCommunicationDialogs"というキーに対して True/False の値をサポートします。"False" が設定された場合、ネットワークエラーが発生しても[リトライ]ダイアログを表示しなくなります。

アプリケーションがネットワークエラー(例えば、オフラインプログラムやクライアント側ストレージを使用している場合)を処理するロジックを定義している場合、この動作が利用できます。

9.3.7 HTTPRequestTimeout

[MAGIC_RIA]セクションに"HttpRequestTimeout"という名前のパラメータが追加され、RIA クライアントが待つ時間を定義することができます。

デフォルト値は、5 秒になります。

以前の HTTP タイムアウトの設定([動作環境／外部参照])は、HTTPGet()や HTTPPost()、HTTPCall()の各関数の動作のみに影響します。

この変更のために、ClientNetworkRecoveryInterval の設定は、サポートされなくなる予定です。

9.3.8 FirstHTTPRequestTimeout

execution.properties に FirstHTTPRequestTimeout というパラメータが追加されました。これは、RIA クライアントが起動時(最初のリクエスト)にサーバへの接続を試みる時間を定義します。

デフォルト値は 2 秒で、アプリケーションを速く読み込ませることができます。遅いネットワーク上で実行する非オフラインが可能なアプリケーションの場合、このキーの値を高くする(例えば、5 秒)ことで通信上の失敗の影響を最小化することができます。

9.3.9 変数項目でのリンクの動作の変更

リッチクライアントタスクでは、[代入]特性の設定がない変数項目の値が変更されると、その値は全てのレコードで表示されます。しかし、更新された変数項目に基づくリンク処理では全てのレコードは再表示されませんでした。

この動作は変更され、変数項目の値が変更されるとリンク項目は表示され、[代入]特性が定義されていない変数項目だけでリンクしている場合、リンク項目とリンク成功フラグは同様に全てのレコードで更新されます。

9.3.10 起動時のスプラッシュ画面

リッチクライアント・アプリケーションが起動されるとき、スプラッシュ画面を表示させることができるようになりました。

Magic xpa Studio では、RIAModules¥Desktop¥Resources フォルダ内の startup.png ファイルを差し替えることで変更することができます。.

リッチクライアントインタフェースビルダに表示させたいスプラッシュ画面を選択するオプションが追加されました。この設定が実行時に反映されます。

PNG ファイルのみをサポートします。

9.3.11 キーボードのバッファリング

［MAGIC_SPECIALS］セクションに SpecialKeyboardBuffering というパラメータが追加されました。

このパラメータに'Y'を設定することで、すぐに処理できない状態でキー入力された場合、コントロールに送るキーストロークをバッファリングすることができます。これは、リッチクライアントタスクで修正可能な［エディット］コントロール上にパークする場合のみに関連します。

9.3.12 前バージョンのクライアントを使用した場合の非互換性

オフラインプログラムのサポートに対応する変更のために、Magic xpa の前のバージョンから、Windows やモバイルクライアントで、Ver2.4 のサーバに接続させることができなくなりました。

このバージョンのサーバを使用するには、クライアントを同じバージョンにアップグレードする必要があります。

9.3.13 ログレベルの追加

MRB やインターネットリクエスタ、RIA クライアント、Magic xpa サーバエンジンに対するログレベルに B/BASIC が追加されました。

このログレベルは、HTTP リクエストの送信と着信に関してのみ記録するものです。このレベルは、製品（そしてテスト)システム、特に通信系とコンポーネント処理に関するトラブルシューティングに対する使用を想定したものです。

9.3.14 ServerFileToClient()関数でフォルダとワイルドカードをサポート

ServerFileToClient()関数は、ファイル名の指定の際にワイルドカード（例えば、"c:¥images¥image*.jpg"）とフォルダ（例えば"c:¥images"）を指定することができるようになりました。複数のファイルやフォルダを指定して ServerFileToClient()を実行することで、プログラミングや実行時のパフォーマンスを改善します。

9.4 Magic xpa 2.5b README

リッチクライアント機能

9.4.1 RIA 内部バージョンのアップデート

ClickOnce を使用しない場合の RIA クライアントの自動アップデートの内蔵メカニズムはサポートされなくなりました。

9.4.2 ローカルデータソースでの範囲、位置付とソートの各関数をサポート

RangeAdd()、RangeReset()、LocateAdd()、LocateReset()、SortAdd()と SortReset()の各関数が、ローカルデータソース上でも使用できるようになりました。

9.4.3 クライアント-サーバとの互換性

クライアントモジュールのサーバとの間の内部通信プロトコルが変更されました。この結果、Ver2.5 以前のクライアント(例、Ver2.4)は、Magic xpa 2.5 のサーバを利用することができず、Magic xpa 2.5 クライアントは、Ver2.5 以前のサーバを使用することができません。

9.4.4 DataViewToDataSource()の動作変更

DataViewToDataSource()関数の動作が変更され、同じ型と保存タイプを持つカラムの間でのみデータをコピーするようになりました。

この変更は、関数のパフォーマンスを向上させるために行われました。

9.4.5 モバイルデバイスでのフローティングライセンス

フローティングライセンスは、モバイルまたは Windows に限定されないライセンスとしてのみ使用できます。

ユーティリティ

9.4.6 環境設定ユーティリティ

動作環境を設定するユーティリティ(mgxpaSettings.exe)が追加されました。Ver2.4 の実行版の製品に添付されていた開発エンジン用のファイルは、このユーティリティに置き換わります。

モバイル

9.4.7 基本色定義 - アルファ値のサポート

基本色定義の画面で、透過色のアルファ値の定義をサポートするように拡張されました。基本色定義のアルファ値は、不透明度レベルを定義して半透過な色の作成を可能にします。

この定義は、Android と iOS クライアントでのみ有効です。

9.4.8 モバイル用の特性

フォームとコントロールには、モバイル関連の全ての特性を含む新しいセクションが追加されました。

これらの特性は、今まで[追加情報]特性に定義された機能に置き換わり、拡張されました。[追加情報]特性は、下位互換性を維持するためにサポートされます。

9.4.9 不透明度レベルのサポート

半透過色は、コントロールに定義して使用することができます。前述のように、定義は基本色定義画面でアルファ値を設定することで行うことができます。半透過色を使用することで、アプリケーション見栄えを拡張することができます。

9.4.10 サジェスチョンテキスト(ヒント)のサポート

新しい[ヒント]特性を使用することにより、[エディット]コントロールにヒントを定義することができるようになりました。ヒントは、入力している間に自動的に削除される[エディット]コントロール上で表示されるテキストです。

9.4.11 ナビゲーションドロワーのサポート

ネイティブなナビゲーションドロワーがサポートされ、フォームの[ナビゲーションドロワーメニュー]特性にメニューエントリを割り当てることで表示されるようになりました。

ナビゲーションドロワーは、画面の左側からスワイプ表示されるメニューです。

9.4.12 フォームアニメーションのサポート

[開始アニメーション]と[終了アニメーション]の各特性を使用することでフォームの開始／終了のアニメーションを制御することができるようになりました。

9.4.13 [タブ]コントロールのサポート

Magic の[タブ]コントロールを使用することで、ネイティブなタブバーがサポートされるようになりました。

Magic xpa の各タブは、モバイルデバイス上のタブページとみなされます。

9.4.14 2 ステートボタンのサポート

2 ステートのイメージボタン(押下/押下されない)が利用できるようになりました。

[チェックボックス]コントロールか[ラジオ]コントロールを使用して、これらの[イメージリストファイル名]特性に 6 ステータスのイメージを設定することによって、コントロールを定義することができます。

9.4.15 キーボードタイプの制御

新しく追加された[キーボードタイプ]特性を使用することで、コントロールを編集する時に開くキーボードタイプ(例、電子メールまたは URL 用のキーボード)を指定することができるようになりました。

9.4.16 キーボードのアクションボタンの制御

新しく追加された[キーボードリターンキー]特性を使用することにより、[Enter]ボタン上に表示されるテキストを定義することができるようになりました。

これは値(例えば、実行)を強制したり、特定の値(例えば、検索や移動)を設定することができます。

"実行"や"検索"または"移動"の値が押下されると、[選択]の内部イベントが発行されこのキーが押下されたときに実行されるロジックユニットを実行させることができます。

9.4.17 タイトルバーの色サポート

[タイトルバー色]特性を使用することにより、フォームのタイトルバーの色を変更することができるようになりました。

9.4.18 カラムの可視性のサポート

[カラム]コントロールの[可視]特性は、モバイルデバイスでもサポートされるようになりました。

9.4.19 編集中での処理コマンドの実行 – 動作の変更

変数項目が定義された[エディット]コントロールでデータを編集するとき、処理コマンドが実行される(例えば、[コントロール変更]ロジックユニット内に処理コマンドが定義されいて)場合に、ビジーを表すスピナーが表示されなくなりました。処理コマンドが実行される間も、文字入力を行うことができるようになりました。

9.4.20 ネイティブコードの実行

ClientNativeCodeExecute()と呼ばれる関数が追加され、ネイティブコード(Android 用 Java や iOS 用の Objective C)を呼び出すことができるようになりました。

Ver2.4 までは、ネイティブコードの呼び出しは、UserDefinedFunction()関数を使用していました。

9.4.21 ネイティブコードの実行 – ユーザイベントの発行

ネイティブコード(Android 用 Java や iOS 用の Objective C)から直接ユーザイベントを発行することができるようになりました。この機能は、ビルトインの[外部]イベントが発行された場合にだけできた処理を置き換えることができます。

9.4.22 ネイティブのコード実行 – Magic コントロールの検索

ネイティブコード(Android 用 Java や iOS 用の Objective C)から直接 Magic コントロールの参照符を見つけることができるようになりました。これによって、Magic コントロールの子としてカスタム・ネイティブ・コントロールを追加することができるようになります。

詳細は、Magic xpa のリファレンスや、リッチクライアントサンプル・プロジェクトのプログラム RMO09 を参照してください。Magic' の[グループ]コントロールの内部にネイティブなラベルコントロールを追加しています。

9.4.23 ネイティブコードの実行 – サンプルの更新

ネイティブコードのサンプル(リッチクライアントサンプル・プロジェクトのプログラム RNCxx)は、前述の新しいネイティブコードの実行機能を使用するために更新されました。

Android のみ

9.4.24 コンボボックス – ネイティブ表示

[コンボボックス]コントロールは、(長方形のプッシュボタンのような表示の代わりに)Android OS のテーマに基づいて表示されるようになりました。

9.4.25 ポップアップウインドウ – 開始時の位置、サイズ、位置のサポート

ポップアップウインドウの最初の位置をカスタマイズすることができるようになり、(iOS の場合のように)ウインドウの開始時の位置やサイズと位置を定義することができるようになりました。

9.4.26 デフォルトテーマ – 動作の変更

ActionBar に対応するため、デフォルトのテーマとして AppCmpat.Light を使用するようになりました。

ActionBar で利用できるテーマは、以下の3つだけになります。

- Theme.AppCompat
- Theme.AppCompat.Light
- Theme.AppCompat.Light.DarkActionBar

RIAModules¥Android¥Source¥MgxpaRC¥res¥values フォルダ内にある themes.xml ファイルを修正することでテーマを変更することができます。

9.4.27 プッシュ通知機能 – 動作の変更

統合されたプッシュ通知機能のプロジェクト ID は、新しく追加された ClientNativeCodeExecute()関数を使用して設定することができるようになりました。

9.4.28 ネイティブコードの実行 – 内部のリネーム

内部の中心的なライブラリパッケージ名が、com.pdac.myact から com.magicsoftware.core に変更され、コアの主要なクラスは GlobalClass から CoreApplication に変更されました。

この変更のために、Android アプリケーションファイル(例えば AndroidManifest.xml と Java ファイル)も変更されました。

ネイティブコードを Android アプリケーションに追加した場合、アップデートされたファイルを変更したり、別々のファイルにカスタムコードを記述し、コードを実行するために新しいネイティブコード実行機能を使用することができます。

9.4.29 SDK レベルの最小値の変更

Android クライアントをビルドする上で必要な SDK レベルの最小値が API19(Android 4.4.2)になりました。これ以前のバージョンの OS でも実行させることはできます。(2.5b)

iOS のみ

9.4.30 Xcode6.1 対応

iOS クライアントを更新する場合は、プロジェクト内の project.pbxproj と AppDelegate.mm を差し替える必要があります。

9.4.31 iPhone6 対応

iPhone6 用の起動イメージが追加されました。

iOS クライアントを変更する場合、project.pbxproj と MagicApp-Info.plist の各ファイルの変更内容を反映するようにしてください。

9.4.32 64 ビット対応

iOS クライアントは、ARM64 アーキテクチャー（Xcode の Build Settings タブ内の Architecutes）を使用してコンパイルすることができるようになりました。

これは、2015 年 2 月 1 日から新しいアプリを App Store にアップロードする場合や、2015 年 6 月 1 日から既存のアプリを更新する場合に必要となります。iOS クライアントを変更をする場合、project.pbxproj ファイルにに対する変更内容を反映されていることを確認してください。

9.4.33 ポップアップフォームのバルーン効果の削除

iPad デバイス上で表示されるポップアップフォームの（フォームの［開始時の位置］特性が「OS デフォルト」または、「OS デフォルト位置」の場合、フォームの端が吹き出しのように表示されされる）バルーン効果は、iOS 8 との非互換性のために削除されました。

RIA オフライン データの同期

Magic xpa

第 2 版　2015 年 9 月 16 日

発行　〒169-0074
東京都新宿区北新宿二丁目21番1号
新宿フロントタワー24階
マジックソフトウェア・ジャパン株式会社
http://www.magicsoftware.co.jp/