`CodeGen_MySQL_UDF` - the MySQL User Defined Functions code generator

`CodeGen_MySQL_UDF` - the MySQL User Defined Functions code generator

Table of Contents

List of Examples

Chapter 1. Introduction

1.1. What is it?

`CodeGen_MySQL_UDF` is a tool that can automatically create the basic framework for MySQL User Defined Functions (UDF) from a rather simple XML specification file. The actual functionality is provided by the script `udf-gen` that is installed by the `CodeGen_MySQL_UDF` package.

The code generated by `udf-gen` is designed to work with MySQL versions 3.23, 4.0, 4.1 and 5.0 without modifications (although it is usually to compile it for a specific server version).

1.2. Features

`udf-gen` tries to support as many UDF writing aspects as possible. This currently includes code generation for simple and aggregate functions and preparation of `configure` and `Makefile` related build files.

1.3. Installation

1.3.1. Online installation

The packages has not yet been accepted by PEAR (its in voting stage right now) so online installation is not yet possible.

1.3.2. Installing from package files

A test package is available at `http://hartmut.homeip.net/CodeGen-MySQL-UDF-0.9.0dev.tgz`. The package depends on `CodeGen` so this needs to be available in your local PEAR installation before you can install the test package.

1.3.3. Installing from PEAR CVS

The package is not yet available from PEAR CVS as has not yet passed the proposal process.

1.4. How to use it

Given that you already have written your XML specs file invocing `udf-gen` is as simple as:

```
udf-gen specfile.xml
```

`udf-gen` will parse the specs file, create a new subdirectory and puts all generated files in there. The generated code is ready to be compiled using the usual

```
configure; make
```

sequence.

Below you find a hardcopy of `udf-gen --help` output:

```
udf-gen 0.9.1dev,Copyright (c) 2003-2005 Hartmut Holzgraefe
Usage:

/usr/local/bin/udf-gen [-hxf] [-d dir] [--version] specfile.xml

  -h|--help          this message
  -x|--experimental  enable experimental features
  -d|--dir           output directory
  -f|--force         overwrite existing files/directories
  -l|--lint          check syntax only, don't create output
  --version          show version info
```

Chapter 2. The XML description

2.1. Basics

The top level container tag describing an extension is the `<extension>` tag. The name of the extension is given in the `name=...` attribute. The extension name has to be a valid file name as it is used the extensions directory name.

You can specify which CodeGen_MySQL_UDF version your specification file was build for using the `version=...` attribute. The `udf-gen` command will not accept specifications written for a newer version of CodeGen_MySQL_UDF than the one installed. If the requested version is older then the current one then `udf-gen` will try to fall back to the older versions behavior for features that have changed in incompatible ways.

> **Note:** So far no such changes have happened.

The tags `<summary>` and `<description>` should be added at the very top of your extensions. The summary should be a short one-line description of the extension while the actually description can be as detailed as you like.

Example 2-1. Extension basics

```
<extension name="sample" version="0.9.0">
 <summary>A sample UDF extension</summary>
 <description>
  This is a sample extension specification
  showing how to use CodeGen_MySQL_UDF for
  extension generation.
 </description>
 ...
```

2.2. Release information

The release information for your UDF extension should include the extension authors and maintainers, the version number, state and release date, the chosen license and maybe a change log describing previous releases.

The `<maintainers>`, `<release>` and `<changelog>` tags specifications are identical to those in the PEAR `package.xml` specification so please refer to the PEAR documentation here.

Example 2-2. Release information

```
...
  <maintainers>
    <maintainer>
      <user>hholzgra</user>
      <name>Hartmut Holzgraefe</name>
      <email>hartmut@php.net</email>
      <role>lead</role>
    </maintainer>
  </maintainers>

  <release>
    <version>1.0</version>
    <date>2002-07-09</date>
    <state>stable</state>
    <notes>
     The sample extension is now stable
    </notes>
  </release>

  <changelog>
    <release>
      <version>0.5</version>
      <date>2002-07-05</date>
      <state>beta</state>
      <notes>First beta version</notes>
    <release>
    <release>
      <version>0.1</version>
      <date>2002-07-01</date>
      <state>alpha</state>
      <notes>First alpha version</notes>
    <release>
  </changelog>
...
```

The `<license>` tag is a little more restrictive as its `package.xml` counterpart as it is used to decide which license text should actually be written to the `LICENSE`. For now you have to specify either `GPL`, `LGPL` or `BSD`, any other value is taken as '*unknown*'.

Example 2-3. License

```
...
  <license>GPL</license>
...
```

2.3. Functions

Two different kinds of functions may be defined using the `<function>` tag: `regular` and `aggregate` functions. The function type is determined by the `type=...` attribute and defaults to `regular`.

The function name is defined using the `name=...` attribute and has to be a valid C function name.

2.3.1. Return type

The return type of a function is defined using `returns=...`, possible values are `string` (default), `int`, `real` and `datetime`. For a function that may return `NULL` values the `null='yes'` attribute has to be set.

The `length=...` attribute can be used to define the max. length for a `string` result or the number of sigificant digits for `int` and `real`. For `real` results the number of significant decimals can be defined using the `decimal=...` attribute.

2.3.2. Parameters

Function parameters are defined using the `<param>` tag. The parameter name is defined by the `name=...` attribute and has to be a valid C variable name. The parameter type can be one of `string`, `int`, `real` or `datetime` and is defined using the `type=...` attribute.

For each parameter set of C variables starting with the same name is generated that can be used within the functions code snippet. The actual variable names and types depend on the parameter type.

For each parameter a variable by the name of the parameter is created, the variable is of type `char *` for `string` and `datetime` parameters, for `int` it is of type `long` and for `real` parameters a `double` variable is created.

For `int` and `double` parameters an additional variable *name*`_is_null` is created

2.3.3. Private data

It is possible to define an associated data structure for a function that can store data to be shared across all calls to this function during the execution of a statement. This data structure can be used to manage allocated buffers across all calls or to store the intermediate data while processing an aggregate group.

The elements of such a data structure are defined using `<element>` tags within a `<data>` section. Each element needs to be given a valid C name, type and default value using the `name=...`, `type=...` and `default=...` attributes.

Within the functions code sippets the private data can be accessed using the `data` pointer that is created and initialized in the generated wrapper code.

Example 2-4. Accessing private data

```
<function ...>
 <data>
  <element name="txt" type="char *" default="NULL"/>
 </data>
 <init>
  data->txt = (char *)malloc(...);
 </init>
 <code>
  strcpy(data->txt, "...");
 </code>
 <deinit>
  free(data->txt);
 </deinit>
</function>
```

2.3.4. Execution of regular functions

A regular function is initialized once for each SQL query it is used in. Before the actual execution starts the function is initialized by calling the functions `init` handle and after execution the functions `deinit` handle is called to clean up. During the execution phase the actual function handle is called for every result row.

Code for the inid, deinit and execution phase can be added to a function using the `<init>`, `<deinit>` and `<code>` tags.

Code in <init> is usually used to allocate and initalize private data. Parameter count and type checking and allocation of the private data structure is handled by the generated code already so the <init> code doesn't have to take care of this anymore.

The <deinit> code section usually only has to take care of freeing any resources held by elements of the private data structure.

The actual <code> section is supposed to perform the actual functionality of the function by processing its parameters and returning a result value.

... RETURN macros ...

2.3.5. Execution of aggregate functions

The calling sequence of the different handlers of an aggregate function is a little more complicated than for a regular function. Both share the <init> and <deinit> handlers that are called before and after executing the actual SQL statement. Two additional handlers <start> and <add> are called at the beginning of each new group in the result and for each row in that group. The <result> handler is called after the last row of each group has been processed and is supposed to return the aggregated result for the group.

Example 2-5. A simple aggregate function

```
<function name="sum" type="aggregate" returns="int">
 <param name="val" type="int"/>
 <data>
  <element name="sum" type="long" default="0"/>
 </data>
 <start>
  data->sum = 0;
 </start>
 <add>
  data->sum += val;
 </add>
 <result>
  return data->sum;
 </result>
</function>
```

2.4. Custom code

Custom code may be added to your extension source files using the <code> tags. The role=... and position=... tags specify the actual place in there generated source files where your code should be inserted.

Possible roles are 'code' (default) for the generated C or C++ code file and 'header' header file. Possible positions are 'top' and 'bottom' (default) for insertion near the beginning or end of the generated file.

2.5. config.m4 fragments

Additional configure checks can be added to the generated config.m4 file used by Unix/Cygwin builds using the <configm4> tag. Using the 'position' attribute it is possible to specify whether the additional code is to be added at the top or bottom of the config.m4 file.

Example 2-6. config.m4 additions

```
<configm4>
  AC_CHECK_PROG(RE2C, re2c, re2c)
  PHP_SUBST(RE2C)
</configm4>
```

2.6. Makefile fragments

Makefile rules may be added using the <makefile> for Unix/Cygwin builds. Using this it is possible to add dependencies or build rules in addition to the default and auto generated rules.

Example 2-7. Makefile fragments

```
<makefile>
$(builddir)/scanner.c: $(srcdir)/scanner.re
  $(RE2C) $(srcdir)/scanner.re > $@
</makefile>
```

Chapter 3. XML input parsing

3.1. Includes

The XML parser used by `CodeGen_PECL` supports inclusion of additional source files using three different ways:

- external entities
- a subset of XInclude
- the `source` attribute of `<code>` tags

3.1.1. External entities

In SGML and early XML system entites were the only include mechanism availabe. System entities have to be defined in the documents DOCTYPE header, later on in the document the entity can be used to include the specified file:

Example 3-1. System Entities

```
<?xml version="1.0" ?>
<!DOCTYPE extension SYSTEM "../extension.dtd" [
<!ENTITY systemEntity SYSTEM "parsing_1.xml">
]>
<extension name="foobar">
...
&systemEntity;
..
</extension>
```

3.1.2. XInclude

The `CodeGen` XML parser supports a simple subset of XInclude, it is possible to include additional specification files using the `href=...` attribute of the `<include>` tag:

Example 3-2. XInclude

```
<extension name="foobar" xmlns:xi="http://www.w3.org/2001/XInclude">
 ...
 <xi:include href="foobar_2.xml"/>
 ...
</extension>
```

The `parse=...` attirbute is also supported, using `<include parse='text' href='...'/>` it is possible to include arbitrary data without parsing it as XML.

Example 3-3. Verbatim XInclude

```
<extension name="foobar" xmlns:xi="http://www.w3.org/2001/XInclude">
 ...
 <description><xi:include href="README" parse="text"/></description>
 ...
</extension>
```

Other `<include>` features and the `<fallback>` are not supported yet, and most of them won't make sense in this context anyway.

3.1.3. `<code>` tags

In most places the `<code>` tag supports loading of its content using its `src=...` attribute:

Example 3-4. Using `<code src="...">`

```
<function name="foobar">
 ...
 <code src="func_foobar.c"/>
</function>
```

3.2. Verbatim text data

C code usually contains quite a few >, < and & characters all of which need to be escaped in XML. This can be done by either converting them into entities all over the place or by embedding the code into CDATA sections:

Example 3-5. CDATA sections

```
<code>
<![CDATA[
  foo->bar = 42;
]]>
</code>
```

Typing <![CDATA[can become rather annoying over time (esp. on a german keyboard), so i introduced the <?data processing instruction into the CodeGen XML parser as an alternative to CDATA:

Example 3-6. <?data processing instruction

```
<?data
  foo->bar = 42;
?>
```

Chapter 4. Usage

4.1. Invocation

The transformation of a XMP specification file into an UDF directory is done by simply calling the `udf-gen` command with the XML filename as argument:

```
udf-gen specfile.xml
```

`udf-gen` will refuse to overwrite an existing UDF directory (as changes made in there may be lost) unless you call it with the `-f` or `--force` option:

```
udf-gen -f specfile.xml
```

4.2. Configuration

You need to configure your UDF for your actual build system before compiling it. `udf-gen` has already created the necessary autotools input files and run `autoconf` and friends on them so that a `configure` file is already available.

You need to run `configure` to configure your UDF source for your system installation. Most of the time just running `configure` will be sufficient as appropriate defaults should be picked by the script.

If the `mysql_config` binary is noth in your `$PATH` or if you want to build the UDF for another MySQL installation than the default one you have to speficy the location of `mylsq_config` when calling configure:

```
configure --with-mysql=/path/to/bin/mysql_config
```

If your extension relies on external libraries installed in non-standard places you may want to run `configure` with the appropriate `--with-...` options.

You can find out about the `--with-...` (and other) options provided by a `configure` script by running:

```
configure --help
```

4.3. Compilation

After configuring your UDF the actual compilation is done by the `make` command. No further parameters are needed at this point:

```
make
```

4.4. Testing

Currently no testing infrastructure is generated, it may be provided by a future version though. I startet to work on a `mysql_udf` PECL extension that allows PHP to load UDF libraries and to call the functions provided by it. This might be used together with the PHP test infrastructure to test UDFs, or maybe test cases for the MySQL test infrastructure could be generated instead. All this requires further investigation and work being done though.

4.5. Installation

There is no `make install` target yet as it is hard to automaticly find the right place to put the generated UDF .so libraries. Putting them into `/usr/lib` is a safe bet but usually you don't want to have them there ...