SQL Anywhere 12 での Ruby の使用

このマニュアルでは、SQL Anywhere 12 のネイティブ Ruby ドライバーのセットアップ方法について説明します。

はじめに

SQL Anywhere は、バージョン 11 から Ruby プログラミング言語をサポートしています。Ruby で SQL Anywhere データベースにアクセスするためのインタフェースが 3 つあります。ネイティブ C のドライバ―ラッパー、Active Record アダプター、および DBI ドライバーです。

このマニュアルでは、ネイティブドライバーのセットアップと単純なデータベース操作の実行について説明します。SQL Anywhere 12 の他に Ruby 1.9.2 がインストールされていて動作可能であり、バイナリーディレクトリーがシステムパスに追加されていることを前提としています。このマニュアルは、他のビルドおよびプラットフォームにも適用可能ですが、具体的には、MinGW 4.5.1 でコンパイルされた Ruby 1.9.2 と Windows XP SP3 上の SQL Anywhere 12 を使ってテストが行われました。

ネイティブ C のドライバーは、SQL Anywhere API for C/C++ を使用します。この API の詳細については、以下を参照してください。

http://dcx.sybase.com/index.html#1201/ja/dbprogramming/sql-anywhere-database-api.html

SQL Anywhere gem のコンパイル

最新版の SQL Anywhere gem を使用するには、ソースからコンパイルする必要があります。Rubygems.org は必ずしも最新のバイナリーではないため、別のコンパイラを使用して、インストールした Ruby との互換性を確保する必要があります。

ソースコードの取得

ソースコードは、GitHub 上にある git リポジトリーから入手可能です。ソースコードをダウンロードするには、Windows の場合は msysgit、Linux および Mac OSX の場合は git パッケージをインストールします。以下のコマンドを実行することにより、リポジトリーのクローンを作成できます。

```
git clone git://github.com/sqlanywhere/sqlanywhere.git
```

Windows 用ツール

Ruby DevKit を使用すれば、最も簡単に Windows 上に SQL Anywhere ドライバーを構築し、一般的な RubyInstaller バイナリーとの互換性を確保できます。これには、MinGW コンパイラーと C ライブラリの構築およびリンクに必要なその他の Unix に相当するユーティリティーが含まれます。

1. RubyInstaller ダウンロードサイトから自己解凍型実行可能ファイルをダウンロードし、インストーラーを実行します。
2. スペースを入れずにターゲットディレクトリーを指定し (例:`C:¥Ruby192¥devkit`)、インストール処理を進めます。
3. コマンドプロンプトウィンドウを開き、DevKit インストールディレクトリーに移動します。
4. `ruby dk.rb init` を実行してから、`ruby dk.rb install` を実行することにより、インストールされている既存の Ruby に DevKit のユーティリティーを増補します。
5. コンパイルを単純化するために、`C:¥Ruby192¥devkit¥bin` と `C:¥Ruby192¥devkit¥mingw¥bin` の両方をシステムパスに追加します。

Linux 用ツール

ほとんどの Linux ディストリビューションは、SQL Anywhere ネイティブドライバーの構築に必要なツールがすべてパッケージ化されて提供されています。もしも、`gcc` や `make` などの基本的なツールが欠けている場合は、インストールする必要があります。ほとんどのよく知られているディストリビューションについては、オンラインでインストール手順を参照できます。

構築とインストール

クローンを作成した git リポジトリーに移動し、`rake gem` を実行します。構築が成功したら、`rake install` を実行して次の処理に進みます。ネイティブ SQL Anywhere ドライバーがコンパイルされ、Ruby gem として追加されます。

インストールしたドライバーが動作することを確認するためのテストを行います。

1. テストディレクトリーに移動します (`cd test`)。
2. 空のデータベースを作成し (`dbinit test`)、データベースを起動します (dbeng12 test.db)。

3. テストスキーマーを挿入し (`dbisql –c “uid=DBA;pwd=sql;eng=test” test.sql`)、テストを実行します (`ruby test.rb`)。

エラーなしでテストが実行されたら、すべてが適切にインストールされており、ドライバーが使用可能な状態です。

SQL Anywhere ドライバーの使用

SQL Anywhere Ruby API を使用する前に、インタフェースを使用する必要がある各スクリプト内で、SQL Anywhere ライブラリーを参照させる必要があります。以下の行に、SQL Anywhere ライブラリーを示します。

```
require 'sqlanywhere'
```

データベースへの接続を行う前に、インタフェースを作成して初期化する必要があります。同じインタフェースが複数の接続を使用できるため、各インスタンスにつき 1 回だけこれを行います。次の項では、インタフェースを使ってデータベースに接続する方法について説明します。

```
api = SQLAnywhere::SQLAnywhereInterface.new()
SQLAnywhere::API.sqlany_initialize_interface( api )
api.sqlany_init()
```

これで、ライブラリがロードされて接続可能になり、API が初期化できるようになりました。データベースを必要とするすべての作業が完了したら、インタフェースを解放する必要があります。接続の切断後に、以下の 2 つの手順を実行します。

```
api.sqlany_fini()
SQLAnywhere::API.sqlany_finialize_interface( api )
```

接続を開いて閉じる

接続オブジェクトには、データベースへのリンクが含まれています。これは、実行されるクエリーがどのように渡されるかを示します。接続オブジェクトは、必要なデータベースごとに作成する必要があります。まず、空の接続オブジェクトを作成します。

```
conn = api.sqlany_new_connection()
```

次に、接続文字列を使って接続を初期化します。

```
pi.sqlany_connect( conn, connection-string )
```

接続文字列は、接続の確立に必要なキーワード=値のペアをセミコロンで区切って指定します。一般的なパラメーターは、以下のとおりです。

- **DataSourceName=dsn** この接続パラメーターの省略形は **DSN**=dsn です。たとえば、DataSourceName="SQL Anywhere 12 Demo" と指定します。
- **UserID=user-id** この接続パラメーターの省略形は **UID**=user-id です。たとえば、UserID="DBA" と指定します。
- **Password=passwd** この接続パラメーターの省略形は **PWD**=passwd です。たとえば、Password="sql" と指定します。
- **ServerName=demo** この接続パラメーターの省略形は **SERVER**=demo です。たとえば、eng=”demo12db” と指定します。
- **DatabaseFile=db-file** この接続パラメーターの省略形は **DBF**=db-file です。たとえば、DatabaseFile=demo.db と指定します。

接続パラメーターの完全なリストについては、 接続パラメーターを参照してください。

データベースとの接続を切断するには、2 つの手順を実行します。まず接続を切断してから、接続オブジェクトを解放します。

```
api.sqlany_disconnect( conn ) api.sqlany_free_connection( conn )
```

データの選択

接続が開かれたら、データベースへの問い合わせが可能です。データベースで SQL 文を実行するためのメソッドは 3 つあり、それぞれに違いがあります。`sqlany_execute()` メソッドは、`sqlany_prepare()` によって準備された初期化文オブジェクトを取得し、クエリーの成功を返します。以下のサンプルコードは、パラメーター付きの文を準備し、結果を出力します。

```
stmt = api.sqlany_prepare( conn, “SELECT * FROM Products WHERE ID >= ?” )
rc, param = api.sqlany_describe_bind_param( stmt, 0 )
param.set_value(302)
rc = api.sqlany_bind_param( stmt, 0, param )
rc = api.sqlany_execute( stmt )
```

```
num_rows = api.sqlany_num_rows( stmt )
num_rows.times {
    api.sqlany_fetch_next( stmt )
    num_cols = api.sqlany_num_cols( stmt )
    for col in 1..num_cols do
        info = api.sqlany_get_column_info( stmt, col - 1 )
         unless info[3]==1 # Don't do binary
            rc, value = api.sqlany_get_column( stmt, col - 1 )
            puts "#{info[2]}=#{value}"
        end
    end
    print "¥n"
}
```

`sqlany_fetch_next()` は、ローポインタを次のローに進めてから、新しいローのデータをフェッチします。デフォルトではローポインタが最初のローよりも前の場所を指しているため、データの取得前にこのメソッドを呼び出す必要があります。

データの挿入

前の項で使用した方法と同様の方法で、新しいローを挿入できます。ここでは、名前のないパラメータをもう一度使用して、製品データの 2 つの配列を挿入します。接続がコミットされるまでは、他の接続から新しいローにアクセスすることはできません。

```
rows = [[550, 'Ski Mask', 'Wool Ski Mask', 'One size fits all', 'Black', 24, 12.00], [551, 'Ski Mask', 'Wool Ski Mask', 'One size fits all', 'Red', 18, 12.00]]

stmt = api.sqlany_prepare( conn, "INSERT INTO Products (ID, Name, Description, Size, Color, Quantity, UnitPrice) VALUES (?, ?, ?, ?, ?, ?, ?)")

num_params = api.sqlany_num_params( stmt )
rows.each {|row|
    num_params.times {|param_idx|
```

```
            rc, param = api.sqlany_describe_bind_param( stmt, param_idx )
            param.set_value(row[param_idx])
            rc = api.sqlany_bind_param( stmt, param_idx, param )
    }
    rc = api.sqlany_execute( stmt )
}

rc = api.sqlany_commit( conn )
```

エラーの確認

データベースの操作を行ったら、返されたエラーステータスコードで操作が成功したかどうかを確認する必要があります。各例では、rc を使用してこの値が保存されていました。成功した場合は値 1、失敗した場合は値 0 が返されます。メソッド `sqlany_error()` は、接続オブジェクトに保存されている最後の SQL Anywhere エラーコードと、それに対応するメッセージを返します。

```
code, msg = api.sqlany_error( conn )
puts "Code=#{code} Message=#{msg}"
```

必要に応じて、最後に保存されたエラーコードを消去できます。

```
api.sqlany_clear_error( conn )
```

結論

このマニュアルでは、ネイティブ Ruby ドライバーを使った基本的なデータベース接続と、SQL Anywhere データベースでの文の実行について説明しました。SQL Anywhere 12.0.1 用の Ruby API リファレンスの詳細については、以下を参照してください。

http://dcx.sybase.com/index.html#1201/ja/dbprogramming/pg-ruby-api.html*d5e38368

ORM (object-relation mapping ：オブジェクト関係マッピング) 用の ActiveRecord アダプターもあります。Rails 開発用アダプターのセットアップの詳細については、以下を参照してください。

http://dcx.sybase.com/index.html#1201/ja/dbprogramming/pg-ruby-rails.html.